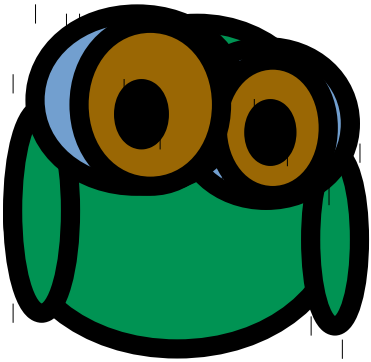
www.COOGL.org

COOGL



pronounced *see-oogl* as in:

"see ogly eyed bird ogling at you"

Concurrent Object Oriented Generic Language

1.3.079-lo-6.0.6.2

This book is dedicated to my mother, Adina de Pantin (R.I.P.), who persevered through life with the single minded purpose to raise her children and aim their lives in the right direction; and to

Professor “Killer” Francisco Hernández (R.I.P.), for showing me the beauty of Mathematics during my five years of high school at Colegio Don Bosco (Caracas, Venezuela).

Ramón G. Pantin

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

1 - Introduction

"For infrastructure work, C will be hard to displace."

-- Dennis M. Ritchie

COOGL is based on a subset of the C language, enhanced with safe, concurrent, object oriented, and generic programming support.

This book does not require that the reader be familiar with the C programming language, the complete COOGL language is described. Nonetheless, familiarity with C is expected from most users of COOGL, the book is organized to satisfy both audiences. The few differences between COOGL and C are explained in Appendix §3D (page 307), which C programmers will want to refer to. Programmers that are not familiar with C might also want to read *The C Programming Language* by Kernighan and Ritchie.

1.1 Rationale for COOGL

The large majority of the world's system software infrastructure is written in C or in C++. Software at risk from the unsafe nature of C and C++ includes: operating systems, virtual machine hypervisors, database servers, transaction monitors, application servers, web servers, file servers, backup systems, compilers, run-time systems, industrial control systems, web browsers, networking infrastructure, security, authentication, encryption, and a large number of applications built on top of these technologies, even if those applications are written in safe languages.

A few safe, or safer, programming languages, such as C#, Java, Eiffel, and Go, are used at the higher levels of application programming, but the core infrastructure continues to be written primarily in C or C++. The gap between C and C++ and these other languages is large, causing large bodies of system software to continue to be maintained and enhanced in C and C++ instead of being rewritten in safe languages. The fundamental problem with those safe languages is that their memory management approach, through mandatory garbage collection, and their memory safety approach, through an extremely tight type system, makes their use inappropriate as an evolutionary path for existing C and C++ code.

As the world's dependence on information systems continues to grow, it is important that an evolutionary path exist for these systems to be reengineered, incrementally, into systems that are safer through the use of a safe programming language for

system software, COOGL is that language.

The problems caused by the unsafe nature of C and C++ are at least an order of magnitude more complex when shared memory multi-processing is involved through multi-threaded programming as a means to take advantage of the additional performance provided by modern multi-core systems. The safe aspect of COOGL's memory management support is specially well suited for shared memory multi-processing systems. COOGL's memory management is based on a stable type memory model and its type safe approach results in a safe programming language suited as an evolutionary path for C code bases. Safety in COOGL is provided while preserving C's rich memory manipulation support where it is most useful, which is when manipulating data with externally imposed representations. For example in the implementation of protocols, storage systems, and other information processing where detailed memory layout control is fundamental.

COOGL's enhancements: concurrent, object oriented, and generic programming support, are all optional, they are used only when the programmer requires them, they don't impose any overheads if they are not used. Of COOGL's enhancements, only safe programming is used by default, most system software can be written completely as safe software, unsafe operations have to be requested through the `unsafe_cast()` operator, unsafe facilities are provided to allow low level system software to be written, for example a memory allocator for objects of any type. The rationale for including unsafe support as part of COOGL is to ensure that system software, at any level, can be written completely in COOGL, possibly with a small amount of assembly language for the lowest level of systems programming such as the lowest levels of interrupt and exception handling, context saving and restoring, etc. Because safety is such an important aspect of COOGL, this book doesn't make use of unsafe code other than in the examples used to describe the `unsafe_cast()` operator itself.

COOGL doesn’t mandate, or even encourage, that garbage collection be the dynamic memory management technique that must be used by programs written in COOGL. Some programs will not use garbage collection at all. Other programs might use some amount of garbage collection, for example type based garbage collection that only garbage collects objects of a few specific types. Some other programs might use garbage collection for all dynamically allocated memory. Garbage collection for all memory, or a few specific types, can be fully implemented in the language, most implementations of garbage collection require writing unsafe code.

A particularly troublesome area with C and C++ is that some compilers take many liberties under the guise of optimization and the excuse of *undefined behavior* to make code that the programmer wrote disappear into thin air, in a way these compilers are introducing security holes and data integrity risks into existing code. Code that might have worked for years might suddenly stop working because it was com-

piled with a newer compiler that optimizes undefined behavior, when it sees it, to delete the programmer's code and make the program go wrong faster. This is intolerable and has no end in sight in C or C++. COOGL addresses all of these issues in a way that those kinds of problems don't exist for code written in COOGL.

C++ is a mind-numbingly and absurdly complex language, its complexity continues to grow without restraint. C++ causes system infrastructure written in it to be at further risk because of the complexity that the language itself engenders. COOGL provides an avenue to reengineer C++ systems away from that run-away complexity train. It seems that the people evolving the C++ language and its compilers have automated themselves into their jobs through a never ending stream of proposals and language changes. These experts in the intricacies of C++ will write examples into their books that they themselves can't tell are broken, they then encourage that style of programming, then some tool finds that the code is not expected to work, none of the experts could tell that the code was not supposed to work, because it relied on unspecified behavior, and that if it worked it was by luck during compilation, and then they go change the language to make the example code and all its derivatives that propagated into actual programs work. The worst aspect of C++ is that no single line of code can ever be examined with certainty of what it does, first a myriad lines of header files and templates in them have to be investigated to ensure that nothing in them causes the line to do something unexpected, between the C preprocessor, operator overloading, function name overloading, templates, and thrown exceptions, the reader of the code can not be certain of anything. Unless the programmer is intimately aware of every aspect of the code in the header files the programmer can't tell what might be happening because of the level of obfuscation that C++ allows and encourages.

1.2 Object oriented terminology

A few terms used in object oriented programming are used in this book. Brief definitions of these terms follow, subsequent paragraphs expand on these concepts:

- *Class* – a programmer or language defined type and the operations on objects of that type.
- *Object* – an instance of a specific type or class, for example a variable or a dynamically allocated entity of that type.
- *Pointer* – data that can contain the address of other data, pointers are objects.
- *Constructor* – a procedure that initializes raw memory into the initial state of an object.
- *Destructor* – a procedure that deinitializes an object and turns it back into raw memory.

- *Member* – an entity, for example a variable, declared within a class declaration. Other kinds of members are: functions, constants, enumerations, and types. The term member applies to all of them, when referring to a specific kind of member, terms such as: *member variable*, *member function*, or *member type* are used.
- *Static member* – is a per class member, not a per object member. A static member can be accessed independent of any object, an object doesn't have to be provided to access a static member. A static member is nothing more than a global entity specific to a class.
- *Non-static member* – a per object member, not a per class member. Each object has its own instance of the member. An object must be specified to access a non-static member.
- *Member function* – a member function is a function declared as a member of the class.
- *Static member function* – is a member function that does not require an object of the class to be provided, it doesn't operate on a specific object.
- *Non-static member function* – is a member function that requires an object of the class to be provided for it to operate on.
- *Inheritance* – a mechanism that allows an existing class to be used in the specification of the interface or the implementation of another class, usually in a way that allows objects of the new class type to be used as if they were objects of another class type, even though their implementation details might be different. When inheritance is used it is said that the new class *inherits* from the other class.
- *Derived class* – a class that inherits from another class can be described as being *derived* from it.
- *Base class* – a class that a derived class inherits from can be described as being the *base* class of the derived class.
- *Ancestor class* – a class that is either a base class of another class, or that is an ancestor class of one of its base classes.
- *Descendant class* – a class that has another class as an ancestor class, is a descendant class of the other class.
- *Related classes* – a set of classes is said to be related to each other if they have a common ancestor, or if one of them is an ancestor of all the others.
- *Polymorphism* – the ability of related classes to have different implementations of the same member functions. Polymorphism occurs when there is a descendant relationship between two classes and the descendant class rede-

fines a member function defined by the ancestor class. Polymorphism causes the member function that is invoked to be determined by the actual type of the object, irrespective of the type of the pointer that is used to refer to the object, frequently a pointer to an ancestor class.

- *Created* – when an object is allocated dynamically from a run time memory heap, and constructed, it is said to be *created*. Usually through a static member function, `create()`.
- *Destroyed* – when a created object is destructed and its memory released to the run time memory heap it is said to be *destroyed*, usually through a non-static member function, `destroy()`.

Some object oriented literature refers to regular variables, integer variables for example, as objects. In COOGL an integer variable is indeed an object of the `int` class type. This book refers to them simply as *variables*, unless some object oriented aspect of them is being emphasized.

Most well designed software has functions to initialize data structures, and deinitialize them when they are no longer needed. In object oriented languages it is not the duty of the programmer to invoke these functions, the compiler produces code to cause their invocations without the possibility of the programmer forgetting to do so. The constructor is invoked whenever memory is designated to be used as an object of a given class. For example, a local variable declaration causes the compiler to produce code that invokes the constructor. Similarly, dynamically allocated memory of a given type, allocated from a memory allocator, causes its constructor to be invoked. In non object oriented languages, such as C or Pascal, the programmer must always explicitly invoke the initialization and deinitialization functions.

Because the compiler generates calls to the constructor and destructor functions, they must be known to the compiler. In some languages the code for the constructor and destructor might need to be generated by the compiler, if they are not provided by the programmer, it is generated to ensure that the construction and destruction of the non-static data members of the class occurs. The constructor function in COOGL is not optional and is never generated by the compiler, simply because the class declaration is also the class constructor. The destructor is optional and is generated by the compiler when it is not provided.

1.3 Member function invocation syntax

Consider the following `stack` class, `int` is a built in integer type. Unless you are familiar with other programming languages, you night not understand the code the code yet. For now just read it, the only important aspects to focus on is that the class defines various members, some are data members, e.g. `entries` and `sp`, `MAXENT` is a compile time literal constant, and some member functions: `empty`, `full`, `push`, `pop`,

and top.

```
class stack promise(empty()) {
    pub lit int MAXENT = 100;   // literal constant not variable
    priv int entries[MAXENT];   // array of MAXENT ints
    priv int *sp = &entries[0]; // int pointer, set to point ...
    return;                     // ... to address of entries[0]
    pub bool empty() { return sp == entries; }
    pub bool full() { return sp == entries + MAXENT; }
    pub void push(int v) require(!full()) { *sp++ = v; }
    pub int pop() require(!empty())
                  promise(!full()) { return *--sp; }
    pub int top() require(!empty()) { return sp[-1]; }
}
```

Preconditions and postconditions, promise() and require(), of the class and some of its member functions are also specified. A stack object promises to be empty() immediately after being constructed, to push() an element onto the stack it is required that the stack not be full(), furthermore, after pushing an element onto the stack it promises that the stack() is not empty(), and finally prior to examining the top() element of the stack or using pop() to remove the top element from the stack it is required that the stack not be empty().

The following code makes use of s, a stack object, and p a pointer to an object of stack type, initialized to point to s, its address, i.e. &s, is assigned to p:

```
void use_stack() {
    stack s;
    stack *p = &s;
    s.push(7);
    int seven = p->pop();
    int max = s.MAXENT;
    max = p->MAXENT;
    max = stack.MAXENT;
}
```

The expressions s.push(7) and p->pop() are member function invocations. The dot operator: . is used with an object, for example s, to refer to a member, for example s.push(). The arrow operator: -> is used with a pointer to an object, for example p, to refer to a member, for example p->pop(). The references to the MAXENT literal: s.MAXENT, p->MAXENT and stack.MAXENT are all valid. The later one uses the class name, stack, instead of an object or a pointer to an object, to refer to MAXENT. Use of the class name to refer to MAXENT is valid because MAXENT is a lit declaration, literal values are compile time constants, their values are not different across objects of the class, thus they are accessible as if they were static declarations, they declare per class literal values, not per object literal values.

When a non-static member function is invoked, the object on which the member

function is invoked is passed implicitly as a hidden first argument to the member function. When a static member function is invoked, an implicit object is not passed as an argument, because the static member function does not operate on an object.

The `s.push(7)` and `p->pop()` invocation forms refer to member function names within the scope of the names of the class type of `s`, i.e. `stack`, and of the underlying class type that the declaration of `p` states that it points to, i.e. `stack`. When referring to an object through a pointer the actual member function implementation that is invoked might vary according to the underlying type of the object in question, i.e. when there is polymorphism, which is not the case in this example.

1.4 Hello world and type safe input and output

The traditional *hello world* program in COOGL is:

```
int main() {
    libc.puts("hello, world");
}
```

COOGL does not support functions with variable number of arguments such as `printf()`, they are a wart from C's ancestral BCPL origins, the `on` statement is how type safe input output is implemented without input output being built into the language. A different version of *hello world* is:

```
int main() {
    "hello, world\n".print();
}
```

The type of the `"hello, world\n"` string in COOGL is `strlit(class const char)` a native generic type, short in this case for *string literal of* `const char`, not `const char[]` which is its type in C. COOGL allows types, including native types, to have member functions added to them, thus the `int` and `strlit(class const char)` types can be extended, to add `print()` member functions to them. The COOGL library adds `print()` and various other member functions to these and other types.

The source code for a COOGL program is stored in a file with a `.cog` extension. An example compilation and invocation on UNIX:

```
$ COOGL hello.cog
$ ./a.out
Hello, World.
$
```

Formatted output uses the `on` statement:

```
int main() {
    float f = 78;
    float c = (f - 32) * 5 / 9;
    on ("temperature in Caracas: ";
        f; "(f) "; c.fmt(4,2); "(c)\n") print();
}
```

The program above is a type safe version of this C program:

```
#include <stdio.h>
int main() {
    float f = 78;
    float c = (f - 32) * 5 / 9;
    printf("temperature in Caracas: %f(f) %4.2f(c)\n", f, c);
}
```

The output of both of these programs is:

```
temperature in Caracas: 78(f) 25.56(c)
```

The `c.fmt(4,2)` expression implies that the built in `float` type, the type of `c`, must have been extended with a member function `fmt()`, which returns the value of `c` formatted as a string according to the `printf()` like specified precision.

The `on` statement causes the invocation of a member function on a list of expressions. Its syntax is:

`on` (*semicolon_separated_expression_list*) *function_invocation_expression*

A list of one or more expressions separated by semicolons must be specified within the parenthesis after the `on` keyword. For example:

```
on (expression1; expression2) function(a, b, c);
```

can be thought of as a short hand for:

```
((expression1).function(a, b, c),
 (expression2).function(a, b, c));
```

but when `function()` returns a value, there is more to it, see §9.2 and §9.3.

The types of the arguments in the argument lists of all of the member functions must be compatible with the specified arguments. There is no difference between the `on` statement and the expression statement that it stands for. The expressions used in the argument list are evaluated on each member function invocation.

The above program with the `on` syntactic sugar removed, is harder to read:

```
int main() {
    float f = 78;
    float c = ((f - 32) * 5) / 9;
    ("temperature in Caracas: ".print(),
        f.print(),
        "(f) ".print(),
        c.fmt(4, 2).print(),
        "(c)\n".print());
}
```

The `on` statement can also produce a numeric value which can be used as the means through which end of file or errors can be reported, or to return the number of bytes written, as in `printf()`, or the number of items scanned, as in `scanf()`, see §9.2.

1.5 Compilation model

Programming language definitions usually don't describe their compilation models, though some programming languages do, for example Java not only specifies the compilation model but also the target instruction set, i.e. the Java Virtual Machine, and various other facilities such as class loading and verification.

The COOGL language definition includes the specification that COOGL is compiled into C11 code. The COOGL facility that supports interfacing with C code depends on this specification. The C code that results from the translation from COOGL, is automatically compiled by the C compiler into instructions for the underlying computer system. The compilation model for COOGL is global, the user specifies a set of source files and libraries, the COOGL library is included by default, but it can be excluded. The user perception of the compilation is that all files are compiled together, whether all the files are actually compiled each time the compiler is invoked or only compiled when needed is not specified by the language, the current compiler only compiles a file if it is required to do so, caching transparently the results of prior compilations. It is expected that all compilers will do this.

Function and type declarations are extracted from the source files where they are declared and used to ensure that function invocations and type uses are correct. The C notion of header files does not exist in COOGL, the user does not have to maintain function prototypes and external variable declarations in header files.

Compilation into C allows the native system compilers to be used for code generation, it also facilitates operating system kernel development in COOGL because specific compilation options required for kernel mode development are supported by the underlying C compiler. Options to the C compiler are passed through by the COOGL compiler.

The C code that the COOGL compiler produces is formatted and indented in such a

way so as to keep it as close as possible to the original C code. The intent is not to use the generated C code as a portable machine language. Instead, by maximizing generated code readability, the engineering and verification of the compiler was a simpler effort. Before the first COOGL compiler existed, it had to be written in some language other than COOGL. It was written as the expected output of itself, in C, it was then hand translated to COOGL, and compiled with itself, with its output compared with the original compiler written in C to ensure that they were the same.

The compiler distribution includes the compiler source code in COOGL, together with its output in C11, a thorough set of test cases for regression testing, and build scripts. Part of the compiler installation verification includes ensuring that the output that it produces is identical to the C version of the compiler distributed with it.

1.6 C versions and COOGL ancestry

The C language has had various minor variations, usually compiler or operating system vendor specific. It has had four standard versions:

- The language described in the first edition of *The C Programming Language* by Kernighan and Ritchie, also known as K&R C. Its first edition was the authoritative language definition, it was the de-facto C standard. The first generation of widely available C compilers were based on the UNIX Portable C Compiler (PCC), which reinforced this de-facto standard.
- C89, the first official C language standard (ANSI C89 and ISO C90, informally C89). Its most notable addition was the adoption of C++ syntax for function declarations and prototypes.
- C99, the second official C language standard. Includes a slew of additions to attempt to allow numerical code to be written in C and be competitive with FORTRAN code: complex numbers, type generic math (i.e. `<tgmath.h>` library functions), variable length arrays, and the `restrict` qualifier. Miscellaneous but convenient changes include intermingling of declarations and statements and BCPL comments, both of which were in wide use in existing compilers. Among all the additions, variable length arrays introduced the most complexity.
- C11, the third official C language standard. Its most important enhancement is multi-threading support and a memory model mostly relevant to concurrency. Atomic data types. The `_Generic` keyword that allows type generic macros to choose between different functions based on the type of an argument, this allows FORTRAN like libraries to be written by programmers instead of only by the compiler vendor (as in `<tgmath.h>` in C99). Minor enhancements include Unicode characters and string literals. Optional support for newer floating point and complex number standards. Anonymous struc-

tures and unions, and alignment specification. Additionally the ability for compilers to indicate that they do not support some C99 features (complex numbers and variable length arrays are optional). Most new features in C11 are optional.

COOGL descends primarily from C89 and Simula67. A few enhancements from C99 and C11 were incorporated into COOGL. Some ideas were borrowed from Eiffel. Other COOGL ancestors from its C family lineage are: K&R C, ALGOL68, B, PL/1, BCPL and CPL. Common ancestors of C and Simula67 are: ALGOL60 and FORTRAN.

Both B and K&R C have elements of PL/1. B's `extrn`, `auto`, semicolon terminated statements and `/* comment */` came from PL/1. C got from PL/1 these aspects: `NULL`, `static`, the `->` operator, and the rule that local variables without `extern` or `static` are `auto` by default. COOGL does not descend from C++. If anything, COOGL learned from C++ what not to do, instead of what to do. Though it could be perceived that the keywords `this`, `priv`, and `prot` come from C++, in reality they come from the Simula73: `this`, `hidden`, and `protected`. The use of `priv` instead of `hidden`, and the `pub` keyword are a nod, but also a cleaning up and simplification, of the C++ terminology used in its label-like syntax, which uses: `public:` and `private:` respectively to dictate the accessibility of subsequent declarations.

1.7 C language schism: concurrency and undefined behavior

There has been a growing schism between C compiler writers and C language users. People involved in the development of compilers, and others involved in the specification of programming language standards for C and C++, specifically some of those involved in efforts to specify a memory model for concurrency (starting with efforts in the C++ standardization community that lead to the memory model present in both C11 and C++11) have come to claim that traditional C could have never been used to write concurrent programs because concurrency had not been specified in the C language standard, thus it was not possible to have been able to reliably write concurrent programs in C.

These people forget that well before there was the first standard for C, C89, the language existed and large reliable concurrent programs were written with it, e.g. the UNIX kernel, relational database systems, etc. It is absurd to say that something is impossible while typing those statements most likely on computer systems where the operating system kernel for those systems was most likely written in C, (e.g. MacOS, Windows, Linux, or UNIX), which have long supported hardware concurrency, then they post their words in web servers most likely written in C, also running on SMP systems.

What the compiler writers actually mean is that now that the language has been specified, poorly, say as it was done in C89, and that it doesn't specify concurrency, they don't know when to stop with their unwieldy optimizations for the sake of getting tiny performance improvements in the compilation of sequential benchmarks. Thus compilers have become so aggressive in their optimizations that operating system kernels and other concurrent programs have to be compiled with a series of compiler options to ensure that the compiler doesn't perform optimizations that are useless to the point that they only work on sequential programs, or code generation strategies that are confused, and could make concurrent programs misbehave.

Even with the schism, C programmers know that the C compiler can not assume anything about code that the C compiler is not allowed to see. Thus a function call to a function, that is not expanded inline and that is not subject to global compilation or link time optimization, is the last resort for programmers and the perfect boundary for preventing the compiler from performing optimizations across function calls to them. At those function call boundaries it can be assumed that data that must by then be in their corresponding memory must be stored there by the code generated by the compiler prior to the function call, or the sequential programming model would be broken. Furthermore, because pretty much any memory that is global, or memory on the run-time stack whose address has been taken and passed to functions or stored globally, could have been modified by the function whose code is unknown to the compiler, the calling function is not allowed to cache those values, for example in registers, and must refetch those values from their proper locations after the unknown function returns. Thus the function call boundary when calling functions whose code is unknown to the compiler, global optimizer, and link time optimizer, is the correct place to implement synchronizers. For example, the acquisition and release operation in a mutual exclusion lock together with any memory barriers that might be required by the hardware on weakly ordered memory systems. Thus all has remained well in the implementation of concurrent programs in C. Some kernels that are heavily optimized might desire to have low level assembly functions that manipulate hardware or implement synchronizers expanded inline into their invocation locations, and that requires a more careful dance with the compiler to ensure that it doesn't perform optimizations that cause the code to be incorrect, for example by moving code around it, or caching values in registers, etc. In practice calling functions directly is heavily optimized by modern computer systems, and if a compiler provides a pragma that indicates which registers are affected by a function, then implementing synchronizers as assembly functions, instead of as inline assembly, is usually just as fast as inlining them, the minuscule overhead saved by avoiding a function call is counterbalanced by the reduced code footprint and its impact on the instruction cache.

Another form of the schism, and in some ways even more dangerous, is the *undefined behavior disease* which means that anything that is written in the C language standard and labeled as *undefined behavior* is an opportunity for optimization by the

compiler. The compiler writers, in their search for optimization in the wrong places, identify undefined behavior, and instead of causing a compilation error, use that knowledge to perform optimizations that makes the execution of the program go faster, or misbehave in possibly subtle ways, or crash horrendously, instead of simply allowing what the underlying hardware in the computer system would do under those circumstances. Notions such as *wobbly* data values and other silliness are invented to justify the compiler's behavior. The compiler writers, instead of producing compilers that are more useful to programmers by producing compilation errors when presented with code that would lead to undefined behavior, instead turn working programs that worked with earlier compilers into programs that no longer work.

For example, what could have been a machine dependent operation that might be different between different underlying hardware, for example a shift by 32 bits of a quantity in a data type that is 32 bits wide, a no-operation on x86 but a proper 32 bit shift on POWER, becomes an optimization opportunity for the compiler, which instead of producing a compile time error, causes every value that depends on the result of that computation to be undefined and to delete as much code as possible based on that, silently. What is at the hardware level machine dependent, and well specified, gets turned into an irrational optimization opportunity.

C is the low level language of choice for programming the lowest levels of operating systems on real hardware, real hardware does not have *wobbly data*, nor does it have shift instructions that cause a bunch of dependent operations to be skipped, or that turn one memory load into two memory loads, etc. If real hardware had such behavior it would be labeled as errata, i.e. a hardware bug.

COOGL does not have any undefined behavior, the C11 code it generates does not have undefined behavior either. Any construct that would lead COOGL source code to be compiled into C11 code with undefined behavior causes a compilation error, for example constructs that would be unsafe. Every memory access performed by a COOGL program, that doesn't use the `unsafe_cast()` and doesn't use `NULL`, is a valid memory access, accessing the `NIL` pointer or the set of *trapping addresses* defined by the language are also valid memory accesses, it is valid to access them they reliably cause an exception they are not undefined behavior, COOGL knows about this, the underlying C11 compiler doesn't know about this, and it can not do anything about it other than to perform the memory access that the programmer programmed. The `NULL` pointer (or the value 0 when used as a pointer) are deprecated and should not be used by safe programs, they remain in the language as a bridge for interoperability with C code.

COOGL is a firewall between the needs of the programmers and the degeneration that is occurring to the C language, which is moving farther and farther away from the real world of concrete hardware, and into a world of a needlessly complicated language specifications mostly to satisfy the needs of compiler writers at the cost of

programmers and risking the stability of existing code bases. Evolving the C language from the standards community perspective seems to have become a full time job for a large number of people and there doesn't seem to be much restraint in those efforts, features are being invented irrespective of them having ever been implemented in practice and most without the benefit of even an experimental implementation.

The needs of existing C programs and C programmers are not always addressed by the C standard writers, though the C compiler writers continue to make accommodations to ensure that existing C code doesn't stop working, mostly through compiler options that disable some of the most misguided optimizations. For example the Linux kernel doesn't work with these compilers that use the notion that signed arithmetic overflow is undefined behavior and perform irrational optimizations based on that assumption. For the Linux kernel to work and many other large code bases they need to be with options which turn off quite a few optimizations predicated on undefined behavior. A problem with the C standard is its definition of `volatile` which it has been known to have been incorrect for almost two decades and even with an error report that could have been included in C17, it wasn't, the compiler writers in this case knew what to do and they implemented the behavior desired by the programmers and ignored what was described by the language. The compilers are right in this case, the standard is wrong, hopefully the compiler writers won't forget this and go break some more code in the future because they chose not to update the standard to reflect the intended language design and actual standard practice.

Some of the most absurd undefined behavior over optimizations cause, what used to be trivial in C, to become obfuscated to prevent the compiler from doing stupid things. For example, in an operating system that wanted to initialize some low memory to some specific contents, for example moving exception handler code there, say at physical address 0, prior to turning on the MMU and enabling interrupts, to have to confront the compiler that sees a pointer with value 0 and decides that any memory references based on that pointer are undefined behavior and the compiler can do whatever it pleases it, not to produce an error, but crazy things, for example not to generate any other code for the function it is compiling, not even a return instruction, of course without a warning, simply because the standard says it is undefined behavior. So the C programmer has to write an assembly function that returns a pointer with value 0 and call that so that the compiler stops doing absurd things with its code, remember, in C, the programmers were supposed to know what they were doing, the compiler was supposed to just do it. The new generation of C compiler writers and C standard writers do not seem to have learned that. To any complaint from a C programmer with their compiler's behavior they mutter: *"Its UB, I could corrupt the contents of your files if I wanted to, the standard says I can do whatever I want, go away."* They say these things so often that *UB*, undefined behavior, is part of their everyday lingo.

This is what the original developer of the LLVM project and, at the time, a member of the Clang compiler development team at Apple had to say about the subject, underlined highlights are by the author of this book:

> *"There is No Reliable Way to Determine if a Large Codebase Contains Undefined Behavior"*
>
> *"Making the landmine a much much worse place to be is the fact that there is no good way to determine whether a large scale application is free of undefined behavior, and thus not susceptible to breaking in the future. There are many useful tools that can help find some of the bugs, but nothing that gives full confidence that your code won't break in the future."*
>
> *"The end result of this is that we have lots of tools in the toolbox to find some bugs, but no good way to prove that an application is free of undefined behavior. Given that there are lots of bugs in real world applications and that C is used for a broad range of critical applications, this is pretty scary."* – Chris Lattner (What Every C Programmer Should Know About Undefined Behavior)

Views typical of compiler writers who have hijacked the meaning of C, and only seem to care about performance, resulting in an unsafer language, mostly because the compiler teams where originally tied to the CPU design teams and achieving the highest SPEC performance numbers was the only thing that mattered, while not breaking too much working code. They are all busy rummaging through the standards to see how else they can make things go faster based on undefined behavior:

> *"**Using a Safer Dialect of C ...**"*
>
> *"A final option you have if you don't care about "ultimate performance", is to use various compiler flags to enable dialects of C that eliminate these undefined behaviors. For example, using the -fwrapv flag eliminates undefined behavior that results from signed integer overflow (however, note that it does not eliminate possible integer overflow security vulnerabilities). The -fno-strict-aliasing flag disables Type Based Alias Analysis, so you are free to ignore these type rules. If there was demand, we could add a flag to Clang that implicitly zeros all local variables, one that inserts an "and" operation before each shift with a variable shift count, etc. Unfortunately, there is no tractable way to completely eliminate undefined behavior from C without breaking the ABI and completely destroying its performance. The other problem with this is that you're not writing C anymore, you're writing a similar, but non-portable dialect of C."* – Chris Lattner

So pretty much every operating system kernel, written in C, is from this compiler writer's perspective not written in C but in a dialect of C, thus the schism between the C compiler writers and the users of the compilers grows. Somehow they can not read

that *undefined behavior* in the standard allows also for "*program execution in a documented manner characteristic of the environment,"* because they prefer to do whatever they want instead of what the programmers want. So when they support, begrudgingly, the behavior expected by the programmers they then state that they are writing code in a language that is not C, but a dialect of C, and that the dialect is not portable, which is hilarious because these large code bases are portable to many CPU architectures and many different operating systems.

The historical reality is that in the rationale documents for both C89 and C99 this is the rationale for undefined behavior:

> *"Undefined behavior gives the implementor license not to catch certain program errors that are difficult to diagnose. It also identifies areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior."* C99RationaleV5.10.pdf:11

C designed by Dennis Ritchie and implemented originally by Ritchie, and reimplemented into the PCC (Portable C Compiler) by a small team at Bell Labs is being attributed, incorrectly, this sentiment by Lattner: *"Undefined behavior exists in C-based languages because the designers of C wanted it to be an extremely efficient low-level programming language."* Lattner's loop optimization concern, expressed in his article, is addressed in §10.9.

Historical reality is that Ritchie and Ken Thompson wanted a language into which UNIX could be rewritten from PDP-11 assembly language, they just needed it to be reasonably efficient. Ritchie, Johnson, Lesk, and Kernighan wrote: *"The language is sufficiently expressive and efficient to have completely displaced assembly language programming on UNIX"* when describing C (in his article *"The C Programming Language"* article received for publication on December 5th, 1977 in the Bell System Technical Journal issue July/August 1978 vol 57, no. 6, part 2). Ritchie continues *"C was originally written for the PDP-11 under UNIX, but the language is not tied to any particular hardware or operating system. C compilers run on a wide variety of machines, including the Honeywell 6000, the IBM System/370, and the Interdata 8/32."*

Ritchie closed his paper *"The Development of the C Programming Language,"* a historical account presented in the ACM SIGPLAN History of Programming Languages Conference (HOPL-II) which took place April 20-23 1993 "*it evidently satisfied a need for a system implementation language efficient enough to displace assembly."* The goal was not for C to be *"extremely efficient"* as Lattner incorrectly claims.

The spirit of C, its original spirit, lives on in a family of C compiler's written by Ken Thompson for Plan 9, the grandfather of C, the creator of UNIX and its C's direct ancestor B. Thompson's report about his then new C compilers: *"produce*

medium quality object code." It also lives in the Go programming language, the spiritual descendant of C, that Thompson and his Bell Labs compatriots, from UNIX and Plan 9 fame, created at Google, a language whose definition doesn't have undefined behavior either.

COOGL is a language for the real world, for the needs of C programmers, it is written and supported by C and COOGL programmers that find it useful, and it doesn't turn your COOGL programs into a morass of undefined behavior compiler optimization opportunities for your code to disappear, it doesn't get in the way of what you do.Defined behavior and implementation dependent behavior

COOGL program constructs translate into code that has either defined behavior, i.e. behavior that can be derived from the source code and is specified by the language, i.e. behavior that doesn't vary across compilers and computer systems. Otherwise the construct has implementation dependent behavior which is as close as possible to what the underlying computer system does, it is also what you would expect on that computer system for the construct to do. See §10.7 and §14.

1.8 COOGL syntax and language design philosophy

COOGL is a language designed to be learned and used. Reducing complexity was one of the most important guiding objectives in its design. The language should be easy to learn, simple, with no surprises and no intricate cleverness that requires deep *language lawyering* over the intricate delicate description of the language in a language standard, as C++ does. The meaning of every language construct should be trivial to understand. The fewer the language constructs the better. Many language constructs were considered for the language, and rejected, because of the complexity that they would introduce was not worth their value.

Language design questions were always answered with *keep it as close as possible to C*. In contrast, other languages such as Objective C and C++ have freely imported syntactical constructs from other languages without much consideration of C's syntax. For example, when choosing between the `pub` or `public` keyword names, `pub` was chosen, because C uses abbreviated names, such as `int`, `float`, and `char` (instead of `integer`, `floating`, and `character`). A modifier syntax was chosen for `pub`, like C got `static` from PL/1, instead of C++'s label like `public:` syntax. The expression of inheritance through the `inherit` keyword in COOGL instead of what C++ choose from Simula67, i.e. the colon (i.e. `:`) character in a specialized context to mean inheritance. Other examples include the use of `defer` to indicate the deferred declaration of a member function instead of the specialized use of `=0` used by C++ for a similar purpose.

The intent is to have *no surprises*, anything in the code that looks like C behaves exactly as it does in C. For example in C#:

```
int* p, n;
```

surprisingly, to a C programmer, declares both `p` and `n` as pointers to `int`, whereas in C it declares `p` as a pointer to `int`, and `n` as an `int`. Thus in C and COOGL the use of whitespace should always be:

```
int *p, i;
```

as shown by Kernighan and Ritchie in their C book. C++ suffers from an idiomatic misplaced space disease propagated broadly by Stroustrup's C++ books, because the meaning of the declaration is the same as in C, nonetheless C++ programmers tend to use the misleading convention: `int* p, n;`.

COOGL includes minimal support for language features, the bulk of the functionality is implemented in libraries. There is no syntactical support in the language syntax for: input output, heap based dynamic memory management, threads, or locks. The only syntactical support for dynamic memory allocation in C is the `sizeof` operator, COOGL requires an additional construct, `argsof`. C included variable argument functions to support formatted input output operations, COOGL includes the `on` statement, a general purpose statement that can be used to implement type safe formatted input output.

COOGL is an evolution of a subset of C, it is not a superset of C. Language evolution requires change, if change is restricted to additions, i.e. if removal is not allowed, then the resulting accumulation of features leads to needless complexity. If animal evolution were like most programming language evolution, we would all be wondering why do we still need a monkey like tail!

1.9 Programming language complexity

Some languages are stillborn because their sheer complexity makes their implementation and adoption impossible. An example of that was Algol68, which took its specification through a complexity path that made its specification flawed and incomprehensible, taking until its 3rd specification iteration, Algol68c, to get to the point of being less flawed but still required a very steep learning curve. Eventually it was implemented and used, but its broad adoption never occurred.

Another language that in some way learned from Algol68's design mistakes was Ada, a much less complex language, but quite complex for its time, it only succeeded because of the sheer perseverance and tremendous investment behind it by the USA's DoD (Department of Defense) with its purchasing contracts and other R&D grants, and the myriad defense contractors, defense systems manufacturers, civilian and military aero-space manufacturers, that took it from an almost stillborn language into the language of choice for those systems for weapons and defense systems. In a way we can all feel more safe from a software catastrophe caused by weapon systems or nuclear reactors because these systems are written in Ada instead of C (or C++). If any-

thing we should all be concerned about the F-35 because most of its new subsystems are written in C++, with its legacy sub-systems, from the F-22, remaining in Ada.

Other languages achieve success because they are able to ride on the coattails of an earlier successful language and its ecosystem, for example C++ was able to ride the coattails of C, and evolve, initially slowly into its current high complexity condition. C++'s current tremendous complexity can be attributed, in part, to its initial set of design choices and the complexity that they subsequently engendered. Starting with operator overloading, which forces the introduction of references, and the eventual addition of exception handling, because there is no way to report an error from an overloaded operator. Note that operator overloading was the last feature added to Algol68.

Stroustrup choose complexity instead of simplicity for C++ at every step: choosing to add multiple implementation inheritance, virtual base class, and all that that engenders. Stroustrup's choice of a template mechanism, that from its inception should have served the language users by specifying the requirements of the parameterized types, which had already been done in the Clu language prior to C++'s templates. The subsequent discovery of meta-programming with C++ templates and the complexity that arises from that is enormous. How often does a language gets discovered inside of another language? Lastly the technique of SFINAE (substitution failure is not an error) in C++ templates is mind boggling, it says, generate code according to the template, if the code does not compile, ignore it, and try another template choice.

Choosing complexity over simplicity at every step of the way, and the design by committee that ensued during and after its initial standardization, resulted in the single most complex language that has ever existed, and with no end in sight to its complexity explosion. The language creator, Bjarne Stroustrup, stated in an interview in April 2010, as C++0x was evolving into C++11:

> *"Even I can't answer every question about C++ without reference to supporting material (e.g. my own books, online documentation, or the standard). I'm sure that if I tried to keep all of that information in my head, I'd become a worse programmer. What I do have is a far less detailed – arguably higher level – model of C++ in my head."*
>
> *"What programmers should know is the basic facilities of the language, the basic of the functioning of the main features, and how to gain more knowledge as needed. In other words: People need a model of the language and have access to information sources. I do not require people to believe in magic. Never! There is far less "magic" in C++ than in other modern languages and I think that is part of the problem. You can look at a standard-library algorithm or a boost library and see exactly how it is put together. Sometimes, reading such code is an expert-level task." – Bjarne Stroustrup*

Basically C++ has become a language for two types of programmers, users of C++

and template library writers, Stroustrup's *"expert-level task"* programmers. The language complexity does leak from the expert level written code into the realm of the regular users of C++, making their life just as thrilling when, as Stroustrup stated:

> *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."*

More recently, the paper by Stroustrup *"Remember the Vasa!"* (March 2018) attempts to sound the alarm about the work towards the ANSI C++2x standard in its working group (WG21):

> *"Many/most people in WG21 are working independently towards non-shared goals. Individually, many (most?) proposals make sense. Together they are insanity to the point of endangering the future of C++."*

Myself, having used and tracked the evolution of C++ for more than 30 years, share Stroustrup's concerns and the concerns of many others in the industry about what has already happened to C++ and is bound to continue to happen to it.

The final word about complexity in C++ goes to Ken Thompson: creator of the UNIX operating system (the direct ancestor of all modern commercial UNIX operating systems, MacOS X, iOS and GNU/Linux a clone of UNIX); the creator of the B programming language (ancestor of B, basically C without types); the co-creator of Belle (five times world chess computer champion and first computer chess program awarded the rank of Master by USCF, a direct ancestor to ChipTest, a predecesor of IBM's DeepBlue); created computer chess end-game tables for 4-6 pieces; co-creator of the Plan 9 operating system; co-creator of UTF-8 (the encoding of choice for Unicode text); co-creator of the Inferno operating system; and co-creator of the Go programming language. From the book *"Coders at Work: Reflections on the Craft of Programming,"* September 2009:

> **Seibel:** *"You were at AT&T with Bjarne Stroustrup. Were you involved at in the development of C++?"*
>
> **Thompson:** *"I'm gonna get in trouble."*
>
> **Seibel:** *"That's fine."*
>
> **Thompson:** *"..."*
>
> **Seibel:** *"Can you say now whether you think it's a good or bad language?"*
>
> **Thompson:** *"It certainly has its good points. But by and large I think it's a bad language. It does a lot of things half well and it's just a garbage heap of ideas that are mutually exclusive. Everybody I know, whether it's personal or corporate, selects a subset and these subsets are different. So it's not a good language to transport an algorithm—to say, "I wrote it; here, take it." It's way too big, way too complex. And it's obviously built by a committee."*

"Stroustrup campaigned for years and years and years, way beyond any sort of technical contributions he made to the language, to get it adopted and used. And he sort of ran all the standards committees with a whip and a chair. And he said "no" to no one. He put every feature in that language that ever existed. It wasn't cleanly designed—it was just the union of everything that came along. And I think it suffered drastically from that."

I shudder to think what Thompson thinks about C++ now.

Sadly, the final final word has to go to the C++ expert community: Gabriel Dos Reis, Herb Sutter and Jonathan Caves who wrote in a proposal to change the language definition to address bugs that have been written by C++ expert programmers for over 30 years without knowing that they were writing those bugs into their code. They write in *"Refining Expression Evaluation Order for Idiomatic C++"* a C++17 language change, underlined highlights are by the author of this book:

"2. A CORRODING PROBLEM "

"These questions aren't for entertainment, or job interview drills, or just for academic interests. The order of expression evaluation, as it is currently specified in the standard, undermines advices, popular programming idioms, or the relative safety of standard library facilities. The traps aren't just for novices or the careless programmer. They affect all of us indiscriminately, even when we know the rules."

"Consider the following program fragment: "

```
void f() {
    std::string s = "but I have heard it works even "
                    "if you don't believe in it";
    s.replace(0, 4, "").replace(s.find("even"), 4, "only")
                       .replace(s.find(" don't"), 6, "");
    assert(s == "I have heard it works only "
                "if you believe in it");
 }
```

"The assertion is supposed to validate the programmer's intended result. It uses "chaining" of member function calls, a common standard practice. This code has been reviewed by C++ experts world-wide, and published (The C++ Programming Language, 4th edition.) Yet, its vulnerability to unspecified order of evaluation has been discovered only recently by a tool. ... Newer library facilities such as std::future<T> are also vulnerable to this problem, when considering chaining of the then() member function to specify a sequence of computation. ... For example, using << as insertion operator into a stream is now an elementary idiom. So is chaining member function calls. The language rules should guarantee that such idioms aren't programming hazards. ... Without the guarantee that the obvious order of

> *evaluation for function call and member selection is obeyed, these facilities become traps, source of obscure, hard to track bugs, facile opportunities for vulnerabilities."*

So apart from 30 or more years of possibly broken code that might not ever be migrated to a C++17 compiler, with its new rules for order of expression evaluation, yet another layer of complexity is thrown into the C++ language. Newly written code will now start to purposely depend on these new rules for the order of evaluation of expressions, some programmers might not understand the new rules and introduce bugs thinking that they know what is going on with the minutiae of the order of expression evaluation.

1.10 Book Organization

Chapter §1, explains the rationale for the COOGL language. Presents object oriented terminology, the member function invocation syntax, the *hello world* program, the compilation model for the language, its lineage, design philosophy, and the *undefined behavior* schism.

Chapter §2, describes CLEAN, the subset of C from which COOGL evolved, code that needs to be used as both as C and COOGL source code is written in CLEAN.

Chapter §3, describes array descriptors, pointer arithmetic, tuples, and literals.

Chapter §4, is detailed presentation of classes and inheritance. Presents contracts, classes as constructor functions, classes are functions and functions are also a special kind of classes, accessibility modifiers and member declarations, object declarations, member functions, introduction to inheritance and member function redefinition, contract specifications and member function redefinitions, constructor restrictions and organization, unification of member declarations and their initialization, nested class declarations, `this` object pointer, iterators and the use of `this` in member functions that are non-static classes, functions as degenerate types and nested member functions, default arguments, and the stringifying operator `#`.

Chapter §5, presents abstract classes, interfaces, and destructors. Describes value like objects, assignment, default constructor, initialization of an object from another object, the `lang.value` interface. These topics are then presented in more detail with the help of a string value like objects, various optimizations that minimize the creation of temporary objects. Lastly literal members are described.

Chapter §6, presents abstract classes, interfaces, and inheritance in detail. Single inheritance and multiple interface implementation, deferred and redefined member functions, accessibility modifiers are revisited also their relationship to inheritance and interface implementation, member access aliases, qualified accessibility modifiers, pointers and inheritance, duplicate member names, and contracts and their relationship to redefined member functions.

Chapter §7, explains class declaration extensions, class declaration continuations, class of pointers and class of array descriptors and their implicit declaration, pointer arithmetic and size of objects as it relates to inheritance and polymorphism, control of class data layout, lookup of identifiers in the context of the function being called instead of the context of the calling function, delegate function pointers, and various other aspects of classes.

Chapter §8, explains various aspects of the language that support programming in the large: name spaces, modules, dynamic linking, global declarations, accessibility controls for a class as a type versus the class as its constructor, class initialization, and global construction order.

Chapter §9, presents control flow aspects of the language: replacing the `goto` `out` idiom with a function destructor, the `on` statement, vital function values, jump statements their restrictions and their relationship to object destruction, lack of structured exception handling syntactical language support, and loop member functions and the `loop` statement.

Chapter §10, describes operators, expressions and keywords, parenthesis requirements under certain circumstances to reduce programming errors, the member lookup operator, fine grained function inline control, checked arithmetic operators, language keywords, and C language keywords that have been removed.

Chapter §11, presents generic programming: type path expressions, type arguments, type variables, type values, restriction on type arguments and type variables, and the `argsof` tuple type compiler declared member, dynamic object allocation, literal arguments to generic classes, field name arguments and `fieldof`, and an implementation of a generic intrusive `list`.

Chapter §12, explains additional aspects related to types: integer, floating, complex, imaginary, enumerations, bit fields, unicode characters, character and string literals, incompatible types, global types, type dimensions (as in units of measure) and literals of specific dimensions. Additionally class hierarchies that relate to various array types, pointers, and various number types and the number type hierarchy, in support of both generic programming, class extensions, and smart pointers.

Chapter §13, presents variable length arrays and dynamically allocated arrays, error reporting due to failure of an object's construction while constructing an array of objects, restrictions on array descriptors and variable length arrays, array memory reinterpretation, dynamic creation and destruction of arrays, and restrictions on walking arrays with pointers when inheritance is involved.

Chapter §14, describes the approach of the language to remain as flexible in its memory management as C is, while being a safe language, all unsafe aspects of C are presented and the approach to safety of the language is presented.

Chapter §15, explains the language concurrent programming support, various de-

sign considerations, hardware aspects, memory models, and examples.

Five appendixes complement the book:

Appendix 1L – Libraries lang, lib, and libc

Appendix 2S – Identifier mapping and calling convention

Appendix 3D – Differences between C and CLEAN

Appendix 4C – Sharing Code and Using C Code

Appendix 5R – Language and Compiler Manual

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

2 - COOGL's C subset: CLEAN

"A designer knows he has arrived at perfection
not when there is no longer anything to add,
but when there is no longer anything to take away."

-- Antoine de Saint-Exupéry

This chapter describes CLEAN, the subset of C implemented by COOGL. CLEAN excludes C's: preprocessor, obsolete constructs, and minor language quirks. Their use causes compilation errors to ensure the meaning of C code does not silently change when used as COOGL code. Code written in CLEAN can be used as C or COOGL code. CLEAN code, when used as C code, is used together with a header file to bridge very minor syntactical differences between CLEAN and C. A few COOGL only details are introduced in this chapter, they are presented in **bold** to make them easier to find.

2.1 Tokens and identifiers

The term *token* is used to refer to sequences of characters considered to be a single entity. For example, the tokens used in comments: `//`, `/*`, `*/`, `/#`, and `#/`. The various language keywords: `enum`, `return`, `if`, `while`, etc. Various operators: `+`, `-`, `++`, `+=`, `=`, `!=`, `==`, `<<`, etc. Certain tokens are defined by the user instead of by the programming language. There are two kinds of user defined tokens: literals and identifiers. Literal tokens correspond to explicit values: numbers, characters, and strings, for example: `1`, `123`, `3.1416`, `'a'`, and `"string"`.

Identifiers are a sequence of one or more characters, digits, and underscores. An identifier can not start with a digit or an underscore, nor can it end with an underscore. Identifiers can not contain two consecutive underscores. Examples of valid identifiers: `MAX`, `create_file`, `CreateFile`, `log2`, `i18n`, and `a`. Identifiers are case sensitive, these are three different identifiers: `MAX`, `Max`, and `max`. Examples of invalid identifiers: `0zero`, `my-cat`, `my dog`, `_`, `x_`, `_x`, and `a__z`.

2.2 Comments

Text prefixed by `//` through the end of the line is a comment, this BCPL comment syntax was later adopted by C++ and C99. The C `/* comment */` can be used to comment out code as long as it doesn't include any other comments. Use of the `/*`

token within a `/* comment */` causes an error, instead of silently allowing code to be turned into a comment accidentally. If allowed, the assignment to `i` would not occur:

```
void example() { /* invalid COOGL code */
    int i = 0;   /* because this comment is not closed here =>
    i = 1;       /* this statement would be commented out! */
}
```

The `//` comments can not include these four tokens: `/*`, `*/`, `/#`, or `#/` For example, the following COOGL code causes a compilation error:

```
a = b //* divisor */ c
      + d;
```

in C89 and C99 that code has two different meanings, a *silent* change between those two languages. Those meanings are:

```
a = b + d;      // C99 meaning
a = b / c + d; /* C89 meaning, which has no // comments */
```

C programmers use pre-processor directives to prevent code from being compiled, for example to be able to compile and test the surrounding code. The C preprocessor is not part of COOGL. **A third form of comment, not part of CLEAN,** `/# comment #/`, syntax can be used to comment out code that has `/* comments */` or `// comment` comments. The `/# comment #/` does not nest within other `/# comments #/`. There is no way to comment out code that already contains `/# comments #/`.

Neither one of the comment start tokens have any significance within a string literal. Comments can not start within a string literal, for example:

```
void use() {
    byte *p = "this /* is not a comment */";
}
```

Comment examples:

```
/* Binary tree node. */
struct node {
    node *right;  // right sub tree
    node *left;   // left sub tree
};
```

Remember that the best comment is sometimes no comment at all, compare:

```
int depth(node *tree) {
    if (!tree) return 0;
    int left = depth(tree->left);
    int right = depth(tree->right);
    return (left > right ? left : right) + 1;
}
```

to the paper-work filling bureaucratic style sometimes confused with programming:

```
/# Please do not do this sort of stuff!!!
/*
 * Name:      depth
 * Argument:  tree, a pointer to a tree of nodes
 * Result:    The depth of the longest branch of the tree.
 * Algorithm: Recursively compute the depth of each branch,
 *            use the depth of the deepest tree branch to
 *            compute the depth of the tree.
 */
int depth(node *tree) {
    /* painfully commented code that I have spared you from. */
}
#/
```

2.3 Source code after comment removal

Comments are processed at compilation time before any other processing is done. Comment removal is equivalent to the replacement of the comment tokens and non-space characters within the comment with space characters. An empty comment does not lead to the concatenation of surrounding text, for example, this causes a compilation error:

```
int i = 1/**/0;
```

After comment removal it is equivalent to:

```
int i = 1    0;
```

It is not equivalent to:

```
int i = 10;
```

After comment removal, the remaining contents of a source file are treated as a possibly empty sequence of *global declarations*, as explained in §2.7.

2.4 Functions and the `return` statement

Functions correspond to the procedures and subprograms of other programming languages. Functions have arguments and return a value, their types are declared by the function. Functions that don't return a value use the type `void` as the type of their return value, they correspond to procedures in other languages. Arguments are passed by value, modification of an argument does not affect the data used by the caller when invoking the function. Functions that return a value must use the `return` *expression*`;` statement to compute the value returned, after the `return` statement is executed the function returns to its caller.

Subsequent sections explain declarations in detail. This introduction to functions makes use of simple declarations of variables whose type is the integer type: `int`. Functions without arguments have an empty argument list, for example, the `one()`

function below. The type of the value returned by a function precedes the function declaration:

```
int one() { return 1; }
```

Functions with a non-empty argument list include a list of comma separated argument declarations within the parenthesis that follow the function name. The `add()` function, below, returns a value of type `int`, its argument list declares `a` and `b`, both of type `int`:

```
int add(int a, int b) { return a + b; }
```

The `return` statement without a value expression can be used by a `void` function to cause the function to return to its caller, see also §9.9. A `void` function doesn't require a `return` statement as the last statement of its function body, if execution control reaches the end of the function, the function returns to its caller. For example:

```
void test_add(int a, int b) {
    int result = add(a, b);
    if (result != a + b) puts("add() is not working");
}
```

The first statement in `test_add()` declares a local variable, named `result`, of type `int`. The value returned by the `add()` function is used to initialize `result`. Examples in this chapter use the standard C library, `libc`, function `puts()`, which writes its string argument followed by a new line character to standard output:

```
int puts(char s[])
```

If the flow of control within a non-`void` function can reach the end of the function, then the last statement of the function must be a `return` statement:

```
int invalid() { /* error: missing return statement */ }
```

2.5 Built-in types

The built-in types are:

- Boolean: `bool`.
- Character types: `char`, `wchar_t`, `char16_t`, `char32_t`, and `unic`.
- Signed integer types: `byte`, `short`, `int`, `long`, `large`, `index`, `ptrdiff_t`, and `ssize_t`.
- Unsigned integer types: `ubyte`, `ushort`, `uint`, `ulong`, `ularge`, `uindex`, and `size_t`.
- Floating point: `float` and `double`. Some platforms might support some of these other types: `float128`, `long_double`, `double_double`.
- Complex floating point: `complex_float` (and its more succinct equivalent

`complex`) and `complex_double`. Complex numbers store both a real and an imaginary component. They require twice the memory than their corresponding floating point type. Some platforms might support some of these other types: `complex_float128`, `complex_long_double`, and `complex_double_double)`.

- Imaginary floating point: `imaginary_float` (and its more succinct equivalent `imaginary`) and `imaginary_double`. Complex numbers are capable of storing both a real and an imaginary component. The imaginary numbers are complex numbers with a real part with value 0. They only require memory for the imaginary value. Some platforms might support some of these other types: `imaginary_float128`, `imaginary_long_double,` and `imaginary_double_double).`.

The types `large` and `ularge` take the place of the types introduced in C99 and refined in C11: `long long` and `unsigned long long`. Together with `uchar`, `ushort`, `uint`, and `ulong`, they are meant to remove the ad-hoc morass of ALGOL68 enabled syntactical combinations of `unsigned`, `int`, `long`, `double` , `complex`, and `imaginary` in a variety of forms that require compiler changes for each new ad-hoc variation. Common code shared with C can use `typedef` if the verbosity of `long long` is desired (e.g. `typedef large long_long;`).

The signed integer types are represented in *two's complement* format, the format that all modern computer systems use. The following restrictions apply to the sizes of the signed integer types, and their corresponding unsigned counterparts. Their specific sizes are determined by the underlying system and its C compiler and the compilation mode being used.

- `byte`, the smallest unit of memory that is directly addressable by the hardware, all modern systems have 8 bit bytes.
- `int`, usually the natural word of the machine, but on 64 bit systems the `int` type is usually 32 bits, instead of 64 bits.
- `short`, its memory requirements can not be larger than the memory requirements of the `int` type, its size is usually 16 bits.
- `long`, its memory requirements can not be smaller than the memory requirements of the `int` type, its size is usually 32 or 64 bits.
- `large`, its memory requirements can not be smaller than the memory requirements of the `long` type, its size is at least 64 bits.
- `index`, the type required to index into the largest possible array supported by the language, including indexing into data in a memory mapped file, same size as a pointer.

The character types:

- `char`, the native character type of the system, its size is the same as the size of a `byte`, it is either signed or unsigned, depending on the underlying computer instruction set, underlying C compiler, and system ABI.
- `wchar_t`, a legacy character type from the C language, it is usually 16 or 32 bits.
- `char16_t`, an unsigned character type that can represent a 16 bit Unicode character, from C11.
- `char32_t`, an unsigned character type that can represent a 32 bit Unicode character, from C11.
- `unic`, a 32 bit type used to store a Unicode character which is the same as `char32_t`, it is the preferred type for Unicode characters, its name is more mnemonic and less machine oriented.

2.6 Integer, floating, character, and string literals

Variables of `bool` type use one byte of memory, their possible values are `true` and `false`. When used in an expression where integers are expected, their corresponding values are 1 and 0.

Integer literals are written in decimal, unless they are prefixed by a base: `0`, `0x`, or `0b` (respectively: octal, hexadecimal, and binary). The prefixes `0X` and `0B` can also be used, but the lowercase prefixes are preferred.

Suffixes can be used to specify the type of integer literals: `l`, `ll`, `u`, `ul`, and `ull`, respectively: `long`, `large`, `uint`, `ulong`, and `ularge`). Alternative forms with different cases and order: suffixes that use `ll` can not have mixed case variations, they are either both lowercase (`ll`) or both uppercase (`LL`). The alternative forms are shown in parenthesis: `l` (`L`), `ll` (`LL`), `u` (`U`), `ul` (`uL`, `Ul`, `UL`, `lu`, `lU`, `Lu`, `LU`), and `ull` (`uLL`, `Ull`, `ULL`, `llu`, `llU`, `LLu`, `LLU`). The types of floating point literals is `double` unless a suffix is specified, if `f` (or `F`) is specified, then the type is `float`. The integer and floating point suffixes are identical to the suffixes in C.

The type of integer literals varies depending on the suffix specified, if any, the base used to express the number, and the magnitude of the number. The type of a literal corresponds to the smallest type, but not smaller than `int`, capable of representing the numeric value of the literal but only if that type is allowed, if unsigned types are allowed, then the signed type is always chosen first, if the value can not be represented, then the unsigned type of the same size is chosen. The following table shows which are the types (designated with `v`) that are considered for the type of the literal and in order of choice, from top to bottom, as a function of: the integer literal suffix (if any), its base (explicit or implicit), and the literal value's ability to be represented by the types allowed to be considered for the combination of base and suffix. Only a

single suffix that represents each class of suffixes equivalent to it is shown (e.g. `ul` is shown, it stands for its 7 variations shown above):

	decimal (no explicit base)						**explicit base: `0`, `0x`, or `0b`**					
suffix	*none*	`u`	`l`	`ul`	`ll`	`ull`	*none*	`u`	`l`	`ul`	`ll`	`ull`
`int`	v						v					
`uint`		v					v	v				
`long`	v		v				v		v			
`ulong`		v		v			v	v	v	v		
`large`	v		v		v		v		v		v	
`ulong`		v		v		v	v	v	v	v	v	v

The overall idea is that values without a suffix are of the smallest integer type that can represent the value, but not of a type smaller than `int`. Decimal integer literals that don't have a suffix are never considered unsigned integer literals. Integer literals with an explicit base (i.e. either hexadecimal, octal, or binary) are considered to be unsigned if required to fit a type. For example, on a system with 32 bit `int` and 64 bit `long`, the value `4294967295` (2^{32} - 1) is of type `long`, but the same exact value expressed in hexadecimal `0xFFFFFFFF` (or its octal or binary equivalents) fits an `uint` and its type is `uint`. The type is tied to the literal value only. The types of other literals or subexpressions in an expression that uses the literal value don't affect the type of the literal.

2.7 Declarations and declaration contexts

Declaration forms and the various contexts in which they occur are described in this section. The contexts in which declarations can occur are:

- *member*, members of a class or function, **only applies to COOGL, not CLEAN**.
- *global*, in the outermost context, i.e. not nested within another context.
- *local,* local variables of a function **or class**.
- *aggregate*, declarations that are within a `struct` or `union` declaration.

Examples of declarations in these various contexts follow.

Classes are not part of CLEAN, included here for completeness:

```
class line {   // line is declared in the global context
    pub point a;  // a and b are declared in the member context
    pub point b;  // of the line class, they are members of the
}                 // line class
```

Examples pertinent to C and CLEAN:

```
int i;          // i is declared in the global context
int sum(        // sum is declared in the global context
        int a,     // a, b, and c are declared in the
        int b,     // local context of the sum function,
        int c)     // they are the arguments of sum
{
    int v;         // v is declared in the local context of
                   // the sum function, it is a local variable
    v = a + b + c;
    return v;
}
struct point { // point is declared in the global context
    int x;         // x, y, and z are declared in the
    int y;         // aggregate context of the point
    int z;         // structure, they are fields of point
};
```

The syntax of declarations is the same within all the declaration contexts, with the exception of which declaration prefix keywords could be used with them. The declaration prefix keywords are:

- **accessibility modifiers, not part of CLEAN**: `pub`, `priv`, or `prot`, used to control the accessibility of entities declared in a class, interface, namespace, or module, see §6.7;
- absence of an accessibility modifier within functions and classes, implies a local declaration of an entity that is not accessible outside of them.

The rules for the use of those keywords are:

- global declarations occur in the outermost context, they may be preceded by a `pub`, `priv`, or `prot` accessibility modifier in COOGL, **but not in CLEAN**;
- **member declarations, not part of CLEAN**, occur only in the outermost block or the argument list of classes and functions. They are preceded by: `pub`, `priv`, or `prot`;
- local declarations only occur within functions and classes, or their argument lists, they are not preceded by: `pub`, `priv`, or `prot`;
- declarations within an aggregate, a `struct` or `union`, can not be preceded by any of these keywords: `pub`, `priv`, or `prot`.

2.8 Declaration kinds

The kinds of declarations are:

- variable;

- **literal**, **not part of CLEAN**;
- function;
- **class**, **not part of CLEAN**;
- type;
- enumeration;
- aggregate.

These declaration forms are briefly explained below, together with examples. The example declarations are shown without a surrounding context, as if they were global declarations.

Variable declaration, variables can be declared of built-in or user defined types. Variable declarations can be accompanied by various syntactical constructs to declare arrays, pointers, and the arbitrary compounding of them. For example:

```
int n;       // n is a variable of type int
int a[10];   // a is an array of 10 int elements
int b[2][3]; // b is an array of 2 arrays of 3 int elements each
int *p;      // p is a pointer to an int
int **q;     // q is a pointer to a pointer to an int
int *t[3];   // t is an array of 3 pointers to int
```

Variables can be initialized at declaration time:

```
int n = 1;   // n is a variable of type int, initialized to 1
int *p = &n; // p is a pointer to an int,
             //   initialized with the address of n
```

Literal declarations. **Not part of CLEAN**. Are similar to variable declarations, but they declare a compile time value that can not be changed, a value must be specified at declaration time. For example:

```
lit int k = 1; // k is a literal of type int with value 1
```

Function declarations, they specify the type of the value that the function returns, if any. For example:

```
int five() { return 5; }   // five is a global function
void f() {}                // f is a global function
```

Functions can also have argument declarations, for example:

```
int factorial(int n) {     // factorial is a global function
    if (n <= 2) return n;  // n is a local declaration
    return n * factorial(n - 1);
}
```

***Class declarations*. Not part of CLEAN**. For an example see `stack` in §1.3.

Type declarations. The `typedef`, **`typenew`**, and **`typeglob`** **(the last two are not part of CLEAN)** declarations are used to declare types based on other types. Their

declaration syntax is based on the declaration syntax of variable declarations:

```
typedef int integer;
```

Declares the `integer` type, which is identical to the built-in `int` type. Variables declared of the `integer` type are identical from their type perspective to variables declared of the `int` type.

```
typedef int *intptr;
```

Declares the `intptr` type, which is identical to the compound type: pointer to `int`, i.e. the type: `int *`. Variables declared of the `intptr` type are identical from their type perspective to variables declared of the `int *` type.

Types introduced with `typedef` are synonyms for the type embodied in the `typedef` declaration, for example:

```
integer n;      // n is a variable of type int
int *p = &n;    // p is a pointer to int,
                // initialized with the address of n
intptr q = &n; // q is a pointer to int, that points to n
```

Enumeration declaration. Provides a type for a set of literal values, for example:

```
enum temperature_unit { // temperature_unit is a type
    FAHRENHEIT = 0,
    CELSIUS = 1,
    KELVIN = 2
};
temperature_unit tu;    // tu is a variable of that type
```

Aggregate type declarations. These can be `struct` or `union` declarations, e.g.:

```
struct person {         // person is a type
    int age;
    byte name[128];
};
person p;               // p is a variable of person type
```

A `union` overlays its fields such that the memory that they use is shared:

```
union int_bytes {       // int_bytes is a type
    int i;
    byte b[4];
};
int_bytes x;            // x is a variable of int_bytes type
```

The only kind of declarations that can occur within a `struct` or `union` is the declaration of its fields, such as `age` and `name` above. Field declarations are syntactically the same as variable declarations, though initialization of them at declaration time is not allowed. No other form of declaration (i.e. literal, type, aggregates, enumerations, functions, or classes) are allowed within a `struct` or `union` declaration.

2.9 Order of declarations

In CLEAN variable, type, enum, and aggregate declarations must precede their use, when CLEAN code is used as C code forward declarations and prototypes are placed in a header file, see §C.1.

The order of declarations, at the global level, does not matter in COOGL, for example, a function can reference a global variable declared after the function's declaration:

```
int genid() {                   // genid is a global function
    id = id + 1;                // invalid in CLEAN, valid in COOGL
    return id;
}
int id = 0;                     // id is a global variable
```

The declaration of local non-static variables must precede their use:

```
int random() {
    static int old = 1;// declaration must precede use in CLEAN
    int v;              // v declaration must precede its
    v = (old * 168071 + 71111111) & 0x7FFFffff; // use here
    old = v;
    return v;
}
```

In COOGL, within a class or function, the order of entities other than non static local variables does not matter, for example, `random()` would still be valid if the `old` static variable declaration was moved to the end of the function.

2.10 Statements within functions and classes

The body of a function, i.e. the code that implements the function is either a possibly empty sequence of code within curly braces; or **a single return statement, without curly braces, not part of CLEAN**.

The code of a function or class body within the curly braces is a possibly empty sequence of constructs, which belong to one of these groups:

- **Member declaration: `pub`, `prot`, or `priv`, not part of CLEAN**.
- Local declaration.
- Compound statement.
- Expression statement.
- Selection statement: `if`, `if else`, and `switch`.
- Iteration statement: `for`, `while`, `do while`, **not in CLEAN**:: `loop` **and** `on`.
- Control flow altering statement: `return`, `break`, `continue`, and `goto`.

Any statement might be preceded by one or more labels of the form *label:* where *label* is a user defined identifier which must be unique within a function.

A brief introduction to some of the control flow statements follows, they are presented first to allow the subsequent presentation of expressions to contain more interesting examples. Nonetheless, a very brief introduction to expressions follows this section to allow the control flow statement examples to be understood.

Subsequent sections present the various types and the expressions that can operate on objects of those types, explaining through them the various ways in which expressions are formed and their meanings.

2.11 Introduction to operators and expressions

The following sections make use of some of COOGL's operators, operators are fully covered in section §2.16 - §2.26. The operators used in the following sections are:

- *addition*, *subtraction*, *multiplication*, and *division*: `+`, `-`, `*`, and `/`.
- *assignment*, *equality*, and *inequality*: `=`, `==`, and `!=`.
- *less than*, *less or equals to*, *greater than*, and *greater or equals to*: `<` , `<=`, `>`, and `>=`.

These operators function as they function in other programming languages, traditional mathematical operator precedence applies. Parenthesis, `(` and `)`, can be used when required. Example expressions follow:

```
void example() {
    int n;
    n = 1 + 2 * 3;     // value of n is 7
    n = (1 + 2) * 3;   // value of n is 9
    n = 9 - 5 - 2;     // value of n is 2
    n = 9 - (5 - 2);   // value of n is 6
    n = 8 / 4 / 2;     // value of n is 1
    n = 8 / (4 / 2);   // value of n is 4
    bool b;
    b = 4 > 1;         // value of b is true
    b = 4 < 1;         // value of b is false
    b = 4 == 4;        // value of b is true
    b = 4 != 4;        // value of b is false
}
```

2.12 Compound statement

A compound statement is a list of statements enclosed within curly braces:

```
{ possibly_empty_statement_list }
```

For example:

```
{ f = f * n; n = n - 1; }
```

2.13 `assert()` function and `...` statement

The COOGL library function (a `#define` in C) `assert(expression)`, does nothing if `expression` is a non-zero value (i.e. a truth value), otherwise it causes an exception, which if not handled, causes the program to terminate with an error message that includes the file name, line number, and the text of the argument expression. See §4.17 for an implementation of `assert()`. The `...` statement, **only available in** COOGL is used to indicate that there is missing code, it causes an exception to be raised if control flow reaches it, if unhandled the file and line number are reported together with a message that indicates that unimplemented code was attempted to be executed.

2.14 `if` and `if else` selection statements

The `if` statement:

```
if (expression) statement
```

Controls the execution of its subordinate *statement*, which is only executed if the controlling *expression* is true (i.e. non-zero). The `if else` statement:

```
if (expression) statement else statement2
```

Provides a second subordinate *statement2*, after the `else` keyword, which is executed if the controlling *expression* is false (i.e. zero). For example:

```
void check(int n) {
    if (n == 0) { puts("n == 0"); return; }
    if (n < 0) puts("n < 0");
    else puts("n > 0");
}
```

2.15 `while` and `for` iteration statements

The `while` iteration statement provides controlled iteration of a statement:

```
while (expression) statement
```

The `while` statement evaluates its controlling *expression*, if it is false, its execution is complete and its subordinate *statement* is not executed, if it is true, the subordinate *statement* is executed and after its execution the control flow proceeds to the `while` statement where the expression is reevaluated, the process repeating until the *expression* is false.

The following function computes the factorial of its argument iteratively:

```
int factorial(int n) {
    assert(n >= 1);
    int f = 1;
    while (n >= 2) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```

The `for` statement:

```
for (expression1; expression2; expression3) statement
```

Has three optional expressions. *Expression1*, also known as the *initialization expression*, is invoked once when the `for` statement execution starts. *Expression2*, the *controlling expression*, is evaluated on each iteration, if it is false, the `for` statement execution is complete; if true, the subordinate statement is executed, after which *expression3*, the *stepping expression*, is executed and the process repeats itself with the re-evaluation of *expression2*.

An alternative form of `for` statement is:

```
for (local_declaration_statement; expression2; expression3) statement
```

Where instead of *expression1* a local declaration statement is used. The `power()` function uses this `for` statement form to raise `v` to the `n`*th* power, i.e. to compute v^n:

```
int power(int v, int n) {
    assert(n >= 0);
    int p = 1;
    for (int i = 1; i <= n; i = i + 1) p = p * v;
    return p;
}
```

2.16 Operators and expressions

The C programming language is rich in operators, COOGL has the same operators. The basic binary arithmetic operators: *addition*, *subtraction*, *multiplication*, and *division*, respectively: `+`, `-`, `*`, and `/`; and the unary *arithmetic negation* operator, `-`, can be used with both integer and floating point operands. Integer division between two integers results in an integer value, the remainder of the division is ignored. The *remainder* operator, `%`, which can only be used with integer values, provides the remainder of integer division.

Expression	Result	Expression	Result
`7 + 3`	`10`	`7 / 3`	`2`
`7 - 3`	`4`	`7 % 3`	`1`
`7 * 3`	`21`		

When one or both of the operands of the division operator is a floating point value the result is also a floating point value with a possibly non-zero fractional part, i.e. the remainder is not dropped, it is used to produce the fractional part of the result.

Both K&R C, and C89 allowed the behavior of integer division, `/`, and the remainder operator, `%`, to be implementation dependent when negative numbers are involved. The C99 language removed that relaxation and mandated the well defined behavior of FORTRAN. COOGL adopts that behavior as well, this is the behavior in all modern hardware.

C99 standard: 6th paragraph under section 6.5.5 and its 78 footnote:

> "When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[78)] If the quotient `a/b` is representable, the expression (`a/b)*b + a%b` shall equal `a`."
>
> "78) This is often called “truncation towards zero”."

The obsolete behavior allowed by K&R C and C89 could lead to results such as `-7/3` equal to `-3` as long as `-7%3` was equal to `2`, even though those results are against the mathematical expected results. The allowance for such strange behavior was to accommodate computer systems that have long since been obsolete.

The allowed behavior of C99, FORTRAN and COOGL mandates these results:

Division	Result	Remainder	Result
`7 / 3`	`2`	`7 % 3`	`1`
`-7 / 3`	`-2`	`-7 % 3`	`-1`
`7 / -3`	`-2`	`7 % -3`	`1`
`-7 / -3`	`2`	`-7 % -3`	`-1`

The logical bitwise operators: *bitwise and*, *bitwise or*, *bitwise exclusive or*, *shift left*, and *shift right* operators, `&`, `|`, `^`, `<<`, and `>>` respectively, together with the unary *bitwise negation* operator `~` must be used with integer operands. Examples of these operators follow, literal values preceded by `0x` are in hexadecimal notation:

Expression	Result
`6 | 3`	`7`
`6 & 3`	`2`
`6 ^ 3`	`5`
`6 << 1`	`12`
`6 >> 1`	`3`
`0xABCD << 8`	`0xABCD00`
`0xABCD >> 8`	`0xAB`
`~0xFFFF`	`0xFFFF0000`
`~0`	`0xFFFFFFFF`

The last two results depend on the `int` type being a 32 bit type, if it had been a 64 bit type those two expressions would have been:

Expression	Result
`~0xFFFF`	`0xFFFFFFFFFFFF0000`
`~0`	`0xFFFFFFFFFFFFFFFF`

2.17 Controlling expressions, relational operators, and truth values

The values of the various comparison and relational operators: `==`, `!=`, `<`, `<=`, `>`, and `>=` is either true or false. The tokens `true` and `false` are literals of the `bool` type, when used as integer values these literals have the values `1` and `0` respectively.

The value of expressions within conditional contexts, e.g. the controlling expression of an `if` or a `while` statement, is compared with zero. A zero value means that the expression is false, a non-zero value means that the expression is true. For example:

```
int main() {
    if (-7) puts("-7 is true"); else puts("-7 is false");
    if (0)  puts(" 0 is true"); else puts(" 0 is false");
    if (1)  puts(" 1 is true"); else puts(" 1 is false");
}
```

Its output is:

```
-7 is true
 0 is false
 1 is true
```

2.18 Logical operators

There are two binary logical operators: *and* `&&`, and *or* `||`. They evaluate their first operand, and if it can be determined the truth value of the operator from the value of the first expression, i.e. if the value is zero for `&&`, and non-zero for `||`, then their second operand is not evaluated; otherwise the second operand is evaluated and its

truth value (i.e. zero or non-zero) determines the value of the operator. The result of these operators is 1 for truth, and 0 for false. The unary logical not operator, `!`, performs the logical negation of its operand, i.e. if the operand is zero its result is 1, if the operand is non-zero its result is 0.

2.19 Assignment and assignment-*op* operators

The assignment operator, `=`, and the assignment-*op* operators `+=`, `-=`, `/=`, `%=`, `|=`, `&=`, `^=`, `<<=`, and `>>=` combine binary operators with assignment, for example:

x is 7 in each expression	value in x after expression
`x += 1`	8
`x -= 3`	4
`x /= 3`	2
`x %= 3`	1
`x |= 8`	15
`x <<= 2`	28
`x >>= 1`	3

they perform the operation specified by corresponding binary operator and assign the result to the left operand.

The value of an assignment or an assignment-*op* operator is the value assigned to its first operator, for example:

```
void example() {
    int i, j, k;
    i = j = k = 8;    // i, j, and k have the value 8
    k /= j -= 4;      // j is 4 (i.e. 8 - 4)
}                     // k is 2 (i.e. 8 / 4)
```

Beware of confusion between the equality comparison `==`, and the assignment operator `=`, particularly in the controlling expressions of various control statements:

```
bool is_one(int v) {
    if (v = 1)        // wrong: assignment =, not comparison ==,
        return true; // always returns true!
    return false;     // this statement is never reached
}
```

The incorrect `is_one()` function always returns `true`, because it assigns the value `1` to `v` and then tests that value to see if it is non-zero, it always is non-zero.

2.20 Increment and decrement operators

The increment and decrement, `++` and `--`, operators cause the value in a variable to be incremented or decremented by one. There is a prefix and a postfix forms of these

operators. There is no difference between the prefix and the postfix form in the effect on the variable, the difference is in the value of the expression. The value of the prefix operators as an expression is the value of the variable after the operation, the value of the postfix operators is the value of the variable before the operation, e.g.:

```
void example() {
    int p = 1, q = 1, r = 1;
    int a = p++;      // p is 2, a is 1
    int b = ++q;      // q is 2, b is 2
    int c = r += 1;   // r is 2, c is 2
}
```

The `factorial()` function, from §2.15, is shown below using `*=` and `--`:

```
int factorial(int n) {
    assert(n >= 1);
    int f = 1;
    while (n >= 2) {
        f *= n;
        --n;
    }
    return f;
}
```

2.21 Ternary selection `?:` operator and the comma operator

The operator *`exp1 ? exp2 : exp3`* is the ternary selection operator, it evaluates *`exp1`* and if its value is non-zero, then the value of the `?:` expression is the *`exp1`*, otherwise the value is *`exp3`*.

The binary comma operator *`exp1, exp2`* evaluates *`exp1`* fully, including its side effects, then it evaluates *`exp2`*, the vale of the comma operator is *`exp2`*. If `exp1` is also a comma operator expression, i.e. if its form was: *`exp1_1, exp1_2, exp2`* its implicit parenthesis are left to right, i.e. as if it were *`(exp1_1, exp1_2), exp2`* and the value of the whole expression is *`exp2`*, similarly for 4 expressions separated by commas, etc.

2.22 C array types, operators and expressions

There are two kinds of arrays in COOGL, traditional C arrays, whose dimensions are known at declaration time, and variable length arrays, whose dimensions are determined at run time, additionally there are array descriptors used to refer to arrays, used when passing an array as an argument to a function, among other uses, see §3. This section deals with C array types, variable length arrays are presented in chapter §13.

The declaration syntax of arrays allows for the declaration of unidimensional ar-

rays. The declaration of multidimensional arrays results form the declaration of arrays of arrays by applying the syntax repeatedly. For example:

```
int table[10];       // array of 10 int
int matrix[50][80]; // array of 50 arrays of 80 int
int d3[10][20][30]; // array of 10 arrays of 20 arrays of 30 int
```

The `[]` operator is the array indexing operator. The first element in an array is at index zero, all arrays are zero based. Array access uses expressions of the form `table[i]` and `d3[i][j][k]`, the syntax `d3[i,j,k]` does not correspond to 3 dimensional array indexing, it is an invalid expression. An example of the traditional multi-level iterative access of all the integers in the `d3` array is shown below:

```
void initd3() {
    for (int i = 0; i < 10; ++i)
        for (int j = 0; j < 20; ++j)
            for (int k = 0; k < 30; ++k)
                d3[i][j][k] = random();
}
```

The order of the underlying elements of a multidimensional array are a reflection of the declaration itself, for example:

```
int d[2][3];  // d is an array of 2 arrays of 3 integers
void initd() {
    int n = 0;
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 3; ++j)
            d[i][j] = n++;
}
```

The integer entries within `d`, after the execution of `initd()`, have these values:

`d[0][0]: 0`	`d[0][1]: 1`	`d[0][2]: 2`
`d[1][0]: 3`	`d[1][1]: 4`	`d[1][2]: 5`

The underlying `int` sized words, in increasing memory address order, are:

`d[0][0]: 0`
`d[0][1]: 1`
`d[0][2]: 2`
`d[1][0]: 3`
`d[1][1]: 4`
`d[1][2]: 5`

The iterative array access in the nested `for` loops shown above for `d` and `d3` are the fastest forms of iterative array access because the memory accesses are to sequential memory addresses, which allows the hardware caches and memory subsystem to

perform optimally. Of course, those considerations matter when dealing with large arrays, not tiny ones like `d` and `d3`.

Most array indexing occurs in iterative loops that walk all or part of an array. The index variable is usually declared in the first expression of a `for` statement, declaring indexes with type `int` is correct unless the number of elements in the indexed dimension is larger than the positive value range minus one supported by `int`. Indexing into memory mapped files or arrays whose dimension size is unknown at compile time should done with variables of type `index`, which uses 64 bits in systems with 64 bit pointers. Indexing variables are usually just local variables held in registers, declaring them with type `index` instead of `int` doesn't cause any extra overhead. The compiler will produce a compile time error if array indexing is attempted with variables whose value range is smaller than the value range of `index` if the compiler can not guarantee at compile time that its value range is large enough be used correctly as an index.

2.23 Pointers: types, operators, and expressions

The C programming language approach to memory manipulation allows direct manipulation of memory that is very close to assembler level programming, but in a typed, structured, and portable manner.

There are various language aspects related to pointers:

- declaration of pointer variables;
- access of the entity referred by expressions of pointer type;
- expressions based on the pointer values themselves, for example through arithmetic operations on pointers to refer to other nearby entities.

The underlying type of entity that a pointer refers to can be a built in type or a user defined type, i.e. one defined using: `struct`, `union`, `class`, arrays, or pointers. Example declarations of pointer variables:

```
byte *bp;             // bp is a pointer to byte
byte **bpp;           // bpp is a pointer to a pointer to a byte
stack *stk;           // stk is a pointer to a stack
int *tab[4];          // tab is an array of 4 pointers to int
typedef int array_of_8_int[8];
array_of_8_int *tp;  // tp is a pointer to an array of 8 int
//int (*tp)[8]; C ONLY: tp is a pointer to an array of 8 int
```

There are several operators that can be used with most kinds of pointer variables, the exception being pointers to functions which are discussed later.

The unary `*` operator is the *dereference* operator. It is used with an expression of pointer type, it dereferences the pointer value to refer to the object whose address is the value of the pointer expression.

The unary `&` operator is the *address-of* operator. It can be used on variables of any type, it is used to obtain the address of a variable, usually to assign it to a pointer variable, or to pass it as an argument to a function, or to use it in pointer arithmetic or pointer comparison expressions.

```
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int p = 1, q = 2;
void example() { swap(&p, &q); }
```

A common use of the `&` and `*` operators is in argument passing where a value is to be returned through the memory that the pointer argument refers to. For example the `swap()` function above exchanges the values that its two arguments point to.

Pointer arithmetic operators are the binary addition and subtraction, `+` and `-`, operators and their related forms: `++`, `--`, `+=`, and `-=`. Use of the unary negation operator, `-`, is invalid with a pointer expression.

Pointer addition is between a pointer and an integer, not between two pointers. The expression `p+n`, where `p` is a pointer value and `n` is a positive integer, or the equivalent expression `n+p`, results in a pointer value that refers to the n^{th} entity after the entity that `p` points to. The expression `p+n`, where `p` is a pointer value and `n` is negative integer, results in a pointer value that refers to the n^{th} entity before the entity that `p` points to. The expression `p-n`, where `p` is a pointer value is equivalent to `p+(-n)`, i.e. the addition of p and the integer `-n`.

Pointer arithmetic can only be performed when the programmer provides compile time or run time proof that the arithmetic is valid and that the underlying object that the pointer refers to is of a type that does not result in a type violation that could result in unsafe programming. The concepts of safe and unsafe programming are explained in chapter §14. One of the ways by which the programmer provides run time proof that pointer arithmetic is valid, subject to run time checks, is by specifying the pointer with the syntax: `type name[]`, which indicates that `name` refers to an *array descriptor* in a way such that the elements within the array can be accessed through pointer arithmetic in a safe manner.

Examples of pointer arithmetic are shown in `exchange()` below, it exchanges the values at `p+i` and `p+j`:

```
void exchange(int p[], index i, index j) {
    int t = *(p + i);
    *(p + i) = *(p + j);
    *(p + j) = t;
}
```

If the `p` argument had been declared as: `int *p`, then the expressions `p+i` and `p+j` would result in a compile time error. The pointer plus integer expression `*(p+i)` is equivalent to the expression `p[i]`, thus `exchange()` could be written as:

```
void exchange(int p[], index i, index j) {
    int t = p[i];
    p[i] = p[j];
    p[j] = t;
}
```

The `exchange()` function is used in the `sort()` function, below which does its work by finding the smallest element and exchanging it with the first element, and then sorting the rest of the array by the same means.

```
void sort(int array[], index count) {
    assert(count <= array.max[0]);
    if (count <= 1) return;
    int *rest = array;
    for (; count >= 2; ++rest, --count) {
        int min = rest[0];
        index min_index = 0;
        for (index i = 1; i < count; ++i) {
            int v = rest[i];
            if (v < min) {
                min = v;
                min_index = i;
            }
        }
        exchange(rest, 0, min_index);
    }
}
```

Pointer arithmetic, `++rest` above, is allowed in this code, even though `rest` was declared `int *rest`, because the value of `rest` was based on an array descriptor, `array[]`. The array descriptor is used by the compiler to validate the pointer arithmetic at run time, or invariants proven at compile time, as is the case in this example, given the `assert(count <= array.max[0])` and that `count >= 2` inside the `for`.

The variable `rest` can be used as an array descriptor argument to `exchange()` even though it is just a pointer to `int`, the compiler converts the value of `rest` and the value of the array descriptor on which its value was based on, `array[]` in this case, into an array descriptor converting `rest` into the `exchange()` argument `p[]`. A version of `sort()` that uses pointers instead of indexes:

```
void sort(int array[], index count) {
    assert(count <= array.max[0]);
    if (count <= 1) return;
    int *first = array, *last = array + count - 1;
    for (; first < last; ++first) {
        int min = *first, *minptr = first;
        for (int *p = first; ++p <= last;) {
            int v = *p;
            if (v < min) {
                min = v;
                minptr = p;
            }
        }
        swap(first, minptr);
    }
}
```

The subtraction of two pointers, which must be of the same type, `n=p2-p1` produces an integer result, `n` in this case, such that `p1+n==p2`, `n` is the number of times that `p1` should be incremented (if `n` is positive) or decremented (if `n` is negative) so that it becomes equals to `p2`. The type of the result of pointer subtraction is `ptrdiff_t` which is an implementation dependent type capable of holding the difference between two pointers, e.g. it is 64 bits when pointers are 64 bits and 32 bits when pointers are 32 bits. Subtraction of pointers is only valid if the programmer provides compile time or run-time proof that both pointers are within the same array descriptor. Safe pointer subtraction is required to ensure that the programmer does not use pointer subtraction between a valid pointer and a `NULL` pointer as a means to obtain a value from which the pointer integer value could be obtained. Obtaining the value of pointers is prevented to ensure that garbage collection libraries and other memory management schemes can be written without concerns about pointer value changes affecting the program, for example when pointer values are used as hash values, which would no longer be valid if the object the pointer refers to was relocated.

The addition of two pointer values makes no sense, it is invalid.

Pointer arithmetic of pointers to types with sizes other than 1 byte imply hidden scaling of the pointer values in ways that might result in multiplication and division instructions that involve the size of the type that the pointer refers to. For example:

```
struct name {    // variables of type name use 40 bytes
    byte b[40];
};
void exchange_names(name p[], index n) {
    name t = *p;
    *p = *(p + n);
    *(p + n) = t;
}
name name_tab[100];
void example() {
    exchange_names(&name_tab[0], 30);
    name *p = &name_tab[random() % 100];
    name *q = &name_tab[random() % 100];
    size_t n = p - q;
}
```

The arithmetic expression `p+n` within `exchange_names()` is translated by the compiler into these integer instructions `integer_value_of_p + n * 40`, of the appropriate width, e.g. 32 or 64 bit instructions. Similarly, the expression `p-q` is translated into `(integer_value_of_p - integer_value_of_q) / 40`.

Pointer arithmetic of pointers that refer to types whose sizes are a power of two have those underlying multiplication and division operations reduced to shift operations, in which case the computational cost of the scaling of values by the size of the underlying pointed type is negligible. All built-in types and pointer types have sizes that are powers of two, thus pointer arithmetic of pointers that refer to those types always benefit from the use of shift instructions instead of multiplication and division instructions. Because the compiler knows about the size of the underlying object, the underlying multiplication and divisions are against a constant value, which many compilers can optimize into simpler instructions, for example to multiply by 40, a compiler *might* choose to translate `n * 40` into:

```
((n << 5) + (n << 3))       // n * 32 + n * 8
```

Array indexing also involves scaling of indexes by the size of array elements, thus `nametab[n]` involves an underlying multiplication by 40.

Walking arrays with `++` and `--` operators on pointer values never involves multiplication, those are translated into the addition or subtraction of the size of the type that the pointer refers to. For example:

```
lit size_t NAMELEN = 40;
struct name {
    byte b[NAMELEN];
};
lit size_t TABLEN = 100;
name name_tab[TABLEN];
// search name_tab for n, return -1 if not found
index indexof(name *n) {
    name *start = &name_tab[0];
    name *end = &name_tab[TABLEN];
    for (name *p = start; p < end; ++p)
        if (name_equals(p, n))
            return p - start;
    return -1;
}
```

Thus the `++p` expression is translated to *`integer_value_of`*`_p += 40`.

As can be seen above, there is an order relationship between pointer values that follows the underlying order of memory addresses. For example when pointer `p` above is used to walk the `name_tab` array starting with its first element, `&name_tab[0]`, its value will be smaller than `&name_tab[TABLEN]`, the value of `end`. `End` does not point to the last element of `name_tab`, that element is `name_tab[TABLEN-1]`, `end` points instead to the memory location immediately after that element, a memory location that contains some other unknown information, unrelated to the elements of the `name_tab` array. Both languages, C and COOGL, allow such expressions and the idiomatic walking of arrays shown above, both languages make the program behavior of dereferencing the `end` pointer invalid, just as it would be if the `name_tab[TABLEN]` invalid array entry was accessed. In C the dereferencing of `end` would cause the underlying memory to be accessed, COOGL causes an exception to be raised and the memory that `end` points to is not accessed.

Pointer arithmetic with the `p` variable is valid because the value of `p` is based on the address of an element within an array, thus it is run-time safe for the pointer arithmetic operations to be performed, the compiler knows the underlying memory of the `name_tab` array and can ensure, at compile time in this case, that all the accesses are within bounds. To be able to have a common subset between C and COOGL the ability to perform pointer arithmetic on variables declared with the `type *name` syntax is required. C only allows for the declaration of pointers with the syntax `type name[]` in the declaration of function arguments. Safe programming in COOGL is presented in chapter §14, this section only touched on some of its aspects to allow pointer arithmetic to be explained.

2.24 Aggregate types and their operators

The `struct` and `union` keywords are used to declare types for an aggregation of data declarations, for example:

```
struct person {      // person is a type
    int age;
    byte name[128];
};
```

Variables of `person` type have two fields, `age` and `name`, they are accessed through the *dot* `.` and *arrow* `->` operators, depending if the expression used to reference them is of the type of the aggregate or the type is pointer to the aggregate, respectively. For example:

```
void example() {
    person p;          // p is a variable of person type
    p.age = 33;
    p.name[0] = 'A';
    p.name[1] = 'n';
    p.name[2] = 'n';
    p.name[3] = 0;     // 0 terminates C strings
    p.age++;           // one year older
    --p.age;           // one year younger
    person *pp;        // pp is a pointer to person
    pp = &p;           // it now points to p
    pp->age *= 2;      // double the age
    pp->name[2] = 'a'; // now the name is Ana
}
```

A `union` declaration overlays its fields so that the memory that they use is shared:

```
union ubytes_of_uint { // ubytes_of_uint is a type
    uint u;
    ubyte b[4];
};
```

A `union` declaration can not contain member variables that are of a class type or that refer to other data (pointers, array descriptors, `index`, or `uindex`), see §14.7. This is a restriction required for safe programming. The `ubytes_of_uint` union can be used to implement a byte swapping function:

```
uint byte_swap(uint u) {
    ubytes_of_int u;
    u.i = i;
    ubyte t;
    t = u.b[0];   u.b[0] = u.b[3];    u.b[3] = t;
    t = u.b[1];   u.b[1] = u.b[2];    u.b[2] = t;
    return u.i;
}
```

Use of `ubytes_of_uint` can be used to access the 4 bytes within an `uint` on a system with 32 bit `uint`, for example to swap its bytes, is better done with bitwise operators than with a `union`:

```
uint byte_swap(uint u) {
    return (u >> 24) | (u << 24) |
           ((u & 0xff00) << 8) | ((u & 0xff0000) >> 8);
}
```

2.25 Expressions

Expressions are used by themselves, for example an assignment or a function call, or as parts of other statements:

- As the initial value given to a variable at declaration time.
- Declarations to specify number of elements in arrays, or bits in a bit field.
- The value returned by a function through the `return` statement.
- In the expression list of the `on` statement.
- Initialization, controlling, and iteration expressions of: `if`, `for`, `loop`, and `while`.
- In a `switch` statement, described further below.
- The value of `case` labels, described further below.

Examples of expressions in various contexts are shown below:

```
index find_last(int value, int array[], index count) {
    index ix = count - 1;
    while (ix >= 0) {
        if (ix == array[ix]) return ix;
        --ix;
    }
    return -1;
}
```

2.26 Expression statements

An expression statement is alone by itself, it is not a part of another statement, e.g.:

```
void example(int a, int b) {
    int i = a + b;          // expression in declaration
    while (i < 10) {        // expression in conditional context
        factorial(i);       // expression statement
        ++i;                // expression statement
    }
}
```

Expression statements must be function calls or assignments, including `++` and `--`, they must do something with the value they produce, it can not just be ignored:

```
void example(int a, int b) {
    a + b;          // error: invalid expression statement
    a == b;         // error: invalid expression statement
    a < b || a > b; // error: invalid expression statement
    factorial(a);   // expression statement, ok to ignore value
}
```

2.27 Default value returned by `main()`

In C and COOGL the signature of main can be declared in any of these ways:

```
int main();
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *envp);
```

The third forms is only valid in what the C standard defines as hosted environments, e.g. supported by an operating system. In C `main()` is special in that even though `main()` is supposed to return a value, if execution control reaches the trailing curly brace the compilation doesn't cause a compilation error, instead a default zero value is returned, which must be handled with special purpose code by the compiler and is needlessly obscure.

Omitting the trailing `return` in `main()` only has value in reducing the number of lines of code in small C examples, and keeping C compatible with the historical *"hello, world!"* program in the K&R C book, and many programs that copied it and exit without a trailing `return 0;`.

2.28 `if` and `if else` selection statements and indentation errors

The `if` and `if else` statements were presented above. The `else` keyword is associated with the nearest `if` statement that precedes it. Incorrect indentation can visually mislead the programmer about the actual `if` statement that an `else` statement is associated with, for example:

```
int worker() {
    int error;
    if (!too_many_workers())
        for (;;)
            if (error = get_and_do_work())
                return error;
    else                        // error: misleading indentation
        puts("too many workers");
    return 0;
}
```

In `worker()` above, the `else` is associated with the `if` within the `for` statement, instead of the first `if` of the function. COOGL produces a compilation error as a result of this misleading indentation. This is the only circumstance under which COOGL pays attention to whitespace characters other than for token delineation. To avoid having to understand how tab characters are expanded, the indentation checking expects that the whitespace characters that precede the `if` keyword must be exactly the same characters that precede the `else` keyword, i.e. the same sequence of spaces and tabs. For programmers that have trouble making up their minds about the use of tabs or spaces, or that simply hate this COOGL feature, it can be turned off, see XXX.

Appropriate indentation removes the compilation error, which makes the logical coding error clear:

```
int worker() {
    int error;
    if (!too_many_workers())
        for (;;)
            if (error = get_and_do_work())
                return error;
            else
                puts("too many workers");
    return 0;
}
```

Curly braces can be used to force the appropriate association:

```
int worker() {
    int error;
    if (!too_many_workers()) {
        for (;;)
            if (error = get_and_do_work())
                return error;
    } else
        puts("too many workers");
    return 0;
}
```

Better code organization makes the code clearer. The idiomatic use of an empty block, `{}`, to indicate that the body of the `while` statement is an empty statement:

```
int worker() {
    if (too_many_workers()) {
        puts("too many workers");
        return 0;
    }
    int error;
    while (!(error = get_and_do_work()) {}
    return error;
}
```

The `else if` indentation style shown below does not cause an error, nor does an error occur when the `if` and its matching `else` are on the same line, as shown below:

```
int match_work(int k, int a, int b, int c) {
    start_work(k);
    if (k == a) a_work(a);
    else if (k == b) b_work(b);
    else if (k == c) c_work(c);
    else if (k < 0) negative_work(k) else positive_work(k);
    return final_work(k);
}
```

2.29 `goto` statement

The `goto label;` statement allows execution control to be transferred:

```
size_t merge(int src[], size_t n, int src2[], size_t n2,
             int dest[], size_t nd)
        require(n <= src.max[0] && n2 <= src2.max[0]
                && nd <= dest.max[0] && !(n1 ?+ n2)) {
    assert(nd >= n + n2);
    int *d = dest, *s = src, *end = s + n
    int *s2 = src2, *end2 = s2 + n2;
    if (n > 0 && n2 > 0)
        for (;;)
            if (*s < *s2) {
                *d++ = *s++;
                if (s1 == end1) goto end;
            } else {
                *d++ = *s2++;
                if (s2 == end2) goto end;
            }
end:
    while (s < end) *d++ = *s++;
    while (s2 < end2) *d++ = *s2++;
}
```

The only restriction on the `goto label` statement is that it can not jump over local

variable declarations that remain in scope, i.e. that are still accessible at `label`. In the `merge()` function above the `goto end;` could be replaced by `break`, see §2.32.

2.30 `switch` statement

The switch statement:

```
switch (expression) compound_statement
```

The `switch` statement evaluates an integer valued `expression` and compares the value with the `case constant_expression:` labels within the `compound_statement`, if the value of the `expression` is equals to the value of one of the constant expressions in the `case` labels, execution continues with the statements with that label, as if a `goto` statement to that label had occurred. If no `case constant_expression:` is equals to `expression`, execution continues at the `default:` label, if there is one, otherwise execution continues after the `switch`'s `compound_statement`.

Any statement within the `compound_statement` or its sub-statements can be labeled with one or more `case constant_expression:` labels or with the `default:` label. The `constant_expression` values must all be different, the `default:` label can be used at most once. An example of the `switch` statement follows:

```
bool is_white_space(byte b) {
    switch (b) {
    case ' ':                   // space
    case '\t':                  // tab
    case '\n':                  // newline
    case '\r':                  // carriage return
        return true;
    default:
        return false;
    }
}
```

The order of the `default:` and `case constant_expression:` labels is immaterial, execution proceeds to the `default:` label if the value of the `switch` expression is not equals to any of the `case constant_expression:` labels, irrespective of their order. Sometimes for brevity when multiple case apply to the same statement they are placed in a single line.

Execution continues down the statement list, uninterrupted, as other labels are encountered, for example in the `zero_memory_small()` function shown below, which is a specialized function that zeros up to 7 bytes of memory.

```
void zero_memory_small(ubyte mem[], size_t size) {
    assert(size <= 7);
    switch (size) {
    case 7: *mem++ = 0;
    case 6: *mem++ = 0;
    case 5: *mem++ = 0;
    case 4: *mem++ = 0;
    case 3: *mem++ = 0;
    case 2: *mem++ = 0;
    case 1: *mem   = 0;
    }
}
```

For example, if the value of `size` is 5 then 5 sequential assignments are executed. The purpose of this form of zeroing memory is to unroll the zeroing of small memory regions for performance purposes.

The `zero_memory_small()` function is used by the `zero_memory()` function to deal with the zeroing of small memory areas of up to 7 bytes in various circumstances. Number 7 corresponds to the value of `sizeof(ularge) - 1` on systems whose native types are 64 bits wide, or less.

```
void zero_memory(ubyte mem[], size_t size) {
    if (size < sizeof(ularge)) {
        zero_memory_small(mem, size);
        return;
    }
    size_t x = cast(size_t) mem % sizeof(ularge);
    if (x != 0) {
        x = sizeof(ularge) - x;
        zero_memory_small(mem, x);
        mem += x;
        size -= x;
    }
    if (size >= sizeof(ularge)) {
        ularge *m = try_cast(ularge *, mem, NULL) mem; //§14.12
        ularge *endm = m + size / sizeof(ularge);
        while (m < endm) *m++ = 0;
        size %= sizeof(ularge);
        mem = cast(ubyte *) endm;
    }
    zero_memory_small(mem, size);
}
```

The more complicated case corresponds to zeroing bytes until a `ularge` aligned boundary is encountered, then zeroing memory in units of `ularge` sized words, and then zeroing the leftover bytes.

The `break` statement causes control to be transferred after the `switch` statement:

```
bool is_white_space(byte b) {
    bool result;
    switch (b) {
    case ' ': case '\t': case '\n': case '\r':
        result = true;
        break;
    default:
        result = false;
        break;
    }
    return result;
}
```

2.31 `do while` iteration statement

The `do while` statement has this syntax:

```
do statement while (expression)
```

The *`statement`* is always executed once. Then the *`expression`* is evaluated, and if its value is non-zero, the *`statement`* is executed again, this process is repeated until the *`expression`* if false. For example:

```
bool prompt(char question[]) {
    char answer;
    do {
        prompt_user_y_or_n(question);
        answer = get_answer();
    } while (answer != 'y' && answer != 'n');
    return answer == 'y';
}
```

2.32 `break` and `continue` statements

The `continue` statement is used to cause control flow to be transferred to the controlling part of the closest surrounding iteration statement. The use of `continue` within the `while` on the left is is equivalent to the code on the right:

```
while (expression) {
    some_statements;
    if (expression)
        continue;
    other_statements;

}
```

```
while (expression) {
    some_statements;
    if (expression)
        goto cont;
    other_statements;
cont: ;
}
```

The use of `continue` within the `for` statement in the left is equivalent to the code one the right:

```
for (initialization_expression;
     expression;
     iteration_expression) {
    some_statements;
    if (expression)
        continue;
    other_statements;

}
```

```
for (initialization_expression;
     expression;
     iteration_expression) {
    some_statements;
    if (expression)
        goto cont;
    other_statements;
cont: ;
}
```

The `break` statement is used to cause control flow to be transferred after the closest surrounding iteration statement or `switch` statement. The use of `break` within `switch` and `while` statements on the left is equivalent to the `goto` based code shown on the right:

```
switch (expression) {
case constant_expression:
    statements;
    break;
case constant_expression:
    while (expression) {
        statements;
        if (expression)
            break;
        statements;
        switch (expression) {
        case constant_expression:
            statements;
            break;
        case constant_expression:
            statements;
            break;
        }
        statements;
    }
    statements;
    break;
default:
    statements;
    break;
}
```

```
switch (expression) {
case constant_expression:
    statements;
    goto switch_end;
case constant_expression:
    while (expression) {
        statements;
        if (expression)
            goto while_end;
        statements;
        switch (expression) { //2
        case constant_expression:
            statements;
            goto switch_2_end;
        case constant_expression:
            statements;
            goto switch_2_end;
        }
        switch_2_end: statements;
    }
    while_end: statements;
    goto switch_end;
default:
    statements;
    goto switch_end;
}
switch_end:
```

3 - Array descriptors, tuples, and literals

"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

-- C. A. R. Hoare

Array descriptors are a built-in data type, they are a building block for the safe programming nature of COOGL. Variable length and dynamically allocated arrays are described in §13. Tuples are a lightweight data structuring construct whose principal use is by functions that return more than one value. Literals are compile time constants.

3.1 Array descriptors

An *array descriptor* is a compiler implemented data type that is used to describe a contiguous area of memory organized as an array. The following sections introduce various concepts related to array descriptors and safe programming. Variable length arrays and other details about array descriptors are presented in chapter §13.

Traditional C arrays, variable length arrays, and array descriptors have these members: `start`, `end`, and `max[N]`, `start` points to the first element of the array and `end` points immediately after the last element of the array. The number of elements for the `i`th dimension of the array, counting from zero and numbered left to right, is `max[i]`. These members don't exist in memory at run time for traditional C arrays, because their values are known at compile time. An example showing some invariants:

```
void f() {
    int a[2][3];
    assert(a.max[0] == 2 && a.max[1] == 3 &&
           a.start == &a[0][0] && a.end == &a[1][2] + 1);
    assert(a.end - a.start == a.max[0] * a.max[1]);
}
```

A declaration that uses the `[]` declarator, without a size between the square brackets, specifies an array descriptor, for example:

```
int tab[10];
void f() { int desc[] = tab; }
```

The use of `[]` in C, where it is used as an alternative syntax to declare an argument of pointer type, is source code compatible when COOGL, see §S.6.

For source code compatibility with C, assuming a unidimensional array (a traditional C array, or a variable length array, or an array descriptor), the array name is the same as the array's `start` value. The name of a multidimensional array is not the same as its `start` value. For example:

```
int tab[10];
void f() { int desc[] = tab; increment(desc); }
void increment(int *p) { (*p)++; }
```

The last element of `tab` is `tab[9]`, `desc.end` points to memory that is not part of `tab`. The range of memory described by an array descriptor is the open ended range: [`start`, `end`) starting with `start` and up to, but excluding, `end`. Array descriptors are capable of describing empty arrays, i.e. when `start` is equals to `end`. For every array descriptor it is always the case that `start <= end`.

An array descriptor can also describe a sub-array within another array or another array descriptor, for example:

```
int tab[10];
int desc[] = tab;
int last5[] = &tab[5];
int same_last5[] = &desc[5];
```

Both, `last5` and `same_last5`, refer to the last 5 elements of `tab[10]`. The sub-array descriptor is specified by the address of the starting element, and extends to the last element of the array.

An array descriptor that refers to elements within an array that doesn't extend to the last element of the array can be specified with two indexes separated by a colon, i.e. `[first : after]`, `first` is the index of the first element, and `after` is the index of the element after the last element that is to be included in the array descriptor. The number of elements included is `after - first`. For example:

```
int tab[10];
void example() {
    int first5[] = &tab[0 : 5];      // tab[0] ... tab[4]
    int last5[] = &tab[5 : 10];      // tab[5] ... tab[9]
    int last2[] = &last5[3 : 5];     // tab[8] ... tab[9]
    int cut = 3;                     // cut in 1 ... 8
    int low[] = &tab[0 : cut];       // tab[0] ... tab[cut-1]
    int high[] = &tab[cut : 10];     // tab[cut] ... tab[9]
    int empty[] = tab[4 : 4];        // empty array descriptor
}
```

The creation of an array descriptor with indexes that are out of bounds or where the `first` index is greater than the `after` index causes a run time exception, see §14.28. Note that the range specified in `&tab[5 : 10]` does not specify an out of bounds in-

dex by specifying 10, it doesn't raise an exception, because the index of the last element to be included in the array descriptor is 9 (i.e. 10 - 1) which is a valid element within the array.

Array descriptors can be walked with various iteration loops, even if they describe an empty range:

```
void work(int desc[]) {
    for (index i = 0; i < desc.max[0]; i++) use(desc[i]);
    for (index i = desc.max[0]; --i >= 0;) use(desc[i]);
    for (int *p = desc.start; p < desc.end; ++p) use(*p);
    for (int *p = desc.end; --p >= desc.start;) use(*p);
}
```

Walking the array backwards with pointers, as shown above in the last `for` loop, is correct, the equivalent code in C is supposed to be undefined behavior in C89 and its descendants (even though on modern systems the generated code does what is expected) because the value of `p`, when it is decremented so that it is not greater or equals to `desc.start`, undefined behavior in the C standard in this area is to accommodate hardware that uses segment based addressing. Idiomatically this kind of code does occur in practice, and because it only has trouble with obsolete segmented architectures, this C89 restriction is not imposed by COOGL, which is not supported on those obsolete segment based systems.

3.2 Multi dimensional array descriptors

Multi-dimensional array descriptors are declared with multiple empty square brackets. An example declaration and use of a multi-dimensional array descriptor:

```
void work(int m[][][]) {
    for (index i = 0; i < m.max[0]; i++)
        for (index j = 0; j < m.max[1]; j++)
            for (index k = 0; k < m.max[2]; k++)
                use(m[i][j][k]);   // walk array with indexes
    for (int *p = m.start; p < m.end; ++p)
        use(*p);                   // walk array with pointer
}
```

The array `int d[4][2]`, is initialized to have sequential values from 0 to 7:

```
int d[4][2];
void work() {
    for (index i = 0, n = 0; i < 4; i++)
        for (index j = 0; j < 2; j++)
            d[i][j] = n++;
    int sub3by2[][] = &d[1];         // same as &d[1 : 4]
    int middle2by2[][] = &d[1 : 3];
    int tab[] = &d[1][0];            // same as &d[1][0 : 2]
}
```

Shown below in increasing address order:

```
d[0][0]: 0
d[0][1]: 1
d[1][0]: 2
d[1][1]: 3
d[2][0]: 4
d[2][1]: 5
d[3][0]: 6
d[3][1]: 7
```

The memory layout of the `sub3by2[][]` sub-array is:

```
sub3by2[0][0]: 2
sub3by2[0][1]: 3
sub3by2[1][0]: 4
sub3by2[1][1]: 5
sub3by2[2][0]: 6
sub3by2[2][1]: 7
```

The layout of the `middle2by2[][]` sub-array is:

```
middle2by2[0][0]: 2
middle2by2[0][1]: 3
middle2by2[1][0]: 4
middle2by2[1][1]: 5
```

The first two sub-arrays above have the same number of dimensions as their base array. The `tab[]` subarray has a single dimension:

```
tab[0]: 2
tab[1]: 3
```

3.3 Array descriptor access restrictions

Array descriptors are value like objects, almost as if they were addresses, but instead of specifying a single object in memory, they describe multiple objects and their organization in memory, i.e. the indexing structure through which they are accessed.

The address of an array descriptor can not be obtained by using the address-of operator, `&`, with an array descriptor, such use obtains the address of the data that the array descriptor refers to, i.e. the underlying array entries.

Array descriptors that are not initialized explicitly are initialized by the language to a zero element array. Array descriptors can be declared in any context, other than as a member of a structure or a union. Array descriptors can only be changed in a controlled way. Their `start`, `end`, and `max[]` members can not be changed individually.

3.4 Restricted array descriptors

The array elements that the array descriptor refers to, and its `start`, `end`, and `max[N]` members can only be accessed when the array descriptor is a local non-static variable of a function. All other array descriptors: members of an object, static local variables, or global array descriptors are *restricted array descriptors*. The only operations that can be performed on restricted array descriptors are:

- use as an argument to a function
- use as the return value of a function
- use as the source value in an assignment to another array descriptor
- use as the target of an assignment from another array descriptor

Copying a restricted array descriptor to a local non-static variable ensures that the copy of the array descriptor can not be changed concurrently by another thread, or even another function invoked by the same thread, because its address can not be obtained. Forcing all accesses to occur through a non-static local array descriptor variable allows for compiler optimization without concern for address aliases or concurrency aspects related to the array descriptor itself.

3.5 Restricted array descriptor accesses are atomic

Assignment to non-local array descriptors is performed in such a way that it occurs atomically. Concurrent code referencing the array descriptor sees all of the old values of its members, or all of their new values, not a combination of them.

The implementation of the atomicity of assignment and of fetching of restricted array descriptors is platform dependent, see §1L.3 for the Intel/AMD x86/64, ARM 64 bit, and IBM POWER implementations. The performance characteristic of these operations can be assumed to be highly optimized by the compiler, and is very close to the performance of the memory operations that would be required if the operations where not atomic, see §1L.3 for performance measurements.

The performance of fetching or storing local array descriptors with `N` dimensions corresponds to the inline memory load or store operations of their underlying members: `start`, `end`, `max[0]`, … `max[N]`.

Atomicity of non-local array descriptor accesses do not lead to deadlocks in their implementations, for example, when two global array descriptors, `a[]` and `b[]`, are

assigned concurrently in separate threads as: `a = b` and `b = a` deadlock never occurs, what is atomic is the individual fetching and storing of them, not the whole combined fetch and store operations. The implementation of `a = b;` can be considered to be equivalent to the following pseudo code, without any function call overhead, where `t` is a temporary local array descriptor:

```
t = b.atomic_fetch();
a.atomic_store(t);
```

3.6 Array and array descriptor indexing is checked

Indexed accesses to arrays and array descriptors are checked, either at run-time, or at compile time. The example functions, above, have their indexed accesses checked at compile time, invariants about the values of `i`, `j`, and `k` are used to determine, at compile time, that all the array element accesses are safe.

Run time checks are performed when it is impossible to determine, at compile time, under global compilation, if the array element accesses are safe. The compiler produces a warning when it generates run time checks as a reminder to the programmers that they have not provided evidence that the access is always valid. For example in:

```
int rand_val(int array[][]) {
    return array[random_index()][random_index()];
}
```

An out of bounds memory access attempt causes an out of bounds exception. The behavior of run time exceptions is explained in §14.33.

3.7 Arrays of arrays vs multidimensional arrays

Conceptually, C does not have multidimensional arrays, what it has is unidimensional arrays. When multiple dimensions are required in an array, what is technically declared are arrays of arrays. For example, `int a[2][4]`, declares `a` to be an array with two elements, each one of those two elements is an array of four elements. This technical and conceptual view is completely proper for C and it is the simplest way to support multi-dimensional arrays in the C programming language.

It is important to emphasize that in COOGL the declaration `int a[2][4]`, declares a multidimensional array, in this case a two dimensional array. It behaves exactly as a C array, for example the expression `a[1]` refers to the second subarray of 4 elements within the `a` array. For all practical purposes there is no difference between both languages, the only difference is that in the context of array descriptors, a concept that does not exist in C, when `a` is used in a context that requires an array descriptor to be created by the compiler, for example to pass it as an argument to a function that requires an array descriptor, the array is considered as a whole to derive and produce a multidimensional array descriptor.

It is important also to indicate that there is no such thing as array descriptors of array descriptors. Multidimensional array descriptors are not implemented as arrays of array descriptors. Jagged arrays, can be simulated in C with arrays of pointers that point to arrays of various sizes, for example:

```
int a0[11], a1[22], *a[2] = {a0, a1};
```

Where accesses like `a[0][10]` and `a[1][20]` are valid and look like a two dimensional array reference. Apart from an intellectual curiosity jagged arrays are not used often. Syntactically they could be declared in COOGL as an array of array descriptors:

```
int a0[11], a1[22], a[2][] = {a0, a1};
```

Note that unless a was a non-static local declaration array indexing expressions such `a[1][20]` would cause a compilation error because the array descriptor `a[1]` is a restricted array descriptor and can not be used in such an expression.

3.8 Array descriptor use in expressions

A non-static local array descriptor can be used in expressions to refer to sub-arrays of fewer dimensions. For example:

```
int example(int a[], int u[][], int x[][][]) {
    int *b = a;         // a stands for &a[0]
    int v[] = u[0];     // u[0] is a unidimensional sub-array
    int y[][] = x[0];   // x[0] is a two dimensional sub-array
    int z[] = x[0][0]; // x[0][0] is a unidimensional sub-array
}
```

3.9 Pointer arithmetic and array descriptors

Pointer arithmetic is only allowed if it is known, at compile time, that the pointer points within an array, for this knowledge to be available, the compiler must be able to determine, at compile time, that the pointer is associated with a specific array descriptor. For example, the following code, which is unsafe C code, is safe COOGL code because the bound check to avoid accesses past the array is inserted by the compiler:

```
large total_until_zero(int a[]) {
    large sum = 0; // add all values until a zero value is seen
    int *p = a;
    int v;
    while (v = *p++)
        sum += v;
    return sum;
}
```

The bound checking C11 code generated by the compiler is shown below. If the

bound check fails a run-time exception is raised, see §14.33:

```
large total_until_zero(int *a, size_t a__max0) {
    int *auto__a__end = a + a__max0;
    large sum = 0; // add all values until a zero value is seen
    int *p = a;
    int v;
    while (v = (lang__bound_check(p, auto__a__end), *p++))
        sum += v;
    return sum;
}
```

If it is possible that a pointer at a specific code location, could refer to array elements within more than one array, a compilation error is produced, for example:

```
int example(bool use_a, int a[], int b[]) {
    int sum = 0;
    int *p = use_a ? a : b;
    int v;
    while (v = *p++)  // error: unknown array bounds
        sum += v;
    return sum;
}
```

All such code can be corrected by using an additional local array descriptor to satisfy this requirement, this simplifies the compiler and doesn't seem to be too burdensome to require the programmer to address this.

3.10 Use of pointers based on array descriptors is always safe

Use of pointers based on an array descriptor are always safe, see §14, run time checks are performed when it is impossible to determine, at compile time, if a pointer is safe, as shown above.

Iterating over the array elements within an array or array descriptor, walking it forwards or backwards, through pointers or indexes (e.g. walking from 0 to `max[0]`), does not introduce run time checks. For example:

```
large total(int a[]) {
    large sum = 0;
    for (int *p = a, *end = a.end; p < end; ++p)
        sum += *p;
    return sum;
}
```

Run-time checks are not required in this case because the array descriptor can not be affected by any other code path. By construction array descriptors always reference valid memory, or if the array descriptor has not been initialized, it refers to a dummy zero element array.

There is always an array element before and another one after every array, with the exception of arrays within a `struct`, `union`, or a `class struct` see §7.6, which must follow the structure layout of the C compiler. An array within a `struct` can not have its address used to form an array descriptor unless the array descriptor is for a subset of the array that guarantees that there is at least one extra element prior to the array descriptor described sub-array and another one after it. In consequence, for every array descriptor, objects at `start-1`, and at `end` always exist, they are either properly constructed objects or uninitialized objects set to their deconstructed values, see §14.22, dereferencing pointers whose values are `start-1` or `end` is not undefined behavior, it could be a programming error or it could be what the programmer intended depending on what they are doing.

3.11 Functions that return array descriptors

Functions that return array descriptors, for example `trim_space()` receives an array descriptor for a character array, and returns an array descriptor that refers to the same memory but excluding and leading or trailing spaces. Note that the array descriptor declarator for the unidimensional array descriptor, `[]`, for the return value goes to the right of the function's argument list:

```
char trim_space(char buf[])[] {
    index first = 0, max = buf.max[0], last = max;
    for (; first < max; ++first)
        if (!libc.isspace(buf[first])) break;
    while (last > first)
        if (!libc.isspace(buf[--last])) break;
    return &buf[first : last + 1];
}
```

A function that returns a two dimensional array descriptor:

```
int d[4][2];
int middle2by2()[][] { return &d[1 : 3]; }
```

3.12 Implicit array descriptor for string literals

When a pointer to a character type is initialized to point to a string literal, an implicit array descriptor is associated with the pointer, allowing pointer arithmetic within the string literal according to the array descriptor:

```
char g() { char *p = "dog"; p += 2; return *p; }
```

3.13 Tuples

Tuples are a lightweight data aggregation construct, their principal use is in functions that return multiple values. Functions that return a tuple value, variables that are

tuples, and type definitions for tuple types are allowed. Tuple expressions are used to form tuples and to extract values from tuples. Tuple declarations allow for the tuple members to be initialized. A function that returns a tuple value, whether the tuple is declared by a typedef or declared as part of the function declaration itself has the members of the tuple as local variables that are used to built the value returned byt the function. The value of a tuple is the value of its last member. All of this is shown in the following example.

```
typedef tuple [int fd = -1, error_t err = libc.EINVAL] fderr_t;

fderr_t topen(char name[], int mode) {
    if (!name) return;            // same as: return [fd, err];
    if (name[0] == '\0') return [-1, libc.ESRCH];
    if ((fd = libc.open(name, mode)) == -1)
        return [-1, libc.errno];
    return [fd, 0];
}

tuple [ int fd = -1, error_t err = libc.EINVAL]
    topen2(char name[], int mode) return topen(name, mode);

void use() {
    tuple [int fd, error_t e] r = topen("file", libc.O_RDONLY);
    fderr_t fe = topen("file", libc.O_RDONLY);
    if (fe.error == -1) return;
    int x;
    error_t e;
    if ([x, e] = topen("file", libc.O_RDONLY)) return;
    fe = [x, e];
    assert(fe.fd == x && fe.err == e);
    [x, e] = [-2, 0];
    assert(x == -2 && e == 0);
    int a = 1, b = 2;
    [a, b] = [b, a];  // result unspecified, values not swapped
}
```

A function can be invoked with tuple values as part of the argument list, the function itself can specify its arguments with, or without tuples, the only requirement is that all the arguments be passed and that the types match. For example:

```
void f(int x, error_t e){ ... }
void g(fderr_t fe){ ... }
void use() {
    fderr_t fe;
    f(fe); f(-1, libc.EINVAL);
    g(fe); g(-2, libc.ENOENT);
}
```

3.14 Literals

A literal declaration is a declaration of a compile time constant, its initialization expression must be a constant expression

```
lit int N = 100;
lit int NBPB = 8;                    // number of bits per byte
lit ularge ULARGE_MSB = 1uLL << (sizeof(ularge) * NBPB - 1);
lit double PI = 3.14159265358979323846264338 32;
```

Note that in C `const` is used to indicate that a data item can not be modified, it is not used to declare constants.

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

4 - Classes and inheritance

> *"The fundamental mechanism for decomposition in ALGOL 60 is the block concept. As far as local quantities are concerned, a block is completely independent of the rest of the program. ... A block is a formal decomposition, or "pattern", of an aggregated data structure and associated algorithms and actions. ..."*
>
> *"The notion of block instances leads to the possibility of generating several instances of a given block which may co-exist and interact, such as, for example, instances of a recursive procedure. This further leads to the concept of a block as a "class" of "objects", each being a dynamic instance of the block, and therefore conforming to the same pattern."*
>
> *"An extended block concept is introduced through a "class" declaration and associated interaction mechanism such as "object references" (pointers), "remote accessing", "quasi-parallel" operation, and block "concatenation."*
>
> -- Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard

COOGL unifies the notions of class and function. A class is a function, and a function is a class. Member functions are a simplified form of nested functions.

4.1 Contract specification: `vital`, `require()`, and `promise()`

A class declared `vital` indicates that every object that could be created by the language specification must be created, without any object being removed because of various optimizations related to value objects. A function declared `vital` indicates that its value must always be used, it can not simply be ignored, see §9.10 and §9.11. If `vital` is used, it must immediately follow the functions declaration and prior to its body or `defer` or `redef` keywords, see §4.6, §6.4, and §6.6.

A function, or a class, can include in its declaration a contract specification, an idea borrowed from the Eiffel programming language. The requirements for the function to be called appropriately, i.e. the requirements on the function's caller, can be optionally specified in a `require(`*`expression`*`)` specification, immediately following the function's declaration (or `vital` if that is specified) and prior to the function's `prom-`

`ise()` specification, if any, or otherwise prior to its body (or `defer` or `redef` keywords). The promises made by the function or class to its caller can be optionally specified in a `promise(expression)` specification after the `require()` specification, if any, or otherwise after the function's declaration (or `vital` if that is specified), and prior to the functions body (or `defer` or `redef` keywords). The requirements and promises are the pre-conditions and post-conditions to the function invocation, or to the construction of an object. The *`expression`* must be true for a correct execution, if they are not true a run-time exception is raised, see §4.2 for an example. A compile time error might occur if it can be determined that the *`expression`* is always false. See §4.8 for contract specification details that relate to inheritance and member function redefinitions.

4.2 Class declarations are function declarations

The `class` keyword is used for the declaration of a class. The fundamental data structure abstraction mechanism in COOGL is the `class`. Use of `struct` is for C language interfacing and traditional C style programming in COOGL.

A `class` declaration provides the data content of objects of the class type, and it is also the function used to construct such objects. Class `stack`, shown below, allows its user to pass the maximum stack capacity as an argument to the constructor, this is an improvement over the earlier version shown in §1.3. The constructor sets `*error` to a non-zero error value if the stack construction failed.

```
class stack(size_t max, int *error) promise(empty()) {
    priv int entries[];
    entries.create(max);        // allocate space for stack §13.8
    priv int *sp = entries;
    *error = !sp ? libc.ENOMEM : 0; // ENOMEM error if no space
    return;

    pub void deinit() { entries.destroy(); }  // free space §13.8
    pub bool empty() { return sp == entries; }
    pub bool full() { return sp == entries.end; }
    pub void push(int v) require(!full())
                         promise(!empty()) { *sp++ = v; }
    pub int pop() require(!empty()) { return *--sp; }
    pub int top() require(!empty()) { return sp[-1]; }
    pub int count() { return sp - entries; }
}
```

4.3 Accessibility modifiers and member declarations

Variables declared with an accessibility modifier, i.e. `pub`, `priv`, or `prot`, are data members of the class. Data members declared without `static`, are *non-static data*

members, i.e. they are per object data. Data members declared with `static` are *static members* of the class, a single instance of the *static member* data exists, irrespective of the number of objects of the class that exist. Both `entries` and `sp` are *non-static data members* of `stack`. Their lifetime is the lifetime of the object that contains them. Local variables and arguments of the constructor function, such as `max` and `error`, are not members of the object being constructed, they are not members because they were declared without an accessibility modifier.

A function argument can also be declared as a member, this allows for the common case of matching constructor arguments and members to be unified as shown below for this modified `stack` and its new member `max`.

```
class stack(priv size_t max, int *error) promise(empty()) {
    // ... rest unchanged ...
}
```

A public member is declared with the `pub` accessibility modifier, which makes it accessible from outside of the class, the member is part of the class specification, code that isn't part of the class code can access `pub` members. A private member is declared with the `priv` accessibility modifier, which renders it inaccessible by any code other than the class constructor and its member functions, `priv` members exist only to implement the class functionality, for example `entries` and `sp` in `stack` above. See §6.7 and §6.8 for more on accessibility modifiers. The `bytes` static data member added to `stack` below:

```
class stack(size_t max, int *error) promise(empty()) {
    priv int entries[];
    entries.create(max);
    priv int *sp = entries.start;
    *error = sp ? libc.ENOMEM :
                  (bytes += max * sizeof(int), 0);
    return;

    priv static size_t bytes = 0;
    pub void deinit() {
        if (entries.start != entries.end) {
            bytes -= entries.max[0] * sizeof(int);
            entries.destroy();
        |
    }
    pub static void info() {
        on ("memory used: "; bytes; '\n') print();
    }
    // ... rest unchanged ...
}
```

Total memory allocated internally for the entries of all `stack` objects is tracked in `bytes`. The lifetime of the `priv static bytes` member is the lifetime of the class

itself, which is the lifetime of the program execution, unless `stack` is in a dynamically loaded module, in that case its lifetime starts at module load time and ends at module unload time. The ability to have members is not a special feature of classes, non-class functions can have them as well, this is explained in §4.15.

Members must be declared within the outermost block of a function or class declaration, i.e. they must be declared within the block that is the class or function body; member variables can also be declared as arguments. Members can not be declared within compound statements, or within statements subordinate to other statements, for example, within the statement that is executed subordinate to an `if` statement.

The notions of members and member declarations have complementary notions, local entities and local declarations, also known as non-member entities and non-member declarations. Local entities are the arguments and other entities that were declared within a class, or a function, without `pub`, `priv`, or `prot`.

Access to a static member does not require an object of the type of the class to be provided because the member is global to the class, it is not a member that is per object. A static member can be accessed through the class type name or through an object. Like any other member, access to static members is subject to its accessibility modifier.

4.4 Object declarations and `decl`

An object declaration and its construction is a combination of the declaration syntax and the function invocation syntax, as shown below for `s`. Note that when the construction function requires arguments at object declaration time, the declaration must be preceded by the `decl` keyword or by an accessibility modifier (if declaring a member), to make it clear that it is a declaration, and not just a function invocation:

```
void test() {
    int error;
    decl stack(100, &error) s;
    if (error) libc.abort("s construction failed");
    s.push(1);
    assert(!s.empty());
    int v = s.pop();
    assert(s.empty() && v == 1);
    s.info();
    stack.info();
}
```

4.5 Member functions

A member function does not have access to non-member entities of the class constructor, for example, the `error` argument of the `stack` constructor function. The

lifetime of local variables ends when the constructor function returns, whereas the object, i.e. the entity formed by the members, continues to exist until the object is destructed, at a later time.

A non-static member function of a class can access all of its members, irrespective of whether they are static or non-static. Non-static member functions operate on objects of the class type, they implicitly refer to the object's members. Non-static member functions can only be invoked on objects.

Static member functions can only access static members of the class. When a static member function is invoked from a function that is not one of its member functions, it can be invoked using the class name or an object of the class. For example, in `test()`, above, `s.info()` uses the `s` object to invoke the static member function `info()` which makes that invocation indistinguishable from `s.pop()`, it doesn't syntactically reveal that what is being invoked is a static member function. The second invocation `stack.info()` uses the class name `stack`, instead of an object name.

The *`classname.member`* form of member access can only be used to access static members. To access non-static members an object has to be specified. For example:

```
void test() { stack.pop() } // error: pop() requires an object
```

An object on which to `pop()` was not specified, resulting in a compilation error.

4.6 Introduction to inheritance and member function redefinition

Inheritance declarations in COOGL are member declarations with the `inherit` modifier, the member is usually unnamed. For example, `polar` inherits from `point`:

```
class point(priv double x, priv double y) {
    pub int print() {
        int n = on ("x="; x; " y="; y) print();
        return lang.on_int_count_result(n, 4);      // see §9.2
    }
}
class polar(double x, double y) {
    pub inherit point(x, y);
    priv double ro = libm.sqrt(x * x + y * y);
    priv double teta = libm.atan2(y, x);
    return;

    pub int print() redef {
        int n = point.print();
        if (n <= 0) return n;
        n = on (" ro="; ro; " teta="; teta) print();
        return lang.on_int_count_result(n, 4);      // see §9.2
    }
}
```

The `polar` class, inherits from `point`, it adds polar representation in radians. The `print()` member is redefined to show both representations, note the `redef` above.

Use of `point` and `polar`:

```
int main() {
    decl point(1.0, 0.0) p1;
    decl polar(1.0, 1.0) p2;
    on ("p1: "; p1; "   p2: "; p2; "\n") print();
}
```

The output is:

```
p1: x=1 y=0   p2: x=1 y=1 ro=1.414214 teta=0.785398
```

The term *named inheritance* refers to inheritance declarations that include an identifier, `polar` uses *unnamed inheritance* in its inheritance of `point`. Use of named inheritance is usually not required unless the inheritance causes name clashes. Whenever there are name clashes, name disambiguation is required, named inheritance can be used to resolve these ambiguities, as shown later in §6.13.

4.7 Access to redefined member functions

Note, above, that the redefined `print()` member function in `polar` is allowed to access the original `print()` member function of `point` by specifying the class name, `point`, and using the dot operator on it to select `point()`, i.e. `point.print()`. Only member functions of the class that redefined the member function are allowed to use this syntax, it can not be used to access redefinitions made earlier by its ancestor classes. For example, `colored_polar` can not access `point.print()`:

```
class colored_polar(double x, double y, priv rgb_t color) {
    pub inherit polar(x, y);
    return;
    pub void print() redef {
        //point.print(); // error: point.print() inaccessible
        int n = polar.print();
        if (n <= 0) return n;
        n = on (" color="; color) print();
        return lang.on_int_count_result(n, 2);      // see §9.2
    }
    pub void print_base() { polar.print(); }  // valid
}
```

4.8 Contract specifications and member function redefinitions

The contract specification of a member function applies also to redefinitions of the member function. A contract specification can only be specified for a member func-

tion within the class declaration where the function was first declared, not in redefinitions of the functions in classes that descend from it. If a member function is declared, deferred or not, without a contract specification, then subsequent redefinitions of the member function can not specify a contract for them. This restriction in contract specifications makes them very simple, it might be tempting to allow relaxing the requirements and strengthening the promises of a redefined function, as in Eiffel, but it is too complex to allow choosing which contracts to honor and which to ignore.

4.9 Restrictions on constructor calls to non-static member functions

While an object is being constructed, non-static member functions can not be invoked on it, with one exception, a `void priv` non-static member function can be invoked as part of the final `return` statement of the constructor, an exception made to allow constructor related code to be in other functions see §4.11. The `void priv` non-static member function can not reference explicitly `this`, see §4.13, within its code and can not call other non-static member functions on the object with the exception of other `priv` non-static member functions that follow these same restrictions. This restriction exists to ensure that member functions always operate on a fully formed object, not a partially constructed object.

For the same reason, member functions can not be invoked from the destructor, with a related exception for modularization of a large destructor, see §5.4. Also see §6.14 for additional technical details about these restrictions and their relationship to inheritance and when pre and post-conditions are evaluated.

This restriction on the constructor is relaxed for member functions of non-class functions, a non-class function can invoke it's non-static member functions as long as they do not access members that have not yet been constructed at the time the member function is invoked, they are allowed to call other non-static member functions that follow this same restriction.

4.10 Constructor organization

To facilitate reading a `class` declaration, its data members are, by convention, declared towards the beginning of the `class`, unless intervening local variables and other code are required for the efficient construction of the members.

Another style convention is for member functions to be declared towards the end of the `class`, after a final `return` statement in the `class` constructor, which indicates the end of executable code and its non-static data members. Any code or non-static data members after the final `return` statement are thus unreachable and result in a compilation error. Thus this final `return` is a reliable indication that no additional constructor code follows after it, this stylistic convention assumes that there are no `goto` statements that target labels beyond that final `return`. If such a baroque con-

structor is required, the label ought to immediately follow the `return` which could otherwise have been perceived as the final `return` statement. A better approach is to move the bulk of the construction work to a `void priv` non-static member function, as explained in section §4.11.

4.11 Complicated constructor and the `ini()` programming idiom

Classes with complicated constructors should declare a non-static member function and invoke it from the class constructor to do within it *most* of the construction, which can make the class declarations easier to read. As a convention such a member function is named `ini()`. The actual construction of the non-static members of such a class can not be deferred to the `ini()` function, thus there usually is some minimal construction that occurs within the constructor function itself. For example:

```
class stack(size_t max, int *error) promise(empty()) {
    priv int entries[];
    priv int *sp;
    return ini(max, error); // use priv void non-static member
                            // function, only allowed here §4.9
    priv void ini(size_t max, int *error) inline {
        entries.create(max);
        sp = entries.start;
        *error = !sp ? 0 : libc.ENOMEM;
    }
    // ... rest unchanged ...
}
```

4.12 Member declarations and initialization are unified

The unification of `class` and function declarations allows for member declaration and initialization to occur in one place, which reduces errors that occur in other languages where declaration and construction must be in separate places (e.g. C++).

The following Hanoi Towers class, `towers`, has three members of type `stack`, which are declared and constructed based on `towers`' `n` argument.

```
class towers(pub lit int n, int *error) {
    int e, e2, e3;
    priv stack(n, &e) left;
    priv stack(n, &e2) middle;
    priv stack(n, &e3) right;
    if (!e && !(e = e2)) e = e3;
    *error = e;
    if (e) return;
    for (int i = n; --i >= 0; ) left.push(i);
    // ... some other code ...
}
```

Syntactically, when an object is being declared the argument list is specified as the argument list of the constructor function, not as an argument list associated with the name of the entity being declared (as is done in C++). Assuming the modified version of class `stack`, below, which only sets `*error` if an error occurred:

```
class stack(size_t max, int *error) promise(empty()) {
    priv int entries[];
    entries.create(max);
    priv int *sp = entries.start;
    if (!sp) *error = libc.ENOMEM; // only set *error on error
    return;
    // ... rest unchanged ...
}
```

Then the `towers` class, which also only sets `*error` on errors, is more succinct:

```
class towers(pub lit int n, int *error) {
    priv stack(n, error) tower[3];
    if (*error) return;
    for (int i = n; --i >= 0; ) tower[0].push(i);
}
```

When a declaration, such as the declaration of `tower[3]` above, declares more than one object, the constructor function is invoked multiple times, once for each object, 3 times in this case. The expressions used as arguments to the constructor are also evaluated multiple times, once per invocation. In the `play()` function, below, variables `toy` and `world` had to be declared in separate declaration statements because the arguments for their construction are different. The `++x` expression in the declaration of the 7 towers, `bunch[7]`, is evaluated once for each, so `bunch[0]` has 3 stacks of 1 element each, and `bunch[6]` has 3 stacks of 7 elements each. When constructing the `bunch[]` array the constructor is invoked `7` times.

```
int play(int *e) {
    int x = 0;
    decl towers(7, &e) toy;
    decl towers(64, &e) world; // world end when solved at 1m/s
    decl towers(++x, &e) seven[7];
}
```

4.13 Object pointer: `this`

Each object has its own instance of the non-static class members. A non-static member function operates on a specific object, the object on which it was invoked. From a language implementation perspective a pointer to the object is passed as the first argument to the non-static member functions. A member function can access the members of its containing class directly, as if they were its local variables, which of course they are not. Sometimes access to the implicit object pointer argument is re-

quired, the keyword `this` is used to refer to it, it can be used as any other pointer variable with the exception that its value can not be changed. The type of `this` is constant pointer to the class type.

For example, some buggy code is invoking `pop()` on some empty stack, `stack` has been changed, temporarily, to print `this` when that happens:

```
class stack(size_t max, int *error) promise(empty()) {
    // ... rest unchanged ...
    pub int pop() {
        if (empty())
            on ("pop() on empty() stack: this==0x";
                (cast(uintptr_t)this).hex(); "\n") print();
        assert(!empty());
        return *--sp;
    }
    // ... rest unchanged ...
}
```

The type of `this` within the non-static member functions of `stack()` is:

```
stack *const this
```

Assignment to `this` or taking its address is invalid, it causes a compilation error.

4.14 A `stack` `iterator` and the use of `this` in the class constructor

The class constructor, i.e. the class function, is not a member function of itself. A class constructor can be a member function of a different class, i.e. when a class is declared within another class. Nonetheless, when the class constructor invokes a member function (see §4.9 and §4.11), the member function is invoked on the object that the constructor is in the process of constructing.

A class constructor does not have a `this` variable that refers to the object that is being constructed. A class constructor only has a `this` variable when the class is a non-static member function of another class, in that case `this` does not refer to the object being constructed, it refers to another object on which the constructor was invoked as a member function. For example, in class `iterator`, a non-static member of `stack`:

```
class stack(size_t max, int *error) promise(empty()) {
    // ... rest unchanged ...
    pub int get(size_t i) require(i < entries.max[0])
        { return sp[i]; }
    pub class iterator {
        priv size_t ix = count();
        priv stack *stk = this; // this' type is: stack *const
        pub bool end() { return ix == 0; }
        pub int get() { return stk->get(--ix); }
    }
}
```

The `iterator` constructor requires a `stack` object to be invoked on, as shown in the declaration of `itor` below:

```
int average(stack *s) {
    int count = s->count();
    if (count == 0) return 0;
    large sum = 0;
    decl s->iterator itor;
    while (!itor.end())
        sum += itor.get();
    return cast(int) (sum / count);
}
```

The type `iterator`, a member class of `stack`, is invoked on the `*s` object to construct the `itor` object, during `itor`'s declaration: `decl s->iterator itor;`. A stack object must be provided to invoke `iterator` on, because it is a non-static member function of `stack`, `iterator` needs access to `this` to keep a copy of it in its `stk` member variable for later use by `iterator`'s members.

The mandatory use of `decl` in the declaration of `itor` makes it easier to identify it as a declaration than if the `decl` keyword was not required. Particularly if the expression that resulted in the `stack` object on which `iterator` was invoked was a much more complicated expression.

In C, the name of a function stands for a function pointer to it, using the function name, by itself, i.e. without parentheses, does not result in the invocation of the function, it results in the address of the function, a value. For non class functions the same rule applies in COOGL. For class function invocations, i.e. the invocation that is part of an object declaration, the use of parenthesis is not required, unless the constructor invocation requires arguments. Thus, the declaration of `itor` did not require parentheses to indicate that `iterator` was a function to be invoked to construct the object. In contexts other than the type used in a declaration, the class name stands for a class literal value, a value that refers to a type that can be used for generic programming. For example, when type arguments are used, the class name stands for a type, see §11.3.

4.15 Functions as degenerate types and nested member functions

The fundamental difference between a `class` as a function and a traditional function, is that a `class` as a function defines a new named type, a type that can be used to declare variables or dynamically allocate objects. A traditional function does not introduce a new named type that can be used in such a way. Nonetheless, a traditional function does introduce a new named scope, its `static` and `lit` members can be referenced from outside the function, see §7.8. A traditional function can be thought of as a degenerate type for a short lived object that doesn't need to exist beyond the time of its construction. The type is made from the member fields declared within the function, just as if it were a `class`. When a traditional function is invoked it can be thought of as if the memory for the object was allocated on the run-time stack, at function return time the object is destroyed and the memory on the run-time stack becomes available for reuse.

The non-static member functions of a non-class function don't have access to the underlying object through a `this` variable. A pointer to the object is passed to them with the same calling convention as when non-static member functions of a class are invoked, but the object itself is not accessible explicitly through `this`.

A member function declared within another function (i.e. nested within it) can access the outer function's members. Thus member functions nested within another function, in COOGL, are no different than regular member functions. Member functions of other functions are *similar* to nested member functions in other languages. Member functions of another function are allowed access only to members of the function that directly contains them. In this respect they are very different than the arbitrarily nested functions of ALGOL68 and Pascal, where any variable or function in scope can be accessed, whether the variable or function is within the immediately enclosing function or within another function that directly or indirectly encloses that function.

Support for arbitrarily nested functions with access to all the state of all of the enclosing functions is of questionable value, it can make the amount of code that might alter the local state of a function much larger than the function's code, verifying the correctness of the function requires a careful examination of all of the functions nested within it to determine their hidden shared state dependencies.

The nested member functions of COOGL are simpler than nested functions in those languages, shared state is restricted to the function's members, and only one level of nested functions can access them. Their implementation is also much simpler, it is identical to the class member function implementation.

A common use of nested member functions is to hide functions used only to aid in the implementation of a function, even when no data is shared through members between them, for example:

```
void rotate(byte a[], size_t n) { // rotate n bytes to the end
    size_t size = a.max[0];
    assert(n <= size);
    priv void reverse(byte b[]) {
        for (byte t, *first = b, *last = b.end - 1;
             first < last; ++first, --last)
                t = *first, *first = *last, *last = t;
    }
    reverse(a, n);
    reverse(a + n, size - n);
    reverse(a, size);
}
```

The constrained lifetime of a function call makes its *object* allocation and deallocation no different than the run time stack management required to allocate and free space for local variables. The overheads of local variables whose addresses are passed to a function, and of function objects on which member functions are invoked, is the same. If more than one local variable needs to be modified by another function it is more efficient to make them members instead of passing multiple pointer arguments.

Passing a few small (e.g. pointers, integers, floating point values, etc.) variables by value and returning their new values in a tuple can be faster in some circumstances, particularly on modern systems with their large register and dedicated register sets (integer registers versus floating point registers) and their calling conventions that allow passing several values in registers and returning several values in registers. From a programming cleanliness perspective, functions are more easily understood, when arguments and tuples are used, instead of arguments that contain pointers where values are to be returned or when members are used.

4.16 Functions with default argument expressions

Function argument declarations can include an initialization expression, a default argument value, used if the argument is not provided at function invocation time. The expression is not restricted to a compile time constant expression, it can make use of the values of arguments that appear before it in the argument list. For example, a memory allocation function that returns an array descriptor that refers to the memory. The `memget()` function allows for optional specification of an alignment requirement, its default value is computed based on the the `size` argument:

```
char memget(size_t size, bool cached = true, align =
            size >= sizeof(large) ? sizeof(large) :
            size >= sizeof(int)   ? sizeof(int)   :
            size >= sizeof(short) ? sizeof(short) : 1)[] {...}
```

The order of evaluation of argument expressions is not specified by the language,

with the exception of omitted optional arguments and the default argument expression, only in that case are the depended upon argument expressions evaluated prior to the omitted default argument expressions that depend on it. Invocations of a function that have more than one default argument can only omit the arguments from right to left, for example:

```
char p[] = memget(16, true);
char q[] = memget(16, , 1);         // error: syntax error
```

See §11.3 for optional arguments and generic programming.

4.17 Stringify operator

The `#argument` expression can be used to direct the compiler to create a string with the text representation of the expression that produce the value of `argument`, stringifying can only occur in the default argument of a function. For example:

```
void assert(bool expr, char msg[] = #expr,
            char file[] = lang.file, int line = lang.line) {
    if (!expr) assert_failed(msg, file, line);
}
```

For another example, `puts_if(expr)`, see §17.

5 - Construction, assignment, and destruction

> *"A data structure, its internal linkings, accessing procedures and modifying procedures are part of a single module. ... The formats of control blocks used in queues in operating systems and similar programs must be hidden within a control block module. It is conventional to make such formats the interfaces between various modules. Because design evolution forces frequent changes on control block formats such a decision often proves extremely costly. ... It is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing."*
>
> -- D. L. Parnas, December 1972

COOGL provides execution control, through member functions, when an object is constructed, initialized from another object, initialized by default, assigned, and destructed. The relevant member functions are introduced briefly in the first sections of this chapter, the final sections of this chapter revisit these member functions in the context of a `string` class example while describing other details of them.

5.1 Value like objects

A *value-like* object is an object that is both *initializable* and *reinitializable*. An object that can be *initialized* at declaration time from another object of the same type is said to be *initializable*. Initialization is a complementary form of construction. An object that has been previously constructed that can be *reinitialized*, i.e. assigned, from

another object of the same type is said to be *reinitializable*.

Initialization of an object from another object also occurs when:

- an argument to a function is initialized from another object of the same type;
- the value returned by a function is initialized from a `return` expression.

The fundamental types (e.g. `char`, `int`, `float`, etc.) are value like, for example:

```
int factorial(int n) {
    assert(n >= 0);
    int v = n;      // v initialized with value of n
    while (--n >= 2)
        v = v * n;  // v reinitialized with value of v * n
    return v;       // return value initialized with value of v
}
void use(int x) {
    // f initialized to value returned by factorial(x)
    // argument n of factorial initialized with x
    int f = factorial(x);
}
```

Structures and unions are also value like, irrespective of the type of their members, they can be initialized and reinitialized.

Arrays are not value-like, an array can not be: initialized from another array, passed by value, returned as a function value, or assigned to another array. But an array that is a member of a structure or union is initialized and reinitialized as part of the initialization and reinitialization of a structure or union that contains them.

Pointers and array descriptors are value like. For pointers the value that is value-like is the pointer not the memory that the pointer points to. Similarly, for array descriptors, the value that is value-like is the description and organization of the memory that the array descriptor refers to, not the memory itself, so when an array descriptor is assigned to another one, or when passed as an argument to a function, etc. the underlying memory that the array descriptor refers to doesn't change. For example:

```
void example(int a[]) {
    int v = 0;
    decl int(++v) positive[10];
    v = 0;
    decl int(--v) negative[10];
    negative = positive;  // error: array assignment is invalid
    int p[] = positive;
    int n[] = negative;
    n = p;                // n and p both refer to positive[],
}                         // negative[]'s values don't change
```

5.2 Abstract classes, interfaces and deferred member functions

The following concepts are introduced informally in this section. They are used in subsequent sections, they are explained in detail in chapter §6, the following brief descriptions are adequate for now.

An `abstract class` is a class that is not allowed to be used to declare objects of its type, pointers to objects of its type are allowed, it is meant to be used as the base class for other classes, see §6.1. Inheritance from a class, abstract or not, is specified with an `inherit` declaration.

An `interface` is a collection of member functions, very similar to a class declaration, other interfaces and classes can choose to provide the interface, see §6.2. An interface can not declare non-static data members. The declaration that a class, or an interface, provides the functionality of an interface is specified with an `is` declaration. The use of `inherit` and `is` is not interchangeable, this makes clear at the declaration location the nature of the entity being inherited, i.e. a `class` (abstract or otherwise), or an `interface`, respectively. An interface can not be used used to declare objects of its type, pointers to objects of its type are allowed to be declared. A class or an interface that specifies another interface with an `is` declaration is said to *provide the interface*.

Interfaces and abstract classes are a specialized form of class declaration, with various restrictions for the purpose of preventing the complexity that arises from unconstrained multiple-inheritance as occurs in C++.

A *deferred member function* is a member function whose implementation is not provided, `defer;` is specified instead of the function's body, see §6.4. A class with deferred member functions must be declared as an `abstract class`. The deferred member function might be declared within it, or inherited by it from another abstract class, or the deferred member function being a member of the class because it was part of an interface that the class provides, directly, or indirectly by providing an interface. If all the deferred member functions were obtained by the class through inheritance or by providing an interface, and they are all actually implemented by the class, then the class doesn't have to be declared as an `abstract class`.

5.3 Destructor, the `deinit()` member function

Inheritance is presented fully in chapter §6. Inheritance is used informally in the following sections. Object deinitialization is provided by implementing the `deinit()` member function. All classes that don't explicitly inherit from another class inherit implicitly from `class void`, the signature for `deinit()` comes from it:

```
pub abstract class void {
    pub void deinit() defer;
}
```

The `deinit()` member function is special, if it is not implemented by the programmer, it is generated by the compiler automatically, and even if implemented by the programmer, some additional code might still be generated at the end of it by the compiler, see §5.12.

5.4 Destructor can not call non-static member functions

The destructor is not allowed to make use of non-static member functions, with the exception of single call to a `priv void` non-static member function which must be invoked as the first statement of the destructor. The `priv void` non-static member function can not reference explicitly `this` within its code, and can not call other non-static member functions with the exception of `priv` non-static member functions that follow these same restrictions. The relaxation of the restriction is to allow for the `deinit()` member function to have some of its code modularized into various non-static member functions that might be required for other destruction purposes. These restrictions also apply to `init_deinit()` and to `reinit_deinit()`, see §5.8.

The `promise(`*`expression`*`)` post-condition of the constructor is allowed to call member functions because the object is fully formed when the constructor returns, which is the time at which the `promise()` *`expression`* is evaluated. The `require(`*`expression`*`)` pre-condition *`expression`* of the destructor is allowed to invoke member functions because the object is still well formed prior to its destruction.

See §6.14 for additional technical details about these restrictions, and their relationship to inheritance and when pre and post-conditions are evaluated.

5.5 Brief introduction to namespaces

A namespace is an outermost declaration of a named scope within which other declarations can be made, its purpose is to reduce the number of names introduced into the global name space. Namespaces are open in the sense that declarations can be added to a namespace from different locations in a source code file and from multiple source code files with the `extend namespace`, see §8.7, syntax shown below. In the code that follows, in §5.6, various interfaces are declared within the `lang` namespace which is where language related declarations are located.

5.6 Default construction, `init_default()` static member function

Classes that support the ability of being initialized by default, i.e. without a value being specified, must provide the `defaultable()` interface and implement the `init_default()` static member function:

```
extend namespace lang {
    pub interface defaultable(genre void type) {
        pub static void init_default(type raw *to) defer;
    }
}
```

This interface is a generic interface, see §11, its member function's arguments are based on a type specified as an argument to the interface. The type of the `to` argument of `init_default()` is `type raw *`, is a pointer to the `raw` object. A pointer to `raw` memory is a pointer that refers to memory that has not yet been initialized, it does not refer to a fully formed object. Calling non static member functions through the `to` pointer is invalid. Section §5.13 describes pointers to raw memory.

5.7 Value classes, `init()` and `reinit()` and member functions

Objects of a type that implement the interface `initializable` are almost value-like in so far as initialization from other objects is concerned. For an object to be reinitializable (i.e. assignable), they also need to implement the `reinitializable` interface.

```
extend namespace lang {
    pub interface initializable(genre void type) {
        pub void init(type raw *to) defer; // init to from this
        pub void init_deinit(type raw *to){// redef to optimize
            this->init(to);
            this->deinit();
        }
    }
}
```

These interface are generic interfaces see §11, its member function's arguments are based on a type specified as an argument to the interface.

```
extend namespace lang {
    pub interface reinitializable(genre initializable type) {
        pub void reinit(type *to) defer;
        pub void reinit_deinit(type *to) { // redef to optimize
           if (this == to) return;
           this->reinit(to);
           this->deinit();
        }
    }
}
```

5.8 Optimization with `init_deinit()` and `reinit_deinit()`

The compiler invokes the compound member functions `init_deinit()` and

`reinit_deinit()` whenever possible instead of invoking `init()` followed by `deinit()` or `reinit()` followed by `deinit()`, respectively. The implementations of `init_deinit()` and `reinit_deinit()` are shown above. The programmer can redefine these member functions for optimization purposes, if required.

5.9 The `lang.value` interface

The interface, `lang.value`, is used by value-like classes:

```
extend namespace lang {
    pub interface value(genre void type) {
        pub is initializable(type);
        pub is reinitializable(type);
        pub is defaultable(type);
    }
}
```

Class `point`, a point in 2 dimensional space, is a value-like type:

```
class point(pub float x, pub float y) {
    pub is lang.value(point);
    pub void init(point raw *to) redef { to->x = x; to->y = y;}
    pub void reinit(point *to) redef { to->x = x; to->y = y; }
    pub static void init_default(point raw *to) redef {
        to->x = to->y = 0.0;
    }
}
```

As described in §6.6, the `redef` keyword must be used when an inherited member function is redefined, as shown above in the redefinitions of `init()`, `reinit()`, and `init_default()`.

5.10 Member functions specified by `lang.value`

The member functions specified by `lang.value` are summarized below for reference (after expanding in place the interfaces that `lang.value` is based on):

```
pub interface value(genre void type) {
    pub static void init_default(type raw *to) defer;
    pub void init(type raw *to) defer;
    pub void init_deinit(type raw *to) defer;
    pub void reinit(type *to) defer;
    pub void reinit_deinit(type *to) defer;
}
```

Note that classes get `deinit()` from `class void`, not from `lang.value`.

5.11 A `string` class example

The following sections implement a simple `string` class, it doesn't use generic programming for the type of its characters to support strings of either `char` or `unic` characters. See §1L.6 for a similar class, `str`, the COOGL library generic string.

```
class string(char cstr[]) {
    pub is lang.value(string);      // strings are values
    index size = cstr.max[0];
    if (size > 0 && !cstr[size - 1]) --size; // don't copy '\0'
    priv index base = 0;            // [base, base+len) is value
    priv index len = size;
    priv char buf[];
    if (size > 0) {
       buf.create(size);
       assert(buf.start);           // no error handling for now
       libc.memcpy(buf, cstr, size);
    }
    return;                                          // continued below
```

5.12 Object deinitialization: `deinit()`

The function `void deinit()` is the class destructor, `string`'s `deinit()` is:

```
    pub void deinit() redef { buf.destroy(); }    // continued below
```

Object deinitialization occurs, i.e. the `deinit()` function is invoked, when:

- A scope where a non-static object was declared as a local variable is exited. This includes objects passed by value as arguments to functions, which are deinitialized when the function returns.
- An object is a member of another object and that object is being deinitialized.
- The execution of an expression that includes objects returned as the values of functions has been fully evaluated, these temporary value objects are deinitialized after the full evaluation of the expression.
- Functions are classes, thus regular non class functions can also have have a `deinit()` member function. When a non class function, that has a `deinit()` member function returns, its `deinit()` member function is invoked. The object implied by the non class function's non-static member variables is destroyed at function return time. For more about the role of a non class function's `deinit()` see §5.23.
- Invoked explicitly, this usually only occurs when an object that was allocated from a memory heap is being destroyed, e.g. internally as part of the implementation of the `destroy()` member function.

- The destructor can also be invoked explicitly for a member object of a class. This is only valid from the destructor of the class whose member is being deinitialized, in that case, the compiler omits the compiler generated object destruction that it would otherwise generate as part of the destructor of a class. Compiler generated `deinit()` invocations are done in the reverse order of the members construction, this is an order determinable at compile time, i.e. an order that is not affected by the flow of control within the constructor. A constructor can not return leaving some of its members unconstructed, doing so causes a compilation error.
- If the object is a global or static variable, when the program terminates normally, i.e. through a `return` from `main()` or an invocation of `libc.exit()`.
- At module unload time, if the object is a global or a static variable, when the module that declared it is unloaded.

The compiler synthesizes the `deinit()` function if an implementation is not provided and if any data member of the `class` needs to be deinitialized. The synthesized `deinit()` deinitializes its members in the reverse order of the order in which they were declared. If the `deinit()` function is provided, but it doesn't explicitly deinitialize some of its non-static members, then the ones that were not explicitly deinitialized have their deinitialization synthesized, the synthesized deinitialization occurs after all user provided code in the `deinit()` function has been executed.

5.13 Pointer to `raw` memory

A pointer to raw memory, for example the `to` argument of `init_default()` (see §5.6) is a pointer to the `raw` memory of an object, prior to being initialized, it does not refer to a fully formed object. Calling non static member functions on a pointer to a `raw` objects is not allowed.

5.14 Some `string` operations

String operations shown below can be used to: obtain its length; trim `n` characters from the start or the end; find the first occurrence of a character, from the start or from a specified `start` index, or the last occurrence backwards from the end or from a specified `last` index; and relationally compare against another `string`. To efficiently support trimming, and later other operations such as appending and prepending, a subset of the `buf[]` array descriptor is described by [`base`, `base + len`), it specifies the current value of the `string`.

```
    pub index length() { return len; }                    // string
    pub index trim(index n) {
        if (n < len) return base += n, len -= n;
        return base = len = 0;
    }
    pub index trim_end(index n) {
        if (n < len) return len -= n;
        return base = len = 0;
    }
    pub int compare(string *other) { // this vs other <0, 0, >0
        size_t min = len < other->len ? len : other->len;
        int result = libc.memcmp(&buf[base],
                                 &other->buf[other->base], min);
        if (result != 0 || len == other->len) return result;
        return len < other->len ? -1 : 1;
    }
    pub index find(char c, index start = 0) {
        string(&buf[base : base + len], /*contruct on:*/ to);
        if (start >= len) return -1;
        if (start < 0) start = 0;
        for (index i = base + start; i < len; ++i)
            if (buf[i] == c) return i - base;
        return -1;
    }
    pub index find_last(char c, index last = len - 1) {
        if (last < 0) return -1;
        if (last >= len) last = len - 1;
        for (index i = base + last; i >= base; --i)
            if (buf[i] == c) return i - base;
        return -1;
    }                                                     // continued below
```

5.15 Initialization constructor: `init()`

The `init()` member function's purpose is to allow the value of an existing object, `this`, to be used to initialize an unconstructed object, whose address is provided in the `to` argument. The implementation of `init()` for the `string` class follows:

```
    pub void init(string raw *to) redef {                 // string
        string(&buf[base : base + len], /*contruct on:*/ to);
    }                                                     // continued below
```

The `init()` member function is invoked implicitly when:

- an existing object is used to initialize another object at declaration time, this includes the case of function argument initialization at function call time; or

an existing object is returned as the value of a function and it is used to initialize an-

other object, possibly a temporary object managed by the compiler.

The `string()` constructor is invoked explicitly by `init()` to do its work, note that an additional argument to `string()` is specified, the string declaration only has one argument, but two arguments are specified, the second one, `to`, is used to specify the memory that is to be used by `string()` to construct the object on.

The type of `to` is `string raw *`, `raw` means that the memory for the object has not been initialized, the object can't be used as an initialized object until all of its data members have been initialized, if a function needs to be invoked to aid in the initialization, then the argument of that function must also be a pointer to the `raw` type. Non static member functions can not be invoked on a `raw` object pointer.

To ensure safety, the compiler prevents the memory that `to` references from being treated as an initialized object, for example by preventing the pointer to be given to other functions through a non-`raw` pointer, unless it can prove that the object at that point has become fully initialized. Access to the object's non-static data members within `init()` is controlled, they can not be accessed unless the compiler can determine that they have already been initialized. Non-static data members can be explicitly initialized by invoking, `init()`, or `init_default()` on them. Similarly, members that are pointers or array descriptors can not be dereferenced or copied unless they have been initialized.

When an object's declaration has an initializer expression the class constructor invocation can not specify constructor arguments, for example:

```
void example() {
   decl string("hello") s;          // string() invoked
   string t = s;                    // s->init(&t) invoked
   decl string("wrong") e = s;      // error
}
```

5.16 Brief preview of strings of generic value types

Generic classes are presented in §11. Briefly, a generic class, interface, or function, has an argument list that consists of two sub-lists. The first sublist is a list of type arguments, i.e. arguments declared starting with the `genre` keyword; the second sublist is the traditional argument list of the class constructor or function.

A generic string class, for example `str` below, that allows the base character type to be specified when an object is declared, and that allows an object to be initialized from another object, must specify the generic type arguments when an object is initialized from another object, it must not specify non-generic constructor arguments (as shown in the incorrect declaration of `e` in `example()` below).

The first argument, `type`, of the generic `str` class is a type argument, it must implement the `lang.value` interface, i.e. it must be value-like:

```
class str(genre lang.value type, type val[]) { ... }
void example() {
    decl str(char, "hello") s;      // str() invoked
    decl str(char) t = s;           // s->init(&t) invoked
    decl str(char, "wrong") e = s;  // error
}
```

5.17 Object slicing along incorrect type boundaries is not allowed

Initialization of an object is allowed only if `init()` exists, and only from another object of the same exact type. Initialization from an object whose type inherits from the object's type is not allowed. If the type is not knowable at compile time, the compilation fails. There is no notion of object slicing along incorrect type boundaries in COOGL. Object slicing (for example in C++) occurs when an object whose type descends from another type is used to initialize, or to be assigned to, an object of an ancestor type. If such operations were allowed information would be lost, at a minimum, and even worse, the information that is not lost might be invalid.

In general, objects are not passed by value or returned as the value of functions, instead pointers to them are passed, preventing object slicing is not particularly burdensome. Most objects that are passed and returned by value are simple *value* objects or handle objects for which this restriction doesn't pose a problem.

5.18 Pseudo constructors

The `integer()` function, together with an implicit invocation of `init()`, can be used as a pseudo-constructor to construct a `string` from an `int`, as shown in the `example()` function below:

```
    pub static string integer(int i) {                    // string
        char m[sizeof(int) * 3], *p = m + sizeof(m);
        do
            *--p = i % 10;
        while (i /= 10);
        return string(p);
    }                                                     // continued below
```

Code that makes use of `init()`:

```
void example() {
    string("hello") s;              // string("hello") invoked
    string t = s;                   // init() invoked
    string u = string.integer(7); // init() might be invoked
}
```

When `integer(7)` is invoked, it returns as its value the object constructed by `string(p)`. Then `init()` is invoked on that object to perform the initialization of

`u`, the temporary object returned by `integer(7)` is then destroyed. That object construction and its immediate destruction is wasteful. The compiler is allowed, by the language definition, to remove temporary objects that are used to initialize another object and are then immediately destructed. A class whose objects are precious and should not be subject to this optimization should be declared `vital`, see §9.11.

When a function returns an object of a user defined class by value, the location where the object is to be placed is given as a hidden argument to the function. In this case the address of `u` is given to `integer()` which then uses that address as the `raw` memory onto which to construct the object constructed by the `return string(p)` statement. Thus, in `example()` above, `init()` is not actually invoked to initialize `u`.

5.19 Default construction

Default construction for `string`:

```
    pub static void init_default(string raw *to) redef { // string
        char empty[];
        string(empty, to);
    }                                                    // continued below
```

The declaration of `empty` uses `init_default()`:

```
string empty;          // init_default()
string("hi") hi;       // constructor, string(), invoked
string ciao = hi;      // init() invoked
```

5.20 Object reinitialization: `reinit()`

The function `void reinit(type *to)`, is invoked on an object when it is to be assigned to another object that has been previously initialized. Continuing with the `string` example:

```
    pub void reinit(string *to) redef {                  // string
        if (this == to) return;    // Assigning to itself.
        to->deinit();              // This code is incorrect
        init(to);                  // for assignment to itself.
    }                                                    // continued below
```

Assignment to another object is an operation on the source object, the argument to `reinit()` is the destination object, the object being assigned to. This makes the signature of `reinit()` similar to the signature of `init()`. An example use of `init()`:

```
void example() {
    string("hello") h; // initialized by: string()
    string("world") w; // initialized by: string()
    string t = h;      // initialized by: h->init(&t)
    t = w;             // reinitialized by: w->reinit(&t)
}
```

Assignment to an object is allowed only if `void reinit()` exists, and only if both objects are of the same exact type. If the type is not knowable at compile time, the compilation fails. There is no notion of object slicing in COOGL.

5.21 Optimizing assignment of returned values: `reinit_deinit()`

In the assignment to `pin`, below, `example()` passes the address of the raw memory for a temporary object where `number()` is to store the object returned by it. The assignment of that temporary object to `pin`, is immediately followed by the destruction of the temporary object:

```
string number(int n) {
    lit size_t N = sizeof(int) * 8 / 3 + 5;    // overestimated
    char buf[N], *p = &buf[N];
    uint u = n > 0 ? n : -n;
    *--p = 0;
    // not assert(p >= &buf[1]), 1 digit + 1 '-' (if n < 0)
    for (; assert(p >= &buf[2]), u > 0; u /= 10) {
        *--p = u % 10;
    if (n < 0) *--p = '-';
    return string(buf);
}
string pin;
void example(int n) { pin = number(n); }
```

The construction of the temporary object, its use as the source of the assignment to `pin`, and its immediate destruction is a source of overhead that can be minimized, if required, by implementing `reinit_deinit()`. The `reinit_deinit()` member function will be invoked, if present, when it would have invoked `from->reinit(to)` immediately followed by `from->deinit()`. For `string`, below, it is faster to take over the source value being assigned than to make a copy of it followed by the destruction of the source object:

```
    pub void reinit_deinit(string *to) redef {          // string
        to->base = base;
        to->len = len;
        to->buf = buf;
    }                                                   // continued below
```

5.22 Optimizing initialization from returned values: `init_deinit()`

In the initialization of `x`, below, `example()` passes the address of the raw memory for a temporary object where `string.integer(n)` is to store the object returned by it. The assignment of that temporary object to `x`, is immediately followed by the destruction of the temporary object:

```
void example(int n) { string key = numer(n); }
```

The construction of the temporary object, its use as the source of the assignment, and its immediate destruction is a source of overhead that can be minimized, if required, by implementing `reinit_deinit()`. The `reinit_deinit()` member function will be invoked, if present, when it would have invoked `from->reinit(to)` immediately followed by `from->deinit()`. For `string`, below, it is faster to take over the source value being assigned than, to make a copy of it followed by the destruction of the source object:

```
    pub void reinit_deinit(string *to) redef {          // string
        to->base = base;
        to->len = len;
        to->buf = buf;
    }                                                   // continued below
```

5.23 Regular function's `deinit()` and `retval`

Regular functions (i.e. non-class functions) can also have a deinitialization function, i.e. a `deinit()` member function, which is a convenient place for cleanup code common to various return paths, it is invoked when the function returns.

The `retval` keyword is a compiler managed local variable, accessible within the `deinit()` member function of a non-`void` regular function, and within a `promise()` contract of the function. It is a pointer to the value returned by the function. Its type depends on the type of the value returned by the function. For example:

```
error_t work() {
    pub void deinit() {
        // the type of retval here is: error_t *retval;
    }
}
```

Extending the `class stack` with a `trypush()` function:

```
extend class stack {
    pub bool trypush(priv int value) {
        priv int cnt = count();
        priv stack *stk = this;
        if (full()) return false;
        push(value);
        return true;
        priv void deinit() { assert(!stk->empty() &&
                                    (!*retval && stk->full() &&
                                     cnt == stk->count() ||
                                     value == stk->top() &&
                                     cnt + 1 == stk->count()));
        }
    }
}
```

The `retval` keyword is similar to the `this` keyword in the sense that it is known and managed by the compiler. The value of `retval` can not be changed, e.g. to make it point to something else, nor can its address be obtained through `&retval`. The `trypush()` function, below, pushes `value` if there is space on the `stack`. It returns `true` to indicate that the value was pushed, `false` otherwise.

The `deinit()` of `trypush()`, above, does various postcondition checks to test the operation. For `deinit()` to be able to reference `stk`, `cnt`, and `value`, they must be members of `trypush()`. For `deinit()` to reference the `stack` object that `trypush()` operates on it is saved in the `stk` member. Nested functions, including their `promise()` post conditions, can only access the members of their directly enclosing function.

The `retval` keyword refers to the address of the value being returned by the function, the value being returned should not be affected because that usually makes the code obscure and hard to follow, but the language doesn't mandate that it not be changed. The value returned might be an object, it is possible to further affect the object. Affecting the returned value is discouraged, it should only be done for sound systematic reasons, for example error injection testing.

5.24 Object arguments and return values

C allows `struct` and `union` variables to be: assigned, initialized, passed by value as arguments, and returned as the value of a function. The meaning of these operations is very simple, the underlying memory is copied, it makes the C type system more orthogonal. Copying structures through raw memory copies can make no logical sense, for example a structure that can be within a list with previous and next pointers as part of the structure.

Performing raw memory copies by default on objects of a user defined class type, is inappropriate. Object copying related to these operations is under programmer control, by implementing the `lang.value` interface, and providing the member functions `init()` and `reinit()`. If raw copying is appropriate, it can be implemented in those functions, but COOGL does not make that the default behavior.

5.25 Literal members

A literal member, i.e. a literal declared within a class as a member, is implicitly a static member. Literal members can not be redefined through `redef`.

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

6 - Abstract classes, interfaces, and inheritance

> *"The class concept ... is a remodeling of the record class concept proposed by Hoare. ... A prefix notation is introduced to define subclasses organized in a hierarchical tree structure. The members of a class are called objects. Objects belonging to the same class have similar data structures. The members of a subclass are compound objects, which have a prefix part and a main part. The prefix part of a compound object has a structure similar to objects belonging to some higher level class. It can itself be a compound object."*
>
> -- Ole-Johan Dahl and Kristen Nygaard

Inheritance declarations are member variable declarations with the modifier `inherit`, the name of the variable can be omitted. Named inheritance allows for name clash resolution when required. Accessibility modifiers dictate the set of classes that are aware of the inheritance. The `pub` and `priv` modifiers lead to fully public or completely private inheritance. Accessibility modifiers allow inheritance relationships to be visible to subsets of classes, a form of partial revelation.

An `interface` is a specialized class declaration that doesn't have non-static data members. A class can inherit from a single class, it can only have a single base class. A class can implement any number of interfaces.

6.1 Abstract classes and concrete classes

An *abstract class* is a class declared with `abstract class`, abstract class declarations can inherit from other classes. If a class contains deferred member functions, then it must be declared as an abstract class.

A *concrete class* is a class that is not an abstract class, a concrete class is a class that is declared with `class`, not with `abstract class`, furthermore all of the member functions of a concrete class must specify their code, none of them can be a deferred member function.

6.2 Interfaces

An *interface* is a specialized class that uses `interface` instead of `class`, interface declarations are not allowed to have non-static data members. Interfaces specify a set of operations that can be implemented by unrelated classes that provide the functionality specified by the interface. Usually all the member functions declared by an interface are deferred member functions, but they don't have to be.

Conventionally most interfaces have names that end in *able*. Interfaces usually convey an ability, something that a class that implements the interface is capable of doing, for example: allocatable, serializable, readwritable, seekable, initializable, reinitializable, etc.

Interfaces can have literal data members. Specifically, literal data members that are arguments to the `interface` function allow choosing of variations or customization of an implementation obtained through the interface at the time the use of the interface is specified. For example, see the arguments to `lib.creatable` for run-time array creation support in §13.8.

6.3 Single inheritance and multiple interfaces

A `class` can inherit from up to one class, whether concrete or abstract, and can implement any number of interfaces. An `interface` can only implement other interfaces, it can not inherit from a class. These restrictions make the language very simple and the object memory layout trivial, the complexity morass of multiple inheritance in C++ is avoided, while allowing objects to provide multiple unrelated interfaces while being part of an inheritance hierarchy. See §XXX about `preclass`, and prefix class inheritance, a limited form of multiple inheritance that is used to implement cross-cutting functionality across many types, used to implement various aspects of language safety and various polymorphic dispatch mechanisms.

Inheritance from a class is specified with the `inherit` modifier. Inheritance from an interface is specified with the `is` modifier. For example:

```
class polar(double xx, double yy) { // changes to polar, §4.6
    pub inherit point(xx, yy);      // point declared in §4.6
    pub is lib.creatable(polar);    // provides creatable APIs
    ... // rest unchanged
}
```

The class `polar` inherits from `point` and provides the `lib.creatable` interface, which allows `polar` objects to be allocated dynamically at run time:

```
int main() {
    point *p = polar.create(-1.0, -1.0);
    p->print();
}
```

The object created is of `polar` type even if its address is stored in a pointer to `point`. Polymorphic invocation of `print()` ensures that `polar`'s `print()` member function is invoked, instead of invoking `point`'s `print()`. The output is:

```
x=-1 y=-1 ro=1.414214 teta=-2.356194
```

6.4 The `defer` and `redef` function modifiers

Abstract classes and interfaces usually don't implement their member functions. Unimplemented member functions use the `defer` keyword followed by a semicolon, instead of the function body. The `defer` keyword only applies to member functions and member class declarations. Global functions and non-member nested functions can not defer their implementations.

An example of member function redefinition follows, `total_stack` inherits from `stack`, it keeps the total sum of the values stored on the stack:

```
class total_stack(size_t max, int *error) {
    pub inherit stack(max, error);
    pub large tot = 0;
    return;

    pub large total() { return tot; }
    pub void push(int v) redef {
        stack.push(v);
        tot += v;
    }
    pub int pop() redef {
        int v = stack.pop();
        tot -= v;
        return v;
    }
}
```

Type `total_stack` inherits from `stack`, thus `&s` is a valid argument to `pop_all()` even though `s` is of type `total_stack`:

```
void pop_all(stack *s) {
    while (!s->empty()) s->pop();
}
void work() {
    int error;
    total_stack(10, &error) s;
    assert(!error);
    s.push(1); s.push(2); s.push(3); s.push(4); s.pop();
    assert(s.total() == 6);
    pop_all(&s);
}
```

6.5 Single inheritance and multiple interfaces example

A larger example follows, it uses both interfaces and a class hierarchy as part of the design of the file system type independent layer in a UNIX-like operating system, it uses these interfaces: `node`, `dir`, `file`, `fifo`, `bdev`, `cdev`, `rdwr`, and `rdwrat`.

The `rdwr interface` specifies the notion of being able to be read and written sequentially:

```
interface rdwr {
    pub size_t read(byte buf[], size_t n) defer;
    pub size_t write(byte buf[], size_t n) defer;
}
```

The `rdwrat interface` specifies the notion of being able to be read and written sequentially, or read written at a specific file offset location:

```
interface rdwrat {
    pub is rdwr;
    pub size_t readat(byte buf[], size_t n, off_t off) defer;
    pub size_t writeat(byte buf[], size_t n, off_t off) defer;
}
```

The `node interface` has several deferred member functions:

```
interface node {
    pub enum flag_t { exec=1, write=2, read=4, trunc=8, ... };
    pub file *is_file() { return NIL; }
    pub dir  *is_dir()  { return NIL; }
    pub fifo *is_fifo() { return NIL; }
    pub bdev *is_bdev() { return NIL; }
    pub cdev *is_cdev() { return NIL; }
    pub err_t open(flag_t flag, cred_t *cred) defer;
    pub void  release() defer;
}
```

A member function, whether deferred or not, of an inherited class (or provided by an interface in an `is` declaration), can be redefined in a descendant class. To ensure that the programmer knows that the function is being redefined, a plain function redefinition causes a compilation error, the `redef` modifier must be used as part of the function declaration, as shown below for `is_dir()`:

```
interface dir {
    pub is node;
    pub dir *is_dir() redef { return this; }
    pub err_t remove(name_t *name) defer;
    pub file *create_file(name_t *name, cred_t *cred) defer;
    pub dir  *create_dir (name_t *name, cred_t *cred) defer;
    pub fifo *create_fifo(name_t *name, cred_t *cred) defer;
    pub node *lookup(name_t *name) defer;
}
```

A previously defined function might be redefined to be deferred in a descendant class, in which case both `redef` and `defer` must be used.

A `node` provides the notion of a file system entity such as a file, a directory, a named pipe (also known as a fifo), a block device, or a character device, each of which is specified by a specialized interface : `dir`, `file`, `fifo`, `bdev`, and `cdev`.

```
interface file {
    pub inherit node;
    pub file *is_file() redef { return this; }
    pub is rdwrat;
}
interface fifo {
    pub inherit node;
    pub fifo *is_fifo() redef { return this; }
    pub is rdwr;        // only sequentially read pr write
}
interface bdev {
    pub inherit node;
    pub bdev *is_bdev() redef { return this; }
    pub is rdwrat;
}
interface cdev {
    pub inherit node;
    pub cdev *is_cdev() redef { return this; }
    pub is rdwrat;
}
```

Operating systems include support for multiple file system types, for example, disk based file systems, network file systems, removable optical media file systems, flash file systems, etc. Each file system implementation would define a concrete class to represent its file system nodes, and would derive from it concrete classes to represent its directories, files, and so forth. For example a network file system, such as NFS, might define these classes:

```
class nfs_node {
    prot nfs_handle_t handle;
    prot nfs_fs_t *fs;
    ...
}
class nfs_file {
    pub inherit nfs_node;
    pub is file;
}class nfs_dir {
    pub inherit nfs_node;
    pub is dir;
}
class nfs_fifo {
    pub inherit nfs_node;
    pub is fifo;
}
```

From the perspective of the file system independent components of the operating system (high level file system system calls, program execution support, etc) any file system entity is accessed and manipulated through pointers to objects that provide certain interfaces (`node`, `file`, `dir`, etc.), without any knowledge of the underlying classes used by each file system type in its implementation.

6.6 Redefining static member functions

The `defer` and `redef` keywords serve the same role for static member functions.

6.7 Accessibility modifiers

There are three accessibility modifiers: `pub`, `priv`, and `prot`. They are used with classes, functions, and interfaces, described in the following sections. They are also used with namespaces and modules, see §8. Access to members declared `pub` is unrestricted. Members declared `priv` are only accessible by the members of the class or interface, `priv` is used to hide implementation details of the class or interface.

Members of a class or interface that are internal to it, but that must be accessible to classes that inherit from the class or that implement the interface are declared `prot`. For example assuming the class `stack`, from §4.2, with the `entries` and `sp` members changed from `priv` to `prot`, and the rest of the class unchanged, users of `stack` are unaffected by this change:

```
class stack(size_t max, int *error) promise(empty()) {
    prot int entries[];
    prot int *sp = entries.create(max);
    // ... rest unchanged ...
}
```

Class `opstack`, below, inherits from `stack`, it uses `entries` and `sp` to provide an `operate()` member function to provide fast access to at least 1 and at most 3 elements on the stack, it allows those top 3 stack entries to be accessed directly, it additionally allows a valid number of elements to be `pop()`'d from an `opstack` in a single operation. Assume that the performance of the `operate()` member function is critical and that access of arbitrary memory is to be avoided. To make `opstack` as fast as possible, its `operate()` member function is `inline`, see §10.3, which would allow its function pointer argument to be expanded in line as well, when possible.

The `opstack` implementation is simplified by allocating three extra entries in the stack and then adjusting the number of entries to the number that the user actually specified, thus ensuring that extra dummy memory is always addressable at `sp-1`, `sp-2`, and `sp-3`, even if the `opstack` is empty:

```
class opstack(size_t max, int *error) {
    pub inherit stack(max + 3, error);
    if (*error) return;
    entries[0] = entries[1] = entries[2] = 0;
    sp += 3;
    return;

    pub size_t count() redef { return stack.count() - 3; }
    pub bool empty() redef { count() == 0; }
    pub typedef size_t (*operation)(int cnt, int *first,
                                    int *second, int *third);
    pub void operate(operation op) inline {
        size_t cnt = count();
        size_t n = op(cnt, sp - 1, sp - 2, sp - 3);
        assert(n >= 0 && n <= cnt);
        sp -= n;
        assert(sp >= entries.start + 3 && sp <= entries.end);
    }
}
```

The `count()` and `empty()` non-static member functions were redefined to compensate for the extra 3 entries. The evaluation of the `promise(empty())` made by `class stack` only occurs after the construction of `class opstack` is complete. The `empty()` member function invoked in the `promise()` is the one redefined by `class opstack`, see §6.14.

6.8 Accessibility modifiers versus `inherit` and `is` declarations

Members inherited from the base class are visible outside of the derived class with the most restrictive access that results from combining the accessibility modifiers of the members of the base class and the accessibility modifier of the member through which the inheritance is expressed.

Similarly, members provided by an interface specified in an `is` declaration are visible outside of the entity that implements the interface, a class or another interface, with the most restrictive access that results from combining the accessibility modifiers of the members of the interface specified in the `is` declaration and the accessibility modifier of the `is` declaration.

The syntax: `pub !inherit` can be used to declare a static member function that is not to be inherited by a class that inherits from the class, see §11.11 for an example.

6.9 Member access aliases

The following declaration syntax, where *`.member2`* through *`.memberN`* are optional, indicates that *`name`* stands for the series of member accesses specified on the right hand side of the `=` operator.

```
alias name = member1.member2 … .memberN;
```

These member accesses must be strictly contained members within the subordinate classes, structures, or unions, no hidden memory references are allowed. It is valid for a *`memberX`* to be an array indexed by a constant expression as long as it is a compile time dimensioned array, not a pointer, nor an array descriptor, nor a variable length array, see §13. The `xyz` declaration is valid only if `x` and `z` are traditional C arrays:

```
alias xyz = x[0][3].y.z[7]
```

This construct is used to replace one of the uses of C `#define`, used to pretend that an inner structure or union member is an outer level member of an outer structure or union. Having `alias` in the language allows symbolic debuggers to receive cut and pasted source code, for example to print an expression, without the programmer having to do the `#define` expansion. The *`memberX`* fields themselves can also be defined through other `alias` definitions.

An `alias` declaration doesn't completely address the whole class of `#define` name substitution possibilities. If indirection through pointers and run time array indexing was supported, then those would be addressed as well. The rationale for these restrictions and the prohibition against indirection is to force the programmer to be more careful about what they are doing. Any requirements beyond the ones met by `alias` are not supported, it is best to force the programmer to do the data structure splitting job completely, instead of hiding the required indirection through a hidden indirection behind an alias declaration. Allowing indirection would lead to hidden costs behind what would otherwise seem like plain member accesses. It is a bad idea to have something in the language whose cost cannot be trivially understood by looking at its use. With the limitations imposed on `alias`, the cost of member references is always a compile time constant expression offset relative to a register, i.e. a cost that is no different than the cost of accessing a regular member.

The scope of the identifier declared by `alias` is the scope within which it is declared. The scope within which the name is introduced must be such that the series of member accesses for which it stands corresponds to existing members and sub-members accessible from the scope that declared the alias. If the scope is global then *`member1`* must be a global variable, not a member.

An `alias` declaration introduces a new name for a member, the access to the underlying member can be modified through an accessibility modifier, i.e. `pub`, `priv`, or `prot`. The `alias` accessibility modifier, can be used to allow access to members that the containing class can access, but that would otherwise not be accessible to code outside of the class.

An example of this is shown below. The `queue` class allows for insertion and removal from its head and tail through `insert_at_head()`, `remove_from_head()`, `insert_at_tail()` and `remove_from_tail()`, it also provides access to the entry at the head or tail of the `queue` through `head_value()` and `tail_value()`.

```
class queue(size_t max, int *error) require(max > 0)
                                    promise(empty()) {
    priv int entries[];
    entries.create(max);
    *error = entries ? 0 : libc.ENOMEM;
    priv size_t free = max;
    priv int *head = entries.end - 1, *tail = entries.start;
    return;

    pub void deinit() { entries.destroy(); }
    pub bool empty() { return free == entries.max[0]; }
```

```
    pub bool full() { return free == 0; }
    pub int head_value() require(!empty()) { return *head; }
    pub int tail_value() require(!empty()) { return *tail; }
    pub void insert_at_head(int v) require(!full()) {
        --free;
        if (++head >= entries.end) head = entries.start;
        *head = v;
    }
    pub int remove_from_head() require(!empty()) {
        ++free;
        int v = *head;
        if (head == entries.start) head = enries.end;
        --head;
        return v;
    }
    pub void insert_at_tail(int v) require(!full()) {
        --free;
        if (tail == entries.start) tail = entries.end;
        *--tail = v;
    }
    pub int remove_from_tail() require(!empty()) {
        ++free;
        int v = *tail++;
        if (tail == entries.end) tail = entries.start;
        return v;
    }
}
```

A `stack` class, functionally equivalent to the `stack` class from prior examples can be implemented with `alias` declarations, as shown below. Members of the `q` member, are selectively renamed and made public by `stack`. A drawback from this is that the interface that `stack` exposes is not easily understood, the `queue` interface has to be examined for that.

```
class stack(size_t max, int *error) promise(empty()) {
    priv queue(count, error) q;
    return;
    pub alias empty = q.empty;
    pub alias full  = q.full;
    pub alias top   = q.head_value;
    pub alias push  = q.insert_at_head;
    pub alias pop   = q.remove_from_head;
}
```

The interface extraction option of the compiler, described in §2, produces the following output for the `stack` interface, which is easy to understand:

```
class stack(size_t max, int *error) promise(empty()) {
    pub bool empty() { ... }
    pub bool full() { ... }
    pub void push(int v) { ... }
    pub int pop() { ... }
    pub int top() { ... }
}
```

6.10 Qualified accessibility modifier

An accessibility modifier can also be used in a class, function, or an interface to give access of a member to named classes, functions, or interfaces, a comma separated list of their names, within curly braces, that follows either the `pub` or `prot` accessibility modifiers is a *qualified accessibility modifier*. The use of a qualified accessibility modifiers are only allowed after a mandatory non-qualified accessibility modifier, which must be a more restrictive accessibility modifier than the qualified ones. For example:

```
class c {
    priv pub {a, b} int p;        // a and b see p as pub
    prot pub {a} int q;           // a sees q as pub
    priv pub {a} prot {b} int r; // a sees r as pub
                                  // b sees r as prot
    pub prot {a} int s;           // error: prot constraints pub
}
```

For example, the `stack` class, below, gives access to its private members: `entries` and `sp` to the `walk()` function:

```
class stack(size_t max, int *error) promise(empty()) {
    priv pub {walk} int entries[];
    entries.create(max);
    priv pub {walk} int *sp = entries.start;
    // ... rest unchanged ...
}
```

The `walk()` function can access the private members, as if they were public:

```
typedef void (*operation)(int v, ularge arg);
void walk(stack *s, operation function, ularge arg) {
     int *p = s->sp;
     int *base = s->entries;
     while (p > base)
         function(*--p, arg);
}
```

If access needs to be given to an unknown set of classes, access can be given to a dummy public `interface`, which is idiomatically called `intrusive`, the classes or functions that need access specify their intrusiveness with the `stack.intrusive` in-

terface, for example:

```
class stack(size_t max, int *error) promise(empty()) {
    pub interface intrusive {};
    priv pub {intrusive} int entries[];
    entries.alloc(max);
    priv pub {intrusive} int *sp = entries.start;
    // ... rest unchanged ...
}
```

The modified `walk()` function provides the `stack.intrusive`:

```
typedef void (*operation)(int v, ularge arg);
void walk(stack *s, operation function, ularge arg) {
     priv is stack.intrusive;
     int *p = s->sp;
     int *base = s->entries;
     while (p > base)
         function(*--p, arg);
}
```

A class that allows access to its members through this mechanism will be harder to maintain, but at least the classes, functions, or interfaces that have access to it are easy to identify by searching for use of `stack.intrusive`. The open ended access to some of the internal details of a class might be seen as bad design, but it is up to the programmer to decide to provide the access or not, it can not be forced by other classes without cooperation of the entity whose internals are being exposed.

The qualified accessibility modifier mechanism can be used with `inherit` or with `is` to make class inheritance or the support for an interface visible to some entities and not visible to others. These forms of constrained member, inheritance, and interface visibility are referred to as *partial revelation*.

6.11 Single inheritance example

The following subsections implement the `io` abstract class and several implementations of it. An `io` object is a form of sequential input output end point, implementations of the `io` interface provide:

- Sequential input from a file and sequential output to another file.
- Network input output over a network end point.
- Message queue input output.
- Shared memory based input output across processes.
- Input and output buffering of another `io` object.

Classes that derive from `io` implement the member functions `read()` and `write()` which are *deferred member functions* of `io`, a deferred member is an unim-

plemented member of a class. Syntactically, the `defer` keyword must be used to indicate that the implementation of the function is deferred.

```
abstract class io {
    return;
    pub err_t read(void *mem, size_t count,
                   size_t *nread) defer;
    pub err_t write(void *mem, size_t count,
                    size_t *nwritten) defer;
    pub err_t flush() defer;
    pub void deinit() defer;
}
```

The `bio` implementation of `io`, shown below, is a class meant to be used as a base class for other classes. It provides support for `read()` and `write()` buffering for other implementations of `io` to use.

The arguments to the `bio` class include a pair of possibly `NIL` `buf` pointers used for buffering. If buffering of reads or writes is required, the buffer is provided externally. It is not managed by the `bio` class because the entity itself that makes use of a `bio` class would have better knowledge about what memory should be used. For example, a memory mapped file, a shared memory segment, memory contiguously allocated to the `bio` object itself, etc. Inheritance of `bio` from `io`:

```
class bio(priv io *other, priv buf *readbuf,
          priv buf *writebuf) {
    pub inherit io;
    priv err_t readerr = 0;
    return;
```

```
    pub err_t write(void *mem, size_t count,
                    size_t *nwritten) redef {
        buf_t *bp = writebuf;
        if (bp) {
            if (count > bp->avail()) {
                err_t err = flush();
                if (err) {
                    *nwritten = 0;
                    return err;
                }
            }
            if (count <= bp->avail()) {
                bp->add(mem, count);
                *nwritten = count;
                return 0;
            }
        }
        // not buffered or it didn't fit after flushing
        return other->write(mem, count, nwritten);
    }                                                  // continued below
```

The `bio.write()` function, above, buffers the writes if its `writebuf` member is non-`NIL`. If the data to be written doesn't fit in the space available in the write buffer, the write buffer is flushed first. If the data still doesn't fit, the write buffer is bypassed and the write is performed directly, i.e. without buffering.

The `flush()` member function of `bio` follows:

```
    pub err_t flush() redef {                          // class bio
        buf *bp = writebuf;
        if (bp)
            while (bp->used() > 0) {
                size_t nwritten;
                err_t e = other->write(bp->base(), bp->used(),
                                       &nwritten);
                bp->buf_trim(nwritten);
                if (e || (e = other->flush())
                    return e;
            }
        return 0;
    }                                                  // continued below
```

The internal `flush()` from the `require` in `deinit()` is an extra safety net to avoid data loss when `flush()` was not invoked prior to the destructor invocation, the `assert()` on the error from this `flush()` is an additional safety net to ensure that those `flush()` errors are not ignored. Delayed read errors are ignored because conceptually they occurred for data that was never asked to be read. The `other` pointer to the underlying `io` object buffered by `bio`, is not deinitialized by `deinit()` it is

not it's responsibility.

```
    pub redef void deinit() {                                  // class bio
        if (readbuf) readbuf->deinit();
        if (writebuf) {
            err_t error = flush();
            assert(!error);
            writebuf->deinit();
        }
    }                                                          // continued below
```

The `bio.read()` member function, below:

```
    pub err_t read(void *mem, size_t count,                    // class bio
                   size_t *nread) redef {
        char *m = mem;
        err_t err;
        size_t rd;
        *nread = 0;
        buf *bp = readbuf;
        if (bp)
            for (;;) {
                size_t used = bp->used();
                if (used > count) used = count;
                if (used) {
                    bp->remove(m, used);
                    count -= used;
                    m += used;
                }
                *nread += used;
                if (count == 0) return 0;
                assert(bp->empty());
                if (readerr) return readerr;
                if (count > bp->capacity()) break;
                rd = 0;
                err = other->read(bp->base(),
                                  bp->capacity(), &rd);
                bp->setsize(rd);
                readerr = err;
            }
        // not buffered; or after emptying it, what is
        // leftover to read is bigger than its capacity
        rd = 0;
        err = other->read(m, count, &rd);
        *nread += rd;
        return readerr = err;
    }
}                                                              // class bio
```

The `bio.read()` function satisfies as much of the read as possible from what is buffered, if any, and if what remains to be read is larger than what can be read ahead in the read buffer, it is read directly into the user's memory bypassing the read buffer. If what remains is smaller than the read ahead buffer, the read ahead buffer is filled to its buffering capacity, and what remains to be moved to the user's buffer is extracted from it.

`Bio.read()` does read ahead if the `readbuf` is non-`NIL`, it is more complicated than `bio.write()` because it deals with partial buffer reads and delayed error reporting to the caller. While data remains to be read the error won't be returned. The error will only be returned when the read ahead buffer is drained. It also deals with non error partial reads that don't fill the read ahead buffer to its capacity.

The UNIX file descriptor based `io` implementation, `fdio`, performs input output on a pair of UNIX file descriptors:

```
class fdio(priv int rdfd, priv int wrfd) {
    pub inherit io;
    return;                                          // continued below
```

The `fdio.read()` function is not shown, it is similar to `fdio.write()` and `fdio.flush()` are:

```
    pub err_t write(void *mem, size_t count,          // class fdio
                    size_t size_t *nwritten) redef {
        size_t nw = unix.write(wrfd, mem, count);
        return nw >= 0 ? *nwritten = nw, 0 :
                         *nwritten = 0, errno;
    }
    pub void flush() redef { unix.fsync(wrfd); }
}
```

The `bfdio` derived class is a compound class that makes use of both `fdio` and `bio` to implement buffered UNIX file descriptor I/O. It simply compounds these other two while allowing its user to choose the underlying buffering memory.

```
class bfdio(int rdfd, int wrfd,
            buf_t *readbuf, buf_t *writebuf) {
    priv fdio(rdfd, wrfd) fdinout;
    pub inherit bio(&fdinout, readbuf, writebuf);
    return;
}
```

The `read()` and `write()` member functions that `bfdio` inherits from `bio` do all the work. A custom destructor is not required either, the destructor that is generated by the compiler is equivalent to the following one, destruction of members is generated in the reverse order of their construction:

```
    pub void deinit() redef {       // generated by compiler for class bfdio
        bio.deinit();
        fdinout.deinit();
    }
```

6.12 Pointers and inheritance

A pointer to an ancestor class can be assigned a pointer to a publicly derived class. A pointer to an interface can be assigned a pointer to a class or another interface that publicly provides the interface. Assuming the classes shown in section §6.11:

```
err_t copy(io *src, io *dest) {
    size_t nr, nw;
    err_t err;
    byte mem[1024];
    while (!(err = src->read(mem, sizeofex mem, &nr)))
        for (; nr > 0; nr -= nw)
            if (err = dest->write(mem, nr, &nw))
                return err;
    err_t ferr = dest->flush();
    return err ? err : ferr;
}
```

Derived class pointers are compatible with the base class pointer arguments in `copy()`, thus `s` and `d` both of `bfdio` type can be passed as if they were pointers to `io`:

```
err_t copy3into1(bfdio *s1, bfdio *s2, bfdio *s3, bfdio *d) {
    err_t e;
    if (e = copy(s1, d)) return e;
    if (e = copy(s2, d)) return e;
    return copy(s3, d);
}
```

6.13 Duplicate member names

COOGL has no support for any form of name overloading. A member inherited from a base class whose name is the same as the name of a member in the derived class causes a compilation error. A member of the base that is not accessible by the derived class, i.e. because it is `priv`, doesn't result in name collisions. The same occurs for members of an interface that a class or another interface provides through an `is` declaration.

Duplicate member names can be easily addressed by using named inheritance. Names that are not duplicate continue to be accessible directly, duplicate members are not directly accessible. An `alias` declaration can be used to make a duplicate name accessible with a different name. For example:

```
class base { pub int i, pub int j; }
class derived {
    pub int i;
    pub inherit base b;       // named inheritance
    pub alias bi = b.i;
    i = bi = 17;
}
int example() {
    derived d;
    return d.i + d.bi + d.j;
}
```

Even though named inheritance of `base` occurs in `derived`, the `j` member of derived is directly accessible, because it is not a subject of name collision.

Similar to the duplicate names that can occur with inheritance, providing one or more interfaces with or without inheritance can also cause name collisions between the accessible members of the interfaces themselves or the base class. Name collision can be addressed through named inheritance or by naming the interfaces provided in the `is` declarations, and the names disambiguated appropriately with alias declarations.

Repeated implementation of the same interface, with multiple `is` declarations is not allowed, irrespective of whether the repeated `is` declarations occur directly or indirectly through inheritance or through intermediate interfaces. This restriction simplifies the language semantics. An object either implements an interface, or not, but if it does there is no question about the single implementation of it, there is no need to choose between multiple equivalent interfaces.

6.14 Constructor and destructor restrictions and contracts

As explained in §4.9 and §5.4, non-static member functions can not be called from the constructor or from the destructor, other than a single call under very restricted circumstances.

The technical rationale for not allowing non static member functions to be invoked while an object is being constructed or destructed is that an object that is not fully formed is not an appropriate object to be operated upon by a non-static member function, particularly if the member function has been redefined, or could be redefined in the future, the amount of confusion and incorrect code that this can lead to is tremendous. Simple questions such as the type of the object while it is being partially constructed would require potentially different answers at different times, which is the case in more complex languages such as C++. The type of objects is fixed from the start of their outermost constructor until it is finally destructed, in COOGL the types of objects never change dynamically.

Furthermore, because of inheritance and redefinition of member functions, the post-conditions of a constructor can not be guaranteed until the outermost object has been constructed.

Similarly the pre-conditions of a base class destructor can not be guaranteed while the destructor of the descendant class has already begun to deconstruct the object, because its invariants might no longer hold, and its implementations and redefinitions might have rendered the base class out of its own invariants at this time.

A base class can not ensure its post-condition, its `promise()`, is met if the member functions it depends on have been redefined. Not allowing calls to member functions from the constructor makes sense. Allowing calls to member functions from `promise(`*`expression`*`)` makes sense only if the *`expression`* is verified at the end of the construction of the outermost object, i.e. after the outermost object is fully formed, which is the time when the `promise()` *`expression`* is evaluated, irrespective of which constructor defined it.

A base class can not ensure its pre-condition, its `require()`, is met if the member functions it depends on have been redefined. Not allowing calls to member functions from the destructor makes sense. Allowing calls to member functions from `require(`*`expression`*`)` makes sense only if the *`expression`* is verified prior to the start of the destruction of the outermost object, i.e. before the object is began to be deformed from its outermost perspective, which is the time when the `require()` *`expression`* is evaluated, irrespective of which destructor defined it.

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

7 - Extension, continuation, and other class topics

"How can one check a large routine in the sense of making sure that it is right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

-- Alan Turing, June 24th 1950

This chapter covers miscellaneous topics about classes. A class can be extended independently of its declaration through the `extend` keyword. The declaration of a class can be continued elsewhere, through the use of the `continue` keyword, for example to split the class code into multiple source code files. The `sizeofex` operator can not be used unless the compiler can determine at compile time its value. The memory layout of objects is chosen by the compiler, independent of the declaration order of its members (unless it is a `class struct`). Declarations can not hide symbols other than global symbols. The name lookup operator ^ looks up names within the scope of a function when it is used in an expression that is an argument to the function. Various aspects of class and array initializers are presented. Object oriented callbacks are supported by delegate functions.

7.1 Class extension: `extend` `class`

A class must be declared once in the set of source code files that make the compiled program. A class can be extended, through the `extend` syntax, for example, assuming the `stack` class shown in §4.2 a `popall()` member function can be added:

```
extend class stack {
    pub void popall() { while (!empty()) pop(); }
}
```

The declarations within an `extend` declaration are limited to member functions, `lit`, `enum` and `static` data members. The body of an `extend` declaration does not contain executable code, i.e. no additional constructor code can be added.

A class extension cannot add non-static data members. Static data and member

functions can be added but only if they don't redefine anything inherited by the class. For example adding `print()` to `int` is valid. Redefining the `write()` member function of an `io` object, from §6.11, is not.

7.2 Class declaration continuation: `continue class`

The declaration of a class can be continued elsewhere, usually in another source code file, through the use of the `continue class` syntax. The `continue class` declaration can not add executable code to the class constructor, nor can it add non-static data members to it, it is similar to an `extend class` declaration, but it is allowed to redefine member functions inherited by the class. For example the `class opstack` can have its declaration from §6.7 continued in a separate file where both `push()` and `pop()` are redefined to maintain a static count of each operation:

```
continue class opstack {
    pub static size_t pops = 0, pushes = 0;
    pub int pop() redef { ++pops; return stack.pop(); }
    pub void push(int v) redef { ++pushes; stack.push(v); }
}
```

A `continue class` declaration is pure syntactic sugar, a `continue class` declaration must be the only declaration within the file that contains it, unless it is located in the same source file that contains the class declaration. The file scope visible to the `continue class` declaration is the same file scope that is visible to the class declaration, thus from a compilation perspective this syntactic sugar could be implemented by appending the contents of the each `continue class` declaration to the end of their corresponding class declaration, prior to their closing curly brace.

The purposes for the continue class syntax are two, one to allow very large classes to have their code split into multiple files, and to support turning regular pointer declarations into smart pointer declarations, described in §Error: Reference source not found.

7.3 Class of pointers and array descriptors implicit declaration location

When a class is declared, the class of pointers to the class is considered to be implicitly declared in the same source file. Recursively, the class of pointers to pointers to the class, etc, is thus also considered to be declared in the same source file. The same occurs for arrays and array descriptors based on the class declaration, and the arbitrary compounding of their declarations, for example, pointers to arrays of pointers to array descriptors of a class, their implicit declaration location is the file where the class is declared.

7.4 Pointer arithmetic

Walking arrays passed as a pointer argument, instead of as an array descriptor, is not allowed, even though it is idiomatic in C. Inheritance makes such walking potentially incorrect because the type of the underlying object might be different than the static type of the pointer used to access the object. If the indexing or pointer arithmetic were to be correct it would require that the `sizeofex` of the objects (or an equivalent internal operation) used to perform pointer arithmetic and array indexing to produce the size of the dynamic type, not of of the static type.

Instead of adding complexity to the language, use of the `sizeofex` operator causes a compilation error when the compiler can not reliably determine, at compile time, the types involved. The same occurs with pointer arithmetic or indexing of arrays when the compiler can not guarantee that the operation is correct. Variable length arrays, and array descriptors, can be reliably walked with indexes, or pointers, without these problems, see §13 for more on this subject.

7.5 `sizeof` and `sizeofex` operators

The grammar of COOGL is context free, a COOGL program can be parsed without the aid of a symbol table. The grammar of C is not context free, in various places the parser can not determine if a name corresponds to a type or not, and the parsing can not progress without that determination. COOGL restricts the use of `sizeof` to be used with types, not with expressions. To determine the size of the result of an expression the `sizeofex` operator should be used. For code meant to used both as COOGL and as C code (i.e. CLEAN code) the `sizeofex` can be `#defined` to be `sizeof` when the code is compiled as C code. The `sizeofex` of an array with zero elements, the `sizeof` of an empty class, and the `sizeof(void)` are all zero.

7.6 Layout control of class objects: `class` `struct`

Layout control of the memory of objects is completely under the control of the compiler which can ensure that the object's data is organized in the most memory efficient order irrespective of the declaration order of its members. The compiler is able to commingle data from various members, for example by using the pad space available because of data alignment constraints within a member to store in that pad space the data of another member. This is something that can not be done with most programming languages without destroying the modularity of the code.

Sometimes layout control is required by the programmer a class can be declared with both keywords: `class struct`. See §Error: Reference source not found for another example:

```
class struct foo { pub char c; pub large l; pub short s; }
void test() { assert(sizeof(foo) == sizeof(large) * 3); }
```

The class will be laid out exactly in the order used by the programmer, any inheritance must be the first non-static data member of the `class struct` and arguments to the class constructor of a `class struct` can not be data members.

7.7 Only global declarations can be hidden

Function arguments, local variables, and members can not be hidden by declarations in nested scopes. Nested scopes allow the introduction of new variables as long as their names are not the same as the names of the function's arguments, its local variables, or its members. Only global declarations can be hidden by a declaration in a nested scope.

Hiding an existing declaration usually leads to confusion and bugs where it is assumed that a single variable exists instead of multiple variables at different times. The restrictions against hiding symbols ensures that most such mistakes are caught. The reason to allow the hiding of global variables is that the introduction of a global declaration should not cause unbounded amounts of unrelated code to have compilation errors. The number of global declarations is significantly reduced in COOGL programs when compared to C programs, their hiding is not a significant source of problems.

7.8 Name lookup relative to the scope of a function

Function argument expressions can use the `^` unary operator to indicate that an identifier should be interpreted relative to the scope of the function being invoked, instead of the scope of the function invoking it. For example:

```
int fileopen(byte *name, int flag, int mode = 0644) {
    pub enum { READ = 1, WRITE = 2, TRUNC = 4, CREAT = 8 };
    ...
}
void use() {
    int fd = fileopen("/tmp/foo", ^READ | ^WRITE);
    ...
}
```

The use of the `^` name lookup unary operator is only valid in function argument expressions, an identical expressions elsewhere results in a compilation error. Normal access rules apply, the name's accessibility is checked against the calling context.

Typically the function whose scope is searched by a `^name` reference in an expression is a member function, the `name` is usually not defined within the member function itself, but in the class that it is a member of. For example:

```
class file(priv byte *name) {
    priv int fd = -1;
    pub enum flags {READ = 1, WRITE = 2, TRUNC = 4, CREAT = 8};
    pub error_t open(int flag, int mode = 0644) { ... }
}
void use() {
    decl file("/tmp/foo") f;
    error_t e = f.open(^READ | ^WRITE));
}
```

7.9 Structure and array initializers

C style structure initialization is part of the COOGL language:

```
struct person {
    char *name;
    int age;
};
person j = {"Jill", 29};
person k = {.name = "Ken", .age = 31};
```

Traditional C array initialization, i.e. with values in curly brace delimited lists is supported. An alternative array initialization syntax, from the Plan 9 C language, allows specific array entries to be initialized by specifying an index within square brackets before the initializer. Both forms of array initialization can be used together, an explicit index specifier sets the base for the subsequent entries that don't include an index specifier. This is similar to what occurs for explicit and implicit enumeration values.

For example a typical character classification table and interfaces to use it:

```
class cis {
    priv lit ubyte L = 0x01;  // lower case
    priv lit ubyte U = 0x02;  // upper case
    priv lit ubyte D = 0x04;  // decimal digit
    priv lit ubyte O = 0x08;  // octal digit
    priv lit ubyte X = 0x10;  // hexadecimal digit
    priv lit ubyte P = 0x20;  // punctuation
    priv lit ubyte S = 0x40;  // space

    priv lit uint N = 128; // must be power of two
    priv lit uint MASK = N - 1;
```

```
    priv static ubyte map[N] = {   // Assumes ASCII
        ['A'] U|X, U|X, U|X, U|X, U|X, U|X,   //  6 = A-F
        ['G'] U, U, U, U, U, U, U, U, U, U,
              U, U, U, U, U, U, U, U, U, U,   // 20 = G-Z
        ['a'] L|X, L|X, L|X, L|X, L|X, L|X,   //  6 = a-f
        ['g'] L, L, L, L, L, L, L, L, L, L,
              L, L, L, L, L, L, L, L, L, L,   // 20 = g-z
        ['0'] D|X|O, D|X|O, D|X|O, D|X|O,
              D|X|O, D|X|O, D|X|O, D|X|O,     //  8 = 0-7
        ['8'] D|X, D|X,                       //  2 = 8-9
        [' '] S, ['\t'] S, ['\n'] S,
       ['\f'] S, ['\r'] S, ['\v'] S,          //  6
         [33] P, P, P, P, P, P, P, P,
              P, P, P, P, P, P, P,            // 15 = [33,47]
         [58] P, P, P, P, P, P, P,            //  7 = [58,64]
         [91] P, P, P, P, P, P,               //  6 = [91,96]
        [123] P, P, P, P,                     //  4 = [123,126]
    }
    priv static bool v(int c, ubyte m) inline {
        return map[MASK & c] & m;
    }                                         // continued below
```

Which allows for `cis.alpha()` style functions, similar to standard C library character classification functions (e.g. `isalpha()`):

```
    pub static bool alpha(int c) { return v(c, L|U); } // class cis
    pub static bool alnum(int c) { return v(c, L|U|D); }
    pub static bool digit(int c) { return v(c, D); }
    pub static bool xdigit(int c){ return v(c, X); }
    pub static bool odigit(int c){ return v(c, O); }
    pub static bool space(int c) { return v(c, S); }
    pub static bool punct(int c) { return v(c, P); }
    pub static bool print(int c) { return v(c, L|U|D|S|P); }
}
```

7.10 Delegate functions: `deleg`

The notion of delegates arises from the need to allow arbitrary code, together with a data context for it, to be invoked by unrelated code in a type safe manner, the invoking code could have been compiled completely separate from the called code, for example the calling code could be in a dynamically loadable module that was developed by a third party who had absolutely no knowledge about the type in question, e.g. the type in question might have been developed by others at a later time.

The `stack` class provides a member function, `iterate()`, below, that allows code to be passed to it such that the code can access the values stored on the `stack`, while restricting its knowledge and preventing direct access to the internal representation of

`stack`. For example to compute the sum of the values on the `stack`, an iterator class as shown in §4.14 could be used, alternatively a delegate function pointer argument to an `iterate()` function could be used:

```
extend class stack {
    pub void iterate(void work(int val) deleg) {
        for (int *p = sp; --p >= ent; ) work(*p);
    }
}
```

Calling code follows, a simpler version is shown further below.

```
void use() {
    int error;
    decl stack(100, &error) stk;
    assert(!error);
    fill(&stk);
    priv class sum {
        priv large total = 0;
        pub void add(int v) { total += v; }
        pub large result() { return total; }
    }
    sum s;
    stk.iterate(s.add);
    on ("sum = "; s.result(); "\n") print();
}
```

The `work` argument is a delegate function pointer. Delegate function pointers are implemented by a pair of values: a C function pointer and an object pointer. For the `s.add` delegate function pointer argument the delegate function is `add()` and the object pointer is `&s`.

To reduce code clutter in the caller, the notion that a function is a class that has its own members is used, resulting in this simpler idiomatic code:

```
void use() {
    int error;
    decl stack(100, &error) stk;
    assert(!error);
    fill(&stk);
    priv large total = 0;
    priv void add(int v) { total += v; }
    stk.iterate(add);
    on ("sum = "; total; "\n") print();
}
```

The object associated with the `work` delegate function pointer is the nameless object associated with `use()`, its members are `add()` and `total`. When the delegate function is invoked the object pointer is provided transparently by the delegate invocation mechanism. Because `add()` is a member function being passed as an argu-

ment within the scope of its class constructor, `use()` in this case, the pointer to the object in question is known implicitly and passed together with the function as the pair that makes the delegate function pointer value.

With the global compilation model, and in line code expansion performed by the compiler, no actual function call, nor does the `add()` function actually end up existing. The C code produced by the compiler is similar to the code shown below:

```
void use() {
    stack stk;
    large total = 0;
    { int *p = stk.sp; *end = stk.ent + stk.MAXENT;
      while (p < end) total += *p++; }
    char__pointer__print("sum = ");
    large__print(total);
    char__pointer__print("\n");
}
```

7.11 Other aspects of delegate function pointers

Regular function pointers and delegate function pointers are different. A function pointer is assigned the address of a function with a signature that matches the signature of the function pointer. A function pointer does not refer to any specific object. A delegate function pointer refers to an object and to a member function of the object. Comparisons between function pointers are well defined and have an easily understood meaning. Comparison between delegate function pointers could have several possible interpretations depending on which values are compared: function, object pointer, or both. For the sake of language simplicity, comparison between delegate function pointers is invalid. Assignment of `NIL`, and comparison between a delegate function pointer and `NIL` are also invalid.

The memory layout of delegate function pointers is defined by the language as a `class struct`, its mandatory layout is required to allow completely unrelated and separately compiled code to interact through them. It is also defined for the obscure circumstances under which actual access to its fields are required. The address of a delegate function pointer can be taken and accessed through an `unsafe` cast to a pointer to a `lang.delegfp`:

```
extend namespace lang {
    pub class struct delegfp {
        pub void (*function)(void *object);
        pub void *object;
    }
}
```

8 - Name spaces, modules, and initialization order

"Controlling complexity is the essence of computer programming."

-- Brian Kernighan

A collection of entities can be grouped into a collection of names with a `namespace` declaration. Modularity aspects related to independently developed binaries, that are loaded together at run time to form a single program, are specified by the language, shared libraries and modules loaded and unloaded at run time have language specified semantics. Complicated static member initialization is done idiomatically through the construction of global objects. Construction order of global objects is under programmer control.

8.1 Modules and name spaces in C

Some programming languages have mechanisms that allow the name space of identifiers to be partitioned in such a way that the likelihood of name clashes between unrelated components is significantly reduced. Name space partitioning together with information hiding are very important for large programs, particularly extensible programs whose development is not meant to end and are continuously adapted to new environments and requirements. These programs are usually developed by a large team of programmers, and sometimes independently by unrelated parties, their code bases brought together as part of a large system, either in source code form or in binary form. Extensible programs, such as operating systems, databases, transaction monitors, web servers, browsers, etc. all benefit from modules and name spaces.

The idiomatic way of avoiding name clashes in C, through naming conventions has been good enough for most programs, the identifier name space is partitioned by short prefixes or some other naming convention. For example the BSD UNIX operating system kernel uses these and many other prefixes: `ufs_`, `vm_`, `net_`, `eth_`, etc. The C89 standard reserved all names that start with `_X` where `X` is any uppercase letter, for example C99 added the `_Bool` type without concern for name clashes.

Given that C symbols declared `static` have only file scope visibility, i.e. are hidden from other source code files, making it the natural mechanism in C for module support. The C `static` symbol hiding feature is sometimes implemented by not placing `static` symbols in the symbol table of the compiled object file. As a conse-

quence, the link editor doesn't see the `static` symbols, they are not visible for link time resolution if referenced from other C source files. The C symbol table based implementation of modules is restrictive in that a function can not be expanded `inline` into a source code file if it refers to `static` symbols declared in another file. Such `inline` expansion would require that the `static` symbols be placed in the object file's symbol table in a special way to make the symbol globally unique to account for the possible global name collisions that could otherwise occur, and for the symbol references in the location where the inline expansion was to occur to be adjusted accordingly.

Constraining a module to a single source file is too restrictive, modules end up being made of multiple files, which leads to entities being accessible needlessly, usually leading to tighter coupling of the software and making it harder to ascertain what the actual interfaces to the module are. Some projects don't use `static` to hide module internals and use instead naming conventions as a way to specify symbols that are not meant to be used outside of a module. For example, a trailing underscore, or a trailing `_p`, or another symbol prefix. Sometimes different conventions are used depending on the nature of the symbol, private fields might use a trailing underscore, and private functions might have a different function prefix, for example `mod_` for public interfaces and `modp_` for private ones where `mod` is some abbreviation of the module name. Because of the need to use header files in C, separate header files can be used to hide the internals of the module and not expose them in the header file through which the module interface is exposed, sometimes partial declarations are exposed without exposing the full declaration, for example through a `typedef struct mod_s mod;` declaration where `mod` is used as part of the interface but `mod_s` is not exposed in the module interface header file, which works reasonably well but doesn't allow for the inline expansion of functions that reference fields of `mod` into code outside the module.

The fundamental problems with name spaces and module support in C is that there is no name space and no module support in the language, it is up to each project to come up with its own conventions to implement them, sometimes tools are implemented to verify that the conventions are not being violated. When code bases of unrelated projects, with different conventions, are brought together into a single project the multiplicity of conventions makes the aggregate project even more complex.

Some C dynamic linking environments allow for all symbols in a program to be hidden from other modules. In those environments the notion of module is a separately bound executable file, which could be a shared library (in UNIX or GNU/Linux or a DLL in Windows) meant to be loaded at program start time, or a module meant to be loaded and unloaded dynamically at run time, usually as a means of extending a program. For example: device drivers for an operating system, graphics drivers for a window system, compiled stored procedures for a data base, language extensions for an interpreter, authentication modules for programs that use an exten-

sible authentication infrastructure, etc. In some environments the symbols within these separately compiled C based modules are only visible to other modules if they are explicitly exported, for example, through export and import mechanisms that are outside of the language. These export / import mechanisms can be found in Microsoft Windows, IBM's AIX, the Linux kernel, and other systems.

8.2 Global declarations in C

Traditional C programs consist of global declarations with few if any nested or hidden declarations, other than local variables. In C information hiding is done through selected inclusion of header files and by using the `static` keyword, which makes functions and global variables inaccessible from other files. In some C implementations, the use of `static` for global variables and functions interferes with the ability to debug the code, their names are not placed in the symbol table, not even for debugging purposes, and are thus inaccessible to debuggers.

The legacy notion of C file based encapsulation is supported by COOGL with some differences described below. The following code shows the use of `static` to implement two C modules `random.c` and `use.c`, one provides a random number generator, the other uses it. The file `random.c` hides its implementation data by using `static`.

```
static int random_prev = 1;                    // C code in file random.c
void random_seed(int seed) { random_prev = seed; }
int random() {
    random_prev = random_prev * 168071 + 71111111;
    return random_prev & 0x7FFFffff;
}
```

File `use.c` attempts to access the `static` variable `random_prev`. When `random.c` and `use.c` are compiled and linked into a program the link step fails because the `random_prev` symbol reference is `use.c` can not be resolved.

```
extern int random_prev;                        // C code in file use.c
int main() { random_prev = 0; }
```

In C the only declarations globally visible across files are those that are part of the program at run time, i.e. functions and variables. C does not make `typedef`, `enum` or `struct` declarations in a C file visible to other source files, because C source files are compiled separately, any such declaration sharing is done through header files and `#include` preprocessing directives; thus `typedef`, `enum` and `struct` declarations are implicitly private when placed in a C source file, and selectively public when placed in a header file meant to be included via `#include`. Portions of header files can also be selectively hidden under the control of `#if` or `#ifdef` C preprocessor directives, which are mechanisms sometimes used for modularity purposes.

8.3 Modules and accessibility modifiers

A module is a set of source files compiled and linked together into an executable, or into a program binary meant to be loaded at run time. At the operating system level modules are contained in dynamic libraries on macOS, shared libraries (aka as dynamic shared objects) in UNIX and GNU/Linux, or DLL files in Windows. Modules can be loaded at program startup, or dynamically at run time.

Accessibility modifiers, when used in global declarations, have a relationship to source file boundaries and modules. Global declarations that don't have an accessibility modifier or that use `prot` are only accessible within the module, those that use `priv` are only accessible within the source file that contains them, and those that use `pub` are accessible by other modules.

Given that COOGL doesn't have header files, declarations other than variables or functions, e.g. `typedef`, `enum`, `class`, `struct`, etc. are visible by default in other files within the same module, they are implicitly `prot` by default. This behavior is different from C's behavior with respect to those declarations in a C source file, which makes them inaccessible to other source files. Access to `typedef`, `enum`, `class`, or `struct` declarations can be restricted to the source files that contains them by declaring them `priv`, or unrestricted so that they be can accessible to other modules by declaring them `pub`.

The use of `static` to support the C notion of file based symbol hiding is supported by COOGL, very similar, but better behavior is obtained by using `priv`. An important difference between `priv` and `static` in global declarations is that, `priv` functions and data declarations have their names adjusted in a simple manner, see §2S.11, to include the file name and module name within the mangled name, thus making the symbols accessible to debuggers. Another difference is that inline functions that access global `static` functions or data can not be expanded in line outside of the file that contains them, whereas inline functions that access global functions or data (declared `priv` or `prot`) can be expanded into any invocation location.

Global declarations can make use of qualified accessibility modifiers to allow access to a global declaration to specific classes, interfaces, functions, or namespace, independent of their source file locations.

8.4 Publicized and published declarations

Global declarations that are `pub` are said to have been *publicized* across modules. Global declarations that are not publicized but whose implementation details are used for the implementation of publicized entities, directly or indirectly, are said to be *publicized indirectly*. Declarations that are publicized directly or indirectly are said to have been *published*.

The internal details of a published declaration are represented and implemented in a well specified way that is standard across compilers, data layout rules are followed strictly and function calling conventions are also followed without exception, no intra-module optimizations are performed on them. The compiler is still free to create specialized versions of a function optimized for intra-module calls that can be optimized because of global knowledge of all the calls within the function. For example, if for all the calls within a module an argument to the function is always the same constant value, then the code within the function can be specialized to take advantage of that fact, possibly causing a series of expressions to become compile time constant expressions and possibly leading to a series of run-time tests to be eliminated.

The compiler compiles each module independently, globally, as if it were compiled all at once, even if internally it compiles files only when required, thus the compiler can determine what are the declarations that have been published. Entities that have not been published can be compiled as aggressively as the compiler is capable of. For example member functions for a class that has not been published, and that are not called within the module, and whose address is not stored in a function pointer or a delegate function pointer, can be removed from the module without harm, members of the class that are only accessed by deleted member functions can also be removed from the class, for example a `stack` that is just declared but never used, even though its constructor and destructor are not completely trivial, the compiler can determine that the work that it does has no side effect so the object can be removed. Classes declared `vital` never have objects of their type optimized this way.

Data within a `struct`, `union`, or `class struct` type are never optimized, i.e removed, even if they are not published because the entity presumably exists to communicate through its memory layout in a way that requires the layout to be honored.

8.5 Module specification

A module is the result of compiling and linking together a set of source code files, the desired name of the module is specified as an argument:

```
$ coogl file1.cog file2.cog file3.cog -o name
$ coogl name    # equivalent, name.spec lists the source files
$ cat name.spec
file1.cog
file2.cog
file3.cog
$
```

The result of the compilation and linking is an executable or a dynamically loadable program file named with the module name and possibly with an extension depending on the underlying operating system (e.g. `.exe` or `.dll` on Windows; or without an extension or with `.so` on UNIX/Linux). A module specification file is also produced

with the `.pubmod` extension. A directory is created with the name of the module followed by `-out`, inside of it there are 5 or more files for each source code file, their names are the same as the source code file name but with a different extension. There is a public specification file with `.pub` extension, a protected specification file with a `.prot` extension, a C source code file with a `.c` extension, a header file with a `.h` extension, and an object file e.g. a `.o` or `.obj` file, there can be an assembly source code file with a `.s` extension if machine level assembler instructions were requested with the `-S` compilation option.

The compilation of a module can specify module specification files for other modules (`.pubmod` files) as part of the source code file list, for example when the entities exposed publicly by the module are to be used by the module being compiled.

A module that has the `main()` function compiles into a module that can be invoked as a program, a module with the `main()` entry point is referred to a *program module*. Modules that are not a program module are meant to be loaded dynamically at run time, or automatically when another module that was built against it is loaded either at program startup time or when the other module is dynamically loaded.

Note that a module specification is extra-linguistic in the sense that it is not expressed within the source code of the module, it is expressed externally as an invocation of the compiler. Modules have no relationship with namespaces, modules and namespaces don't have to be in a one-to-one relationship with each other, but sometimes they are when required by the programmer, for example by having the C standard library in a module that also contains all of its source code within the `libc` namespace, and when there are no other namespaces declared within its source code.

Arguments to the compiler can be specified in the command line or in the `name.spec` file, the arguments have to be consistent across recompilations of the module, the file `name.args` is produced by the compiler to contain the arguments that were used in the prior compilation and it is used to decide if a full recompilation is required, or if incremental recompilation is appropriate.

A module might be compiled with various compilation arguments, for different instruction sets, various levels of optimization or debugging information, etc. To allow multiple such compilations to coexist with each other the `--target` *`target`* argument can be specified, the output files of the compilation are then: *`name-target`*, *`name-target`*`-out`, *`name-target`*`.pubmod`, and `name-`*`target`*`.args`.

8.6 Controlling access to class as type vs as constructor

Sometimes it is required that objects of a given class type not be allowed to be declared in a specific way, for example on the stack, globally, or as members of other classes, while still allowing objects to be manipulated through pointers to them. If the class is declared `priv`, then not even pointer variables that refer to the class type can

be declared. If the class is declared `priv` in the outer most scope, i.e. globally, then this restriction applies only to other source files, but it still allows declarations within the same source file to use it.

Use of a class as a type to declare objects or arrays of objects of that class, can be constrained independently from the use of the class to declare pointers to objects or array descriptors that refer to arrays of objects of that class. Objects and arrays of objects require that the constructor be invoked at declaration time. Pointers and array descriptors don't require that the constructor be invoked at declaration time.

An accessibility modifier can be placed immediately after the closing parenthesis of the class constructor argument list, or immediately after the class name for classes whose constructor doesn't have an argument list. These accessibility modifiers, when present, control the use of the class: as a generic argument type, and as a constructor when it needs to be invoked to construct an object, i.e. when an object or an array of objects is declared. The declarations of `xp` , `ip`, and `gp` are valid, the declarations of `x`, `i`, and `g` are not:

```
pub class xval priv { pub int xx = 0; }
pub class ival(pub int ii = 0) priv {}
pub class gen(genre void type) { pub type tt; }
int f(xval xp[], ival *ip, decl gen(class ival *) gp) {}
xval x[10];                     // error: xval not accessible
decl ival(0) i;                 // error: ival not accessible
pub gen(class ival) g;          // error: ival not accessible
```

8.7 Name spaces

A collection of related entities can be declared with a `namespace` declaration:

```
pub namespace libc {                          // standard C library
    pub struct FILE { /*...*/ };
    priv FILE filetab[3];
    pub int lit EOF = -1;
    pub int fgetc(FILE *fp) inline ...;
    pub int fputc(int c, FILE *fp) inline ...;
}
```

A `namespace` declares a named scope in which other entities can be declared, entities declared within a namespace must be declared with an accessibility modifier, they are global declarations as if they were declared in the global name space, they are not member declarations as if they were members of a class. A namespace declaration is not a class declaration, it doesn't contain constructor code, a namespace is not a type declaration, the name of the namespace can not be used as a type.

COOGL language related compile time and run time support declarations are all in the `lang` name space. The COOGL library is defined within the `lib` name space. The standard C library is defined within the `libc` name space, part of it shown

above.

```
pub namespace lang { ... }// COOGL compile and run time support
pub namespace lib { ... } // COOGL library
```

Declarations can be added to a name space, in a source code location that is separate from its declarations, with `extend`:

```
extend namespace libc {
    pub lit FILE *stdin  = &filetab[0];
    pub lit FILE *stdout = &filetab[1];
    pub lit FILE *stderr = &filetab[2];
    pub int getchar() inline return fgetc(stdin);
    pub int putchar(int c) inline return fputc(c, stdin);
}
```

The name of the namespace can be used to refer to entities declared within the namespace with the dot operator. For example, as shown below for `libc`:

```
int cat() {
    int c;
    while ((c = libc.getchar()) != libc.EOF)
        if (libc.putchar(c) == libc.EOF) return libc.EOF;
    return 0;
}
```

The `cat()` function, can `import` into its scope the `libc` namespace, allowing it to access entities declared within `libc` without the `libc.` prefix as shown below. The `import` statement must be in the outermost scope of a function, class, or interface, it can not be in a nested scope.

```
int cat() {
    import libc;
    int c;
    while ((c = getchar()) != EOF)
        if (putchar(c) == EOF) return EOF;
    return 0;
}
```

A `namespace` that has a very long name can be imported with a shorter name. Thus preventing the names from polluting the local name space while making their use less cumbersome. For example, assuming:

```
namespace C99_standard_library { ... };
```

The excessively long name can be shortened:

```
int cat() {
    import C99_standard_library c99;
    int c;
    while ((c = c99.getchar()) != c99.EOF)
        if (putchar(c) == c99.EOF) return c99.EOF;
    return 0;
}
```

Use of `import` in the global scope is allowed. Legacy C code, on a file by file basis can be adjusted, as COOGL code, to use the C library definitions as if they were a bunch of global symbols, for example:

```
import libc;
int cat() {
    int c;
    while ((c = getchar()) != EOF)
        if (putchar(c) == EOF) return EOF;
    return 0;
}
```

An `alias` declaration can be used to provide access to an individual symbol in the namespace without having to `import` the whole namespace.

```
int cat() {
    priv alias getchar = libc.getchar;
    priv alias EOF = libc.EOF;
    while ((c = getchar()) != EOF)
        if (putchar(c) == EOF) return EOF;
    return 0;
}
```

8.8 Modules and namespaces are independent concepts

Modules and namespace are unrelated facilities, a module might contain declarations of entities within multiple namespaces, and a namespace might be defined in one module and extended within other modules. Sometimes programmers choose to place all the declarations for a namespace within a single module, and that module is made to contain only declarations within that namespace.

8.9 Class initialization

A class might have some data structures that need to be initialized before objects of its type are constructed. Declaring such data structures as static members is all that is needed, i.e. their construction causes the class data structures to be appropriately initialized. If the initialization is particularly complex, to the point that it is impossible through individual object constructions, then an auxiliary class can be declared and its constructor made to invoke a static member function of the original class to do the

complicated initialization. Thus the construction of a static object of the auxiliary class triggers the initialization. The class `keyword`, below, uses the auxiliary class `hinit()` for its initialization which occurs when the object `dohinit` is constructed:

```
class keyword {
    return;
    priv static hash(str, int) h;
    priv static str("if") if_kw;
    priv static str("while") while_kw;
    // ... same for all COOGL keywords ...
    priv enum { IF, WHILE };
    priv static class hinit {
        h.insert(&if_kw, IF);
        h.insert(&while_kw, WHILE);
        // ... same for all COOGL keywords ...
    }
    priv static hinit dohinit;
}
```

8.10 Global construction order

Construction of global objects follows this order:

- Objects declared within classes are constructed in declaration order.
- Classes and namespace are sorted in a user specifiable class order, by default: `pragma`, `lang`, `libc`, and `lib`. Classes and namespace that are not in the user's list are added, sorted, to the end of the list, sorting uses the filename where the class is declared as the primary key and the class name as the secondary key. Note that the order of files is also specifiable by the user.
- Static objects are constructed according to the order of their call type in the order described above. All the static objects of a class type are constructed prior to the construction of the static objects of the next class in the order.
- Construction of global objects whose class is not in the class order proceed in file order. The order of the files is specifiable by the user.
- Unspecified files, if any, are sorted by their names, and follow the list of files specified by the user, if any.

Compilation fails if objects are used prior to their construction. A naming convention would be required to make name based sorting more usable, name based sorting is mostly for stability and determinism. The user specifiable lists are specified outside of the language syntax, the lists are specified in files as part of the command line to the compiler. To facilitate the maintenance of the lists, even though there are two logical lists described above, the lists themselves can be composed from multiple files which are appended in order to form the two logical lists.

9 - More about control flow and input output

"At one point, I took BCPL from Martin Richards at MIT and converted it into what I thought was a fairly straight translation, but it turned out to be a different language so I called it B, and then Dennis took it and added types and called it C."

--Ken Thompson

The COOGL control flow extensions to C are: function destructors, the `on` statement, and the `loop` statement which works in conjunction with `loop` member function that encapsulate iteration. `Goto` statements can not jump ahead of an object declaration that would still be in scope at the target of the jump. The syntax `return` *`expression;`* is valid within `void` functions. The value of `vital` functions must be explicitly used, objects of a vital class can not be the subject of optimizations, each and everyone of them must be constructed and the destructor invoked on it. Jump statements cause the destruction of objects that are no longer reachable. COOGL does not have nor does it need structured exception handling, because it does not have constructs such as operator overloading or conversion operators which are incapable of reporting errors other than by throwing exceptions.

9.1 Replacement of `goto out` idiom with `deinit()`

A function's `deinit()` is invoked at function `return` time, it can be used to replace uses of the `goto out` idiom. This idiom consists of one or more forward `goto out` statements where the `out` label is towards the end of the function. The code after the `out` label is where cleanup and resource releasing occurs for various paths that fall through it or that `goto` it. After the cleanup and release of resources the function returns, which is part of the idiom.

The `goto out` idiomatic code can be replaced with a `deinit()` function together with `return` statements replacing the `goto out` statements. The C example below, is rewritten in COOGL using `deinit()`, it is as efficient as the C code shown. The reason for that is the compiler's single copy function in line expansion ability which makes the compiled COOGL version equivalent to the C version.

An example of the `goto out` idiom, written in C and with `deinit()` in COOGL:

```
void func() {       /* C code */
    char buf[1024];
    char *m = buf;
    bool open = false;
    file_t f;
    int error;
    error = file_open(&f, "a");
    if (error)
        goto out;
    open = true;
    size_t len = file_size(f);
    if (len > sizeof(buf)) {
        char *x = malloc(len);
        if (!x) {
            error = ENOMEM;
            goto out;
        }
        m = x;
    }
    error = file_read(&f, m, sz);
    if (error)
        goto out;
    work(m, sz);
    if (invalid_condition) {
        error = EINVAL;
        goto out;
    }
    final_work(m, sz);

out:
    if (open) file_close(f);
    if (m != buf) free(m);
    file_deinit(&f);
    return error;
}
```

```
void func() {    // COOGL code
    priv char buf[1024];
    priv char *m = buf;
    priv bool open = false;
    priv file f;
    int error
    error = f.open("a");
    if (error)
        return error;
    open = true;
    size_t len = f.size();
    if (len > sizeof(buf)) {
        char *x = malloc(len);
        if (!x)
            return ENOMEM;
        m = x;
    }
    error = f.read(m, sz);
    if (error)
        return error;
    work(m, sz);
    if (invalid_condition)
        return EINVAL;
    final_work(m, sz);
    return error;

    priv void deinit() {
        if (open) f.close();
        if (m != buf) free(m);
    }
}
```

9.2 `on` statement

Input and output that is terse, type safe, and extensible, requires a syntactical notation of some kind. The C `printf()` and `scanf()` functions are not type safe, they require that the type of each argument be specified in the format string. The format string is interpreted at run time, errors encountered at run time include: incorrect output and invalid memory accesses. Even C compilers that support type checking of `printf()` and `scanf()` format strings, such as the GNU C compiler (gcc), are limited to the specific types and format options known by the compiler. Furthermore,

they can not type check format strings specified at run-time. The C `printf()` and `scanf()` functions are not extensible.

The `on` statement was added to COOGL to support extensible and type safe input output, but it has no knowledge about input output, and can be used for other purposes unrelated to input output. The `on` statement specifies, within parenthesis, a semicolon separated list of expressions, followed by a function invocation expression. Each expression in the list is evaluated in left to right order, after each expression is evaluated the function invocation expression is invoked, as a member function, on the object that is the result of the expression evaluation.

In this example `on` is used to implement `class point` input output, it assumes the input output support on fundamental types provided by the COOGL library, the reason why these member functions return `int` will become clear shortly when the value that `on` can produce is explained further below.

```
class point(priv int x, priv int y) {
    return;
    pub int print() return fprint(libc.stdout);
    pub int scan() return fscan(libc.stdin);
    pub int fprint(libc.FILE *f) {
        int n = on ("x="; x; " y="; y) fprint(f);
        return lang.on_int_count_result(n, 4);
    }
    pub int fscan(libc.FILE *f) {
        int n = on ("x="; x; " y="; y) fscan(f);
        return lang.on_int_count_result(n, 4);
    }
}
```

An example program that does input and output on `point` objects is:

```
int main() {
    point(1, 2) a;
    point(10, 11) b;
    on ("we have two points:\n";
        "  a is "; a; '\n';
        "  b is "; b; '\n';
        "using this form: x=4 y=5 x=14 y=15\n";
        "  enter new values for a and b: ") print();
    on (a; " "; b; '\n') scan();
    on ("new value of a: "; a; '\n';
        "new value of b: "; b; '\n') print();
}
```

For the examples above to work, the types `int`, `strlit(class const char)`, and `char` require `print()` and `scan()` member functions, COOGL allows types to be extended, to have member functions defined for them. A simple implementation of those members, using the C library, are shown further below.

The objects in the expression list of the on statement are of arbitrary types. The member function invocation follows after the parenthesized list of expressions. For the on statement to be valid, each expression in the list must have a member function whose name and signature matches the member invocation expression. The on syntax is superficially similar to the for(*expr1; expr2; expr3*) syntax in the way in which the list of expressions is specified, i.e. a semicolon separated list within parenthesis.

9.3 on expression

The on statement can be used to produce a value through which results of multiple operations are reported, the value can be used to initialize a variable being declared, or in an outermost assignment expression, or in a return expression, but not in a nested assignment expression. This restriction simplifies the language, it also prevents code from using multiple on statements in unboundedly complex expressions. When on is used in these circumstances it is called an *on expression*. Examples of on expressions producing a value follow:

```
int main() {
    point(1, 2) a;
    point(10, 11) b;
    int n = on ("we have two points:\n";
                "  a is "; a; '\n';
                "  b is "; b; '\n';
                "using this form: x=4 y=5 x=14 y=15\n";
                "  enter new values for a and b: ") print();
    assert(n >= 0 && n <= 9 || n == EOF);
    if (n != 9) libc.exit(1);

    n = on (a; " "; b; '\n') scan();
    assert(n >= 0 && n <= 4 || n == EOF);
    if (n != 4) libc.exit(2);

    n = on ("new value of a: "; a; '\n';
            "new value of b: "; b; '\n') print();
    assert(n >= 0 && n <= 6 || n == EOF);
    if (n != 6) libc.exit(3);
}
```

The member functions applied to the objects in an on statement all must have the same return value type. They must all be void, or if they are not, they must all return a signed integral type. The control flow aspects of the on statement are dictated by whether a value is returned or not, i.e. signed integral vs void, they are not affected by whether the value is used or not by the on statement. By definition the value is always used in an on expression.

If a value is returned by the member function, this means that an error might have occurred as the result of invoking the member function on one of the expressions, invoking the member functions in subsequent expressions would only add to the confusion, for example by producing confusing output or misreading values into the wrong variables, thus it is important to stop the processing when an error is reported.

The `on` syntax is a general purpose mechanism, it can not be tied intimately to the requirements of specific legacy C interfaces to the point that its generality is affected, but it is designed in such a way that the values they return can be produced by the `on` statement. For example, in C, `printf()` returns `EOF` or the number of bytes written, and `scanf()` returns `EOF` or the number of variables that were scanned successfully. For the purposes of this discussion, assume that `EOF` is a negative value, it is `-1` in every platform ever encountered by the author. Certainly every platform currently supported or intended to be supported by COOGL defines it as a negative number.

When a negative value is returned from a member function invocation it means that an error occurred, and processing must stop. If a value of zero is returned, it means that the operation was not performed, and processing must also stop. A positive value means that the operation was performed and processing should continue.

If a positive value is returned by the first member function invocation of the `on` statement or expression, that value and all subsequent positive values are accumulated, i.e. added, and the accumulated value is the value of the `on` statement. This implies that when at least one member function succeeded, the value of the `on` statement will never be negative, even if the processing of a subsequent member function call returned a negative value.

At this point a translation example will help the explanation of how an `on` that produces a values is translated, this code fragment:

```
n = on (a; " "; b; '\n') scan();
```

Is equivalent to this COOGL code:

```
{
    if ((n = a.scan()) > 0) {       // scanned something or EOF?
        int c;                      // same type as type of n
        if ((c = " ".scan()) > 0) {
            n += c;                  // accumulate scanned count
            if ((c = b.scan()) > 0) {
                n += c;
                if ((c = '\n'.scan()) > 0)
                    n += c;
            }
        }
    }
}
```

If the value returned by a `scan()` member function invocation is not positive, the

remaining `scan()` operations are skipped. If a negative value is returned by any of the `scan()` member function invocation, other than the first one, it doesn't affect the value of `n`, thus the number of elements processed will be returned if a partial number of elements were processed. The individual `scan()` operations could have returned the number of bytes read, instead of the number of elements processed, the generated code would accumulate the number of bytes read or the number of elements processed irrespective of the meaning of the value returned. The return value convention for `scan()`, `fscan()`, `print()`, and `fprint()` is based on the C `<stdio.h>` interfaces, they are meant to be compatible and similar to them because they operate on open files based on the `<stdio.h>` `FILE` type. They return:

- `EOF`, a negative value, indicates end of file or some other error condition, the `libc.feof()`, `libc.ferror()`, and `libc.clearerr()` functions can be used to determine what actually happened and how to proceed.
- `0`, value couldn't be scanned or printed, for example an `int` was to be scanned but the next characters to be read did not form an integer value, i.e. not one or more digits possibly preceded by a `+` or `-` sign.
- `1`, value was successfully scanned or printed.

Which results in a return value convention for an `on` expression, with one or more objects being `print()` or `scan()`, returning the number of elements processed, or `EOF` if an error occurred with the first element, thus proper error handling on these `on` expressions entails checking for the value being different than the number of expressions in the `on` expression, and if different, then doing whatever is appropriate, particularly interactive programs can re-prompt the user for valid input.

A helper function is provided to aid in producing the correct return value, its `result` argument is the value produced by an `on` expression used to implement the operation, `wanted` is the number that `result` should be equals to if all the member functions in the on expression did their work, see §9.2 for example uses of it.

```
extend namespace lang {
    pub int on_int_count_result(int result, int wanted)
                promise(result > 0 && result <= wanted) inline
        return result == wanted ? 1 : result >= 0 ? 0 : result;
}
```

The code shown above is for illustration purposes, the code generated might be similar to it, when the number of expressions in the `on` statement is small, but if it is large, a table of delegate function pointers is created and a run time helper function is invoked instead, which performs the work inside the first `if` with a `for` loop. The equivalent COOGL code would then be:

```
{
    typedef int (*scandfp)() deleg;
    scanfp scantab[4] = {&a.scan, &" ".scan,
                         &c = b.scan, &'\n'.scan};
    n = lang.on_array(int, scantab);
}
```

The helper function is similar to the function that follows, but it uses generic types instead of the specific type `int`, see §1 for the actual generic helper function which, also handles argument passing to the member functions.

```
extend namespce lang {
    int on_array(on_array.delegate a[]) require(a.max[0] > 0) {
        pub typedef int (*delegate)() deleg;
        int n = a[0]();
        if (n > 0)
            for (delegate *dp = a; ++dp < a.end; ) {
                int c = (*dp)();
                if (c <= 0) break;
                n += c;
            }
        return n;
    }
}
```

String literal `scan()` and `print()`, their type is `strlit(class const char)` :

```
extend class const char[] {
    // type of this is: const char(*this)[]
    pub int print() return fprint(libc.stdout);
    pub int scan() return fscan(libc.stdin);
    pub int fprint(libc.FILE *f) {
        return libc.fputs(f, *this) != EOF ? 1 : EOF;
    }
    pub int fscan(libc.FILE *f) {      // matches chars doesn't
        const char mem[] = *this;      // need to store them
        int c;
        for (const char *s = mem; s < mem.end; s++) {
            char e = *s;
            if ((c = libc.fgetc(f)) == e)
                continue;
            if (c == EOF) return EOF;
            libc.fungetc(f, c);        // unexpected character
            return 0;                  // didn't match
        }
        return 1;                      // matched
    }
}
```

Support for `int` `scan()` and `print()`:

```
extend class int {
    // type of this is: int *this
    pub int print() return fprint(libc.stdout);
    pub int scan() return fscan(libc.stdin);
    pub int fprint(libc.FILE *f) {
        char buf[64];
        size_t len = lib.inttostr(*this, buf);
        return libc.fwrite(buf, len, 1, f);
    }
    pub int fscan(libc.FILE *f) {
        libc.flockfile(f);
        libc.fskipspace(f);
        char buf[64], *p = buf, *last = buf.end - 1;
        bool first = true;
        *p++ = '+';  // implicit sign
        int v = 0, ret = 1;    // assume int will be scanned ok
        for (;;) {
            int c = libc.getc_unlocked(f);
            if (c == EOF) {
                if (p == buf) ret = EOF;
                break;
            }
            if (first) {
                first = false;      // + or - can only be first
                if (c == '-' || c == '+') {
                    buf[0] = c;     // save explicit sign
                    continue;
                }
            }
            if (!libc.isdigit(c)) {
                libc.ungetc_unlocked(c, f);
                break;
            }
            if (p >= last) ret = 0; // too big, skip all digits
            else *p++ = c;
        }
        if (ret == 1 && p >= &buf[2]) { // sign and >= 1 digits
            assert(p < last);
            *p = 0;
            errno_t error;
            if ([v, error] = lib.strtoint(buf)) ret = 0;
        }
        libc.funlockfile(f);
        *this = v;
        return ret;
    }
}
```

String literal scanning is used to do input format matching, it doesn't store the characters anywhere, it is used to do string literal character matching, for example for the `"x="` literal specified by `point.scan()` above.

This example uses the `fmt(4,2)` precision specification:

```
int main() {
    float f = 78;
    float c = ((f - 32) * 5) / 9;
    on ("temperature in Caracas: ";
        f; "(f) "; c.fmt(4,2); "(c)\n") print();
}
```

The native type `float` can be extended to support a `fmt()` member function used for printing with a precision specification, a set of auxiliary formatting classes are provided in the COOGL library. The invocation of `fmt()` causes the construction of an object of type `fmt`, then the `print()` member function is invoked on that object through the `on` statement.

A minimalist implementation of the `fmt` formatting class for the `float` type is shown below. It only supports `"%left.rightf"` `printf()` equivalent formatting, where left and right specify the number of digits left and right of the decimal period. A trivial implementation based on `libc` follows:

```
extend class float {
    pub class fmt(priv int left, priv int right) {
        priv float value = *this;
        pub int print() return fprint(libc.stdout);
        pub int fprint(FILE *f) {
            char buf[128];
            lib.floattostr(value, left, right, buf);
            return libc.fputs(buf, f) == EOF ? EOF : 1;
        }
    }
}
```

9.4 Arguments to `on` statement member function and `str` strings

The expressions in the argument list of the member function invoked as part of the `on` statement are evaluated for each of the expressions that control the `on` statement. In the example below, the argument expression `f ? f : stdout` of `fprint()` is logically evaluated for each invocation of `fprint()`. Compilers are capable of evaluating an expression once if it is safe to do so, as it is in this case:

```
void print7primes(FILE *f) {
    on (2; 3; 5; 7; 11; 13; 17) fprint(f ? f : stdout);
}
```

Errors can occur on both input and output, the underlying `FILE` streams become

disabled until the error status is fetched, through `stdout.ferror()` and cleared through `stdout.clearerr()`, COOGL library functions. This form of error handling is appropriate for input output and doesn't require any additional language support. By stopping further operations, output doesn't become garbled with missing information, and input is not confused by values, going to the wrong variables and input being consumed beyond the error condition.

The on statement can also be used for string manipulation for example by `sprint()` and `sscan()` member functions that have as their argument a string one which they append the formatted values they produce, or a string from which the data is consumed and after the data is consumed it is truncated so that the data consumed is no longer part of the string so that the next sscan() operation works on whatever comes next on the string. The language library provides a str type that supports these operations very efficiently. In this way on also subsumes the unsafe C variable argument functions `sprintf()` and `sscanf()` functionality.

9.5 Byte count vs operation count `on` value convention

The value convention of `print()`, `fprint()`, `scan()`, and `fscan()`, when used in an `on` expression is based on the number of operations that succeeded. It is the same as the C standard library `scanf()` return value convention, C's `printf()` return value convention is based on the number of bytes written, which is not very useful.

9.6 Compile time and run time enabled traces with `on`

Traditionally in C the preprocessor is used to define macros for the generation of traces. These macros usually have two versions, one that does nothing and another that does the tracing, the version of the macro chosen depending on some compile time configuration, maybe through `#ifdef` to choose between production and debug builds. The trace macros themselves, when they generate code, usually have a fast way of being disabled at run-time, usually through a test of some global state. The evaluation of arguments to the macro, including function invocations does not occur when the tracing is disabled at run-time.

Compile time infrastructure can be done in COOGL with the `on` statement and a dedicated `trace()` member function for that purpose, similar to how `print()` is done. Alternatively with a family of functions whose argument expressions are only evaluated when tracing is enabled, see the §9.7 section for more about that.

To cause the whole on statement to disappear, the first object in the `on` expression list can be a dummy type, a version of which that causes its `trace()` member function to always return `-1`, and that is inlined, allows the compiler to see that the `on` is to do no work after the first `trace()` on the first expression, and because that func-

tion does nothing, the whole `on` statement produces no code. For example:

```
pub namespace tr {
    pub class start { pub int trace() return -1; }
    pub class stop  { pub int trace() return -1; }
    pub start go;
    pub stop  end;
}
```

Two versions of the code above would exist, one as shown above, another with actual tracing support. Assuming that other types have `trace()` member functions added to them:

```
import tr;
int doit(int n, char name[]) {
    on (tr.go; "n="; n; "name="; name; tr.end) trace();
}
```

The version of `namespace tr` that supports tracing would have its `start` class test whether traces are enabled at run-time, and if not, return -1. The function `tr.go.trace()` would prepare whatever memory is required for the tracing, insert common information for all traces (global sequence number, cpu, thread id, time, function name, etc.), use of `trace()` on the data to be traced would then go into that memory, and `tr.end.trace()` would complete the trace record. If tracing occurs to per-thread tracing buffers then other than a global sequence number amount of shared state no interference would occur between the threads. If the traces are to be in a global trace buffer, then more complicated shared state management would be required.

9.7 Optional argument expression evaluation

The declaration of a function's argument list can have one of the comma argument declaration delimiters be a semi-colon instead of a comma. Invocation of such a function requires that the same argument delimiter be a semicolon. A function with such a declaration indicates to its user that the arguments after the semicolon might not be evaluated unless their values are required by the function. A valid degenerate form of this syntax allows for a starting semicolon prior to the argument declarations and correspondingly prior to the argument expressions in the function invocation.

An example use might be a function that has a fast path that only requires the arguments after the list for its slow path, so the cost of evaluation those expressions, which might involve function calls, etc. does not have to be incurred in the fast path.

An example is a collection of trace functions that are split into an inline function and an out of line function which is only invoked by the inline function when tracing is actually to occur, i.e. when run-time traces are not disabled, at compile time, or at run-time. The inlined fast path could generate no code whatsoever, for example in a

production build. Or whose fast path tests to see if tracing is enabled, and only in that case, allows the cost of the argument expression evaluation to occur when the inlined code calls the actual tracing function.

A sketch of a tracing example follows. It is important to emphasize that requiring the semicolon to be used both in the declaration of `trace()` and in its invocation makes the contract between it and its user explicit, the programmers that call `trace()` knows that `f()`, `g()`, and `h()` will not be invoked under some circumstances, in this case when traces are disabled, so their code should not depend on their invocation.

```
bool e = false;                    // run-time trace control; or
// lit bool e = false;             // compile-time trace control

void trace(bool enabled; uint a0, uint a1, uint a2) inline {
    if (enabled) tracex(a0, a1, a2);
}
void tracex(uint a0, uint a1, uint a2) {
    on ("a0="; a0; "a1="; a1; "a2="; a2; "\n") print();
}

// f(), g(), and h() are not invoked unless e is true
int main() { trace(e; f(), g(), h()); }
```

9.8 `Goto` target restrictions

COOGL ensures that objects declared on the run time stack are constructed when their declarations are executed and are destroyed when the scope in which they are declared is exited. A `goto` that reaches a statement in which an object is in scope is valid only if the object had already been constructed prior to the `goto` statement. If the `goto` causes the object construction to be skipped, a compilation error occurs.

Frequent use of declarations intermixed with statements reduces the ability of `goto` statements to be used. For example, a `goto` statement into the body of a `for` that has some variables declared before the label makes such a control transfer invalid.

The most common forms of `goto` are:

- The `goto out` idiom, and the similar `goto error` idiom, which goes to a label towards the end of the function to do cleanup and then `return`; and
- The `goto restart` idiom, where some operation that usually is performed by straight line code in the function might need to be restarted from almost the beginning because of some rare condition that is most easily handled by restarting the operation.

Though a simple iteration statement coupled with `continue` instead of the `goto restart` would be a normal rewrite of this second form of `goto`, the added level of

indentation and the visual misconception caused by the loop construct leads some systems programmers to prefer the `goto restart` idiom. This form of `goto` does not run into trouble related to jumping past object declarations because it is a backwards jump to the same or an outer scope.

The `goto out` idiom can run into the trouble of jumping past object declarations. The notion of destructors for functions (code that runs at function return time) alleviates most of the `goto out` needs without the visual clutter of a nested scope, it also tends to make such functions shorter and easier to follow. If the `goto out` is absolutely required for some code, it can still be used by:

- Moving all declarations before the `goto` statement so that there are no intervening declarations; or
- Enclosing intervening declarations in scopes that are exited before the target label of the `goto` statement;

If trouble still occurs, it is probably time to rewrite the code and simplify it, because most likely it has accumulated too many independent isolated changes over time and a rewrite would likely reduce its size, complexity, and bugs.

9.9 Use of `return` *`expression;`* in `void` functions

A `void` function can use the `return` *`expression;`* form of the `return` statement, for example:

```
void f() {...}
void g() {
    if (a()) return f();
    ... complicated code follows ...
}
```

This relaxation simplifies the language by allowing `void` functions to be thought of as returning a value, a `void` value. With that in mind, the `return` *`expression;`* statement in a `void` function can use any `void` *`expression`*. Without this simplification of the language the `g()` function above would have to be written this way:

```
void g() {
    if (a()) {
        f();
        return;
    }
    ... complicated code follows ...
}
```

9.10 Function values that are `vital`

A function that has the `vital` keyword immediately after the closing parenthesis of

its argument list indicates that the value can not be ignored, it must be used. Something must be done with it, otherwise a compilation error occurs.

For example, a function whose value is required as an argument to another function, to ensure that nested uses are correct. In `example()` below `disable_lock()` disables interrupts to a certain level and then acquires a lock, it returns the previous interrupt enabled level; `unlock_enable()` unlocks the lock and restores interrupts to the level specified in its argument, these are AIX kernel APIs, many operating system kernels have APIs similar to them:

```
int disable_lock(simple_lock_t *lock, int pri) vital { … }
void example() {
    int pri1 = disable_lock(&lock1, INTIODONE);
    some_work();                        // needs lock1
    int pri2 = disable_lock(&lock2, INTMAX);
    more_work();                        // needs lock1 and lock2
    unlock_enable(&lock1, pri2);
    final_work();                       // needs lock2
    unlock_enable(&lock2, pri1);
}
```

The re-enablement of interrupt levels must be done in the reverse order shown above, even though the locks are released in a different order. This locking idiom occurs when a hash or a hash chain is locked, then an object found through the hash is locked, then the hash lock is released and the object lock is retained while some object manipulation occurs.

The object oriented idiom of encapsulating the lock operations in an object and using the object destruction to cause the lock to be released, other than being obscure, would be incorrect for the example above.

9.11 Classes whose objects are `vital`

A class declared with `vital` indicates that objects of its type must all be created and individually destructed, no optimizations are allowed on them to reduce the number of short lived objects that are created, for example, if they are value like, `init_deinit()` and `reinit_deinit()` won't be invoked on them. For example:

```
class trace() vital { ... }
```

9.12 Jump statements cause object destruction

Jump statements, i.e. `break`, `continue`, `goto`, and `return` cause the destruction of any constructed in-scope objects that won't be in scope after the jump. Destruction occurs in the opposite order of their construction. Between the jump statements and their target an arbitrary amount of code can be executed, including infinite loops and even program normal or abnormal termination. Thus instead of the C assembler like

almost immediate control transfer, additional work could occur during their execution. This is a significant expansion of the behavior of these constructs when compared to their C behavior, it has both advantages and disadvantages, particularly because control transfer is no longer obvious. Other work that occurs under cover is the destructor invocations when objects are no longer in scope, which is not apparent when the scope is closed. In contrast, object construction is correlated to the object declaration because the declaration and construction occur at the same time.

The standard C library functions `setjmp()` and `longjmp()` can not be be used in COOGL they are unsafe. Even though they are useful for very drastic error handling, or to build higher level mechanisms for error handling and they can be of good use in the hands of a competent system designer they are best left out of the language.

9.13 Loop-member functions and the `loop` statement

To aid in the encapsulation of containers, (lists, trees, hashes, etc.), a class can implement a special kind of member function, a loop-member function, used to abstract the control flow and details related to iterating over the contents of the container. Assuming the `iterator` non-static member class of `stack`, from §4.14, the class `stack` can implement a loop-member function to iterate over the values of a stack, top to bottom. Note that the loop-member function, `values()`, produces a series of values, each value is produced by the `continue itor.get()` statement:

```
class stack(size_t max, int *error) promise(empty()) {
    // ... rest unchanged ...
    pub loop int values() {
        decl this->iterator itor;
        while (!itor.end())
            continue itor.get(); // produce int values
    }
}
```

Use of the loop-member function `values()` is shown below:

```
int average(stack *s) {
    large total = 0;
    int count = stack->count();
    if (!count) return 0;
    loop (int v = stack->values()) total += v;
    return cast(int) (total / count);
}
```

There are two forms of the loop control flow statement, a very simple one, shown above. A second `loop` statement allows the 3 `for` statement optional expressions to follow, separated by semicolon, the `loop` control declaration:

```
void add_top_n(stack *s, size_t n) {
    int v, sum = 0;
    loop (int v = stack->values(); size_t i = 0; i < n; i++)
        sum += v;
    return sum;
}
```

The controlling expression of the loop statement can also be an assignment, it doesn't have to be a declaration, as shown in `add_top_n()` above. The result of compiling `show_top_n()` is code similar to the code shown below:

```
int add_top_n(stack *s, size_t n) {
    int v, sum = 0;
  { decl this->iterator itor;
    for (size_t i = 0; !itor.end() && i < n; i++) {
        v = itor.get();
        sum += v;
    } }
    return sum;
}
```

The current version of the compiler, requires that the loop-member function produce its values from within an iterative control flow statement, either a `for` or a `while` statement, and not from a `do while` or another `loop` statement. An additional restriction is that the values be produced from within a single location within the iterative control flow statement, and not through recursive calls to itself, or other functions. A future versions of the compiler could support multiple value producing locations relatively easily. It is unlikely that a future version of the compiler will support recursively producing the values, it would require a co-routine like environment with its own run-time stack to be preserved while the loop-member function was active, i.e. while the loop-member function was being used to iterate on the container, and the disposal of it if the `loop` statement terminated prematurely through a `break`, a `goto`, or a `return`.

Its invalid to invoke a loop-member function other than from the `loop` controlling declaration. The restrictions against recursion could be removed, but they don't seem worthwhile, it would require allowing loop-member functions to be invoked from places other than loop-member functions, which would be meaningless if executed from outside a loop-member function call-chain.

Recursive algorithms can be made non-recursive, with some extra effort, and even though usually the code is not as easily comprehended, non-recursive algorithms can be used to work-around this restriction. Generic high performance containers are usually very well written code, and the extra effort to remove recursion usually pays off in higher performance, thus this restriction is not as bad as it might seem.

If an inherently recursive algorithm needs to be used by the loop-member function,

it can arrange its own co-routine based implementation, it might also benefit (in run-time performance) if the underlying iterator implementation uses a buffer that is managed by the loop-member function, emptied by it, and replenished with multiple values (recursively) each time its emptied, thus reducing the costs of co-routine switching, increasing instruction cache locality, and benefitting from the inlining of the `itor.get()` and of the loop-member function into the loop control flow statement.

9.14 No structured exception handling

COOGL does not have structured exception handling mechanisms, it doesn't need them because it doesn't have other mechanisms, such as operator overloading or conversion operators, which in other languages make exception handling a requirement. The COOGL design excluded structured exception handling support because of the flow control complexity morass that it engenders. The language complexity that results from language based exception handling is very high, particularly for a feature to be used only to handle *exceptional* conditions.

Structured exception handling, as a programming language facility, is a language design mistake, at least in COOGL. The fundamental reason is that once exceptions are thrown freely by a multitude of code, libraries, and even by language facilities themselves, no code can be written without worrying about what exceptions might be thrown by functions down the call chain that ought to require continued consideration in the current code.

Reliable systems can easily be written with traditional error handling paths, out of line exception handling functions, and consistent use of return codes. Well designed and well constructed systems might have one or more layers that represent failure domains where errors are handled through out of line execution. Those systems are usually built with custom allocators, lock stacks, and `setjmp()` `longjmp()` like facilities. They are much easier to build, understand and maintain because it is not the preoccupation of every function, and sometimes every block of code, to worry about these exceptional circumstances. By definition, exception handling code is exceptional, and its correctness is correlated with its actual testing, which implies that only a sideband orthogonal mechanism used between a few layers is the best way to organize such code. The alternative of having exception handling code spread throughout the whole code base, and making it the source of concern for every line of code written, is impossible to maintain, it is, after all, a form of a non local `goto`, but even worst because the target of these non-local `goto`, the label is determined at run-time, with the ever present possibility in some cases that there might not be a target for the exception being thrown.

Providing language support for just a few exception handling layers is nothing more than to add a feature that will be hardly used, so it shouldn't be in the language. Providing it so that a moderate number of exception handling paths be written, thus

possibly needing it in the language, is nothing more than an invitation for an untested morass of paths, or even worse, code that actually actively uses these glorified non-local `goto` statements under normal product operation. In some languages, such as Java, its standard library interfaces include exceptions being thrown as a means of reporting errors, for example when a file open fails, which is not something exceptional, it is something that should be expected. Structured exception handling is similar to allowing a drunken pilot fly an airplane full of passengers, a very bad idea.

A system whose error handling would have been based on `setjmp()` and `longjmp()` together with lock stacks and cleanup functions can still be implemented by performing the cleanup work in the exception handler and coordinating with a control thread to terminate the thread that caused the exception. Alternatively, the thread can move its run-time stack to a different stack and start over from there after disposing of its prior stack.

10 - Operators, expressions, keywords, and behavior

"At one point, I took BCPL from Martin Richards at MIT and converted it into what I thought was a fairly straight translation, but it turned out to be a different language so I called it B, and then Dennis took it and added types and called it C."

"B is a computer language directly descendant from BCPL. B is good for recursive, non-numeric, machine independent applications, such as system and language work. B, compared to BCPL, is syntactically rich in expressions and syntactically poor in statements."

--Ken Thompson

COOGL inherits C's operators, their behaviors are identical. Some operators, when used with a few other operators must parenthesize their sub-expressions. Additional operators added by COOGL: symbol lookup in function's scope; absolute symbol referencing; fine grain function inline control; checked operators that perform arithmetic of signed or unsigned integers, and indicate if the operation overflowed or involved a division by zero.

COOGL doesn't have undefined behavior, wherever the C11 standard states undefined behavior, COOGL code behaves in a documented manner characteristic of the environment where the compiled code runs.

10.1 Parenthesis requirement in certain error prone expressions

Parenthesis are mandatory in certain expressions to prevent certain programming errors. The operators and their precedence levels are shown in the following page. The precedence choices in C for the binary bitwise operators (groups 8, 9 and 10) and the shift operators (group 5) is a common source of programming errors. Their uses without parenthesis together with certain other operators is usually contrary to what some programmers might have intended. Those choices are the result of C's evolution from B, and B from BCPL, and the later incorporation of the `&&` and `||` operators into C, replacing `&` and `|` as the natural logical operators.

The following additional rules apply, they, mandate the use of parenthesis for the troublesome cases:

- Uses of a binary bitwise operators: `&`, `|`, or `^` (groups: 8, 9 and 10) together with operators: `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `<`, `<=`, `>`, `>=` , `==`, or `!=` (groups: 3, 4, 5, 6 and 7) must have parenthesis. The binary bitwise operators have a very low precedence, out of place between relational operators (groups 6 and 7) and logical operators (groups 11 and 12).

Compilation Error	Meaning in C	Programmer Intent
`i & j + 1`	`i & (j + 1)`	`(i & j) + 1`
`i & j == k`	`i & (j == k)`	`(i & j) == k`
`i | j * 3`	`i | (j * 3)`	`(i | j) * 3`
`i & ~1 < k`	`i & (~1 < k)`	`(i & ~1) < k`

  Historically the binary bitwise-and and bitwise-or operators, `&` and `|`, were used also for logical operations in the ancestors of the C language (B, BCPL and CPL). Those languages didn't have separate logical-and and logical-or operators, i.e. `&&` and `||`, because the relational and comparison operators produce a true or false value in those languages, 1 or 0 in the languages that lack a boolean type, then expressions such as these were common and appropriate, the precedence of `&` and `|` didn't cause trouble in those languages, just like this operation doesn't cause trouble in C, but this kind of use is not idiomatic in C:

```
if (min <= n & n <= max)
```

  A problem with C is that C's historical lack of boolean types, and conversions and strong type checking, causes code such as this to be silently wrong:

```
/* C code, some parentheses are needed to be COOGL code */
int a = 2, b = 1;           /* non-zero means true */
assert(a & b);              /* wrong! (2 & 1) is zero */
assert(a && b);             /* right */
if (a & b > 0) go();        /* wrong! (2 & 1) is zero */
if (a && b > 0) go();       /* right */
if (a != 0 & b > 0) go();   /* right, not idiomatic */
if (a && b > 0) go();       /* right, idiomatic */
```

  In K&R C no compilation errors occur when a `&` (bitwise and) operation is performed between an integer and a relational expression, e.g. the first `if` above, which is incorrect because even when both operands are non-zero, the result of a `&` (bitwise and) might still be zero. Thus instead of salvaging in COOGL these last vestiges of CPL and BCPL present in C, including the correct but non idiomatic uses in the 3rd `if` above, it is best to prevent these misuses.

- Uses of the binary bitwise-exclusive-or operator: `^` (group 9) together with

one of the other binary bitwise operators: `&` and `|` (groups 8 and 10) must have parenthesis. The `^` is a more complicated operation than `|` (bitwise-or) and `&` (bitwise-and). The `^` is defined in terms of `&`, `|`, and `~` (bitwise-not) as:

```
(a ^ b) == ((~a & b) | (a & ~b))
```

Having `^` precedence between the more fundamental operators of `|` and `&` is counter intuitive, having `^` with higher precedence would make sense from the mathematical perspective that more complicated operators defined in terms of simpler ones have higher precedence than the ones on which their definition is based. For example, multiplication is more complicated than addition, multiplication is defined as repeated addition, multiplication has higher precedence. Thus `^` should have had higher binding than both `|` and `&`. But because operator precedence cannot be changed in COOGL and continue to aspire to be an evolution of C, mandating parenthesis in this and the various other cases that tend to cause confusion among some C programmers, is a simple improvement that prevents programming errors of this nature.

- Uses of `<<` or `>>` (group 5) together with `*`, `/`, `%`, `+`, or `-` (groups 3 and 4) must have parenthesis. For example: `i << j + k` produces a compilation error. The C language indicates that this expression means `i << (j + k)` instead of `(i << j) + k`. Considering `i * j + k` means `(i * j) + k`, in C and in math, and that `p << k` means $p * 2^k$ i.e. that the shift left operator is a multiplication by a power of two in disguise, then the precedence of `<<` lower than `+` or `-` is mathematically counter intuitive. The same argument applies to right shift which is a division by a power of two:

Compilation Error	Meaning in C	Programmer Intent
`i << j + k`	`i << (j + k)`	`(i << j) + k`
`i + j << k`	`(i + j) << k`	`i + (j << k)`
`p - q << r + s`	`(p - q) << (r + s)`	`p - (q << r) + s`
`p * q >> r / s`	`(p * q) >> (r / s)`	`p * (q << r) / s`

Many code sequences don't need any extra parenthesis, for example in:

```
if (x & 1 && b == 0) go();
```

the C precedence already does what is expected, i.e.:

```
if ((x & 1) && (b == 0)) go();
```

10.2 Member lookup operator `^`

The `^` member access operator is a unary operator (i.e. `^member`) that is only valid in expression arguments within function invocations. The member is looked up within the name space of the function being invoked, instead of the name space of the

calling function. Examples are shown in §7.8.

10.3 Fine grained function inline control

Section §Error: Reference source not found describes the `func()inline` form of the function call operator. The use of `inline` is a syntactical form where the keyword enhances the expression syntax, a kind of modifier used at function invocation time. From a language perspective `()inline` is another form of the function call operator.

10.4 Checked arithmetic operators

Arithmetic operators that indicate if the operation overflowed are: `?++`, `?--`. `?-` (unary checked arithmetic negative), `?+`, `?-`, and `?*` and the assignment-operation operators: `?+=`, `?-=`, and `?*=`. These operators can only be used with signed or unsigned integer types, to efficiently determine if the operation resulted in an overflow. These operators are particularly important for secure coding, where overflow detection is very important but awkward, expensive, and error prone to program in C. Overflow detection is important to ensure that large value computations don't wrap around in unexpected ways, possibly allowing, through the resulting unexpected values, attack vectors into the software that contains them.

Checked arithmetic operators that indicate if a division by zero would have occurred are: `?/` and `?%`, and their corresponding checked-assignment-operation operators `?/=` and `?%=`.

The precise C instruction sequence produced by these operators is carefully tuned, through C intrinsics and/or `asm` expressions, to ensure that the underlying registers are accessed as efficiently as possible. A base portable implementation of these instruction sequences expressed purely in C is also provided to allow the porting of the compiler to new computer architectures and new underlying C compilers to be an easier two step process, by allowing the tuning of these expressions to be performed at a later phase after the compiler has been fully ported and tested.

The value of `?+`, `?-`, `?*` , `?/`, and `?%` is a tuple with two members, the type of the first is the type of corresponding arithmetic operator as dictated by the types of its arguments. The second value of the tuple is a boolean, which is true if the operation overflowed (for the first three) or if division by zero would have occurred (for the last two), false otherwise. The value of the assignment-op operators: `?+=`, `?-=`, `?*=`, `?/=`, and `?%=` is a boolean that indicates if the operation overflowed (for the first three) or if division by zero would have occurred (for the last two). The `|=` operator is used idiomatically to accumulate the overflow condition. When overflow occurs the actual value computed is arbitrary, but it is valid to use it in further computations, its use is not undefined behavior. These operators are not allowed with floating point types.

The checked-assignment, `?=`, operator is an assignment operator whose value is a boolean that indicates if the value being assigned is larger than the values that can be represented by the target of the assignment. It is used when a larger type is used to perform arithmetic involving values of a smaller type, and the final result is then assigned to the smaller type, at that point checking if the value was too large to fit. If the value doesn't fit the result is truncated, when the types are integral.

```
tuple [int r, bool error]
a_times_b_minus_c_plus_d_div_e(int a, int b,
                               int c, int d, int e) {
    [r, error] = a ?* b;  // type of ?* is tuple [int, bool]
    error |= r ?-= c;     // type of ?-= is bool
    error |= r ?+= d;     // type of ?+= is bool
    error |= r ?/= e;     // type of ?/= is bool
    return [r, error];
}
```

In the following example all the operations are performed with variables of type `large` and the final value, `lr`, is then attempted to be assigned to a, presumably, narrower variable, `r`, of type `int`. Note that for systems where the representations of `int` and `large` are the same, the code below is equivalent to the code above, but if `int` is not capable of storing every `large` value, and because both a multiplication and a division are involved, it is possible that the final value might still fit in an `int` while the intermediate values might have been too large for an `int`.

```
tuple [int r, bool error]
a_times_b_minus_c_plus_d_div_e(int a, int b,
                               int c, int d, int e) {
    large la = a, lb = b, lc = c, ld = d, le = e, lr;
    [lr, error] = la ?* lb;
    error |= lr ?-= lc;
    error |= lr ?+= ld;
    error |= lr ?/= le;
    error |= r ?= lr;
    return [r, error];
}
```

10.5 Keywords

COOGL inherits from C all of its keywords and their semantics. COOGL introduces several new keywords used for object oriented, generic programming, and a few other features. The semantics of several C keywords are enhanced. A few C keywords, that are no longer needed, were removed.

Many of COOGL new keywords and C keywords with extended or restricted semantics have already been touched upon. The C keywords removed in COOGL are shown in the table below, keywords removed from the language remain in the lan-

guage as reserved keywords, any use of them causes a compilation error. C keywords whose semantics have been enhanced or restricted in COOGL are also shown.

C keywords affected by COOGL	
removed	**enhanced**
auto const extern long long register signed unsigned volatile	enum union static void

Section §10.6 describes the removed keywords.

Use of enum to declare compile time constants is described in §20. Scoped enumerations, specifically typed enumerations, and bit field structured enumerations can be specified, see §Error: Reference source not found. Pointers can not occur within a union declaration, see §14.7. Static members are declared with the static modifier, see §4.3. The void keyword also corresponds to a very special class: class void.

Types added by COOGL in addition to the types that it inherited from C are shown in the following table. From a language grammar perspective, these types (and void) are not actual keywords, they are built in global types, see §Error: Reference source not found. The new integer types: bool, byte, ubyte, ushort, uint, large, ularge and unic are introduced in §18 and described in chapter §12. The C keywords char and long are typedefs in COOGL as are uchar and ulong, which were not keywords in C, they are also described in the same section.

Types added by COOGL, in addition to its C types				
bool unic	byte large	ubyte uchar	ushort uint	ulong ularge

Keywords added by COOGL in addition to the keywords that it inherited from C are shown in the following table:

Keywords added by COOGL, in addition to its C keywords				
identifiers	**declarators**	**access**	**modifiers**	**statement**
this retval argsof	class extend genre fieldof typenew	decl pub priv prot	inherit defer redef inline vital	loop promise require

The new keywords are described in detail throughout this book, they are briefly de-

scribed in the sections enumerated below.

The statement keywords:

- The `loop` control flow statement is presented in §9.13.
- The `on` statement is described in §1.4.
- The `promise(expression)` and `require(expression)` specifications are used as part of the function declaration, prior to the function body, see §4.1.

Special identifiers:

- The current object can be referred through the `this` identifier, see §4.13.
- The `argsof` identifier stands for the a tuple whose members are identical to the argument list of a function that `argsof` is a member of, see §11.9.
- The `retval` identifier is used to access the value returned by a function, from the function's destructor, if it has one, see §5.23.

Declarators:

- Chapter §4 describes `class`, which is the central object oriented programming facility.
- Classes can be extended through the `extend` keyword, see §7.1.
- Function and class arguments can be type names for generic programming purposes. A type argument is declared with `genre`, see §11.3.
- Generic programming also makes use of the special `fieldof` declarator, which allows arguments to be field names. This is described in §11.13. A generic linked list implementations that make use of `genre` type arguments and `fieldof` field name arguments in its implementation is in §11.15.
- Incompatible number types can be declared with `typenew`, see §Error: Reference source not found.

Accessibility modifier keywords:

- The access modifiers `pub`, `priv` and `prot` are described in §6.7.
- Local declarations within a function or class, i.e. non member declarations, must start with the `decl` keyword when the base type for the declaration is a type expression, i.e. an expression whose value is a type, see §4.4.

Other modifier keywords:

- Inheritance and polymorphism are specified through `inherit`, see §4.6 and §6, together with `defer` and `redef`, see §6.4 and §6.6.
- The `inline` keyword provides control over function inlining, see §Error: Reference source not found.

10.6 Removed keywords

Several C keywords are removed from the COOGL language, they are preserved as reserved keywords, their use results in a compilation error. The rationale for preserving them as reserved keywords is to minimize confusion when seen by C programmers as variable names and to allow them back in the language if the user base demands it. The rationale for their removal is:

- `auto` - this modifier can only be used for local variables, it indicates that a variable should be allocated on the run time stack. Its absence has the same meaning as its presence, unless `static` is used, in which case the variable is not allocated on the stack, it is allocated globally instead. This keyword is hardly ever used by C code, it belongs mostly to the prehistory of C. The ancestral roots of `auto` are in PL/1 where it was required for stack variables.
- `extern` - this modifier is used to indicate that what might otherwise seem like a local variable declaration or a global variable declaration is actually only a type definition for a global variable declared elsewhere. It is used when a variable or function needs to be used in source code compiled separately from the source code that actually defines it. Uses of `extern` usually appear in header files, though uses within a function are not uncommon, local uses introduce complexity in the language because it goes against the normal scoping rules. COOGL compilation is global, `extern` is not required.
- `Long long` - this is not actually a keyword in C, it is a special type that corresponds to a 64 bit type. COOGL introduces the `large` type which serves this purpose without having to have special case ad hoc parsing for a single type in the language. The ancestral roots of `long long` are from ALGOL68.
- `register` - this belongs to the early of C. It used to be used to direct early C compilers to place a local variable in a register instead of in the run time stack. Modern compilers do much better register allocation than a programmer can express through these means. Modern compilers simply ignore uses of `register`, though C mandates that the address of such a variable cannot have its address taken. Most C compilers honor that language definition leftover.
- `signed` and `unsigned` - uses of these as types or as modifiers is not allowed in COOGL. Instead explicitly typed integer types were added to the language: `ubyte`, `ushort`, `uint`, `ulong`, and `ularge`. Declaration forms such as these:

```
unsigned int ui;
unsigned     u;
signed char  sc;
```

Are not valid in COOGL, they should be written this way instead:

```
uint        ui;
uint        u;
schar       sc;
```

10.7 Undefined behavior and implementation dependent behavior

The definition of undefined behavior in C11 follows:

> *"**3.4.3 undefined behavior**"*
>
> *"behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements"*
>
> *"NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)."* – C11 3.4.3 n1570.pdf:4

Every undefined behavior aspect of the C language inherited by COOGL is treated as *"behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)."* No aspect results in *"ignoring the situation completely with unpredictable results."* Furthermore *"terminating execution"* doesn't occur in an uncontrolled way, instead an exception, that can be caught by an exception handler is documented to be raised when specific undefined behavior occurs, for example when an array is indexed with an invalid index, or when a `NIL` pointer is dereferenced.

Not every aspect of undefined behavior in C is addressed in this section, a significant source of undefined behavior in C relates to the unsafe aspects of C, for example indexing an array out of bounds, accessing memory after it has been released, etc. Invalid memory access aspects of C are addressed fundamentally, at the core of those issues, by its safe language design, see §14.

COOGL source code is compiled into C11 code that does not contain any constructs that would be seen by the C11 compiler as undefined behavior. For every instance of undefined behavior in the C11 standard a specific code generation approach is chosen that prevents its occurrence. The approach might include causing the compilation to fail and directing the programmer to address the issue at the COOGL source code level.

For example, the underlying C11 compiler's limits for internal and external identi-

fier lengths are determined and known by the COOGL compiler. The identifiers in COOGL source code, after any adjustments required by the translation to C11 code, (see Appendix §2S in page 291), are checked to ensure that the length of the adjusted identifier doesn't exceed its length limit. If the limit is exceeded, a compilation error occurs. Internal and external identifiers are described in §S9.

With respect to undefined behavior:

> *"Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined."* – C11 6.4.2.1 n1570.pdf:60

To ensure that undefined behavior does not occur, the COOGL compiler produces a compilation error instead of producing code with identifiers whose length exceed the limits supported by the platform or the limits chosen by the programmer.

One of the most common and unexpected sources of undefined behavior in C programs is integer overflow:

> *"EXAMPLE An example of undefined behavior is the behavior on integer overflow."* – C11 3.4.3 n1570.pdf:4

The `overflows()` function:

```
bool overflows(int n) {
    if (n + 100 < n) return true;
    return false;
}
```

Is compiled by undefined behavior optimizing C compilers into this x86/64 code:

```
_overflows:
        xorl    %eax, %eax      // always return false
        retq
```

The compiler sees that a positive value, `100`, was added the `int n`, and takes *advantage* that integers in math are infinite and that $n+100 > n$, is always true, in the domain of the mathematical integers, and causes the function to always `return false`. The compiler writers forget that the computer does not deal with infinite mathematical integers, it deals with finite numbers in a modular wrap around way. The compiler doesn't even bother to tell the programmer that code was removed, the whole `if (n + 100 < n) return true;` statement was removed. By doing this the compiler is in essence hiding behind the standard description to insert what could be a security hole or backdoor into code that in prior versions of the compiler would have been compiled correctly. Particularly on computer systems where the underlying hardware does not raise an exception on overflow and where signed integers are implemented in two's-complement, e.g. all modern computer systems, the code above is idiomatic code that determines if the addition of a positive number and an integer overflows. The compiler writers might assume that this kind of micro-optimization

and silent code removal amounts to adding value, but what they are actually doing is ignoring the programmer and silently introducing bugs.

In COOGL signed integer overflow or underflow does not cause undefined behavior. The behavior is well defined and corresponds to what every modern computer system does when performing arithmetic with two's complement numbers, i.e. on overflow it wraps from the largest positive number representable to the smallest negative number representable, and vice-versa for underflow. Assuming 32 bit `int`, the largest `int` is 2^{31}-1 = `0x7fffFFFF` = `2147483647` the smallest `int` is -2^{31} = `0x80000000` = `-2147483648`. For example adding 4 to 2^{31}-1 after overflow results in -2^{31}+3 which is = `0x80000003` = `-2147483645`. This means that even when some arithmetic might overflow, later arithmetic might cause it to underflow and the final value could be the correct mathematical value, for example the overflow that produces the value of `b` is canceled by the underflow that produces the value of `c`:

```
int main() {
    int a = 2000111222;
    int b = a + 1000333444; // b == -1294522630 (overflowed)
    int c = b - 1000000000; // c == 2000444666 (underflowed)
    assert(a + 333444 == c);
}
```

Some C compiler writers will tell you with a straight face that their compiler might, now, or in the future produce arbitrary values for `b` and `c`. The compiler could determine that the `a + 1000333444` causes integer overflow, so the value of `b` doesn't have to be determined, nor does the value of `c`, so both could be arbitrary values. Other compiler writer's will snicker and say: *"oh, that program, it is allowed to corrupt all of your files, I can do whatever I want in that case."* Of course, computer systems don't behave that way, compilers are not supposed to behave that way either, neither does COOGL.

Another case of undefined behavior is storing into C string literals, whose type is `const char` array:

> *"It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined."* – C11 6.4.5 n1570.pdf:71

The following code has undefined behavior in C, in COOGL the behavior is specified, an exception is raised, usually SIGBUS in UNIX and UNIX-like operating systems, the specific exception is platform dependent but is documented:

```
int main() { char *p = "hello"; *p = 'H'; }
```

Another example of undefined behavior is where this shift left operation, one of the most primitive hardware instructions, which is well defined in every hardware instruction set, gets turned into some aberration far from the hardware reality of computer systems. C was supposed to be a low level language close to the hardware, not

the playground for Computer Science students to evolve it into a language that gets in the way of the programmer whenever a group of people managed to write "undefined behavior" is a standard document that the complier writer's then use to their perverse advantage to introduce bugs into your program under the guise of optimization. For example:

```
int shift32(int a) { return a << 32; }
```

Causes a well known C11 compiler to return an arbitrary value when compiled with optimization enabled, completely ignoring the shift operation, which can be determined at compile time that it would have produced the value 0, or if the underlying instruction was issued on an Intel x86 CPU, it would have returned the value `a`, because on that computer system shift of a 32 bit sized `int` doesn't do anything, the value is left unchanged. Certainly having the hardware do what the computer system does is better than having the compiler return a random value. The same code without optimization causes the underlying hardware instruction to be compiled and what the computer system does is the result of the shift.

The opportunities for undefined behavior with left shift are:

> *"E1 << E2"*
>
> *"The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand.* ***If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined****."*
>
> *"The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is* $E1x2^{E2}$*, reduced modulo one more than the maximum value representable in the result type.* ***If E1 has a signed type and nonnegative value, and*** $E1x2^{E2}$ ***is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.****"* – C11 6.5.7 n1570.pdf:95

Under the same mantra of *optimization* compiler could prove an invariant based theorem that for a specific `E1 << E2` shift of an `int` $E1x2^{E2}$ is not "*representable in the result type*," and allow itself to produce a random value making the code go "faster" by omitting code generation for the shift. Of course that would be even more nonsense, but it would seem that compiler writers would be eager to be consistent with their undefined behavior useless optimization effort related to shift of a 32 bit `int` by 32 bits, what is the performance benefit of that?

In the real hardware world the shift left instruction is the same exact instruction for both `int` and `unsigned int`, and if it happens to be different in some ancient, no longer relevant machine, then the compiler should just generate that instruction and let it do what it does.

In the paper *"Undefined Behavior: What Happened to My Code?"* (Xi Wang Hao-

gang Chen Alvin Cheung Zhihao Jia# Nickolai Zeldovich M. Frans Kaashoek) the authors review a variety of undefined behavior "optimizations" performed by compilers that have caused the directions of the programmer to be ignored under the guise of optimization. They explain the approach of 4 large projects (the Linux kernel, the FreeBSD kernel, the PostgresSQL database, and the Apache web server) to problems introduced into their code base and their approach to address them. The first three of the projects chose to disable most of the optimizations that result from the undefined behavior optimization features, only Apache chose to attempt to address these issues as they are discovered, which seems counterintuitive because a web server has a larger attack surface and vulnerability because web servers are meant to serve requests from completely untrusted computer systems. The other 3 projects, the compiler writers would say, are no longer written in C because they depend on undefined behavior being defined in certain ways not defined by the standard (signed integers are implemented in two's complement, they don't raise overflow or underflow exceptions, they wrap around, etc., NULL pointer related optimizations are disabled, and strict aliasing optimizations are also disabled).

There are quite a few other undefined behavior situations in the C11 specification, they are all addressed by COOGL, the description of how they are addressed is in a separate document.

10.8 Implementation-defined behavior and unspecified behavior

The definitions of unspecified behavior and implementation-defined behavior in C11 follow:

> *"**3.4.1 implementation-defined behavior**"*
>
> *"unspecified behavior where each implementation documents how the choice is made"*
>
> *"EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right."* – C11 3.4.1 n1570.pdf:3
>
> *"**3.4.4 unspecified behavior**"*
>
> *"use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance"*
>
> *"EXAMPLE An example of unspecified behavior is the order in which the arguments to a function are evaluated."* – C11 3.4.1 n1570.pdf:4

Important unspecified behaviors and implementation-defined behaviors of the C language inherited by COOGL, that are important to address to make COOGL a more useful language are explained and specified in this section.

This unspecified behavior could lead to data leaks if the compiler doesn't actually copy the pad fields, for example if it skips them when structures are assigned:

> *"When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values."* – C11 6.2.6.1 n1570.pdf:44

To ensure that this does not occur when a structure has internal padding bytes, or when bytes that contain bit fields have padding bits, the translated structure at the C11 level has all of that storage accounted for by explicitly inserting declarations of extra fields where the padding would have occurred. This prevents scenarios where a programmer has allocated zeroed memory used it as a structure, and later assigns that structure's value to another structure, the target structure will not have padding bytes with unspecified values. This is important in scenarios where the data in the structure might be externalized and information might have leaked unbeknownst to the programmer through the padding bytes because of liberties the underlying compiler might take in this situation, for example not copying every byte of a structure when a structure is assigned to another structure.

Converting from an unsigned integer to a signed integer, the 3rd paragraph below:

> *"When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged."*
>
> *"Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type."*
>
> *"Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised."* – C11 6.3.1.3 n1570.pdf:51

In COOGL the conversion does not raise an exception, instead the conversion results in the integer having the exact same representation, i.e. bit pattern, in the signed integer value as it had in the unsigned integer value.

C11 gives liberty to the underlying system to handle right shifts of signed values in implementation defined ways. Machines that don't implement two's complement and that don't support right shift with sign extension, i.e. arithmetic right shift, are no longer relevant. In COOGL shift right of a signed value always causes sign extension.

> *"The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1 / 2^{E2}. If E1 has a signed type and a negative value, the resulting value is implementation-de-*

fined." – C11 6.5.7 n1570.pdf:95

It is important to emphasize that there is a tremendous amount of lore and very useful algorithms and programming techniques that simply can not be expressed if sign extending shifts are not supported, see *"Hacker's Delight"* by Henry S. Warren, Jr.

Note that if you use shift right of a negative value to try to implement division by a power of two you won't get the same result that you would get with signed integer division, rounding of the remainder will not be towards zero, divisions with a remainder will require an off by one adjustment in those cases. But if you are scaling coordinates by a power of two, and you want the scaling to be uniform everywhere, instead of leaning towards zero, then an arithmetic right shift is the correct way of doing it.

10.9 Loop optimization concern

About the only optimization that seems to matter about integer overflow relates to walking arrays and their indexing with signed variables, the effects it can have in loop unrolling loops. Also issues related to indexing with a 32 bit `int` on some 64 bit platforms, i.e. overheads related to sign extension:

> *"Signed integer overflow: If arithmetic on an 'int' type (for example) overflows, the result is undefined. One example is that "INT_MAX+1" is not guaranteed to be INT_MIN. This behavior enables certain classes of optimizations that are important for some code. For example, knowing that INT_MAX+1 is undefined allows optimizing "X+1 > X" to "true". Knowing the multiplication "cannot" overflow (because doing so would be undefined) allows optimizing "X*2/2" to "X". While these may seem trivial, these sorts of things are commonly exposed by inlining and macro expansion. A more important optimization that this allows is for "<=" loops like this:"*
>
> > *"for (i = 0; i <= N; ++i) { ... }"*
>
> *"In this loop, the compiler can assume that the loop will iterate exactly N+1 times if "i" is undefined on overflow, which allows a broad range of loop optimizations to kick in. On the other hand, if the variable is defined to wrap around on overflow, then the compiler must assume that the loop is possibly infinite (which happens if N is INT_MAX) - which then disables these important loop optimizations. This particularly affects 64-bit platforms since so much code uses "int" as induction variables."*
>
> *"The cost to making signed integer overflow defined is that these sorts of optimizations are simply lost (for example, a common symptom is a ton of sign extensions inside of loops on 64-bit targets). Both Clang and GCC accept the "-fwrapv" flag which forces the compiler to treat signed integer overflow as defined (other than divide of INT_MIN by -1)."* – Chris Lattner

To address Lattner's concerns. The loop below, when compiled with GCC produces identically unrolled optimized code with or without the `-fwrapv` compiler option, CLANG doesn't infer from the `abort()`, which makes the `for` unreachable, that `n` must be smaller than `INT_MAX`. Without the `if (n == INT_MAX) abort();` both GCC and CLANG do not unroll this loop when `-fwrapv` is used.

```
#include <stdlib.h>
#include <limits.h>
void f(int n, double a[], double s) {
    if (n == INT_MAX) abort();
    for (int i = 0; i <= n; i++) a[i] *= s;
}
```

Note how you have to suspiciously setup the loop to test `i <= n`, if the range worked on was `i < n`, both compilers produce identical code with and without `-fwrapv`. Given that Lattner talks about a *"ton of sign extensions"* he must be referring to the address arithmetic and having to widen `int` variables to 64 bits to compute array element addresses. Which implies that in his problematic loop, the body of the loop must be working on an array. Having arrays that have more than 2^{31} elements is going to become more and more common, walking within the inside of such an array with wrap-around `int` indexes that go negative must be quite unusual, so the assumption that these compilers ought to be making is that wrapping array indexing does not occur, and if a programmer wants that, a `-fwrapping-array-indexes` compiler optimization disablement can be provided. Furthermore, the compiler could warn when it sees `i <= n` instead of `i < n` in loops involving `int` indexes, and indicate that it is enabling that optimization, and provide a warning for the loop in question, the programmer might realize that it is actually a bug in his code, as array indexing in C is zero based, not one based, and `i < n` might be what is needed.

When COOGL is compiled into C11 code, and the underlying C11 compiler requires `-fwrapv` and related (or similar) options, because it is one of these undefined behavior optimizing compilers, then the code translated into C11 code is always compiled with those options, and to address Lattner's concern, warnings will be produced when loops iterate with signed (32 or 64 bit) variables and have termination conditions predicated on a `<=` test instead of a `<` test, or if walking backwards, if `>=` is used instead of `>`. The programmer can then decide to claim via a `require()` contract or an `expect()` assertion that indeed the ending value is not the largest value (or smallest value) possible for the iterating variable's type, e.g. `INT_MAX` or `LONG_MAX`.

11 - Generic programming and object allocation

> *"A module is parameterized by a type parameter ... if the module is to do anything with objects of the parameter type, certain operations must be provided by any actual type. Information about required operations is described in a where clause, which is part of the heading of a parameterized module. For example:*
>
> ```
> set =
>    cluster[t: type] is
>       create, insert, elements
>    where t has
>       equal: proctype(t, t) returns(bool)"
> ```
>
> -- Clu Reference Manual, October 1979

Generic programming allows general purpose code applicable to unknown types to be written. The name of a type, built-in or user defined, is a type object. Types as data items are no different than any other native data item in the language, they can be used as variables, members, and as arguments to functions or classes. Types as variables are typed, they can only be assigned compatible types. The names of fields of generic types can also be arguments to functions or classes.

Dynamic memory allocation and deallocation of objects and arrays of objects is not built into the language. The `lib.creatable` interface provides heap based dynamic object creation and destruction.

11.1 Type dot expression

Types are used in generic programming to specify the type that a generic type argument must be compatible with. For example, the generic argument to the class `stack`, in §11.3, has to be compatible with the `lang.value` interface type.

A type dot expression, *`type_dot_expr`*, is an expression that refers to a type, it can just be an identifier that refers to a type, or it can be an expression formed by a series of identifiers separated by the dot operator. The first identifier can be preceded by the dot-dot operator: `..` to indicate that that identifier is to be searched for in the outermost scope instead of being searched for relative to the current scope.

For example:

```
pub ..libc.FILE *fp = ..libc.stdin; // absolute type expression
pub class c {
    pub class libc {                // hides global libc ...
        pub typedef int FILE;       // purposely confusing!
    }
    pub ..libc.FILE *get_stdin() {
        return ..libc.stdin;        // absolute type expression
    }
    pub libc.FILE integer;          // relative type expression
}
```

11.2 Constructor invocation syntax with built-in types

The construction syntax for classes `type(argument_list) variable` is also applicable to built-in types, for example these declarations are equivalent:

```
decl int(2) i;          // declaration of i equivalent to ...
int i = 2;              // ... this declaration of i
int tab[3] = {7, 7, 7}; // declaration of tab equivalent to ...
decl int(7) tab[3];     // ... this declaration of tab
```

To keep the code readable, and because this construct matters only in the realm of generic programming, this syntax is not generalized to pointer declarations:

```
char *cp = "abc";             // valid
decl char *("abc") cp;        // invalid
```

This syntax is valid only when the type specification doesn't compound the base type beyond array declarations, `typedef` can be used for pointers:

```
typedef char *char_ptr;
char *p = "abc";          //  declaration of p equivalent to ...
decl char_ptr("abc") p;   // ... this declaration of p
```

11.3 Type arguments, type variables, and type values

The syntax `genre type_dot_expr ident` is used to declare an argument, `ident`, that refers to a type that is compatible with the type that `type_dot_expr` refers to. This means that the type of `ident` must be the same type, or a subclass of the type, or implement the interface, that `type_dot_expr` refers to. The declaration `genre void type` declares `type` to be a universal type variable. All types are compatible with it because all types descend from `class void`. The generic argument declaration `genre lang.value type`, below, declares `type`, an argument that refers to a type that implements value semantics, i.e. types that implement the `lang.value` interface, this is required by `swap()` because it implements swapping through assignment and also by initializing `temp` from another object of the same `type`, i.e. `*a`:

```
void swap(genre lang.value type, type *a, type *b) {
    type temp = *a;
    *a = *b;
    *b = temp;
}
```

A type value can have one of these forms:

- a type name (native or user defined);
- an expression that refers to a type, for example: `list->type`, or `stk.type` where `type` is a type name;
- an expression that refers to a variable whose value is a type;
- a type specification that uses type declarators, as shown below.

This invocation of `swap()` passes `int` as the `type` argument:

```
void test() { int i = 1, j = 2; swap(int, &i, &j); }
```

Types that use type declarators to specify pointer or array types have to be specified by prefixing them with `class`, for example, `class char *` as shown below.

Type arguments can be omitted, in which case they are deduced by the compiler:

```
void test() {
    char *c = "cat", *d = "dog";
    swap(char *, &c, &d);           // error
    typedef char *char_ptr;
    swap(char_ptr, &c, &d);         // workaround: needs typedef
    swap(class char *, &c, &d);     // better: without a typedef
    swap(&c, &d);                   // NICER: type deduced §11.5
}
```

A generic version of the `stack` class is shown below:

```
class stack(genre lang.value type,
            size_t max, int *error) promise(empty()) {
    priv type entries[];
    entries.create(max);
    priv type *sp = entries.start;
    *error = !sp ? libc.ENOMEM : 0;
    return;

    pub void deinit() { entries.destroy(); }
    pub bool empty() { return sp == entries.start; }
    pub bool full() { return sp == entries.end; }
    pub void push(type v) require(!full()) { *sp++ = v; }
    pub type pop() require(!empty()) { return *--sp; }
    pub type top() require(!empty()) { return sp[-1]; }
}
```

The type argument specifies that the type must implement the `lang.value` inter-

face, it must be a value type so that objects of its type can be assigned, passed by value as arguments, and returned as function values.

Note that when an expression refers to a variable whose value is a type the specific type that it contains might only be determinable at run time, but can not vary during the program's execution, because type variables and type arguments are not value like, they can not be assigned to or changed in any way after their initialization.

11.4 Restrictions on type arguments

Type arguments must be specified first in argument lists. This restriction allows type arguments to be omitted, from left to right, and be deduced instead. Also pointers to specific instances of generic types can be declared by omitting the non type arguments in an intuitive manner, i.e. by omitting the trailing arguments. For example:

```
class point(genre lang.value type, priv type x, priv type y) {}
void use() {
    decl point(int, 1, 2) ipoint;
    decl point(float, 1.23, 4.56) fpoint;
    decl point(12.34, 56.78) fpoint2;   // type deduced §11.5
    decl point(int) *p = &ipoint;       // valid: omits x and y
    decl point(int, 0, 0) *q = &ipoint; // error: extra args
}
```

The declaration of `p` as a pointer to a `point(int)` doesn't specify the `x` and `y` arguments, which are not pertinent to `p`'s type. The declaration of `q` is invalid.

Objects of the `point` type are not value like, a stack of `point` can not be declared:

```
void example() {
    decl stack(int, 10, &error) istk;    // int is a value type
    decl stack(int) *istkp = &istk
    decl stack(point(int), 10, &error) pstk;
    // error: point(int) is incompatible with lang.value
}
```

The `point` class can be enhanced so that objects of its type can be used as values:

```
class point(genre lang.value type, pub type x, pub type y) {
    pub is lang.value(point);       // point is a value type
    pub void init(point raw *to) redef {
        x.init(&to->x), y.init(&to->y);
    }
    pub void reinit(point *from) redef {
        x = from->x, y = from->y;
    }
}
```

11.5 Type argument omission and deduction

A type argument can have a default value, for example:

```
class bitmap(genre lang.whole type = ularge, pub size_t n)
         require(n > 0) {
    priv type data[(n + type.bits - 1) & ~(type.bits - 1)];
    for (type *p = data; p < data.end; ) *p++ = 0;
    return;

    priv typedef tuple [size_t ix, type mask] ix_mask;
    pub bool get(size_t b) require (b < n) {
        ix_mask im = get_ix_mask(b);
        return data[im.ix] & im.mask;
    }
    pub void set(size_t b) require (b < n) {
        ix_mask im = get_ix_mask(b);
        data[im.ix] |= im.mask;
    }
    pub void clear(size_t b) require (b < n) {
        ix_mask im = get_ix_mask(b);
        data[im.ix] &= ~im.mask;
    }
    priv ix_mask get_ix_mask(size_t b) require(b < bits)
        return [b >> type.bits,
                cast(type) 1 << (b & (type.bits - 1))];
}
```

As shown earlier with `swap()` and `point`, type arguments can be omitted, from left to right, when a function or class with type arguments is used. Omitting the first type argument causes its type to be deduced based on the type of the first non-type argument whose type is based on the omitted type argument. If there is a second type argument, and it is omitted too, its type is deduced based on the type of the second non-type argument whose type is based on the omitted second type argument. Similarly for additional type arguments. For example:

```
class stuff(pub genre lang.value type1,
            pub genre lang.value type2,
            int intarg, type1 *t1ptr,
            float floatarg, type2 t2arg[]) {
    pub int    intval = intarg;
    pub type1 *pointer = t1ptr;
    pub float  floatval = floatarg;
    pub type2 *t2val[] = t2arg;
}
```

In the use of `stuff` below to declare `s`, `s.type1` is `double` and `s.type2` is `char`:

```
decl stuff(1, cast(double *) NIL, 3.141593, "hi") s;
```

Of course that is a contrived example, a practical one:

```
max.type max(pub genre lang.number type, type a, type b)
            return a > b ? a : b;
int main() {
    int x = libc.rand(), y = libc.rand();
    on ("max("; x; ", "; y; ") = "; max(x, y); '\n') print();
}
```

See §Error: Reference source not found for more details about `lang.number`, the ancestor class of all number types in the language.

11.6 Specialization of generic classes and functions

The `bitmap` generic class presented in §11.5 includes a data member, `n`, that holds the number of bits in the `bitmap`, at run-time. The type of `n` was chosen to be `size_t`, an appropriate type for sizes in most circumstances. A specialized version of `bitmap` that allows the type of `n` to be chosen, for example to be a `ubyte`, because the bitmaps are known to have at most 128 bits, follows:

```
class bitmap(genre lang.whole type = ularge,
             genre lang.whole size_type,
             pub size_type n) require(n > 0) {
    pub typedef bitmap bitmap_type;
    ...                                 // rest of class unchanged
}
```

Note that this version of `bitmap` when used with various types leads to its specialization into incompatible types, for example a `bitmap` whose `size_type` is `ubyte` is a different `bitmap` type than one whose `size_type` is `size_t`, note that the type of `size_type` is deduced in both `decl` declarations:

```
void f(ubyte ubsz, size_t size) {
    decl bitmap(ubsz) ubm;    // size_type is ubyte
    decl bitmap(size) sbm;    // size_type is size_t
    decl ubm.bitmap_type *p = &ubm;
    p = &sbm;                 // error: incompatible types
}
```

A specialized `bitmaplit` that allows the type to be specialized with a `genre lit`, so that the number of bits is not actually stored as part of the `bitmap` but instead becomes part of the type at compile time:

```
class bitmaplit(genre lang.whole type = ularge,
                genre lit size_t n) require(n > 0) {
    priv inherit bitmap(size_t, n) inline bm;
    pub alias get = bm.get;
    pub alias set = bm.set;
    pub alias clear = bm.clear;
}
```

Note that `bitmaplit` is implemented by privately inheriting from bitmap and inlining the implementation, which allows the compiler to realize that `n` is a compile time constant and doesn't need to be stored as part of the underlying bitmap object, which is not visible outside of `bitmaplit`. Thus a `bitmaplit(256)` uses $2^8 / 8 = 32$ bytes:

```
void x() { decl bitmaplit(256) b; assert(sizeof(b) == 32); }
```

Specialized generic programming where different implementations are provided and the most appropriate one is chosen is supported by the language. Specialized templates in C++ and its SFINAE mechanism are too complex, error prone, and leads to hard to read code code, the programmer can not usually figure out what code is actually being used, short of reading every header file that might have been included and doing the work that the compiler does to choose the actual template that generates the code.

11.7 Type variables must be initialized, never assigned

Type variables and arguments always refer to a concrete type during their lifetime, they are never invalid, the value is set during their initialization, e.g. at function invocation time. The type value that they refer can not be be changed through assignment. Type variables are implemented internally as pointers, when they are required to exist at run time. Static type checking can be applied to multiple objects declared to be of the same type without concern for the type in question changing based on program control flow. If assignment to the type variable was allowed between multiple declarations based on the type, static type checking could not be performed.

An alternative language design, that would be desired by the followers of the school of language orthogonality, would require that type variables be allowed to be changed at run time. The amount of complexity that this would add to the language is very high. Such a design and complexity are mentioned here only to make explicit that such a design and evolution is not desired, furthermore it goes completely against the design goal of simplicity for the language.

Note that by allowing type arguments, which by declaring them with an accessibility modifier can be made into type members of a generic class, and by allowing for the nature of the type arguments that is allowed to be specified, (for example that they be value like, comparable, relationally comparable, hashable, etc), then all the needs for type safe generic programming that produces incompatible type errors at compile time are satisfied. This is much better than C++ link time errors produced after the files are silently compiled successfully because the type that a generic argument must be compatible with can not be specified.

11.8 Function names vs class names

Even though the notions of function and class are unified by the language, there are

differences between them. These properties apply to class declarations:

- Introduces a new type.
- A class name can be used to specify other types, e.g. `stack *`.
- The class name is a type value, it is not a function pointer.

In contrast, a non-class function:

- Does not introduce a named type, it introduces an unnamed type.
- The function name is a function pointer, it is not a type value.

11.9 The `argsof` tuple type member

Classes and functions have a compiler generated tuple type public member, `argsof`, it corresponds to the argument list of the class or function. The calling convention for passing tuple values to a function is such that the values are passed individually as if each was an argument. The calling function can receive them in a tuple, or with arguments whose types correspond exactly to the type of the tuple's values. The `argsof` tuple type is important for dynamic object creation and destruction, i.e. on the heap, as described in §11.10. An example use of `argsof`:

```
void f(int i, float f, char c) {
    on ("i = "; i; ", f = "; f; ", c = "; c; '\n') print();
}
void example() {
    decl f.argsof args = [1, 3.141593, 'x'];
    f(args);
}
```

11.10 The `lib.creatable` interface

Allocation and deallocation support in C does not require special language facilities, other than the `sizeof` operator which is very convenient to prevent programming errors from unmaintainable code that knows the size of a type. The C allocator returns a value of type `void *`. In COOGL object creation is typed correctly, it does not require an unsafe pointer cast or unsafe pointer assignment as is the case with C. The additional language support in COOGL is the `argsof` tuple type member, see §11.9, and the ability to specify the memory on which a constructor is invoked.

Classes that support the `create()` static member function and the `destroy()` non-static member function, by providing the `lib.creatable` interface, can have objects of its class created and destroyed under arbitrary program control. Otherwise objects of the class type can not be created or destroyed at run-time under arbitrary program control. Optional arguments, see §13.8, to `lib.creatable()` are not shown here.

```
pub namespace lib {
    pub interface creatable(genre void type) {
        extend class type {
            pub !inherit static type *create(type.argsof args){
                type raw *r = lib.object.alloc(type);
                return type(args, r); // constructor invocation
            }
        }
        pub void destroy()
          require(lib.object.allocated(type, this)) redef {
            deinit();                  // destructor invocation
            lib.object.free(type, this);
        }
        // array allocation support not shown here, see §13.8
    }
}
```

The implementation of `create()` and `destroy()` obtained by providing the `lib.creatable` interface can be customized by redefining those member functions to mediate between them and the additional required functionality. For `create()` and `destroy()` to be used by clients, the provision of the interface must be accessible, the `is` declaration must be `pub`.

The outermost code of `lib.creatable` is shown above. The implementation provided with the compiler is part of the design and implementation of the language safety, it is required to ensure COOGL's type safety, and central to its very memory efficient object layout and very fast polymorphic member function dispatch. Note how `type.argsof`, a tuple type equivalent to the argument list of the constructor of `type`, is used to specify an identical argument list for the `create()` static member function. Support for array allocation and deallocation is presented in §13.8.

Memory allocation support is added to a class by providing the `lib.creatable interface`:

```
class ratio(pub int numerator, pub int denominator) {
    pub is lib.creatable(ratio);
}
```

An example use follows, the `r` pointer points to a heap allocated and constructed object, unless the allocation fails, in which case the value of `r` is `NIL`:

```
void example() {
    ratio *r = ratio.create(3, 4);
    if (!r) return;
    on (r->numerator; " "; r->denominator; '\n') print();
    r->destroy();
    decl ratio(5, 11) rr;
    rr.destroy();    // causes run-time exception to be raised
}
```

If `destroy()` is invoked on an object that is a member of another object, or if it was not allocated by `lib.object.alloc()`, a run-time exception is raised by the `require` precondition of `destroy()`.

11.11 Public static member functions that can't be inherited

A class `derived` that inherits from another class `base` which provides the `lib.creatable` interface does not have a `derived.create()` member function that creates `derived` objects, nor does it have one that creates `base` objects. The `create()` member function, added through an `extend class` by `lib.creatable`, is declared to be publicly accessible but not inheritable, i.e. `pub !inherit`, which causes `create()` not to be inherited by `derived`.

11.12 Literal arguments to generic classes

A literal argument to a generic argument, can be used to parameterize a generic type, the literal value is a per class value, not a per object value, it can be used to dimension statically sized arrays within the generic class, see §Error: Reference source not found for an example.

11.13 Field name argument declarations with `fieldof`

The names of fields of generic types can used as arguments to functions or classes. A `fieldof` argument or member argument declaration is used to specify an argument that stands for a field (usually declared with a different name than the argument name) of a generic type, the `fieldof` declaration also specifies the type that the field should be compatible with.

For example in class `list`, below, its `field` argument stands for the name of a field of the generic argument specified by `type`, the type of `field` is `link`, a static member class of `list`.

```
class list(priv genre void type,
           priv fieldof type list.link field) {
    priv inherit link;       // next and prev used by list head
    return;
    ...
    pub static class link {
        priv pub { list } link *next = NIL, *prev = NIL;
        return;
        ...
    }
}
```

Class `entry` can have its members in 3 different lists at the same time, its links

within those lists are `link1`, `link2`, and `link3`. These field names are used as arguments to the class `list` when using it to access its nested static class `link`. The argument `field` when instantiated by `link1` indicates that `link1` is a field of `entry` (the first argument of `list`, i.e. the `type` argument), furthermore its type has to be `link` or a class that descends from it.

```
class entry(put byte *data) {
    pub list(entry, link1).link link1;
    pub list(entry, link2).link link2;
    pub list(entry, link3).link link3;
    pub is lib.creatable(entry);
}
```

A complete implementation of `list` is in §11.15.

11.14 Generic intrusive lists

Fine grained generic programming is troublesome in many languages. One of the goals of the generic programming facility in COOGL was to be able to support generic data structures, for example lists, hashes, trees, etc that are as efficient as hand crafted data structures, specifically they should not impose memory overheads that don't exist in hand crafted ones. Additionally, their coding should be straightforward, not the result of an accidentally discovered language within another language as it is the case with template meta-programming in C++, which even with all of its impenetrable obscure programming mechanism, doesn't allow for the most common ways of implementing certain data structures which are easily programmed in C.

The best approximation of intrusive lists in C++ is provided by Boost but the complexity compounding across C++ features leads to these problems:

> *"However, member hooks have some implementation limitations: If there is a virtual inheritance relationship between the parent and the member hook, then the distance between the parent and the hook is not a compile-time fixed value so obtaining the address of the parent from the member hook is not possible without reverse engineering compiler produced RTTI."*
>
> *"Apart from this, the non-standard pointer to member implementation for classes with complex inheritance relationships in MSVC ABI compatible-compilers is not supported by member hooks since it also depends on compiler-produced RTTI information."* -- www.boost.org

It is quite usual for an object to be linked into more than one linked list, such that when found through one list it might need to be removed from another list. Data structures of this nature are quite common in system software. An implementation where the pointers are within the objects is usually desired because it has the smallest overhead. These lists, with linkage within the objects themselves are called intrusive

lists. When the lists themselves are unknown, it is just known that the object is in some list, which is quite common, the usual programming idioms in C for `#define` based reusable list manipulation macros are type unsafe. The challenge is then the type safety of the generic code, and to do so with no overhead, i.e. functionally identical code but without the risk of unsafe memory accesses as a result of programming error or concurrency. Various forms of generic intrusive lists are presented in this chapter.

11.15 Generic doubly linked list: `list`

The `list` class is a generic list type, with member functions to insert an object as its first or last element, and to remove the first or last element, if the list is not empty, the removed element is returned, if any, otherwise `NIL` is returned. The `list` class uses `list.link` for its links.

```
class list(priv genre void type,
           priv fieldof type list.link field) {
    priv inherit link;
    next = prev = this;
    pub bool empty() inline return this == next;
    pub static class link {
        priv pub { list } link *next = NIL, *prev = NIL;
        pub bool in_list() inline return next != NIL;
        prot void remove() require (in_list()) inline {
            link *n = next, *p = prev;
            n->prev = p, p->next = n;
        }
        prot void ins(link *p, link *n)
          require(!in_list()) inline {// insert between p and n
            prev = p, next = n, n->prev = p->next = this;
        }
    }
    pub type *insert_first(type *ent) inline
        return ent->field.ins(this, this->next), ent;
    pub type *insert_last(type *ent) inline
        return ent->field.ins(this->prev, this), ent;
    priv type *rem(link *e) inline return empty() ? NIL :
          (e->remove(), field_to_obj(type, link, field, e));
    pub type *remove_first() inline return rem(next);
    pub type *remove_last() inline return rem(prev);
}
```

Classes whose objects want to be in a `list` declare their links with `list.link`, as shown further below. This form of intrusive `list` has `prev` and `next` pointers to form a doubly linked list. To make insertion and removal as fast as possible, at the start or at the end of the list, the list head itself has the same previous and next point-

ers, an empty list is just the list pointing to itself, thus insertion and removal have no special cases. The previous pointer of the first element points to the list head, and the next pointer of the last list element points to the list head. Thus the list is circular. No tests need to be performed when inserting or removing a list element, furthermore, the list removal code doesn't need to know (i.e. have the address of) the list that the entity is being removed from, this is the most common and most efficient doubly linked list used in system software, particularly if adding an entry at the beginning and at the end of the list needs to be performed in constant time. The fundamentally unsafe aspect of this kind of list (in C and C++) is that the list head could end up being manipulated as if it were an object of the wrong type, i.e. as it if were a list element. See §1L.2 for `field_to_obj()`.

11.16 Use of `list`

An example of `list` that has objects on 3 different lists at the same time follows:

```
class entry(pub byte *data) {
    pub list(entry, link1).link link1;
    pub list(entry, link2).link link2;
    pub list(entry, link3).link link3;
    pub is lib.creatable(entry);
}
```

There are 3 `pub` members of `entry` based on the `list.link` type. Their types are different because they refer to different member names.

List declaration and initialization follows. The types of `list1`, `list2`, and `list3` are all different because they are a function of the member names `link1`, `link2`, and `link3` respectively.

```
decl list(entry, link1) list1;
decl list(entry, link2) list2;
decl list(entry, link3) list3;
```

List manipulation:

```
int main() {
    entry *a = entry.create("a");
    entry *b = entry.create("b");
    entry *c = entry.create("c");
    entry *e;
    list1.insert_first(a);          // list1: {a}
    list1.insert_first(b);          // list1: {b, a}
    list1.insert_first(c);          // list1: {c, b, a}
    e = list1.remove_last();        // list1: {c, b}
    b->link1.remove();              // list1: {c}
}
```

Test that the types of the lists are different:

```
test() {
    decl list(entry, link1) *pl1 = &list1;
    decl list(entry, link2) *pl2 = &list2;
    pl1 = pl2; // error: incompatible pointer types
}
```

12 - More about types and smart pointers

"The machines on which we first used BCPL and then B were word-addressed, and these languages' single data type, the `cell,' comfortably equated with the hardware machine word. The advent of the PDP-11 exposed several inadequacies of B's semantic model. First, its character-handling mechanisms, inherited with few changes from BCPL, were clumsy: using library procedures to spread packed strings into individual cells and then repack, or to access and replace individual characters, began to feel awkward, even silly, on a byte-oriented machine."

-- Dennis Ritchie

All types descend from `class void`. User defined types don't descend directly from it, they descend indirectly through one of these intermediate classes: `lang.classes`, `lang.array`, `lang.number` and `class void *`. The type hierarchy exists to aid native type extension, generic programming, and the treatment of all variables, including pointers, arrays, and array descriptors, as objects. The ability to treat native types as objects allows generic programming to use native types as type arguments. The treatment of pointers as objects allows for the management of pointers and their lifecycle, supporting programming idioms that sometimes referred to as smart pointers.

12.1 Integer types

The hardware and compiler dependent integer types of the target language, i.e. the C integer types, obey the C size restrictions: `sizeof(char)` ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)` ≤ `sizeof(large)`.

The language also has a native boolean type, `bool`, and its literal values: `true` and `false`. Various standard integer `typedef` definitions are also provided: `byte` a signed 8 bit integer, `large` the largest supported integer type, and `index` for variables capable of indexing the largest possible arrays that can be addressed, its size is the same as the size of pointers. The integer types are: `byte`, `short`, `int`, `large`, and `index`; and the corresponding unsigned types: `ubyte`, `ushort`, `uint`, `ularge`, and `uindex`.

The compiler includes options for multiple COOGL compilation modes on systems that support multiple C compilation modes, for example 32 and 64 bit modes. The COOGL compiler's expression evaluation is semantically identical to the native C compiler in each supported compilation mode.

12.2 Indexing types

Historically, in all mainstream computer systems `int` is a 32 bit sized type. At the same time, systems with pointers that are 64 bit wide are also mainstream, legacy systems and deeply embedded systems remain with 32 bit wide pointers. Most systems, even handheld computers, e.g. tablets and smart phones, have physical memories larger than 1GB, at the time of this writing they have between 2GB and 8GB, and there doesn't seem to be a reason why their physical memory won't continue to grow, particularly because digital media, pictures and movies, continue to have higher resolutions and quality (e.g. higher resolutions, higher frames per second, slow motion movie capture modes, etc). At the other extreme, large computer systems with physical memories larger than a TB, i.e. 1024 x GB, are common, even mid-size computer servers are have physical memories in the TB order too.

A consequence of all of this, is that arrays whose numbers of elements are larger than 2^{31}-1 elements might become more and more common. Indexing such arrays with a 32 bit signed variable of type `int` will become problematic over time. The natural progression would be for C to eventually have its `int` type be 64 bit wide, but there is too much legacy software whose binary interfaces require `int` to be 32 bit wide, it is very unlikely that this will change in the foreseeable future.

Additionally, if `int` were 64 bits wide, there would not be a native type for 32 bit words or 16 bit words, because `short` could not serve both roles. Some ALGOL68 based aberration similar to `long long` could be devised, for example, `short` could be 32 bits and `short short` could be 16 bits wide.

The types `index` and `uindex` serve the purpose of having array indexing types that won't run into trouble for arrays that can not be indexed correctly with variables of type `int`. Because most array indexing is done with local variables, using `index` or `uindex`, does not have run-time costs associated with them.

Out of bounds indexing of arrays and array descriptors causes a run time exception, see §14.33. Could also have compile time options to forbid arrays larger that 2^31-1 for software that doesn't require such large arrays, hardware integer overflow exceptions is valuable.

Size restrictions: `sizeof(int)` ≤ `sizeof(index)` ≤ `sizeof(large)` and behave exactly the same as their C counterparts, the same relationship holds for their unsigned counterparts.

12.3 Floating point, complex, and imaginary types

The hardware and compiler dependent floating point types of the language, i.e. the C floating point types, they behave exactly as their C counterparts, they follow these restrictions:

```
sizeof(float)             <= sizeof(double)
sizeof(imaginary)         == sizeof(imaginary_float)
sizeof(imaginary)         == sizeof(float)
sizeof(imaginary_float)   <= sizeof(imaginary_double)
sizeof(imaginary_double)  == sizeof(double)
sizeof(complex)           == sizeof(complex_float)
sizeof(complex_float)     <= sizeof(complex_double)
sizeof(complex)           == 2 * sizeof(float)
sizeof(complex_double)    == 2 * sizeof(double)
```

12.4 Enums

The space and layout rules for variables of `enum` type, when used in a `struct` declaration, strictly follow the rules of the native C compiler. An integer type, a floating type, or a pointer type can be associated with an `enum` declaration. An enumeration declared with `enum class` causes the enumeration identifiers to be accessible only through the enumeration type name, the identifiers are not added to the scope where they are made, instead they are scoped by the `enum` type being declared:

```
enum pet { CAT, DOG, HORSE };
pet p = CAT;                // type of pet dictated by C compiler
enum ularge page {
    SIZE   = 4096,
    OFFSET = SIZE - 1,
    MASK   = ~OFFSET,
};
page pg = SIZE;             // integer type: ularge
enum class double math { pi =  3.14159265358979323846264338 };
double pi_x_2 = math.pi * 2;
enum class byte kind {
    EXPLICIT = 0,
    USERDEF  = 1,
    NATIVE   = 2,
    COMPOUND = 3
};
pub kind k = kind.NATIVE;  // integer type: byte
pub kind u = USERDEF;      // error: USERDEF is undefined
```

The initialization can take advantage of the member lookup operator, see §10.2:

```
pub kind(^USERDEF) u;
```

Within a `struct` declaration `enum` typed fields cannot be based on enums whose

integer type is explicitly specified, this is a feature not available to the `struct` bridge to the native C compiler.

An `enum` used to name a set of known values, among a larger set of values, can be specified by using `...` at the end of its *`name = value`* list, for example:

```
enum ularge page {
    SIZE   = 4096,
    OFFSET = SIZE - 1,
    MASK   = ~OFFSET,
    ...                         // other values are valid
};
page pg = 0;                    // integer type: ularge
```

Absence of `...` implies that other values are not valid, potential assignment of a value outside the valid value set produces a compilation error. The compilation error can be disabled with a cast:

```
void example() {
    kind k = 17;            // error: invalid value
    kind k = kind.USERDEF;  // ok
    k = rand();             // error: invalid value
    k = cast(kind) rand();  // tell compiler it is ok
}
```

COOGL debuggers are encouraged to interpret as bit masks `enum` declarations that include `...` and whose values are disjoint in their underlying bit representations. For example:

```
enum class mode {
    r = 4,
    w = 2,
    x = 1,
    ...    // other values are valid
}
mode rw = mode.r | mode.w;
mode rwx = rw | mode.x;
```

Or:

```
decl mode(^r | ^w) rw;
decl mode(^r | ^w | ^x) rwx;
```

In a COOGL aware debugger:

```
(db) print rwx
mode.r | mode.w | mode.x
(db) print cast(int) rwx
7
(db)
```

12.5 Bit fields

Bit fields are a C language feature used to specify one or more fields that use a specified number of bits and where multiple fields can share the same underlying fundamental units of memory supported by the computer system (bytes, words, etc). For example to specify the bits in a 64 bit IEEE `double`:

```
struct ieee64 { uint sign:1; uint exponent:11; fraction:52; };
```

Modern computer systems do not support loading or storing into bit fields directly, from or to memory, the memory unit that contains the bit field first has to be loaded, the required bits have to be extracted from the register that holds the value to actually be able to interpret the value appropriately. To store a value into a bit field the containing storage word has to be fetched, the bits merged into their correct place, and the word stored back into memory. Usually the surrounding bits contain bits of other bit fields and must be preserved. Thus bit fields cause additional computations more expensive than a simple load or store instruction of a fundamental memory word of the underlying computer system.

The base type of a bitfield must be an integer type, or an enumerated type whose base type is an integer type. Nameless bitfields can be declared to specify unused bits. The address of a bit field can not be taken. The rules for bit fields strictly follow the rules of the native C compiler, and might require them to be specified in a different order:

```
struct ieee64 { fraction:52; uint exponent:11; uint sign:1; };
```

A bit field specified with zero bits causes the remaining bits of the underlying base types to be skipped, if there are any, as shown in this program:

```
union bitfields {
    struct {
        ubyte   field1:1, :0; // skip ubyte's leftover 7 bits
        ubyte   field2:2, :0; // skip ubyte's leftover 6 bits
        ushort  field3:3, :0; // skip ushort's leftover 13 bits
        uint    field4:4, :0; // skip uint's leftover 28 bits
        ulong   field5:5, :0; // skip ulong's leftover 59 bits
        ulong   field6:6;
    };
    ulong words[3];
};
bitfields b = { .f1=1,.f2=3,.f3=7,.f4=0xF,.f5=0x1F,.f6=0x3f };
int main() {
    on (b.word[0]; b.word[1]; b.word[2]) printx();
}
```

Its output is:

```
0000000f0007030100000000000001f000000000000003f
```

COOGL extends `typedef` declarations to allow integral types to be declared with a specified number of bits which are convenient for various purposes, for example when specifying variables whose values are used as shift counts to ensure that shift counts are within valid ranges, 0-31 and 0-63, for 32 and 64 bit types, helpful to ensure that shifts in COOGL don't lead to the undefined behavior disease that is infecting modern C compilers while also ensuring that the shifts are as efficient as the underlying C shift operations. This is achieved by ensuring that the values of shift amounts that are precomputed and used repeatedly are always valid without having to mask them to ensure defined behavior. Example declarations:

```
typedef uint uint5: 5;        // sizeof(uint5) == sizeof(uint)
typedef uint uint6: 6;        // sizeof(uint6) == sizeof(uint)
typedef uchar uchar5: 5;      // sizeof(uchar5) == sizeof(uchar)
typedef uchar uchar6: 6;      // sizeof(uchar6) == sizeof(uchar)
```

Variables whose type is a bit field specified with a `typedef` declaration always consume a whole memory unit of its specified base type, fully, multiple such variables declared next to each other in a `class` occupy their own dedicated memory units, thus their bits don't interfere with each other, furthermore the pad bits are guaranteed to be zero, so the cost of fetching their values is the same as fetching the underlying memory unit, plus an additional sign bit propagation cost if the bit field is signed, which are very unusual. The cost of storing into them clearing the high bits to ensure the pad bits remain zero, it is a simple sore, not a fetch-mask-store operation.

To prevent undefined behavior in COOGL with respect to shift counts, if the compiler can not prove that the shift count results in a valid range, a compilation error occurs. The programmer can provide sufficient proof through `require()`, `promise()`, or `assert()` to establish the domain of various arguments, functions, and expressions, to help the compiler prove that the shift count is valid. Worst case the programmer can reduce the shift count to the correct range with a bit-and operation.

12.6 Unicode characters

C89 introduced the syntax `L"wide"` for a character string literal to mean an array of `wchar_t` initialized to the characters between the quotes, and zero terminated, i.e. the same as the traditional C `"string"` literal, which means the same thing but for `char` being the underlying character type. The actual size of `wchar_t` is not dictated by C89. On AIX it is 16 bits on 32 bit compilation mode and it is 32 bits on 64 bit compilation mode. On Solaris it is 32 bits irrespective of compilation mode. COOGL introduces a new integer type, `unic`, for a Unicode character. The `unic` type is a 32 bit unsigned type (it is the same as C11's `char32_t` which is also supported, its exactly the same). The string and character prefix notation for Unicode characters is similar to `L'x'` and `L"x",` but it uses uppercase `U` instead:

```
unic u = U'x';                  // 32 bit Unicode character literal
unic up[] = U"32 bit Unicode literal";
```

12.7 Unicode 16 bit characters

C11 also introduced `char16_t` 16 bit unsigned character to represent the 16 bit subset of the Unicode character set. Its literals use lowercase `u`:

```
char16_t u = u'x';              // 16 bit Unicode character literal
unic up[] = u"16 bit Unicode literal";
```

12.8 Character and string literal

C multi-character character literals are not supported, for example:

```
void invalid() {
    int c = 'abcd'; // error: multi-character literal
}
```

Multi line string literals, with or without the `L`, `u`, and `U` prefixes are supported:

```
void use() {
    char *p = "this is a long literal, split into multiple "
              "lines, the compiler concatenates them\n";
    p.print();
    unic *up = U"hello "          // U must be only at the start,
                "wide world\n"; // U can not be here.
    up.print();
}
```

There is minimal support in the run time `lang` library for string literals used in the construction of the `str` string type, see §XXX.

12.9 Incompatible and global types

Use of traditional number types to represent values of different kinds, for example age, weight, height, force, speed, etc, can lead to subtle errors not caught by the compiler when variables of these inherently incompatible types are mixed incorrectly. For example, a function with 3 `float` arguments: `age`, `weight`, and `height` could be invoked mistakenly with the arguments out of order.

A `typenew` declaration allows a new number type to be defined that is incompatible with the base type used to declare it, usually the base type is a number type. No default conversions are allowed from or to a variable of such a type. For example:

```
typenew float age_t;
typenew float weight_t;
typenew float height_t;
```

```
// error: incompatible type arithmetic: a += w
void func(age_t a, weight_t w, height_t h) { a += w; }
void use() {
       age_t a = cast(age_t) 23;
       weight_t w = cast(weight_t) 180;
       height_t h = cast(height_t) 6;
       a = 24;          // error: incompatible types
       ++a;             // ok
       func(a, w, h); // ok
       func(w, h, a); // error: incompatible types in
}                       // the three arguments
```

The `typenew` syntax is a subset of the `typedef` syntax, it allows for all kinds of declarations with the exception of function pointers, additionally only one type can be declared at a time.

Types can be declared globally, irrespective of the location of their declaration. Global types can not be hidden by other declarations in non global scopes. Global type declarations are used to declare types that are used pervasively, for example: `int`, `char`, `float`, etc. The syntax for a `typeglob` declaration is identical to the syntax for a `typenew` declaration.

12.10 Types and literal dimensions

To reduce programming errors related to incorrectly used specifications `typenew` and `typeglob` declarations can include a dimension specification that causes them to be unique incompatible types and also allows for literals to be specified of those dimensions, or related to those dimensions, or computed in a way that ensures that the result of the computation is of the correct dimension. A dimension identifier is declared within the curly braces after the base type, in the following example `m` is a dimension specifier for meters and `min` is a dimension specifier for minutes:

```
typeglob float {m} meter_t;
typeglob float {min} minutes_t;
```

The dimension identifier exists in a unique scope of dimension specifiers and its name doesn't collide with other kinds of identifiers. The dimension identifier is global, if declared in a `typeglob` declaration, or local to its scope, if declared with a `typenew` declaration. Dimension identifiers can not be hidden by other dimension specifiers declared in subordinate scopes. A dimension specifier can be specified as a dimension expression based on other dimension identifiers, or literals, and formed with multiplication, division, and parenthesized sub-expressions. For example:

```
typeglob float {km = 1000`meter} km_t;
typeglob float {hr = 60`min} hour_t;
typeglob float {kph = 1`km / 1`hr} kph_t;
```

To specify a literal with a specific dimension, the literal is followed by a back quote

and then by the dimension specifier. For example `60`min` above is a literal that represents 60 minutes and is of the type `minutes_t`. Note that syntactically the back quote is an operator that can only be used with literals, thus there can be space prior and after it, for clarity it is always used without spaces.

The compiler understands the relationships between dimension specifiers that have been defined as a function of other dimension specifiers, and implicitly uses that information to scale compatible units, for example to scale hours to minutes, or vice-versa. For example:

```
minutes_t hr_to_min(hours_t h) { return h * 60`min / 1`hr; }
minutes_t minutes_to_arrive(speed_t s, km_t d) {
    return d / s;       // compiler scales to minutes: (d/s)*60
 // return s / d;       // error: 1/hr incompatible with min
 // return hr_to_min(d / s);  // ok, but conversion not needed
}
```

Dimensionally incompatible expressions cause a compilation error.

12.11 `class` `void`

As mentioned in §5.1 all types inherit, usually indirectly, from `class void`. All types, other than `class void`, descend from these four types:

- `lang.classes` – ancestor to all classes defined by a `class` declaration;
- `lang.number` – ancestor to character, integer, floating point and `enum` types;
- `class void *` – ancestor to all pointer types;
- `lang.arraylike` – base class of `lang.array` and `lang.arraydesc`;

These classes descend directly from `class void`. What these intermediate classes are useful for is explained in the following sections. These intermediate classes when extended enhance their descendants. These descend from `lang.arraylike`:

- `lang.arraydesc` – base class of all array descriptors;
- `lang.array` – base class of all static and dynamically allocated arrays;

12.12 User defined classes descend from `lang.classes`

All classes defined by a `class` declaration that don't inherit explicitly from other classes inherit implicitly from the `lang.classes` class. Classes defined by a class declaration that inherit explicitly from another class inherit `lang.classes` indirectly, i.e. from its base class, `lang.classes` doesn't implement the `lang.value` interface, which results in the non-copying default behavior of classes. This default behavior is different from the corresponding assignment, argument passing, and function value returning C copying behaviors of `struct` and `union`, which are the same in C and

COOGL.

The rationale for user defined classes not having raw memory copy implementations for assignment, value passing, and value returning, is that many classes don't need it, and providing them by default would lead to raw memory copies that might be incorrect for them. If needed, the class designer can implement the copying behavior by implementing the `lang.value` interface.

12.13 Base class of all arrays: `lang.array`

The class `lang.array` is the base class of:

- `lang.carray` – ancestor to all compile time sized array types;
- `lang.dynarray` – ancestor to all dynamic array types;

12.14 Base class of all compile time sized arrays: `lang.carray`

The base class for all compile time sized arrays, also known as C arrays, is `lang.carray`. These arrays can not be initialized from another array, nor assigned from one, nor passed by value as arguments or returned as the value of a function. Arrays that are fields within structures that are assigned to each other, or passed by value, or returned by value, cause the implied array copying required by those operations.

Arrays within structures in COOGL can only contain the subset of types compatible with C, i.e. not user defined classes, this design decision completely separates structures and the layout control that they give programmers and classes, whose layout is under the control of the compiler and whose dynamic allocation and object oriented dispatch mechanisms are not the programmer's concern. Keeping structures and classes separate simplifies the language.

12.15 Base class of all dynamically sized arrays: `lang.dynarray`

Dynamically allocated arrays, see §13, have the number of elements within them determined at run time, not at compile time. The base class for all dynamically allocated arrays is `lang.dynarray`.

12.16 Construction and destruction of `lang.carray` and `lang.dynarray`

Default construction for statically and dynamically sized arrays is allowed only if default construction is allowed by its array element type. Arrays whose element type is initializable, i.e. implements the `initializable` interface can have their elements initialized at array declaration time through the traditional C array initialization syntax:

```
void example(size_t n, type a, type b, type c) require(n >= 3){
    type c[5] = {a, b, c};  // type must be lang.defaultable
    decl type(a) d[n];      // n elements initialized with a
    type e[2] = {a, b, c};  // error: too many initializers
    type f[n] = {a, b, c};  // error without require above
}
```

If the number of elements in the array is larger than the number of values present in the value list, then the additional values in the array are initialized by the default constructor, which the base type of the array must implement, i.e. type must be `lang.defaultable` directly or indirectly, usually through `lang.value`. If the number of elements in the initializer list of a statically sized array is larger than the number of elements in the initializer list a compilation error occurs. If the array is a dynamic array and it is initialized with an array initializer, for example `f[n]` above, a compilation error occurs if the compiler can not prove that the number of elements is greater or equals to the number of elements in the initializer, the proof can be provided by the programmer through a `require()` as shown above. If the number of elements can be larger than the number of elements specified in the initializer and the type is not `lang.defaultable` then a compilation error also occurs.

Destruction is synthesized for arrays just as it is for classes, unless `deinit()` is invoked explicitly on it, for example:

```
class building {
    pub apartment apt[10];
    pub void deinit() { apt.deinit(); }
}
```

12.17 `lang.arraydesc` and `lang.vecdesc` array descriptors

Multi dimensional array descriptors descend from the `lang.arraydesc` class, unidimensional array descriptors descend from `lang.vecdesc`, both are generic classes:

```
namespace lang {
    pub class struct arraydesc(pub genre void type,
                               size_t n) require (n >= 2) {
        pub type *start = NIL;
        pub type *end = NIL;
        pub size_t max[n];
    }
    pub class struct vecdesc(pub genre void type) {
        pub type *start = NIL;
        pub size_t max[1];
    }
}
```

The implementation of unidimensional array descriptors as a specialized type that only uses two fields, instead of 3, is to allow global array descriptors to be updated

atomically faster than multidimensional array descriptors, because most hardware platforms support atomic two-word memory updates.

12.18 Number type interface hierarchy: `lang.number`

Computer representation of numbers is approximate. COOGL numeric types descend from various interfaces that represent various aspects of their number nature. These interfaces are not implemented through COOGL code, they are special interfaces known by the language and implemented natively by the compiler. They exist to allow numeric types to be enhanced through `extend` additions to them, and to unify native C types and interface concepts.

The C behavior that allows fundamental types to be initialized, assigned, passed by value and returned as the value of a function is obtained by their implementation of the `lang.number` interface which implements the `lang.value` interface, compiler generated code implements the `init()`, `deinit()` and `reinit()` member functions.

This hierarchy of interfaces, with `lang.number`, at its top level, and with classes at the bottom, allows various characteristics of numbers to be required by generic classes for their `genre` arguments.

The second level in this interface hierarchy has:

- `lang.sign` - capable of representing negative values;
- `lang.nosign` - not capable of representing negative values;
- `lang.integral` - capable of only representing whole numbers.

The third level in the hierarchy has:

- `lang.whole` - only capable of storing zero and positive whole numbers;
- `lang.integer` - capable of storing positive and negative whole numbers;
- `lang.real` - stores numbers in floating point representation;
- `lang.cmplx` - stores complex numbers in floating point representation;
- `lang.imgnry` - stores imaginary numbers in floating point representation;

The fourth and last level in this hierarchy has five sub-trees of classes, each one containing the COOGL fundamental numeric types:

- `ubyte`, `ushort`, `uint` and `ularge`;
- `byte`, `short`, `int` and `large`;
- `float`, and `double`;
- `complex`, and `complex_double`;

- `imaginary`, and `imaginary_double`;

All of the interfaces in the top two levels of the hierarchy are defined within the `lang` name space. Classes defined in the last level are in the global name space.

A partial declaration showing the interfaces and implementation relationships and a few member functions:

```
namespace lang {
   // root of COOGL numeric interface hierarchy
   // adds arithmetic operators: + - * /
   // adds relational operators: ! == != < <= > >=
   pub interface number { pub is lang.value(number); }

   // 2nd level
   pub interface sign   { pub is number; }
   pub interface nosign { pub is number; }

   // adds bitwise operators:             ~ & ^ |
   // adds checked arithmetic operators: ?+ ?- ?* ?/ ?%
   // adds arithmetic operator:          %
   pub interface integral { pub is number; }

   // 3rd level
   pub interface whole   { pub is integral;  pub is nosign; }
   pub interface integer { pub is integral;  pub is sign; }
   pub interface real    { pub is number;    pub is sign; }
   pub interface cmplx   { pub is number;    pub is sign; }
   pub interface imgnry  { pub is number;    pub is sign; }
}
```

The 4th level of the hierarchy is outside of the `class lang`'s name space:

```
pub class ubyte  { pub is lang.whole; }
pub class ushort { pub is lang.whole; }
pub class uint   { pub is lang.whole; }
pub class ularge { pub is lang.whole; }
pub class byte   { pub is lang.integer; }
pub class short  { pub is lang.integer; }
pub class int    { pub is lang.integer; }
pub class large  { pub is lang.integer; }
pub class float  { pub is lang.real; }
pub class double { pub is lang.real; }
pub class cplx   { pub is lang.cmplx; }
pub class cplxd  { pub is lang.cmplx; }
pub class imag   { pub is lang.imgnry; }
pub class imagd  { pub is lang.imgnry; }
```

The fact that the fundamental types belong to an interface hierarchy has no run time implications in memory use or performance. The `lang.number` type hierarchy exists

to make them no different than user implemented classes, so that they can be used as part of generic programming.

Built in operators such as addition and bitwise-and are introduced in the class hierarchy to allow a generic class that is only applicable to certain number types to be implementable. For example a generic class that only applies to number types that support bitwise operators, for example the `bitmap` class from §11.6 which allows for the specification of the underlying integer type used to store the bits in the set requires that the generic type descend from `lang.whole`.

12.19 Pointers descend from `class void *`

All pointers in COOGL descend directly, or indirectly, from `class void *`. which implements class `lang.value` as if it was declared:

```
class void * {
    pub is lang.value(genre void *);
}
```

A pointer is an object, its members are accessed with the `.` (dot) operator just like any other object member would be accessed given an expression with an object value. Access to the members of an object that a pointer points to is via the pointer member access operator, i.e. `->`. For example:

```
void ex(stack *s) {
    s.print();    // Print the pointer to the stack object.
    s->print();   // Print the stack object.
    (*s).print(); // Print the stack object.
    on (s, *s) print(); // Print the pointer and the object.
}
```

When pointers are treated as objects, their member functions, could be confused by the programmer with member functions of the object that the pointer points to, particularly if both have members with the same name and signature. The wrong member could be called by mistake when `->` is used instead of `.` (dot) or vice versa. Because of this pointers to objects usually are not extended with a `print()` member function to ensure that the object is printed and not the pointer value, which is usually not what is desired.

12.20 Smart pointers and their `priv` member: `ptr`

A smart pointer is a pointer that executes code at pointer construction, assignment, value passing, value returning, and destruction time. Smart pointers are implemented by deriving from pointers. Complete control is provided through the constructor, and these member functions `init()`, `init_default()`, `deinit()`, `init_deinit()`, `reinit()`, and `reinit_deinit()`. By redefining them, appropriate control is pro-

vided for the lifetime of pointers. Smart pointer programming idioms, for example, where reference counting or locking occur at pointer construction time, and reference releasing or unlocking occur at pointer destruction time are easily supported. Control of pointer value use, i.e. whenever data is fetched based on it, is too expensive.

In the following example a simple reference count based garbage collected class `stk` is shown, it tracks all the pointers to its stack objects and forces its objects to be allocated from the heap by making its constructor `prot`. When the last pointer reference to a `stk` object is destroyed, the object is destroyed and the memory released. The `stk` class inherits its implementation from the `stack` class from §4.2.

```
class stk(size_t *n, int *errp) prot pub {stk *} {
    priv pub {stk *} int refs = 0;
    pub inherit stack(n, errp);
    priv is lib.creatable(stk) allocator;
    return;
    pub static stk *create(size_t *n, int *errp) {
        return allocator.create(n, errp);
    }
    priv pub {stk *} void destroy() { allocator.destroy(); }
}
```

Objects of `stk` type are created with `create()`, disposal of `stk` objects is done through their `destroy()` member which is only publicly accessible to the class of pointers to `stk`, i.e. the type `stk *`, a smart pointer in this case. An example use:

```
void use() {
    int err;
    stk *sp = stk.create(20, &err);
    sp->push(1);
    sp->push(2);
    sp->pop();
    sp->pop();
}
```

A smart pointer class is a class declaration that continues the declaration of a class of pointers, it is not a class extension through `extend class`, it is a continuation of the class declaration through `continue class`, see §7.2. All pointer classes have a `prot` member, `ptr`, that can be used to access and change the pointer value without causing `init()`, `reinit()`, etc. to be invoked. Note that the declaration of a smart pointer class causes all declarations of pointers of that class to be smart pointer declarations. The smart pointer class can declare pointers to the class that are not smart pointers by declaring them as `raw` pointers, as shown below in the `equal()` member function of `class stk *`.

The code of `stk *` follows, the type of `this`, within its members is `stk **this`:

```
continue class stk * {
    pub is nilable(class stk *);                    // See §Error:
Reference source not found.
    pub is equalable(class stk *);                  // See §Error:
Reference source not found.
    priv static int conserr;
    priv static stk(1, &conserr) nil;
    priv static stk *nilptr = &nil; // nil.refs is 1
    priv static void init_default(type raw **to) redef {
        to->ptr = &nil.stack;
        ++nil.refs;
    }
    pub void init(stk **to) redef { to->ptr = ptr; hold(); }
    pub void deinit() redef { release(); }
    pub void init_deinit(stk **to) redef { to->ptr = ptr; }
                        // the reference from this is to's now
    pub void reinit(stk **to) redef {
        hold();          // order matters when this == to
        to->release();
        to->ptr = ptr;
    }
    pub void reinit_deinit(stk **to) redef {
        to->release();
        to->ptr = ptr;   // the reference from this is to's now
    }
    priv void hold() { ++(*this)->refs; }
    priv void release() {
        if (--(*this)->refs == 0) {
            assert(!isnil());
            (*this)->destroy();
        }
    }
    pub bool is_nil() redef { return this == &nil; }
    pub void nil_it() redef { *this = nilptr; } //init(&nilptr)
    pub bool equal(stk *raw that) redef {
        return ptr == that->ptr; // raw, non-smart pointer
    }      // chosen for performance reasons, not correctness
}
```

The `stk *` smart pointer class member functions, `is_nil()` and `nil_it()`, test if the pointer is *nil* and to set it to *nil*. The notion of what *nil* means for `stk *` is fully defined by the class. In the example above, it does not correspond to the `NIL` value, but to a dummy static object, `nilstk`, that makes the treatment of *nil* `stk *` objects less of a special case, for example both `hold()` and `release()` can freely increment and decrement `refs` without worrying about the pointer being an actual `NIL` value.

The value of `stk.allocator.create()`, of `stk *s` type, causes during its con-

struction, the reference count of the newly created `stk` object to be 1. The returned value is then used by invoking the member function `init_deinit()` on it when it is returned as the value of `stk.create()`, and again through init_deinit() when it is assigned to `sp` in the `use()` example above. The original reference count now counts as `sp`'s reference count. The more complicated and expensive sequences of invocations of `init()` followed by `deinit()` are avoided. Finally, when `use()` returns, `sp`'s `deinit()` releases the last reference, causing the `stk` object to be destroyed, i.e. it is deinitialized and its memory is released.

12.21 Control during pointer dereference XXX

This requirement is addressed usually by using a handle instead of a pointer, which allows a smart pointer to be constructed from the handle, and causes locks to be acquired, or objects to be held, etc. Destruction of the smart pointer causes the unlock, or object reference to be released, etc.

12.22 Explicitly declared classes and smart pointer restrictions

The language allows objects from an explicitly declared class, including a smart pointer class, to be used in certain expressions, depending on which member functions are defined:

- Use as a function argument or return value, if `init()` is defined.
- Use as a value that is used in the initialization of an object of the same type, if `init()` is defined.
- Use as a value that is assigned to another object of the same type, if `reinit()` is defined.

These operations are always valid:

- Obtain their address with the address-of operator, i.e. `&`.
- Invoke member functions on it.

All other uses are invalid, including:

- Use as a value in an explicit comparison or relational expression, i.e. `==`, `!=`, `<`, `<=`, `>`, or `>=`.
- Use as a value in a conditional expression context, where it would need to be determined if its value is `NIL` or not, i.e. when used as the value tested in an `if`, `while`, `for`, or `loop` statement or by the `&&`, `||`, `!` or `?:` operators.
- Use as a value in arithmetic, pointer arithmetic, or bitwise expressions, i.e. `++`, `--`, `+`, `-`, `*`, `/`, `%`, `~`, `&`, `|`, `<<`, or `>>` ; or their related assignment-operation expressions: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `<<=`, or `>>=` ; or their corresponding

checked operators: `?+`, `?-`, `?*`, `?/`, or `?%` ; or their corresponding assignment checked operators: `?+=`, `?-=`, `?*=`, `?/=`, or `?%=`.

These restrictions were designed into the language to completely encapsulate and prevent misuse of smart pointers. Use of the pointer indirection operators: `*` and `->` are, of course, allowed with smart pointers. The ability to redefine the other operators doesn't seem to carry its weight in a language that aims to be simple, that ability is not present as it would introduce operator overloading and its complexity.

Examples of invalid expressions are:

```
void push_stk(stk *src, stk *dest)
        require(dest != NIL) { // error: smart pointer compared
    if (!src) return;          // error: smart pointer NIL test
    ++src, --src;              // error: arithmetic on object
    // avoid infinite loop:
    assert(src != dest);       // error: smart pointer compared
    while (!src->empty()) {
        assert(!dest->full());
        dest->push(src->pop());
    }
}
```

Idiomatically `is_nil()`, `nil_it()`, and `equal()` are provided by smart pointers by implementing the `equalable` and `nilable` interfaces:

```
namespace lib {
    pub interface nilable(genre void type) {
        pub void nil_it() defer;                // make it NIL
        pub bool is_nil() defer;                // is it NIL?
    }
    pub inteface equalable(genre void type) {
        pub bool equal(type *raw that) defer; // this == that?
    }
}
```

The `push_stk()` function can be written as:

```
void push_stk(stk *src, stk *dest) require(!dest.is_nil()) {
    if (src.is_nil()) return;
    assert(!src.equal(dest));       // avoid infinite loop:
    while (!src->empty()) {
        assert(!dest->full());
        dest->push(src->pop());
    }
}
```

13 - Variable length and dynamically allocated arrays

In spite of this advice, C99 adopted the scheme:

"The rules for both the GCC and MacDonald schemes are difficult to use and comprehend, and are difficult to formalize even to the level of the current ANSI-standard; in particular, the type calculus for variable-sized arrays is murky for both. In the existing ANSI-C language, the type and value of an object p suffice to determine the evaluation of operations on it. In particular, if p is a pointer, the code generated for expressions like p[i] and p[i][j] depend only on its type, because any necessary array bounds are part of the type of p. In the MacDonald and GCC extensions, the values of non-constant array bounds are not tied firmly to its type."

-- Dennis Ritchie

COOGL supports arrays whose dimensions are determined at run time, array dimensions are members of the array. COOGL support is different from the variable length array support of C99, because it is too complex, and its use is error prone.

13.1 Variable length arrays

Variable length arrays were added to C in C99, the second official C language standard and third since the de-facto K&R C standard. Variable length array support is, by far, the most complex extension made to C as part of the C99 standardization process.

The problem that variable length arrays addressed in C99 was the lack of support in C89 for multidimensional arrays whose dimensions are only known at run-time. Given that C has been used mostly for systems programming, this limitation caused little or no trouble, but it is one of the principal reasons that prevented the use of C for numeric programs which were, and sometimes still are, written in FORTRAN instead of C.

13.2 `v[]` declaration syntax in C

In C the meaning of the `type v[]` declaration syntax has changed with C's evolution, the meaning depends on the context of the declaration.

A global declaration of the form `type v[]`:

```
/* Global declaration, it is an external declaration of v
   a uni-dimensional array of unknown size. */
int v[];                          /* Valid in K&R-C/C89/C99/C11 */
```

An argument declaration of the form `type v[]`, is equivalent to an argument declaration of the form `type *v`:

```
/* Argument declaration, equivalent to: void f(int *v) {...} */
void f(int v[]) {...}             /* Valid in K&R-C/C89/C99/C11 */
```

The declaration of the last member of a structure, of the form `type v[]`:

```
/* Declaration of v[] as the last structure member.
   Invalid in K&R-C/C89.
   Valid in C99/C11, it is a flexible array. */
struct s1 { int b; int v[]; }
```

To reduce complexity and for language safety reasons COOGL does not support C99/C11 flexible arrays, nor does it support, zero sized array members as the last member of a structure.

A structure member declaration, that is not the last member, of the form `type v[]`:

```
/* Declaration of a[] not as the last structure member: */
struct s { int v[]; int b; } /* Invalid in: K&R-C/C89/C99/C11*/
```

A local variable of the form `type v[]`:

```
/* Local variable declaration: */
void function() { int v[]; } /* Invalid in: K&R-C/C89/C99/C11*/
```

The various contexts in which a declaration of the form `type v[]` can appear in C code, and their validity, are summarized in the table below:

Is `type v[]` Declaration Valid?		K&R-C C89	C99 C11
Context	**Example**		
global	`int v[];`	yes	yes
argument	`void f(int v[]) { ... }`	yes	yes
last member	`struct s { int i; int v[]; }`	no	yes
not last member	`struct s { int v[]; int i; }`	no	no
local	`void f() { int v[]; ... }`	no	no

13.3 `type v[][]` declarations are always invalid in C

Irrespective of context, array declarations with two or more dimensions with un-

specified sizes are invalid in C. For example:

```
int a2d[][];   /* all of these declarations are invalid in C */
int a3d[][][];
int a4d[][][3][4];
int a5d[][2][][3][4];
```

13.4 Variable length arrays in COOGL

C99 stretches C compile time casts into run time casts that include array dimensioning cast expressions evaluated at run time, COOGL does not follow that baroque design. instead it provides support for variable length arrays in a simpler way. Variable length array declarations in COOGL are a kind of entity that is different from the traditional C statically dimensioned arrays. Nonetheless, statically dimensioned arrays can be used as arguments to functions that expect variable length arrays.

In COOGL the number of elements in each array dimension can be obtained from the array object. The statically dimensioned array member `max[K]` provides the number of elements in each of the `K` dimensions of the array.

Variable length array matrix multiplication in COOGL:

```
void multiply(float a[][], float b[][], float r[][]) {
    // r[I][J] = a[I][K] * b[K][J]
    index I = a.max[0], K = a.max[1], J = b.max[1];
    expect(r.max[0] == I && r.max[1] == J &&
           a.max[0] == I && a.max[1] == K &&
           b.max[0] == K && b.max[1] == J);
    for (index i = 0; i < I; ++i)
        for (index j = 0; j < J; ++j) {
            float t = 0;
            for (index k = 0; k < K; ++k)
                t += a[i][k] * b[k][j];
            r[i][j] = t;
        }
}
void use(index n, index m) {
    float data[n][m], trans[m][n], result[n][n];
    get_data_and_trans(data, trans);
    multiply(data, trans, result);
    print_result(result);
}
```

Local array variables declared with run-time expressions as their dimensions are variable length arrays. Variables declared with array declarators without dimensioning expressions are array descriptors. For example the `a[][]`, `b[][]`, and `r[][]` are array descriptor arguments of `multiply()`. Local variable length array variables within `use()` are: `data[n][m]`, `trans[m][n]`, and `result[n][n]`.

Arrays of array descriptors and array descriptors of arrays are not supported, even though alluring from a language orthogonality perspective, they have little value and don't merit the complexity that they would add to the language. The declarations of `a[n][m][]`, `b[n][][]`, and `c[][n][m]` are all invalid:

```
void f(index n, index m) {
    float a[n][m][];     // error: array of array descriptors
    float b[n][][];      // error: array of array descriptors
    float c[][n][m];     // error: array descriptor of arrays
}
```

In C the `[]` declarator is most commonly used in argument declarations, the other two cases mentioned above in §13.2 are not as common. Because in C the `[]` declarator when used in an argument declaration is a synonym of the `*` pointer declarator, there is no language level semantic difference between them, though it is sometimes used as a visual cue to the programmer that the pointer in question is actually the address of the first element in a unidimensional array. For example, these two C declarations of `vector` in `sum()` are equivalent in C:

```
float sum(int n, float vector[]) { ... }
float sum(int n, float *vector) { ... }
```

The declaration of `m[][]` is an invalid argument declaration in C:

```
/* error: array type has incomplete element type  */
int f(int m[][]) { ... }
```

In C, the number of elements can only be omitted from the first array declarator, for example:

```
int f(int m[][20]) { ... }
```

Which is equivalent to:

```
int f(int (*m)[20]) { ... }
```

In COOGL, arguments declared with `[]` are array descriptor arguments, arguments declared with `*` are a pointer to a single object, not a pointer to an element within an array.

To provide C source code compatibility and *some* run-time calling convention compatibility, a function that takes arguments declared with a single empty `[]` passes a pointer to the first element of the array in that argument. An extra hidden argument, is passed in addition to the function's arguments specified in its signature with the array descriptor's value of `max[0]`, see §2S.6.

In the declarations of `multiply()`'s arguments, above: `a`, `b`, and `r`, are array descriptors, they are objects, not pointers to objects, the references to the dimensions, for example `a.max[0]` are thus in the form `object.member`.

When native C arrays, or variable length arrays, are passed to `multiply()`, as occurs in `use()` above, it is the compiler's job to create array descriptors and pass those

by value. The memory for the underlying array elements is not contained within the array descriptors, thus even though the array descriptors are passed by value the net effect is that the underlying elements of the array that was the input argument is what is referenced or affected by the function.

A source of common bugs is removed by having the array dimensions be part of the array descriptors instead of passing them explicitly as additional arguments, as occurs in C99. Given that the relationship between the dimensions of the arrays is not knowable by the compiler, the `expect()` in `multiply()` validates that the array arguments are valid with respect to each other.

The number of entries in an array or in an array descriptor is `total`, i.e. the result of multiplying: `max[0] * max[0] * ... * max[N-1]`. For example:

```
void add(float a[][], float b[][], float r[][]) {
    // r[I][J] = a[I][J] + b[I][J]
    index I = r.max[0], J = r.max[1];
    expect(a.max[0] == I && a.max[1] == J &&
           b.max[0] == I && b.max[1] == J);
    float *ap = a.start; // same as: ap = &a[0][0];
    float *bp = b.start;
    float *rp = r.start;
    float *endrp = rp + r.total;    // example of of total
 // float *endrp = r.end;           // this is the same
    while (rp < endrp) *rp++ = *ap++ + *bp++;
}
```

13.5 Idiomatic error setting by constructor and arrays of objects

Class `stackx` inherits from `stack`, its constructor only sets `*errnop` if an error actually occurred, otherwise it is left unchanged. This allows users of `stackx` to create multiple stacks and only check the accumulated construction error.

```
class stackx(size_t max, int *error) {
    int e;
    pub inherit stack(max, &e);
    if (e) *error = e;
}
void use() {
    int error = 0;
    decl stackx(10, &error) stk2d[50][50];
    if (error) return;
    for (int i = 0; i < stk2d.max[0]; i++)
        for (int j = 0; j < stk2d.max[1]; j++)
            stk2d[i][j].push(i + j);
}
```

The idiomatic setting of `*error` only when errors occur allows for the checking of

construction errors for the whole array to be done more easily.

13.6 Restrictions on array descriptors and variable length arrays

C99 extends K&R and C89 casts, `sizeof`, and array declarations with run time behaviors, to support variable length arrays. The COOGL extension is simpler, the C89 core is left unchanged in these areas. In C99 casts can be used to mutate memory into an array whose dimensions are only known at run time, the use of casts for this is needlessly obscure, that syntax is not supported in COOGL, the reinterpretation of memory as arrays of various dimensions is done through array descriptors, see §13.7.

To make the use of array descriptors as simple as possible and prevent programming errors, these restrictions exist on them:

- Use of `sizeof` on an array descriptor or on a variable length array is invalid, `sizeof(a)` produces a compile time error to prevent confusion between the size of the array descriptor and the size of the underlying array itself.
- The address of an array descriptor, or of a variable length array, cannot be taken, they are strictly value objects. To pass them as arguments to a function an array descriptor argument receives a copy of the array descriptor.
- The dimensions of a variable length array are set at construction time, the array dimensions can not be changed.

The variable length arrays and array descriptors in COOGL are close to Dennis Ritchie's proposal for a variable length array extension to C. Ritchie's proposal was not adopted for C99, even though Ritchie explained the problems and complexity in GCC's and MacDonald's proposals, they were the base for what was eventually adopted by C99.

13.7 Array memory reinterpretation

The declaration of a variable length array causes the allocation and construction of the underlying memory for the array elements. Array descriptors allow for the underlying memory to be associated with it at a later time. The underlying array entries of an array can be reinterpreted by an array descriptor, for example to refer to fewer elements, or to have a different number of dimensions. In the example below the `b[2][3][5][7]` array descriptor refers to the memory of `a[10][21]`.

Note the last two array descriptor arguments of `lib.array.make()`, the last one is generic based on the first argument. Passing `a.start` there causes it to be received as a unidimensional array descriptor thus its `max[0]` can be used to ensure that the array descriptor that is requested does not give access to memory outside the array.

```
void use() {
    int a[10][21];
    int b[][][][] = lib.array.make(int, {2,3,5,7}, a.start);
}
```

The syntax `{2,3,5,7}` argument to `lib.array.make`, above, is an array initializer passed as an argument and received in an array descriptor argument, the array element's type must be compatible with the type of the array elements of the corresponding argument. In C a cast would have been required, because in C the array expression can occur in subexpressions thus its type can not be determined in general, from the context of where it occurs. This divergence from C and non-silent incompatibility might be addressed in the future, but there is very little use for it in practice.

13.8 Dynamic creation and destruction of arrays

Support for allocating the underlying memory for an array on the heap, when an array needs to be allocated dynamically, even if the array doesn't require construction or destruction, is provided by `lib.creatable`, through `create()` and `destroy()` member functions added as an extension to the array descriptor of the specified type.

An example of the use of `create()` and `destroy()` on array descriptors:

```
float create_and_init_matrix(size_t n, size_t m)[][] {
    float a[][], b[][], r[][];
    if (a.create({n,m}) && b.create({m,n}) && r.create({n,n}))
        work(a, b, r);
    a.destroy(), b.destroy();
    return r;
}
```

the `{n,m}` array is the `dims[]` array descriptor argument to `create()`, see below.

The invocation of `destroy()` is valid even if `create()` failed, or if it was never invoked, `destroy()` must be invoked explicitly, the destruction of `a` and `b` does not invoke `destroy()`, because the underlying memory for the array might still be referenced by other array descriptors, as is the case with `r`, the array data it references is returned as the value of the function by returning a copy of the `r` array descriptor.

Invoking the `create()` member function is invalid, and raises an exception, if the array descriptor already refers to some memory. The `type.argsof` tuple type, the argument list of the constructor of `type`, is used to declare the argument list of `create()`, a non-static member functions added to the array descriptors for `type` by the `lib.creatable` interface.

When a type is extended within some scope, or namespace, the type extension is always global, not scoped. Thus a generic interface such as `lib.creatable` can extend related types, array descriptors in this case, appropriately. The `unsafe_cast()` operator and `uninit()`, used below, are described in §14.21 and §14.23.

```
pub namespace lib {
    pub interface creatable(pub genre void type,
                            pub lit uint extra = 1,
                            pub lit bool uninit_extra = false)
                                        require(extra <= 2) {
        // rest of lib.creatable is in §11.10
        extend class lang.arraydesc(genre void type) {
            pub bool create(size_t dims[],
                            decl type.argsof args)
                                    require(!start &&
                                            dims.max[0] > 0) {
                size_t total = 1;
                bool overflow = false;
                size_t *d = dims;
                do {
                    size_t sz = *d++;
                    expect(sz > 0);
                    overflow |= total ?*= sz;
                } while (d < dims.end);
                expect(!overflow);
                lib.object.array_alloc(type, this, extra,
                                       total, dims);
                if (!start) return false;
                type raw *p = unsafe_cast(type raw *) start;
                type raw *prior = p - 1;
                type raw *after = unsafe_cast(type raw *) end;
                for (; p < after; ++p) type(args, p);
                if (!extra) return true;
                if (uninit_extra) {
                    if (extra == 2) type.uninit(args, prior);
                    type.uninit(args, after);
                } else {
                    if (extra == 2) type(args, prior);
                    type(args, after);
                }
                return true;
            }
            pub void destroy() redef {
                if (!start) return;
                for (type *p = start; p < end; ++p)p->deinit();
                lib.object.array.free(this);
            }
        }
    }
}
```

13.9 Array descriptors and polymorphism

Array descriptors, in general, are not polymorphic, the types of the underlying objects are known, with the relaxation that if the objects have the same exact size, then polymorphism is allowed. The types of all the elements in the array are the same, but they can be of a type that descends from the type of the base element of the array descriptor, for example, an array of objects of a type can be passed as an argument and received by an array descriptor whose type is an ancestor of the array's type. The restriction on the size is required because indexing of array descriptors, or walking them with pointers, is based on the size of the underlying elements of the array, they must be known at compile time, not at run-time. Even though this restriction could possibly be relaxed, it doesn't seem to merit the language complexity of doing so.

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

14 - Safe programming

"The first principle was security: The principle that every syntactically incorrect program should be rejected by the compiler and that every syntactically correct program should give a result or an error message that was predictable and comprehensible in terms of the source language program itself. Thus no core dumps should ever be necessary. It was logically impossible for any source language program to cause the computer to run wild, either at compile time or at run time. A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency on production runs. Unanimously, they urged us not to--they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law."

-- C.A.R. Hoare

Memory safety ensures that incorrect memory accesses do not occur. Most C and C++ programs have bugs that cause incorrect memory accesses. COOGL programs do not contain invalid memory accesses, the language prevents them by design.

14.1 Safe programming

Memory safety ensures that invalid memory accesses do not occur, but without a definition this is no more than a loose concept. This chapter is organized in a bottoms up manner, first low level concepts and mechanisms are explained, incrementally

building up to an explanation at the end of the chapter about COOGL safe programming and a precise definition of *invalid memory access*. Preventing invalid memory accesses prevents a whole class of security weaknesses that are frequently exploited.

C shines in its ability to manipulate memory in any way that the programmer thinks is appropriate, irrespective of whether the memory accesses make sense or are completely wild. The success of C as the systems programming language of choice, over other programming languages (ALGOL60, ALGOL68, PL/1, Pascal, Ada, Modula II, etc.), can be partially attributed to its ability to allow programmers to do whatever they want, without the language getting in the way.

The most difficult design aspect of COOGL was to preserve the ability of C to manipulate memory, while ensuring that the memory manipulation is safe. This need drove the design choices of its approach to memory safety.

14.2 Modern computer system hardware

Modern computer systems organize memory as a flat relatively clean address space, with the units of memory addressing being the 8 bit byte, all of them implement integers as two's complement arithmetic. Long gone are the days of computer systems with one's complement arithmetic; with only word addressable memory; with 12, 16, 18, or 24 bit addresses and words, and 6, 7, or 9 bit characters. Gone also are the days of non-flat, segmented, and possibly capability based, addressing schemes (the IBM iSeries system being the only surviving system with a hardware supported capability memory system). Simplicity won over baroqueness, and only backwards compatible leftovers of segment based addressing remain in the x86/64 system computer architecture, they are mostly ignored, or only used by a tiny amount of system software.

COOGL is a language for modern systems, some of the restrictions in the C language definition that attempted to make accommodations for these defunct computer architectures are not required to be present in COOGL, which is an evolution of C, not an accumulation of additional features on top of it.

Hardware supported memory segments for secure sharing of large amounts of data, or to implement protected subsystems, within clean flat address spaces, for example as provided by the POWER architectures, has been pushed under the flat address space and into the realm of the operating system kernel for their management, where they belong, instead of in the hands of every programmer and every programming language and its compiler, as was done in the ancient segmented architectures.

Modern computer system architectures place data type placement restrictions on the various data types supported by them, or at least make strong performance recommendation about it. Usually native data type entities must be located in memory in addresses that are a multiple of their size, this address *alignment* requirement allows the data item to be accessed as efficiently as possible. For example, 64 bit floating

point numbers stored in addresses that are multiples of 8 (eight 8-bit bytes = 64 bits), or 32 bit integers stored at addresses that are multiples of 4. Hardware designed with these data placement restrictions is simpler to implement than hardware that operates without them. For example, access to a 32 bit integer that crosses a cache line boundary and a virtual memory page boundary is much more complicated than an aligned 32 bit integer access, which would never cross those boundaries. Most modern computer architectures raise an alignment exception when an unaligned access is performed. Certain legacy architectures don't cause alignment exceptions, but their unaligned accesses can be slower, sometimes much slower, than an aligned access. Unaligned data item accesses are not prevented by COOGL, if the access results in a hardware exception, the exception is delivered to the program, see §14.33.

Input and output of characters, integers, and floating point data, in binary form, is required by applications. Whether the data being read is actually well formed for those data types, is a concern for the application, not for the COOGL programming language because such improperly formed data can not lead to invalid memory accesses. For example, an application that expects that the characters that it reads be ASCII characters, valid UTF8 strings, or that its floating point values are valid, can choose to validate them prior to their use, or assume that they are valid and simply allow the program to misbehave if made to operate on an invalid data file.

COOGL does not assume that the computer system includes a Memory Management Unit (MMU) to translate between a virtual address space and a physical address space. If such an MMU exists, it usually does, then some specific implementation approaches to COOGL safe programming take make use of it, either through the memory mapping interfaces provided by modern operating systems, or by directly making use of the MMU in its run time language support, for example in an operating system kernel, a hypervisor, firmware, or some other low level software that runs without the support of an operating system. Systems without an MMU, or with a a very primitive address validation scheme (for example a few bounds and/or mapping registers) use other implementation approaches to achieve COOGL safe programming.

14.3 Safe programming approach

A way to think about safe programming languages is that, when a problem occurs with the program, the problem can always be fully investigated and understood at the programming language level. There is never a need to examine the state at the machine level, for example, the programmer never has to examine a corrupted run time stack, computer register values, instruction sequences, and the machine instructions that the program was translated into. For example, to understand the nature of a runaway program that ended up crashing after executing some arbitrary data as if it were instructions, as occurs in C and C++. The programmer only debugs logic errors that are fully understandable at the programming language level, not at the machine level.

Most safe programming languages include automatic memory management, i.e. garbage collection, as a means to ensure that an object's memory not be reused if the underlying memory where the object is stored could still be referenced through a pointer that is still accessible by the program; and conversely that memory that is no longer accessible can be eventually reused for other purposes. Some safe programming languages contain substantial run-time systems: virtual machines, just-in-time code generators, language interpreters, and large libraries, written in unsafe languages, problems in those bodies of code, can not be debugged as logic errors at the language level. The larger the run-time code written with an unsafe programming language the less safe the language is as a whole.

The approach to safe memory management used by COOGL does not mandate garbage collection, instead memory management remains in the control of the programmer, but in a safe way. General purpose or custom garbage collection can be implemented by an application if they choose to do so. By providing minimal mechanisms in the language, the programmer can choose between traditional explicit memory management, general purpose garbage collectors, or allocators specialized for the application that offer garbage collection like behavior, without the costs of general purpose garbage collection.

There is little value in a new language that evolves C, but that in its evolution causes C's rich memory manipulation abilities to be removed, or to become so crippled so as to become unusable as an evolutionary path for C code. COOGL's approach to safe programming walks a careful design balance of preserving the value and efficiency of C's memory manipulation while ensuring that the memory manipulation is safe and efficient. This chapter explains how that is done.

It is important to emphasize that garbage collection is not a panacea for programming, it prevents certain errors, but leads to other kinds of errors. For example, if the programmer is not careful enough to ensure that data, when not longer needed, is not referenced through accessible pointers, then the memory never is reused, which slowly but surely leads to the program's memory needs to grown continuously, eventually leading to the program thrashing the underlying virtual memory system, or failing in other ways when memory allocations start to fail unexpectedly. Systems that depend on garbage collection tend to require a much larger amount of memory than systems that don't require garbage collection, requiring 1.5 to 2 times as much is not unusual.

Finally, it is also important to realize that one size fits all solutions are limiting and when implemented by the language itself, rob the programmer from the possibility of implementing alternative approaches to safety that might be more appropriate, smarter, performant, safer, than the approach chosen by a language, including the approach chosen by COOGL. Thus only a small amount of mechanism is implemented and defined by COOGL. Through the use of `preclass` inheritance (see §14.30) and

pointer life cycle control (i.e. smart pointers), alternative approaches can be implemented without having to modify the compiler or change the language definition.

14.4 Bad memory accesses in C

In C it is easy to reference memory that should not be accessed.

Run time stack smashing in C:

```
void wrong() { char x[1]; x[200] = 'x'; }
```

More stack smashing:

```
void store_1_at_index_100(int *p) { p[100] = 1; }
void wrong() { int v = 1; store_1_at_index_100(&v); }
```

Abridged version of classical `malloc()` and `free()` problem in C:

```
void stomp() { char *p = malloc(10); free(p); *p = 'x'; }
```

Malformed string causing unrelated memory to be affected in C:

```
char s[] = {"hi"};
char d[3];
void stomp() {
    s[2] = 'x';
    strcpy(d, s);
}
```

Pointer smashing through `union` in C:

```
void smash() {
    char c;
    union { char *p; int i; } u;
    u.p = &c;
    u.i = 17;   /* smash the u.p pointer */
    *u.p = 'x'; /* use smashed pointer */
}
```

All of these programs are incorrect, they are abridged versions that present the essence of programming problems found in many large C programs.

A more complex variation of stack smashing in C:

```
char *set(char *p) { *p = ' '; return p; }
char *bad() { char c; return set(&c); }
void store(char *p) { *p = 'x'; }
int main() { char *p = bad(); store(p); }
```

The address of the local variable `c` is only valid while `bad()` is active, when `bad()` returns, the address of `c` is returned, i.e. `p` in `main()` points to a variable, `c`, whose function, `bad()`, is no longer active in the run time stack. When the value of `p` is passed to `store()` it could easily point to something that should not be altered, for example the return address of `store()` within the function call run time stack.

C compilers are incapable of reporting an error for this class of code, particularly if the functions are separately compiled, all of it is valid C code, programmers are supposed to know what they are doing. For example, if the programmer is doing some intelligence agency's biding, similar code might be written as a hidden security hole to exploit later as a backdoor to take over the system.

14.5 Plain and non-plain data and types

Variables of the base types: character, floating point, and integer types (with the exception of `index` and `uindex`) are *plain data*. Pointers, array descriptors, variable length arrays, and index variables (of `index` or `uindex` types) are not plain data. Structures, unions, and traditional C arrays that only contain plain data are also plain data. Plain data are entities that only contain, directly, or indirectly, other entities that are also plain data. For example, traditional C arrays of plain data, and structures and unions whose members are all plain data, are plain data. The definition of plain data is recursive, allowing for traditional C arrays of structures with traditional C array members, and so on, that are plain data to be plain data. The definition is also intuitive, simply meaning that there are no pointers, no indexes, no array descriptors, and no variable length arrays anywhere within an object that is plain data. The data described by an array descriptor or contained in a variable length array can be plain data, and manipulated as such, the data, the array elements, remain plain data, even though the array descriptor and variable length array are not plain data. A complementary concept, *non-plain data*, refers to any data that is not plain data. Two related definitions are *plain data types* and *non-plain data types*; they are the types of objects that are plain data and objects that are non-plain data, respectively.

Data of a type declared by the programmer, or the language, in a `class` declaration is never plain data, irrespective of whether or not it contains pointers, indexes, array descriptors, or variable length arrays. Even though built in types, `int`, `float`, etc. can be thought of as being of a class type, they are not actually declared in a class declaration, they are plain data, the fact that they can be extended through `extend class` does not affect their treatment as plain data.

These are non-plain data types:

```
typedef byte *byteptr_t;
struct bytebuf_t { size_t size; byteptr_t mem; };
struct range_t { index start; index end; };
class point { pub float x, y; }
```

These are non-plain data:

```
byteptr_t bp;
bytebuf_t bb;
range r;
point p;
int *ip;
```

These are examples of plain data types:

```
struct dirent_t {         // UNIX v6 directory entry
    ushort inum;
    char   name[14];
};
```

```
lit size_t DATABUF_SIZE = 512;
struct databuf_t {
    char   data[DATABUF_SIZE];
};
struct dirbuf_t {        // a databuf full of directory entries
    dirent_t de[sizeof(databuf_t) / sizeof(dirent_t)];
};
```

These are plain data:

```
dirent_t de;
dirbuf_t db;
```

A class can not have the address of its data members, whether they are of a plain data or not, to be used in such a way that their addresses, directly or indirectly in other functions, would end up being used with the `cast()` or the `try_cast()` operators. Any such use causes a compilation error, see §14.12.

A consequence of not allowing any of the memory within a `class` to be manipulated in ways that other plain data can be manipulated is that this forces a complete separation between `class` and `struct`, and the compiler can be allowed to perform more aggressive alias analysis and optimizations for memory accesses that relate to a `class` than those that relate to a `struct`, at least for the plain data parts of a `struct`. The compiler can be as aggressive as it is for classes with the non-plain data members of a `struct`.

14.6 Insight for safe, C style, memory manipulation in COOGL

The rich memory manipulation of C allows simple and efficient organization and placement of data in memory in whatever way that is required, without the language getting in the way of doing so. Usually the data placement has been defined elsewhere and the programmer is not at liberty of choosing a different organization for it. For example, data to communicate with other computer systems or devices, such as data associated with network and storage systems, network communication protocols, distributed file systems, distributed transaction coordinators, file formats, database engines, file system metadata, volume manager metadata, etc.

Externally imposed memory layouts share the common characteristic that they are meaningful outside of the computer system, or at least across unrelated processes. Pointers, array descriptors, and variable length arrays (organized in a programming language mandated way) are not included in such layouts because they would be meaningless in them. The externally defined memory layouts can all be thought of as being plain data. Even if the plain data contained various variable length components to it, they often do, the actual description of such data, its size, its location, would be part of some external specification, described with other plain data for example offsets within the data, explicit or computed, not with pointers or a language specific

representation of variable length arrays nor with array descriptors. This is the **key insight on which safe programming in COOGL is based**: C's rich memory manipulation is needed, almost exclusively, when detailed control of externally defined memory layouts is required, and those only contain plain data.

Other circumstances under which C's memory manipulation is used, are much less important, and don't occur as often, for example to implement memory allocators. This use is supported by COOGL through unsafe code, but are not required to implement most programs.

Some other infrequent uses that require C's rich pointer manipulation include being able to determine from a pointer within a memory area the base of the memory area, for example from a pointer to a field within a structure to compute a pointer to the structure (supported in a safe way in COOGL by `field_to_obj()`, see §1L.2); or from a pointer to a structure to find another structure that has been placed immediately prior to it; or from a pointer to an object within an area to find control information placed near it, for example by clearing a number of low bits within the pointer to compute the pointer to the control information. Most of these uses are uncommon enough that they don't need to be supported by safe code.

Indexing with plain data is unsafe, the plain data could be addressable from elsewhere, type based alias resolution could lead the compiler to assume that the index has not changed by intervening stores and cause the index to be refetched after it has been validated and cause an unchecked out of bounds array reference. Thus `index` and `uindex` are not plain data. Indexing with plain data is only allowed when the compiler can prove that the specific data item has never had its address taken, for example when indexing with a locally declared `int` variable.

The types `index` and `uindex` are non-plain data types, their size depends on the underlying system (for example 32 vs 64 bit memory addresses), externally imposed layouts should never include system specific types. Indexes are used to index into arrays and to perform pointer arithmetic. Because they are not plain data, they are safe from being affected in unexpected ways by code creating external memory layouts through COOGL's rich memory manipulation means. If the external memory layout is to be used only within the same system, for example through some shared memory, then the `ssize_t` and `size_t` types can be used, they are plain data, instead of `index` and `uindex`.

14.7 Unions can't contain indexes, pointers, or array descriptors

COOGL unions are not allowed to contain members of type `index` or `uindex`, pointers, or array descriptors, directly or indirectly. Thus all COOGL unions are plain data, this is a restriction compared to C. COOGL is an evolution of C, not a superset.

14.8 Global memory can't refer to memory on the run-time stack

Global memory are variables declared globally, or anywhere with `static`, or memory allocated dynamically from the heap. Run-time stack allocated memory is the memory that contains locally declared variables (non-`static` ones), function arguments, or memory that was allocated with `alloca()`.

The term *refer* in this section means that it *points to* memory (i.e. it is the address of the memory), or that it is an array descriptor value that can be used to access memory (i.e. it refers to elements within an array). The term *address range* is used in this section to mean an individual address value that refers to a single data item, or an array descriptor value that refers to multiple contiguous data items.

Global memory can not refer to memory allocated on the run-time stack, this is guaranteed by the language, attempting to cause global memory to refer to run-time stack memory causes a compilation error. The requirements listed in this section are imposed by the language, and enforced by the compiler, they aid in the implementation of this property.

An address range that refers to data within run-time stack allocated memory, can not be used other than to:

- Access the underlying memory.
- Store the address range into a variable that resides on the run-time stack. The destination variable must be declared after the variable whose data the address range refers to, this ensures that the lifetime of the variable is shorter than the lifetime of the data it refers to. The destination variable must meet the restrictions on variables that refer to data on the run-time stack, described below.
- Pass it as a non-member argument to a function, but only if the argument meets the restrictions on variables that refer to data on the run-time stack.
- Call a member function on an object that the address range refers to, but only if uses of `this` in the member function meet the restrictions on variables and addresses that refer to data on the run-time stack.

When an address range that refers to data within run-time stack allocated memory is used for any other purpose than those listed above, a compilation error occurs. When used to call a function or a member function that doesn't meet the requirements enumerated above, it causes a compilation error in the calling location, not in the called function's code itself.

The function and member functions mentioned above, have their names adjusted with compiler generated information that indicates that the function meets the restrictions, and for which of their arguments it meets them.

A function argument or a `this` object pointer, whose value refers to memory on the run-time stack, or an expression whose value was derived from the argument's value, including through the use of local variables that refer to that memory, can be returned as a value by the function in a safe way, see §14.9 for details about this.

Run-time stack allocated variables, function arguments, and `this` pointers that at any time might refer to data on the run-time stack can only be used to:

- Access the memory that they refer to.
- Use their value in pointer tests and comparisons.
- Pass their value to other functions or call member functions on them, but only if the corresponding arguments or `this` pointers meet the restrictions in this list, because they will refer to run-time stack allocated memory.
- Obtain an address range that refers to memory within the memory that they refer to. The address range is subject to the restrictions listed above on address ranges that refer to run-time stack allocated memory. The address range can be assigned to a run-time stack allocated variable which must meet the restrictions enumerated on this list. If a local variable was the source of the address range used to derive the assigned address range, and the source variable could be referring into run-time stack allocated memory allocated by the same function, then the variable into which the derived address range is assigned must have been declared after the location where the run-time stack allocated memory was allocated, this ensures its lifetime ends prior to the lifetime of the run-time stack allocated memory.

These restrictions imply that:

- The value of run-time stack allocated variables, function arguments, and `this` pointers, that could possibly refer to run-time stack allocated memory, can not be stored in the members of local variables that are structures, class objects, or arrays, or into any global memory.
- An address range that refers to run-time stack allocated memory can not be used used as an argument to a function, or to call a member function on it, if the invocation of the function or member function, could lead to an address range that refers to the stack allocated memory to be stored in the members of local variables that are structures, class objects, or arrays, or into any global memory.
- Address ranges that refer to run-time stack allocated memory only exist as non-member arguments to functions, run-time stack allocated variables within functions, or the value of `this` within functions, and their lifetimes end prior to the lifetime of all the run-time stack allocated memory that they might have ever referred to.

Traditionally, in C, the address of local variables are used as arguments to functions

as a means to return values through them. The ability to do this is preserved in COOGL to allow CLEAN code to be shared between C programs and COOGL programs. For example, when a large body of code is being incrementally migrated from C to COOGL. In COOGL it is idiomatic for functions that return multiple values to return them through a `tuple`.

The address of a memory buffer into which I/O should be performed, or into which some data should be built, is usually passed as an argument to a function that does the work, passing an array descriptor that refers to a non-`static` local buffer variable for these purposes doesn't have a better alternative solution, if it were not allowed, this kind of memory buffers would end up being global memory allocated from the heap, an alternative would be to allow each run-time stack to control a dedicated stack organized heap from which to allocate these memory buffers. A dedicated stack organized heap, associated with each run-time stack, implemented outside of the language, could be used as a discipline that forces such buffers to be isolated from the run-time stacks, helping secure the run-time stack from security attacks, particularly when the code uses pre-existing libraries written in unsafe languages, such as C and C++. A compiler option is provided to produce a warning if addresses within run-time stack allocated memory are ever passed as an argument to a C function, this helps find any data that might end up being used insecurely in a library written in an insecure language.

14.9 Returning addresses of run-time stack allocated memory

Address ranges that refer to memory allocated on the run time stack can not be returned by the function that allocated the memory. Attempting to do so causes a compilation error, irrespective of whether intermediate functions are used to attempt to do so. This prevents a function from referring to memory allocated on the run-time stack that has been deallocated when the function that allocated it returned. The language guarantees this property for every program.

For example, `bad()` attempts to cause the address of `c` to be returned by it through its invocation of `set()`, causing a compilation error:

```
char *set(char *p) { *p = ' '; return p; }
char *bad(char c) { return set(&c); } // error: returns address
                                      // of local: c
```

This is the case even if `bad()` and `set()` reside in separate source code files, and even if they are compiled separately into two different module binaries meant to be loaded dynamically at run-time under program control. The compiler generated information for `set()` includes information that indicates that the value returned by the function refers to memory within the memory that its first argument refers to. This information is propagated at compile time to `bad()`'s invocation of `set()` which results in a compilation error because the return statement attempts to return the ad-

dress of `c`.

Assume the following version of `set()`, which doesn't lead to the problem, is compiled into its own module that can be loaded dynamically at run-time:

```
char buf[] = {"hello"};
char *set(char *p) { *p = ' '; return buf; }
```

The following version of `bad()` is compiled, by itself, into a separate dynamically loadable module, with the compile time information about the version of `set()` that returns `buf`.

```
char *bad(char c) { return set(&c); } // no error produced
                                      // at compile time
```

After compiling the dynamically loadable binary that contains `bad()`, the code of `set()` is recompiled with this code which could lead to invalid memory accesses:

```
char *set(char *p) { *p = ' '; return p; }
```

A program dynamically loads this version of `set()` and then attempts to dynamically load the version of `bad()` which has not been recompiled, this causes an error and the dynamic loading of the binary fails because the symbol for `set()` required by `bad()` is incompatible with the symbol for `set()` that is currently in the program.

Another example, the `strchr()` C library function, is invoked frequently with a local C string as its argument. The function `wrong()`, below, by itself doesn't reveal that it attempts, through `strchr()`, to cause an address within its local variable `buf[6]` to be returned, which is invalid and causes a compilation error. The relationship between the `str[]` argument and the value returned by `strchr()` is determined by the compiler when `strchr()` is compiled. The relationship between `buf[]` and the value returned by `strchr()`, is used by the compiler when compiling `wrong()`, the compiler knows that it might possibly be returning an address within a local variable, so a compilation error is produced.

```
char *strchr(char str[], int c) promise(retval == NULL ||
                                        str.start <= retval &&
                                        retval <= str.end) {
    char v;
    char *s = str;
    char *send = str.end;
    for (; s < send; ++s) {
        if ((v = *s) == c)
            return s;  // address of c in s, even if c == 0
        if (!v) break;
    }
    return NULL;       // return NULL otherwise
}
```

```
char *wrong() {
    char buf[6] = {"hello"};
    char *p = strchr(buf, 'e');
    return p;       // error: address of buf[] could be returned
}
```

Note that the `promise()` specification that is part of `strchr()` is not required for the compiler to be able to determine the relationship between `str[]` and the value returned by `strchr()`.

14.10 Run-time stack allocated memory and execution contexts

The address of run-time stack allocated memory can not be communicated, by any means, to other execution contexts, for example: concurrently executing code in another thread, coroutines, exception handlers, interrupt handlers, signal handlers, etc. Operating system environments that allow processes with per execution context private run-time stack's that are only addressable by the execution context that owns the stack are supported by COOGL programs. See §15.9 for more about execution contexts.

14.11 Run-time stack growth is checked

The COOGL run-time stack growth is checked against overflow through red-zones (MMU page frames that are inaccessible). When the stack frame increment size could skip the size of the red zone, compiler generated code, or `alloca()` code, ensures that the red zone is not skipped by accessing the underlying memory incrementally in the direction of stack growth.

14.12 Casts and safety: `cast()` and `try_cast()`

A `cast()` is allowed from the address of some plain data to a pointer to a plain data type only if the cast does not result in a pointer that could access memory outside of the memory that the source address refers to. If the type of the memory that the target pointer refers to is larger than the memory that the source address refers to, then the use of `cast()` causes a compilation error, `try_cast()` should be used instead.

A `try_cast(type *, mem, value) addr` is like a `cast(type *) addr` that tests if the memory at `addr` (which must be within the plain data memory described by the `mem` array descriptor) when interpreted as a pointer to `type` (which must be a plain data type) is fully contained within the memory described by `mem`, if it is, the value of the `try_cast()` is the value `addr` with type pointer to `type`, if not, the value is `value`, whose type must be, or be compatible with, pointer to `type`. Usually `value` is `NIL` (see §14.16) or `NULL`, it can also be the address of some other data,

usually a dummy data variable of the same `type`. The result of a `try_cast()` can be checked explicitly, or just used to access the substitute dummy data, or if it is `NIL` to cause an exception (see §14.33) to be raised without testing the value explicitly, for example:

```
void example() {
    short s, *sp = &s, sa[10];
    int *ip = cast(int *)sp; // error: source object is smaller
    sp = &sa[0];
    ip = try_cast(int *, sa[], NIL) sp;     assert(ip != NIL);
    sp = &sa[9];
    ip = try_cast(int *, sa[], NIL) sp;     assert(ip == NIL);
}
```

The code generated for `try_cast()` is optimized to assume that it will succeed. When multiple `try_cast()` are used in a function, it is very common for the same array descriptor within which the memory should be contained to be the same in all of them, array descriptors are used as values they are never aliased. This allows the contained within tests for that are performed in a series of `try_cast()` to be optimized easily, for example, assuming these structures used to form some kind of message where the `header` and `footer` are mandatory but the `prefix`, `body`, and `postfix` are all optional, presumably with the `header` indicating what parts are present:

```
struct header  { uint h1, h2, h3; };
struct prefix  { uint pre1, pre2; };
struct body    { uint b1, b2; };
struct postfix { uint post1, post2; };
struct footer  { uint f1; };
```

In the code below it is expected that `data[]` has enough space within it:

```
void make_message(uint data[], header *h, prefix *pre,
                  body *b, postfix *post, footer *f) {
    uchar *ptr = cast(uchar *) data;
    *try_cast(header *, data, NIL) ptr = *h;
    ptr += sizeof(header);
    if (pre) {
        *try_cast(prefix *, data, NIL) ptr = *pre;
        ptr += sizeof(prefix);
    }
    if (b) {
        *try_cast(body *, data, NIL) ptr = *b;
        ptr += sizeof(body);
    }
    if (post) {
        *try_cast(postfix *, data, NIL) ptr = *post;
        ptr += sizeof(postfix);
    }
    *try_cast(footer *, data, NIL) ptr = *f;
}
```

In the worst case scenario all the optional parts are present, the compiler can see that if any of the `try_cast()` fails it will cause a store into memory with address `NIL`. The optimized generated C code would reduce the 5 `try_cast()` to one `<` test:

```
void make_message(uint data[], header *h, prefix *pre,
                  body *b, postfix *post, footer *f) {
    size_t size = sizeof(header) + sizeof(footer);
    if (pre) size += sizeof(prefix);
    if (b) size += sizeof(body);
    if (post) size += sizeof(postfix);
    lang__COND_STORE(data.max[0] < size, NIL, 0); // one < test

     //  lang__COND_STORE() is a compiler and hardware barrier,
     //  stores below only issued if no exception was raised

    uchar *ptr = (uchar *) data;
    *(header *) ptr = *h;
    ptr += sizeof(header);
    if (pre) {
        *(prefix *) ptr = *pre;
        ptr += sizeof(prefix);
    }
    if (b) {
        *(body *) ptr = *b;
        ptr += sizeof(body);
    }
    if (post) {
        *(postfix *) ptr = *post;
        ptr += sizeof(postfix);
    }
    *(footer *) ptr = *f;
}
```

The generated code could alternatively have a very precise translation of `make_message()` in `make_message_slow()` which causes the unavoidable `NIL` dereference in the exact location that the program dictates (including compiler and hardware barriers), for example to facilitate debugging under a debug support compilation flag:

```
    if (data.max[0] < size)
        return make_message_slow(data, h, pre, b, post, f);
```

The underlying C compiler is allowed to reorder memory accesses, as long as the execution by the thread perceives the execution as if it were in program order, and unless specific memory ordering is enforced through memory and compiler barriers. All the structure assignments in `make_message()` are to disjoint memory, reordering them by the compiler, or by the hardware is allowed from the perspective of other processors (e.g. because of hardware store buffers and cache coherency protocol induced delays together with the data straddling cache lines). Those stores can only occur after `lang__COND_STORE()` has not raised an exception.

14.13 Restrictions on class members whose type is a plain data type

Data members of a user or language defined class, where the data members' type are plain data types are not allowed to be used as plain data, their addresses can not be used in a `cast()` or a `try_cast()` operation. Their address can not be passed as an argument to a function, this ensures that in the function the data that the pointer refers to is not treated as plain data. This further helps segregate high level code that can be heavily optimized by the compiler because the plain data within user or language defined classes can not have pointers to it other than pointers of their correct type. Furthermore it forces the programmer not to get needlessly clever with underhanded manipulation of data members in a way that is not expected by the compiler.

A plain data member within a user or language defined class could have a member function defined on it, for example an `int` with a `scan()` function, this means that a `this` pointer within the member function points to the plain data item within the object, which leads to another language restriction: plain data types when manipulated as objects, i.e. when they are the entity that `this` points to, can not have their address used for any purpose other than to affect the whole data item through its type. The address can not be passed as an argument to a function, stored in a local or global variable, etc. This restriction guarantees that long lived pointers to the member won't exist once the member function invocation returns, it also guarantees that no loads or stores of the data are performed through types different than its own type, for example an `int` data member is never accessed through its underlying bytes individually.

14.14 Implicit pointer conversions without casts

Implicit pointer conversions, without casts, between pointers are allowed if the source type and the target type are both pointers to objects, and if the type of the object that the target pointer refers to is an ancestor class of the type of the object that the source pointer refers to. For example:

```
class base { pub int v; }
class derived { pub inherit base; pub int info; }
void example() {
    derived d;
    base *bp = &d;   // implicit conversion, cast not required
}
```

Similarly, a pointer to an object that provides an interface can be assigned to a pointer to the interface, without a cast; and a pointer to an interface that provides another interface (directly or indirectly) can be assigned to a pointer to the other interface, also without a cast. For example, using the classes from, §6.5, to obtain the `rdwr interface` through which sequential read or writes can be performed on the `file`, a pointer to `file` can just be assigned to a pointer to `rdwr`. Furthermore a pointer to an object that implements an interface directly or indirectly can be assigned

to a pointer to the interface, for example:

```
void nfs_pointer_example(nfs_file *nfp) {
    nfs_node *nnp = nfp;     //    object to ancestor class
    file *fp = nfp;          //    object to provided interface
    rdwrat *rwatp = nfp;     //    object to indirect interface
    rwatp = fp;              // interface to provided interface
    rdwr *rwp = fp;          // interface to indirect interface
    rwp = nfp;               //    object to indirect interface
    nfs_pointer_is_cast_example(nfp, nnp, fp, rwatp, rwp);
}
```

14.15 Pointer to base cast to pointer to derived: `is_cast()`

An `is_cast(type *, value) ptr`, from `ptr` a pointer to an object (where the `ptr`'s type is a pointer to an ancestor class of the actual object that it points to, or is a pointer type to an interface that the actual object implements), can be converted to a pointer to its actual type, or to some type in the inheritance chain or interface implementation chain of the object's actual type. If the `is_cast()` can be performed, because it is of the appropriate type, or implements the appropriate interface, then the value of the `is_cast()` is the value of `ptr` with type pointer to `type`. If the `is_cast()` can not be performed, then the value of the `is_cast()` is `value`, which must be `NIL`, a trapping address, or the address of a valid data item whose type is `type`. For example, when this function is called from `nfs_pointer_example()`, above with the arguments specified there:

```
nfs_pointer_is_cast_example(nfs_file *nfp, nfs_node *nnp,
                            file *fp, rdwrat *rwatp, rdwr *rwp){
    nfp = is_cast(nfs_file *, NIL) nnp;          assert(nfp);
    nfp = is_cast(nfs_file *, NIL) fp;           assert(nfp);
    rwatp = is_cast(rdwrat *, NIL) rwp;          assert(rwatp);
    nfp = is_cast(nfs_file *, NIL) rwp;          assert(nfp);
    nnp = is_cast(nfs_node *, NIL) rwp;          assert(nnp);
    fp = is_cast(file *, NIL) rwatp;             assert(fp);

    // this fails, the rwp did not come from a nfs_dir object
    nfs_dir *ndp = is_cast(nfs_dir *, NIL) rwp;  assert(!ndp);
}
```

Other than `is_cast()` no other casts are allowed from the address of a non-plain data item. No other assignments between pointers are allowed, with or without casts, other than those described in this and the previous sections.

14.16 Trapping addresses, `NIL`, `NULL`, and `uptr_cast()`

Conversion from a non-pointer value, implicit or through a cast, to a pointer value

is not allowed unless the source non-pointer value is:

- The value zero, usually through the `NULL` literal, which can be assigned to a pointer or used as a pointer argument, with or without a cast. Use of `NULL` or zero as a pointer value is strongly discouraged and deprecated, see §14.18.
- The `NIL` value, is the preferred invalid pointer value, `NIL` can be assigned to a pointer, or used as a pointer argument, without a cast.
- A *trapping pointer value*, of the unsigned integer type `uptr`, in this range:

  [`uptr.trap.BASE`, `uptr.trap.BASE + uptr.trap.COUNT`),

  can be converted to a pointer, using the `uptr_cast()` operator:

  ```
  uptr_cast(type *, val) uptrval
  ```

  If `uptrval` is trapping pointer value, the result is a pointer with that value. Otherwise, the result of the operation is `val`, typically chosen to be `NIL`, sometimes `NULL` or a valid pointer to `type`, or some other trapping pointer value literal.

Use of `NULL` or zero as a pointer value is deprecated, `NIL` should be used instead, unless code needs to interface with C or C++ code which requires the use of `NULL`, Using `NIL` instead of `NULL` isolates the language from non-standard behavior when `NULL` pointers are dereferenced in different platforms, from unsafe aspects of using `NULL`, and from the undefined behavior compiler optimization campaign (see §1.7) which negatively impacts on the reliability of C profgrams, by allowing C compilers to turn any `NULL` pointer dereferencing into something potentially much worst than what would have traditionally occurred in earlier versions of the C language and its compilers.

A *trapping address* is either: `NIL`, a trapping pointer value, or an address within the range of addresses referred to by a pointer whose value is: `NIL`, a trapping pointer value, or a trapping address. Given a pointer whose value is a trapping address, the address of: a field of a `struct`, a non-`static` member of a `class`, or of an array element within a traditional C array, are all trapping addresses. Note that similarly computed addresses based on a `NULL` pointer are not trapping addresses.

Fetching, storing, or executing, any memory, of any type, through a pointer whose value is a trapping address causes a run-time exception to occur, the exception is delivered to the execution context that caused it. The raising and delivery of the exception is not undefined behavior, it is defined behavior, see §14.33. A range of trapping pointer values is provided to allow pointers to contain invalid values that are guaranteed to cause an exception if used to access memory through them, or through other pointers whose values are trapping addresses computed from them, while allowing the programmer to store information in pointers when not being used as pointers, for example multiple invalid values with various meanings, or indexes into auxiliary information kept elsewhere.

Trapping addresses exist as a safety net, catching the program, trapping it, reliably and dependably, when it misbehaves, i.e. when it dereferences a pointer whose value is a trapping address. The program does not continue to run subsequent code as if it had not misbehaved. The programmer can depend on the raising of an exception (see §14.33) to build software that is more reliable than if the behavior was undefined.

Note that a pointer set to a trapping pointer value is not related in any way to an array descriptor within which the pointer can be subject to pointer arithmetic. Performing operations such as `p += n` won't cause the pointer to refer to some unknown memory, the expression simply won't compile. Pointer arithmetic is only allowed if the compiler knows the array descriptor within which the objects that the pointer can point to are, and the compiler ensures at compile time that pointer arithmetic won't cause the pointer to have values other than the values allowed by the range of values described by the array descriptor, i.e. [`start-1`, `end`].

The size of memory that an object can occupy is limited by the compiler, the range of trapping addresses is much larger than the range of trapping pointer values shown above. This guarantees that referencing every field of the largest supported object based on a trapping address always causes an exception (see §14.33). Furthermore there are also trapping addresses around `NIL`, whether there is a single range of trapping addresses or two disjoint ones, one around the trapping pointer values and another one around `NIL`, is not defined by the language.

Pointers to traditional arrays of objects, i.e. where the number of elements is known at compile time, and where the size of an array with that number of elements plus one (so that `&array[N]` is also covered by trapping addresses) is larger than the maximum object size, can not be set to trapping pointer values, attempting to do so causes a compile time error.

All empty array descriptors are initialized so that their `start` and `end` members have the value `NIL`, this ensures that any attempt to dereference them will cause an exception (see §14.33). The range of trapping addresses around `NIL`, is such that for a `NIL` pointer `p`, the expression `(p-1)->field` is guaranteed to cause an exception, this ensures that for any empty array descriptor the `start-1` expression refers only to trapping addresses.

The language guarantees that the trapping pointer value range supports:

- Storing 16 bit unsigned values in a 32 bit system, and 32 bit unsigned values in a 64 bit system.
- The values of a `ushort` on a 32 bit system, and the values of an `uint` on a 64 bit system, can be stored and retrieved efficiently as trapping values.
- The type `uptr.uval` is `ushort` on 32 bit systems and `uint` on 32 bit systems.
- The range of `uptr` values supported is: [`uptr.MINUVAL`, `uptr.MAXUVAL`].

- The value `NIL` is different than all the possible `uptr` values that result from using `uptr.uval_set()`, see below.

Most 64 bit processor architectures, allow processors that implement the architecture not to implement the whole 64 bit virtual address space, resulting in some very large contiguous address ranges that are invalid, addresses within such an invalid range are chosen for each architecture to implement the trapping addresses. On a system where every address is valid, wrappers around the memory mapping operating system interfaces might be required to ensure that a set of addresses can be reliably reserved to implement the semantics of the trapping pointer values by preventing the program from mapping memory into those addresses. It is not unusual for the operating system itself to restrict the range of valid addresses supported by it, in which case operating system specific invalid addresses can be used for the trapping pointer values.

14.17 Trapping pointer value interface and implementation

Common code file:

```
extend class uptr {                                          // uptr.cog
    pub lit uptr BITS = sizeof(uptr) * 8;
    enum class uptr trap {
        ONE = 1;
        SHIFT = BITS / 2 + 1;       // 33(64b) or 17(32b)
        COUNT = ONE << SHIFT;       // 8G(64b) or 128K(32b)
        HALF = COUNT / 2;           // 4G(64b) or 64K(32b)
        MNU = BASE | HALF;          // 0x3FF10000(32b)
        MXU = MNU + HALF - 1;       // 0x3FF1FFFF(32b)
        MINUVAL = cast(uval) MNU;   // 0
        MAXUVAL = cast(uval) MXU;   // 65535(64b)
        BASE = uptr.TRAP_BASE;
    }
    pub uval get_uval() return cast(uval) *this;
    pub void set_uval(uval v) { *this = MNU + cast(uptr) v; }
}
```

A platform dependent file used on some 32 bit systems:

```
extend class uptr {                                  // uptr-32.cog
    typedef ushort uval;
    pub lit uptr TRAP_BASE = 0x3FF00000u;
}                           //  ----++++
```

A platform dependent file used on 64 bit systems:

```
extend class uptr {                                  // uptr-64.cog
    typedef uint uval;
    pub lit uptr TRAP_BASE = 0x3FFFFFF000000000uLL;
}                           //  ----++++----++++
```

14.18 Use of `NULL` and zero as pointers is deprecated

Use of `NULL` and number zero as pointer values is deprecated in COOGL, they are allowed as a transition mechanism, the compiler option `--NULL` flag is required for their use. Support for them is particularly important to allow the use of C interfaces which might require or return `NULL` values. If the `--NULL` flag is not specified, testing a pointer, explicitly for being equals or not equals to zero, or equals or not equals, to `NULL`, causes a compilation error. For example, if `--NULL` is not used, the following functions all cause compilation errors:

```
bool is_a_0(int *a)    { return a == 0 ? true : false; }
bool is_b_ne_0(int *b) { return b != 0 ? true : false; }
bool is_c_NULL(int *c) { return c == NULL ? true : false; }
bool is_d_NULL(int *d) { return d != NULL ? false : true; }
```

It is not allowed to implicitly test a pointer to variable of a type that is not a user defined or a language defined class in a conditional context or to convert it to a `bool`, if the `--NULL` flag is used, e.g. all of these would cause compilation errors:

```
bool is_a(int *a)     { return a ? true : false; }
bool is_not_b(int *b) { return !b ? true : false; }
bool is_c(int *c)     { return c; }
bool is_not_d(int *d) { return !d; }
```

The fundamental reason for deprecating `NULL` and zero as pointer values is that, in some platforms, a `NULL` pointer value could be used to fetch memory at or near address zero. If `p` is `NULL`, `p->table[ix]` might be readable if `ix` is large enough, even if it is a valid index for `table[]`, even if `p->table[0]` is not readable. On some platforms it might even be possible to store into memory at or near address zero.

A `NULL` pointer dereference, because of a programming error, becomes a potential security hole. Even if fetching data at address zero, or near it, can not be exploited as a security hole, the fact that it doesn't cause an exception, but silently allows the program to continue to run, means that the program will most likely misbehave, possibly in a subtle way, for example by computing incorrect results and performing incorrect actions, possibly crashing subsequently, or much later, none of which can be tolerated in a safe programming language. Note that any such incorrect behavior could not be reasoned about from the programming language perspective, the programmer would have to understand, at the machine level what happened, what memory was accessed incorrectly, what values were found there, and what happened subsequently, for example if a chain of incorrect values was found and led to one of them to be considered a function pointer and the program ended jumping to an arbitrary location and crashing immediately or later after executing some arbitrary code..

The layout of a running program's address space is under the control of the operating system, it is not reasonable to require operating system changes to implement the

best possible run-time environment for the language, at least not int the foreseeable future. Some operating systems even allow for memory to be mapped at virtual address zero, through `mmap()`, or through `shmat()` (e.g. MacOS X).

In consequence, programs compiled with `--NULL` should be considered unsafe. Even if they were safe in some systems, they might no longer be safe after operating system components have changed, or their behaviors have changed, for example through: tunable options, system behavior control mechanisms, address space randomization variations, etc.

14.19 Addresses of members based on `NULL` or trapping addresses

If a pointer's value is `NULL` or a trapping address (`NIL` is a trapping address), then computing the address of any field of the object is well defined, its a valid operation. Some programmers read the C standard and believe that computing the address of a field based on a `NULL` pointer is undefined behavior in C, there is merit to their reading of the standard but that interpretation goes against the spirit of C, but is the interpretation most likely to be defended by the compiler writers who have taken over the future of the C language, who have hijacked the language from its users. This is not undefined behavior in COOGL.

For example, `offset_of_next()` and `offset_of_prev()` are valid, the second is valid only if compiled with `--NULL`, both result in the computation of a constant value at compile time, they are both equally efficient:

```
size_t offset_of_next() return cast(size_t) (cast(uptr)
                            &(cast(node *)NIL)->next - NIL);
size_t offset_of_prev() return cast(size_t)
                            &(cast(node *) 0)->prev;
```

14.20 Use of `NULL` with objects of a class type is invalid

Because `NULL`, and the value `0`, are deprecated pointer values, which are only supported as a bridge to legacy C code, there is no benefit in allowing pointers to objects of user or language defined classes to have those values. Assigning or comparing, `NULL` or `0,` to a pointer to an object of a class type is invalid, even when compiled with the `--NULL` flag.

A pointer to an object of a user or language defined class can be converted to a `bool` value by assigning it to a `bool` variable; or by evaluating the pointer with the `!`, `&&`, or `||` operators; or by testing it in the control expression of control flow statements (`if`, `for`, `while`, and `do-while`); or in the controlling expression of the `?:` operator. If the pointer is `NIL`, the result is `false`. The result is `true` otherwise. Note that because these pointers can never have the value `NULL`, that value is not considered when they are tested.

Traditional C-like code can be written that tests pointers to objects of a user or system defined class type, for example to determine the end of a list, without worrying whether to use `NIL` or `NULL`, because only `NIL` can be used with them. In this code the first two `for` statements are valid, the third one causes a compilation error:

```
class node {
    pub node *next;
}
void walk(node *list) {
    for (node *p = list; p; p = p->next) work(p);
    for (node *p = list; p != NIL; p = p->next) work(p);
    for (node *p = list;
         p != NULL;          // error: node * compared to NULL
         p = p->next) work(p);
}
```

Use of `NULL` with pointers to the native data types, structures, unions, or pointers to them (recursively) is allowed, but only when `--NULL` is used. `NIL` can always be used with any of these.

If `--NULL` and `--NULL-implicit` are used, then testing implicitly in conditional expressions, or converting to `bool`, pointers to objects whose type is not a user or language defined class is allowed, but if the value `NIL` is ever used with one of these types anywhere in the program, then testing pointer values of that type in a conditional context or causing them to be converted to `bool` causes a compilation error, they have to be explicitly compared against `NULL`, zero, or `NIL` instead. The goal is to provide an orderly way to transition away from `NULL`, because its use is unsafe, and replace its role with `NIL`, incrementally, particularly when migrating and reengineering a very large code base from C into COOGL.

14.21 The `unsafe_cast()` operator and disabling unsafe features

The `unsafe_cast(type)` operator behaves exactly as the underlying C language cast operator, its use requires the `--unsafe_cast` compiler option to be used. Use of this option and `--NULL` and `--NULL-implicit` are the only way that a COOGL program can be unsafe. The use of these options can be disabled in the compiler by various means to ensure that they are not used mistakenly, see XXX.

14.22 Deconstructed values and uninitialized variables

When an object is destroyed, all of the entities that form it take their *deconstructed values*. Pointers take the `NIL` value, array descriptors are reshaped as empty array descriptors (with `start` and `end` set to `NIL`, and the values in `max[]` are set to zero), entities of any other type are set to zero. All of this is done transparently by the language. When an object is first allocated, prior to construction, those are their values.

As a transparent performance optimization, the compiler will not pre-initialize a field of an object on the run-time stack to its deconstructed value, instead the compiler will ensure that the field be initialized by the constructor prior to there ever being a possibility of it being accessed during its construction or its initialization, directly or by other functions invoked at that time. The compiler will report an error if some of the members of an object are not initialized by the constructor or by the `init()` or `init_deinit()` member functions. Because an object on the run-time stack can not have pointers to it at its destruction time its fields are not set to their deconstructed values.

Plain data variables on the stack are not pre-initialized by the compiler, use of uninitialized plain data variables causes a compilation error with the exception of arrays, they are commonly used for reading information into them, requiring them to be pre-initialized would be wasteful. Use of non-plain data variables on the stack that have not been initialized or assigned prior to their use causes a compilation error.

Global or static uninitialized plain data variables are all set to zero. Global or static uninitialized array descriptors are pre-initialized as described above.

Global or static pointers, arrays of pointers, and pointer fields within structures, must be initialized explicitly, pointers must be set to `NIL` or `NULL`, explicitly, according to the convention for them, see §14.18. Arrays of pointers don't require every element to be initialized, but at least the first element of the array must be initialized, every range of non explicitly initialized values must be preceded by an initialized value, which must be either `NIL` or `NULL`, that value is used for the non explicitly initialized values that follow it. For example, a large global array of pointers can have all of its entries initialized to `NIL` by initializing its first element to `NIL`:

```
struct node { node *next; node **prevpp; id_t id; val_t val; };
lit size_t NHASH = 1024;
node *hash[NHASH] = {NIL};      // every array entry is NIL
```

14.23 The `uninit()` member function

A class can implement the static member function:

```
pub static void uninit(type raw *to) { ... }
```

If it is not implemented, then the compiler generates the code for `uninit()` which uninitializes the objects non-static members described in §14.22. A programmer might choose different uninitialized values, for example a `float` might be set to a SNaN (signaling not a number) value, which are more convenient to detect programming errors.

14.24 Permanent association of heap virtual addresses and types

The language provides an implementation of dynamically allocated memory in the `lang.creatable` `interface` its design and implementation are key to the language safety, polymorphic member function invocation implementation, and the zero per-object memory overhead polymorphism support.

When an object, or an array of objects, is allocated through `lang.creatable`, the addresses that they occupy, once the memory is freed, can only be reused to allocate objects or arrays of the same type. This implies that any pointers to the memory of these objects, even after the objects are freed, can never be used to refer to memory of other objects of a different type, they can only refer to the deconstructed memory of the objects of the type that they refer to.

The permanent association of virtual addresses and types does not mean that the underlying physical memory is permanently dedicated to objects of a given type. The underlying pages of memory, if they only contain free objects, can be unmapped by the allocator thus allowing the operating system, or the allocator itself, to remap them elsewhere (if the operating system supports such remapping) for use by objects of another type. This rebalancing of the underlying physical memory might be important for certain software that might create many objects of a specific type, then release them all, and later allocate many objects of another type.

Most operating systems don't provide support for physical page remapping for anonymous memory (i.e. memory that doesn't have a long lived home location, memory mapped files are not anonymous memory) only page un-mapping, the addresses associated with the unmapped page subsequently become inaccessible and any attempt to fetch or store from them results in an exception. To prevent the underlying virtual addresses to be reused for a different purpose, for example through the `mmap()` UNIX system call, which could lead to a security compromise of the running program, if it can be caused to follow pointers to the free objects that previously occupied those addresses, the memory mapping interfaces are wrapped by library code that ensurers that the contiguous address range managed by the allocator can never be reused, or manipulated in any way, by any code other than the allocator itself.

The addresses of a page within the heap that have been unmapped, and that the allocator subsequently wants to reuse, by memory mapping a page of anonymous memory into it, would be immediately accessible and zero filed. Until the pointers and array descriptors for the object carcasses that previously lived there are set to `NIL` the underlying dead objects will have, briefly, a different set of uninitialized values. Particularly pointers would have zero, i.e. C's `NULL` value, in them. Because of the undefined behavior associated with `NULL` by modern C compilers incorrect code might be able to cause arbitrary values to be interpreted as object pointers, defeating the language safety.

On such platforms, unless other mechanisms are provided such as `userfaultfd(2)` on Linux, the allocator managed memory needs to be backed by one or more persistent files that are created as needed and deleted while their file descriptor is kept open, to ensure that when the program exits the storage space that the files use is released. Because those pages can be remapped at will at other addresses, they can be given their proper values prior to remapping them at the dead object addresses. The ability to create holes within these backing files is important, and it is present in most modern systems, to cause underlying file backing storage and physical memory used to cache it to be released all the way back to the operating system itself. The file descriptor for these persistent files must also be guarded through system call wrappers to ensure that they are never used to affect the program's memory, for example by writing into them. An alternative implementation on these systems is to never release memory from the heap back to the operating system for it to reuse, once associated with objects. On these systems the allocator could use memory advisories to indicate that it won't use the memory, hopefully causing the pages to more quickly become candidates to be paged out and reused than other memory.

14.25 Array walking through pointer ranges is always valid

The array descriptor `start` and `end` values are always well defined, furthermore, the address `start-1` is also well defined, thus code that walks arrays forwards and backwards does not suffer from strange address space warps or theoretical segment underflows from stone-age segment based addressing hardware architectures, simply because those architectures no longer exist or are no longer relevant.

```
tuple [int *first = NIL,
       int *last = NIL] find_first_last(int ad[], int val) {
    for (int *p = ad.start; p < ad.end; ++p)
        if (*p == val) {
            first = p;
            break;
        }
    for (int *p = ad.end; --p >= ad.start; )
        if (*p == val) {
            last = p;
            break;
        }
    return;
}
```

Note that the first `for` loop exit condition requires that `p >= ad.end` for the loop to terminate, it will terminate when `p == ad.end`. The second `for` loop requires that `p < ad.start` to terminate, it will terminate when `p == ad.start-1`, which requires that it be a well defined address value that is arithmetically prior to `ad.start`, even though the undefined behavior C compiler optimization religion might

want to call that undefined, it is defined in COOGL. The compilation of COOGL code into C code guarantees that this is done in a way that the underlying C compiler undefined behavior can not arise and cause the compiled code to be incorrect.

A related aspect to these pointers that are off by one, before and after, a valid array of objects is that once these pointers are constructed, they could be used to refer, in principle, to memory that belongs to other objects, in an attempt to subvert the safety of the language, but it can not be subverted this way as explained in §14.27.

14.26 Invalid pointer value computation

The values `ad.end+n` and `ad.start-1-n`, where `n > 0`, are invalid, the compiler does not allow them to be computed, any attempt to do so causes a compile time error, if the compiler can not determine at compile time whether those values are computed at run time, then the compilation fails. For example, this code causes a compilation error:

```
int *find_first_start_at_n(int ad[], int val, size_t n) {
    for (int *p = ad.start + n; p < ad.end; ++p)
        if (*p == val) return p;
    return NIL;
}
```

A `require(n <= ad.max[0])` contract can be used to place requirements on the calling code, which would allow the code to compile, but in this case defensive programming is better:

```
int *find_first_start_at_n(int ad[], int val, size_t n) {
    if (n >= ad.max[0]) return NIL;
    for (int *p = ad.start + n; p < ad.end; ++p)
        if (*p == val) return p;
    return NIL;
}
```

14.27 Use of objects at `start-1` and at `end`

The pointer values computed from an array descriptor: `start-1` and `end` are valid pointer values. Dereferencing pointers with those values is usually incorrect, but does not lead to undefined behavior, nor to invalid memory accesses. The language guarantees that a valid properly constructed object exists at the address, or the memory for a deconstructed object of that type is located there. Making use of those objects is incorrect, unless the programmer knows that those objects are properly constructed objects. For example, because it is actually operating on an array descriptor, that by program design, the programmer has chosen to always be surrounded by valid objects, e.g. if the code was designed that way. Note that this kind of programming is not unusual at all, having valid or degenerate sentinel values around an array or oper-

ating on a subarray of another array is quite common in practice.

The rationale for this language design choice is:

- Array descriptors are frequently created to refer to a subset of objects within a larger array, having valid surrounding objects is a common case.
- Once pointers with the values `start-1` and `end` are computed, extra code would need to be generated to ensure that they are not dereferenced.
- The `lang.creatable` allocator allocates all memory for arrays from a given type in such a way that it guarantees this invariant, the array is surrounded by a deconstructed object of the same type, by default. The programmer can choose for each type: to have the object be constructed (in which case it must implement `init_default()`); or that there be two objects between arrays instead of one (both constructed or not); or that there be no objects at all between arrays, but that it be guaranteed that there is always an object prior and after every array. See §13.8.
- Arrays with more than just a few elements are common, arrays with few elements are uncommon. The cost of the an extra object for individually allocated arrays is one per array, by default only one additional deconstructed object is required per array.

Contiguous arrays share the deconstructed object in between them, at `end` for the first array and at `start-1` for the second. An additional deconstructed object, at the start of every address range from which arrays are allocated, is provided by the allocator, to establish the invariant for the first array allocated in the range.

Arrays with few elements are uncommon, but even if they were common for some application, to the point that the memory consumed by the extra unconstructed objects required in between individually allocated arrays becomes a performance burden (maybe because the objects themselves are extremely large, or because a tremendous number of tiny arrays is created), the programmer can work around the memory waste in various ways, as described above; or by reorganizing the objects into a small part that refers through a pointer to a larger part which is allocated at object construction time, thus the cost for the optional part would not be required for the unconstructed objects between arrays.

Note that when `start-1` or `end` refer to deconstructed memory, their dereferencing is no different than the dereferencing of a pointer to dynamically allocated memory that has been freed. Both refer to deconstructed memory, access to deconstructed memory does not lead to *invalid memory accesses*. It is just a well defined programming error, it is not undefined behavior, it is an error in the logic of the program, no different than any other logic error in the program.

In the array: `int m[10][20];` there is only one additional `int` element prior to the

array, and another one after it. There aren't two 20 element dummy arrays, one prior to `&m[0]`, and another at `&m[10]`. If there had to be dummy arrays to cover the worst case scenario, then the `b[][]` array descriptor below, which reinterprets the memory of `a[100]` into a two dimensional array, `b[2][50]`, would require that there be a dummy 50 `int` array prior to `&b[0]` and another one at `&b[2]`, this would mean that for any array the memory that would have to be allocated would be 2 times the memory specified by the program, which is unreasonable. Worst case, with `c[][]` the required memory would be 3 times the requested memory.

```
void f() {
    int a[100];
    int b[][] = lib.array.make({2, 50}, a.start);  // b[2][50]
    int c[][] = lib.array.make({1, 100}, a.start); // c[1][100]
}
```

To ensure that the required memory overhead is just one additional element per array, the array descriptor associated with arrays of arrays, for example for the array `int a[10][20]`, can be used to walk the array multi-dimensionally through indexes, using `a.max[0]` and `a.max[1]`, or uni-dimensionally with pointers that refer to the underlying base elements of the array (the `int` elements in this case) using `a.start` and `a.end`. Use of an array `typedef`, as shown below, doesn't make a difference.

```
typedef int array_of_20_int[20];
array_of_20_int a[10];              // same as: int a[10][20];
void f() {
    int *start = a.start, *end = a.end;   // valid
    assert(end - start == 200);
    array_of_20_int *s = a.start;  // error: incompatible types
    array_of_20_int *e = a.end;    // error: incompatible types
}
```

Compatibility with C mandates that arrays within structures and unions don't have extra elements around them. Pointers prior to or after arrays within structures or unions produced by walking these arrays are unsafe and need extra code to ensure that they are not used. Within the typical loop that walks such an array the compiler knows that the pointers are valid. After exiting the loop the pointer might be within the array or immediately after, or before if the array is being walked backwards. The code generated by the compiler ensures that the pointer is not used to reference data unless it is within the array, an exception (see §14.33) is raised otherwise.

Arrays of constant size within a class object do have additional elements surrounding them, but only when required, the compiler determines if array descriptors are ever based on the array and if `start-1` or `end` are ever computed, it only allocates the extra element after the array, if `end` is computed, and the one before, if `start-1` is computed. For very small arrays the compiler has options that force it instead to produce code to ensure that data at `start-1` and `end` are never referenced. This can also be accomplished by declaring the `class` as a `class struct`.

14.28 Out of bounds indexing causes an exception

Every bound is checked individually. Bounds checks are optimized by the compiler when the whole array, or sequential parts of the array are walked iteratively, frequently resulting in no bounds checking code at all. These optimizations are facilitated by the fact that the bounds are either known at compile time, or if they are only known at run-time, the array walking must be done through a local copy of the array descriptor which usually doesn't change inside the loop. For example, the bounds checks for the evaluation of `v[i][j]` are optimized away in:

```
double sum(double m[][]) {
    double total = 0;
    for (index i = 0; i < m.max[0]; i++)
        for (index j = 0; j < m.max[1]; j++) total += v[i][j];
    return total;
}
```

14.29 Invalid memory access definition

Memory that is readable or writeable and that contains non-plain data, constructed or deconstructed, can not be accessed as if they were of a type different than its type, other than through a pointer to an ancestor type. All other memory accesses to non-plain data are invalid memory accesses, the language and its run-time support code (i.e. dynamic memory allocation support and the management of run-time stacks), make invalid memory accesses impossible.

Access to plain data as if its type were of a different plain data type is not an invalid memory access, this is a feature of the language to allow for carefully laid out memory to be crafted to support external data representation requirements.

Access to dynamically allocated non-plain data memory that has been freed is not an invalid memory access, it is a valid memory access of the deconstructed memory of a previously existing object of the same type. Access to an object immediately prior to, or immediately after, an array is not an invalid memory access, it is a valid memory access of either: a constructed object, or the deconstructed memory of an object, of the same type as the type of the objects in the array. Dereferencing a `NIL` pointer, or a pointer whose value is a trapping address, is not an invalid memory access, it is a well defined memory access that always causes a run-time exception (see §14.33) to be raised.

Use of an uninitialized pointer is not allowed, it causes a compile time error. Use of a variable that has not been initialized is not allowed, it causes a compile time error.

If a program has any code compiled with `--NULL` it implies that pointers with the value `NULL` might exist within it, such programs might be caused to perform invalid memory accesses, those programs are not safe, unless the code in question is care-

fully localized and proven to not cause directly or indirectly invalid memory accesses. Typical code that might be compiled with `--NULL` are wrappers for C library functions so that they can be provided as COOGL functions, the wrappers would map `NIL` to `NULL` and `NULL` to `NIL` appropriately so that they can be used by COOGL code. Code that uses properly written wrappers is safe and can be guaranteed not to perform invalid memory accesses unless the underlying C code itself performs them.

14.30 Prefix classes: `preclass`

XXX

14.31 Extending the language safety model

XXX

14.32 Dynamically unloaded modules and safety

A module that is unload while a program is executing can cause a program to misbehave if it references code or data at the addresses where they previously resided. If another module is loaded in those same addresses, then pointers to data of the previous type could now refer to data of a different type in the new modulo which is unsafe. To simplify the language, at this time, unloading a module is an unsafe operation, the program that supports such unloading has to ensure that data or code references to the addresses that belonged to the modulo do not occur after it is unloaded. Ensuring that the code and data in question is not accessed is no different than the garbage collection problem, in general, in that potentially all of the memory in the program and elsewhere (thread contexts, CPU registers, etc), would have to be examined. Code references might be the current program counter of a running or blocked thread, the program counter value in an execution context or in a `longjmp` like jump buffer, return addresses in the run-time stack, function pointers, code addresses in general purpose registers in running or blocked threads, etc. Data references could be anywhere in global data or in run-time stacks, there could also be data references in the registers of running or blocked threads, `longjmp` like buffers, etc.

Depending on the design of the program in question, ensuring that a module can be safely unloaded might be a property easier to verify. There are many designs, particularly in operating system kernels, that make the safe unloading of modules a more tractable problem.

The simplest solution is to simply not support the run-time unloading of modules.

14.33 Hardware and software exceptions and exception handlers

When an exception occurs the program is terminated, unless the exception is caught by an exception handler. An exception handler executes in the execution context that caused it, see §15.9 and §14.33 for details about execution contexts and exception handlers.

Exceptions detected by hardware, i.e. without additional instructions in the instruction stream to detect them, include: division by zero, illegal instructions, trap instruction, memory access to a trapping address, invalid memory access, and unaligned memory accesses. Exceptions detected by software, usually through additional instructions compiled into the instruction stream, include: out of bounds array indexing, assertion failure, `expect()` or `promise()` failure, and excessive run time stack memory use.

For performance reasons, software exceptions are sometimes raised through hardware means (for example through a trap instruction or by performing load from a trapping address) to minimize run-time overheads when the exception is not raised, for example overheads related to calling conventions and register usage. If the handler of a software exception handler returns, the underlying software or hardware mechanism that was used to raise it will cause the exception to be caused again, and the exception handler will be invoked again, without reevaluating the condition that caused the exception.

15 - Concurrent programming

> "Weakly-ordered processor architectures provide a relaxed view of the memory subsystem, where different processors may have different views of shared storage. One of the motivations for having weak storage ordering is to allow storage subsystem optimizations, which enable better scaling of the memory nest design. It is important to ensure that modern programming models do not artificially constrain the scalability of these system, which would ultimately undermine their success."
>
> -- R. Silvera, M. Wong, P. McKenney, B. Blainey

15.1 Concurrent programming

Sequential programming, where only a single flow of control exists within a program when it executes is the common programming model for traditional computer systems and programming languages. Execution of programs in a computer system was eventually formalized into the concept of a *process*, and operating systems evolved to support the concurrent execution of unrelated processes, with little or no sharing of resources between them, usually limited in their interactions on the operations they performed on shared resources, for example files provided by the operating system. As operating systems evolved, additional facilities were introduced to allow independent processes to communicate with each other, through services provided by the operating system, for example through message passing, pipes, record locks for files, isolated areas of shared memory, and synchronizers to allow coordination of their work on the shared memory. These facilities are referred informally as *interprocess communication*, or IPC. Communication between processes possibly located on different computer systems with a procedure-like interface, *remote procedure call*, RPC emerged as a high level mechanism for clients and servers to communicate, and for servers to provide services on behalf of the client processes. Communication between processes, within a system, or across computer systems required the notions of identity, trust, and many aspects that can be described broadly as distributed security. Higher level facilities such as distributed transactional systems, durable queues, distributed database operations, etc. continued to evolve as applications and their services transcended the boundaries of individual computer systems to gain scalability and availability.

All of the above are facilities that allow concurrent programming. From the perspective of programming language design these facilities are provided by the operating system or the distributed applications themselves and it would be a mistake to support them natively through built-in syntax and mechanisms by a traditional programming language. The programming language has to end somewhere, and this kind of facilities are meant to be implemented through libraries, server processes, etc.

A parallel thread of computer evolution, pun-intended, was that computer systems evolved from single CPU systems to multi-CPU systems, where the CPUs could all execute concurrently and share a common memory between all of them. One way to program such computer systems is simply by supporting sequential processes that exploit the parallelism by communicating, i.e. *communicating sequential processes*, which is no different than what occurs on single CPU systems. This indeed is the simplest way to program such systems and is appropriate for most programs that don't need to scale because of their computational requirements beyond the processing capability of a single CPU. Even if multiple CPUs could benefit an application, it can sometimes be structured as independent processes, maybe with some shared memory, or none at all, and partition the work in a multi-process way. Nonetheless, sometimes, applications can benefit by having multiple threads of execution within a single process. The benefit might be structural in that the problem being solved is best implemented with multiple concurrent threads of execution, or the benefit might be purely a performance benefit if the application can make effective use of more than one CPU for computational intensive operations.

Certain large programs, database servers, transaction monitors, operating system kernels, designed to operate on these multi-CPU shared memory systems, usually referred to as *symmetric multi-processor* (SMP) systems. The term SMP is dated, the term processor is often used to refer to the physical VLSI chip that implements one or more physical CPUs, the term CPU is sometimes also confused with the VLSI chip and thus the term *multi-core* has emerged to refer to a VLSI chip that contains multiple CPUs, or *cores*. A computer system might contain one or more VLSI chips, each with one or more cores in it. Because of the level of integration in modern systems most-multi chip systems have 4 or more physical cores in each one of them. Sometimes a single core can have multiple register sets within it, which are known as logical cores, or hardware threads, each hardware thread having all the architected general purpose, floating point, vector registers, program counter, condition registers, and special registers in each register set. A single core that implements multiple hardware threads can dispatch and execute instructions for each hardware thread in a finely interleaved way. When one a hardware thread stalls (usually because its waiting for loads from memory to complete, or stores to be be moved from write buffers and into the cache, so that a stalled store can be placed into a write buffer), other hardware threads can have more hardware resources dedicated to their instruction execution. The fundamental weakness of hardware threads is that unless the workloads

have a good amount of cache affinity and the cache hierarchy is very large and highly associative, the interference of the hardware threads and their competition for cache resources can cause significant slow downs, thus hardware threads might include hardware scheduling priorities, and their dynamic enablement and disablement, usually under the control of the operating system but sometimes directly by programs directly.

Multi-threaded processes where all the threads share all (or most) of the memory within their address space require that the programming languages used to write them don't get in the way of their concurrent execution, they require that the code generated by the compiler doesn't assume that there is only a single thread of execution. This is the only are that merits support from a systems programming language such as C, C++, or COOGL.

15.2 Language design considerations

Built-in support for high level concurrent programming constructs in a programming language has been shown to be inflexible and problematic in languages such as Ada, Java, C#, and Go. It usually interoperates poorly, or not at all, with concurrent programming interfaces and the internal synchronization of libraries provided by the operating system.

Operating system provided interfaces are language independent, they support concurrency control within a process through threads and synchronizers, and across multiple processes, sometimes with the same, but usually different synchronizers.

The fundamental balancing act between language provided concurrency mechanisms and operating system provided ones is the design tension of: *what belongs where?* Languages are meant to transcend and be independent of the operating systems where they are supported. Operating systems are meant to be language independent, successful ones support many languages.

Given that the computer system hardware, at its lowest levels, is controlled by the operating system, the lowest levels of concurrent execution support must be exposed by it. The operating system is inextricably involved in system wide concurrent execution across one or more threads of execution, per process, and across all processes within the system, it is the operating system's job to manage the scheduling of threads of execution to the system's CPUs, and their blocking when they have to wait (typically for input or output operations to complete). This classical operating system construction doesn't mean that alternative ones or more refined ones are not possible. The operating system could expose concurrency control in a very low level form, for example virtual CPUs, together with some means of controlling their scheduling for execution onto physical CPUs, low level virtual CPU context management, cross virtual CPU notifications (interrupt like), and notification delivery management.

There are many concurrent programming mechanisms, among them: multiple concurrently executing processes with or without shared memory, multiple threads within a single process, co-routines (fibers), continuations, monitors, locks (spin locks, blocking locks, and adaptive locks), condition variables, events, shared/exclusive locks, semaphores, synchronous or asynchronous message passing, message selection through rendezvous, interprocess procedure calls (IPC), remote procedure calls (RPC), asynchronous procedure calls (APC), typed or untyped messages, sending and receiving of capabilities, lock free programming, read copy update (RCU) techniques. Interacting with many of these mechanisms are other concerns, such as exceptions, timers, asynchronous I/O, blocking I/O, abnormal interruption of operations, security, serving requests on behalf of various requestors, tying work together such that it occurs atomically or not at all, operations across multiple systems, data integrity, availability, scalability, fault-tolerance, etc.

The computer system processor itself, without operating system intervention, usually provides low level computer instructions (load-linked / store-conditional, compare exchange, test-and-set, or some other form of atomic read-write operation) that allow for the implementation of a single load-modify-store operation on a small data item to occur atomically from the perspective of concurrent execution contexts, usually only on system word sized data, sometimes on smaller data, and sometimes on data that is a very small multiple of the system word size, usually with alignment constraints such that the data has to be at an address that is a multiple of its size. New computer architecture enhancements provide hardware transactional memory support, which allows for one or more loads and stores to a few small data items to occur atomically, i.e. with their side effects observable only all at once, or none at all, if the operation failed.

There is a language design tension between concurrency support provided by the operating system and built-in concurrent programming support through a builtin language feature. When concurrency support is expressed syntactically as a series of builtin mechanisms, the language designer seems to believe that the programs exist in isolation completely separated and unaware of the operating system on which the programs run, even if concurrency between multiple processes are supported by the language, it is unreasonable to expect that every piece of software in the system is going to be written in the same language. For example a database and transactional support system might need to be used and the concurrency control and interfacing with it might require a completely different set of concurrency control mechanisms; or operating system features might require certain synchronizers to be used. This language design tension has been traditionally resolved in the direction of the language being as minimal as possible so that programs written in it can be as native as possible in whatever operating system supports them, which is what C has traditionally done. Other languages, Ada, Java, C#, etc take the opposite approach and are sometimes found lacking when having to interact with other software written in other lan-

guages or when accessing native operating system facilities.

15.3 Allowing concurrency support through libraries

Even if there is no concurrent programming support built into the language, and all the support is provided through libraries, it is important that the language itself doesn't preclude those libraries from being written, the compiler should not generate code that depends on there being only a single thread of execution control. For example, the compiler should follow the load and store instructions that the programmer specified and should not create fictitious memory accesses that were not specified by the programmer. If a data item is accessed once, the compiler should not access the data item a second time and assume that the value will be the same as it was before, doing so assumes that no other entity could be changing the memory concurrently. For example:

```
uint fx;
void f() {
    switch (fx) {
    case 0: f0(); break;
    case 1: f1(); break;
    case 2: f2(); break;
    case 3: f3(); break;
    case 4: f4(); break;
    case 5: f5(); break;
    case 6: f6(); break;
    case 7: f7(); break;
}
```

The value of `fx` must be fetched once into a register or a local temporary on the stack and used for the two implied computations, one to see if it is less or equals than 7 and another to compute the branch target (or to fetch a branch address from a table with 8 branch targets, or in this case 8 functions pointers). If the value of `fx` was fetched twice, once to determine if it is less or equals than 7, and a second time to determine where to jump to, the value could have changed and cause the program to misbehave immediately, or worst, in subtle ways (possibly corrupting data), without any guarantee of an exception being raised immediately or at all, possibly continuing to run with corrupted data for an extended period of time.

Another example, this code:

```
class wsbx {
    pub uint w;
    pub ushort s;
    pub ubyte b;
    pub ubyte x;
}
```

```
pub void set_low_wsb(wsbx *d) {
    d->w |= 3;
    d->s |= 2;
    d->b |= 1;
}
```

Could be compiled, incorrectly, into this C code:

```
pub void set_low_wsb(wsbx *d) {
    ularge u = *(ularge *) d;
    u |= 0x300020100uLL;
    *(ularge *) d = u;
}
```

This is incorrect because the `ubyte x` is being read and its value written back, this could cause a concurrent store into `x` to be lost, i.e. as if it never happened.

A very important circumstance when the compiler can not prove that data is not affected, are function invocations, it can not assume that the data is not affected by them. For example, it can not keep data values in registers that are preserved across the function invocation, if the function changes the data, the values in registers would be incorrect. Function invocations, at least invocations of function whose code the compiler knows nothing about, are a boundary where the compiler has to put any data that it has changed and has kept in registers into their proper location, it must also stop using any data values cached in registers. For example:

```
large total;
void large doit(int vec[]) {
    total = 0;
    for (int *p = vec.start; p < vec.end; )
        total += *p++;
    work();    // compiler knows nothing about what work() does
    return total / vec.max[0];
}
```

The variable `total` doesn't need to be affected on each iteration of the `for` loop, the code can be safely compiled into:

```
large total;
void doit(int vec[]) {
    large tot = 0;                  // ok to compile into this
    for (int *p = vec.start; p < vec.end; )
        tot += *p++;                // ok to compile into this
    total = tot;                    // ok to compile into this
    work();                         // could change total
    // return tot / vec.max[0];     // can not compile into this
    return total / vec.max[0];
}
```

Functions that the compiler knows nothing about are the proper places to imple-

ment concurrency control operations. For example, `lock(&p->lk)` and `unlock(&p->lk)` functions that implement a mutual exclusion lock. After the compiler has been forced to synchronize its caching of memory values in registers with the flow of execution, prior to the function invocation, other actions required by the hardware for properly accessing shared memory under a lock can be performed by the function. For example, memory barriers that prevent memory accesses to be moved by the hardware to occur prior to the lock being acquired, and other barriers to ensure that memory affected under the protection of the lock has been flushed into the hardware coherency domain (e.g. incoherent write buffers have been flushed into the cache coherent memory domain) prior to the lock being unlocked.

The fact that the C programming language easily allowed concurrent execution and its compilers didn't get carried away generating code that would make concurrency support impossible is a testament to the fact that C has always supported concurrent execution, even if compiler writers choose to denigrate the C language and claim that it could have never supported concurrent execution until C11 and its memory model. They write those statements into documents on computer systems running highly concurrent operating systems kernels written in C, directly descendant from UNIX in code or spirit (MacOS X, Linux, Solaris, AIX, BSD, Irix, etc), or Windows systems, whose kernel is also written in C. C was written to reimplement the UNIX kernel from assembly into a portable language, it was a multiprocess kernel from the beginning, those processes, while executing in the kernel, behaved like a concurrent program, each with its kernel stack, no different than a multi-threaded program. C has never gotten in the way of concurrency support implemented by functions unknown to the compiler, only recently, with C11, have the compiler writers confused themselves enough to the point that they even forget the history of C. If newer versions of compilers get in the way of concurrency support implemented at function call boundaries, it is because the compiler writers have been sacrificing classical C at the altar of mostly useless optimizations to make benchmarks a bit better, they have forgotten of the users of C and the enormous code bases written in C, some of which stop working mysteriously when compiled by newer versions of compilers, and can only be kept working only when lots of misguided optimizations made by the compilers are turned off. The hijacking of C, and the spirit of C, by compiler writers is an additional motivation for the COOGL programming language, to have a better language for the users of C to move their code bases to.

15.4 Concurrency support in libraries is optional

Concurrency support by the language libraries is optional, there is value in very efficient support of single threaded programs. The libraries it uses should not contain needless synchronization that would only be required by multi-threaded programs, for example internal locking and unlocking of heap management data structures

when heap memory is allocated or released. The language provided libraries, when used for single-threaded programs contain none of that synchronization overhead. See §6.

15.5 Weakly ordered concurrent memory accesses

Weakly ordered memory, as opposed to *strongly ordered memory*, is present in most modern hardware architectures (ARMv8, IBM POWER, etc), it relaxes the program observable memory model under concurrent execution in a multi-processor system. For example processor 1 might perform two stores first into `a` and after that into `b`: through `processor1_stores()`:

```
int a1 = 0, b2 = 0;
void processor1_stores() { a1 = 1; f(); b2 = 2; }
int processor2_fetches() { return a1 + b2; }
```

Processor 2 fetches both variables through `processor2_fetches()`, adds the values of both variables, it can return: 0, 1, 2, or 3 depending on the timing of the concurrent execution of both functions and the underlying hardware operation, different concurrent executions might return different results. Note that for processor 2's function invocation to return 2 it must be that these were the values fetched: `a1 == 0` and `b2 == 2`, a counterintuitive result that would not occur on older computer architectures which provide strongly ordered memory systems (such as IBM zSeries and Intel/AMD x86). The invocation of function `f()`, a function unknown to the compiler, is there between the assignments to `a1` and `b2`, to ensure that the compiler performs the stores in the specified order.

A circumstance under which processor 2 might see the store to `b2` and not the store into `a1` is when each one of those variables is in a different cache line, the cache that contains `b2` is already held in an exclusive state by the cache of processor 1, while the cache line that contains `a1` is held in exclusive state by the cache of a different processor. The store into `a1` is held by processor 1 in a write queue while the cache line is acquired in exclusive mode by the processor, the store into `b2` occurs almost without delay into its cache line. While processor 1 has not yet acquired the cache line that contains `a1`, processor 2 sees `a1`'s old value, 0, and then proceeds to fetch the new value of `b2`.

This is how weakly ordered memory works, some compiler writers call this a data race and decide that it is undefined behavior and that the compiler can do whatever it wants, not what it was asked to do. The hardware has well defined behavior, any of the values 0, 1, 2, or 3 can be returned, there is nothing undefined about it, the author has never seen a computer system whose memory works in a way that results in the equivalent of C's undefined behavior. There is high value in a systems programming language that accepts every possible hardware behavior, and doesn't use some of that behavior as an opportunity to call it undefined behavior, and then uses that as an ex-

cuse for the compiler writers to cause behaviors that the hardware could not have caused by itself. A computer system execution mode with completely defined behavior, for example user mode, can not just be turned into an execution model with undefined behavior as a sacrifice to the altar of ungrounded compiler optimizations. A high level language should not have behaviors that are harder to understand than the behaviors of the underlying computer architecture supported by it. It is supposed to be higher level than the machine itself, its behavior can not be orders of magnitude more complex or confusing than the hardware behavior itself.

Weakly ordered memory improves performance and only needs additional special consideration in the presence of concurrent data accesses to provide a programming paradigm that is easy to reason about. Memory barriers and flushing of store buffers in the implementation of synchronizers and some other concurrent programming mechanisms is all that is usually required. For example if message passing between concurrent threads is supported, and the messages can contain pointers to shared memory, for example memory where the sender placed some data, that the receiver is going to use when it receives the pointer, then the sender must ensure that no data remains in its hardware write buffers before the message is sent. This is usually accomplished as a side effect of the use of synchronizers to implement the message passing. If the message passing is implemented with some lock free data structures that don't use synchronizers, then the functions that queue and dequeue the data would have to explicitly have the appropriate memory synchronization instructions.

15.6 Concurrency support in C

Programming language compiler code generation optimizations can interfere with concurrency support, for example compiler optimizations that result from caching values in registers, or fetching or storing data in an order different than the order specified by the programmer. An area that interferes with the optimization of C code is that any store into memory could affect most memory, or at least any memory whose address could be determined by other code. The generated code might waste a few cycles refetching data values after functions are invoked, i.e. after they return to the caller, because those values might have been changed as a result of the function invocation, this level of under optimization allows concurrency support in C to be implemented in libraries without any concurrency support in the language itself.

Lock free optimization algorithms have been developed that allow for concurrency with less interlock than the interlock that occurs in most traditional concurrent code in existence today. New processor architecture revisions and hardware implementations incorporate hardware transactional memory (HTM), which is the only new significant development in hardware support for concurrent programming. In essence it allows for all memory accesses performed by a small amount of code to be performed atomically with respect to any other execution, for example, the `remove`

function if invoked as a hardware transaction, the element p is removed from a doubly linked list without the partial steps of the removal being seeing by any other code, including other concurrent insertions or deletions into the list.

```
void remove(node *p) {
    p->prev->next = p->prev;
    p->next->prev = p->next;
}
```

15.7 Language design dilemma

A language design dilemma arises, should the language designer make all these choices and bolt his ideas into the language, or should he just provide the tools for many concurrency and synchronization models to be easily implemented? From threads bound to kernel threads, to pure user mode threads, to hybrid N:M thread models, to activations, to work item oriented pure asynchronous execution models that only have a stack and state while running but immediately loose their stack when blocked and later resume from formalized state, etc. From lock free programming, to message based, to traditional concurrent programming (with locks, shared exclusive locks, condition variables, etc).

Certainly various operating system aspects also need to be addressed, for example, interactions with operating system supported exception handling mechanisms, such as signals in UNIX. Input output completion mechanisms and their indications via various callbacks for various operating system APIs.

Various ways to manage stacks, thread contexts, cooperative blocking and resuming, preemption, etc. might be supported fully or partially by the operating system through APIs such as the UNIX `setcontext()`, `getcontext()`, `makecontext()`, `siglongjmp()`, `sigalstack()`, `sigmask()`, etc. An additional aspect related to all of this is that for a safe language such as COOGL, further considerations are required to ensure that these, and other, operating system interfaces are not used to defeat the safety of the language, for example by allowing contexts and longjmp buffers to be affected to affect program execution so that values that are in the restored CPU registers end up with pointers that refer to memory that is not supposed to be accessed through pointers of the wrong type. COOGL's library doesn't expose unsafe interfaces instead it exposes similar interfaces that are safe and are not subject to tampering by a malicious programmer attempting to introduce a backdoor into the code through through the underlying C interfaces.

The answer to the dilemma is that the language can not force any specific synchronization model, paradigm, or set of interfaces, it just has to not get in the way of their support through libraries. A minimal base synchronization library has to be provided to allow other libraries provided with the language to be correct in the presence of concurrency.

15.8 Concurrent programming building blocks

The language design consideration for concurrent programming support aspects was to allow for the underlying hardware facilities to be able to be used efficiently from COOGL code without the aid of assembly language.

Some of those hardware dependent facilities are:

- Underlying hardware synchronization primitives should be exposed in their raw form, or as close as possible to their raw form. This includes compare and swap operations, test and set, load linked and store conditional, atomic operations, etc.
- Underlying hardware memory barriers and other instructions related to the memory model of the computer system should be exposed.
- Underlying hardware mechanisms that support hardware transactional memory (IBM POWER, IBM zSeries, Intel TSX-NI, etc), database memory (ancient IBM RS/6000 POWER1 systems) if it were to re-emerge, capability based memory (IBM iSeries) if it ever becomes open, all should be exposed.
- Memory protection systems that allow for programs to isolate parts of themselves from each other should be exposed (IBM POWER).
- Intrinsic compiler functions that allow for thread context to be obtained, manipulated, or created without resorting to assembly language.
- Hardware aids required to deal with hardware stacks and their register windows such that user level threads can be implemented, for example by forcing the drainage of registers to the memory stack, etc.
- Hardware aids that allow for the support of thread local storage, either through registers or special per-CPU memory mapped pages, or through stack pointer rounding and fetching of per thread memory.
- Low level interfaces to exception dispatching mechanisms, such as dispatching to functions when various errors occur, for example divide by zero, or when invalid memory, or memory that is paged out and the page results in an I/O error.

Given that the number of CPU architecture diversity continues to shrink, to have these hardware mechanisms exposed by the compiler for each architecture, as appropriate, is not an unreasonable requirement to provide this level of deep architecture support.

Given these primitives, higher level primitives, such as stack creation, context binding to a stack, stack on demand growth, stack disposal, stack passing, discontinuous stacks, etc. should also be exposed by the compiler through its libraries, preferably in a platform independent interface.

Given the global compiler nature of COOGL, alias analysis can be performed deeply and the resulting C code can have the results of those optimizations expressed

as C local stack variables thus avoiding the needless re-fetching that C would otherwise do across every function call, when it is safe to do so.

15.9 Execution contexts

An execution context represents a unit of independent execution, for example an operating system kernel implemented thread, a user mode implemented thread, a coroutine, a hardware interrupt handler, a hardware exception handler, a kernel organized execution of an exception handler to handle an exception that originated in user mode, callouts to user mode coordinated by a kernel to allow user mode like interrupt handlers to deal with asynchronous events such as asynchronous input output completion handlers, or user mode interrupt handlers for device drivers implemented in user mode, etc.

An execution context has associated with it a run-time stack where local non-static variables are stored for the execution of a function, and the functions called by it, directly or indirectly. In some environments an execution context could have more than one run-time stack associated with it. For example an operating system environment where an alternative run-time stack can be used to handle exceptions caused by a thread during its execution, both stacks belonging to the same execution context, for example a thread.

The rationale for leaving the concept of an execution context purposely vague, is to ensure that the language is not attempting to dictate operating system concepts, doing so could make supporting the language difficult if the language were to to define the same concepts in ways that would make their support compatibility difficult.

15.10 Threads, mutexes and condition variables

The counterpart to having concurrency support built into the language is providing it through a library, possibly different libraries that implement various standards or concurrent programming models. If concurrency support is not provided, one way or the other, then other libraries can not be written in a way that would make their use correct, when concurrency is present. COOGL provides a thread and synchronizer library that is a subset of the most common operating system provided interfaces and can be easily mapped into them. The goal is completeness and simplicity, instead of a very rich and complex API. It is based on threads, mutexes, and condition variables which are available in every mainstream operating system. They are the common subset between POSIX threads and Windows. C11 `<threads.h>` is also based on the same synchronizers.

15.11 Weaknesses and complexity in C11 `<threads.h>`

In some areas C11 `<threads.h>` is under specified, and has some design flaws, for example acquiring a mutex should not ever return an error, allowing for error returns provides flexibility in the wrong place, concurrent programming is delicate enough, requiring every lock acquisition to have the possibility of failure and in consequence error handling to deal with it is unreasonable. Incorrect use of a mutex should not produce an error, for example acquiring a non-recursive mutex that the thread already holds should cause the thread to self-deadlock, or an exception to be raised.

Another area of underspecification or incorrect specification is that their description of `mtx_t` and `cnd_t` indicate that they hold an identifier for a mutex and a condition variable, respectively, as if they were values that could be copied around, assigned, used as arguments, or returned as the value of functions, as if they contained handles, similar to a file descriptor, to the synchronizers that are actually elsewhere. This extra level of indirection is not needed.

Lastly C11 `mtx_t` is a type that can not make up its mind, so it has four flavors reflecting its indecision: recursive or not, supportive of `mtx_timdlock()` or not, even `mtx_init()` is poorly specified, it uses 3 flags to specify the 4 possible flavors of a `mtx_t` when two would have sufficed, having both `mtx_plain` and `mtx_recursive` makes sense only if in the future they wanted to add another flavor. Wouldn't it be funny to see a mutual exclusion lock with a `mtx_sharedexclusive` flavor? Lastly, the memory for the `mtx_t` implementation data would have required additional memory in every `mtx_t` to support the `mtx_timed` lock flavor, and the lock owner and recursion count. The icing on the cake is that `mtx_lock()` would be more complex because even in the fastest path, mutex not being locked, it would have to obtain some kind of thread id to store in the lock instead of an immediate value.

15.12 Concurrency support in COOGL `lib.concur`

The problems in C11 `<threads.h>` are addressed by the COOGL library `lib.concur`, its interfaces are based on classes, it is similar to the C11 `<threads.h>`, but it is a much simpler interface.

Synchronizers, `lib.concur.mutex` and `lib.concur.cond` are meant to be declared within other data as members, they, are not meant to exist on the run-time stack.A `lib.concur.thread` object can only be allocated dynamically, it is not meant to live on the run-time stack, its constructor is `prot`, it `is lib.creatable`.

```
extend namespace lib {
    pub namespace concur {
        pub class mutex {
            pub void deinit() {...}
            pub void lock() {...}
            pub void unlock() {...}
            pub bool try_lock() {...}// if free locks it
            pub bool owned() {...}   // owned by this thread?
        }
        pub class cond {
            pub void deinit() {...}
            pub void wait(mutex *m) {...}
            pub void wait_timed(mutex *m, time_t time) {...}
            pub void wake_one(mutex *m)require(m->owned()){...}
            pub void wake_all(mutex *m)require(m->owned()){...}
        }
        pub class thread(void start() deleg,
                         bool detached = true,
                         thread_info *info = NIL) prot {
            pub is lib.creatable(thread);
            return;
            priv void deinit() {...}
            pub static void exit() {…}
            pub void join() {...}
            pub void yield() {...}
            pub void sleep(time_t time) {...}
            pub static thread *current() {...}
        }
    }
}
```

The `thread_info` optional argument specifies whatever run-time stack related arguments or scheduling information might be required when creating a thread, e.g. when the default choices need to be overridden, it is not described further in this chapter.

The following two pages describe `lib.concur`, a multi-threaded Sieve of Eratosthenes example program is shown in the next section. The `start()` function delegate function pointer argument to `lib.concur.thread` is used to pass whatever information the thread requires to do its work and to return whatever results it returns to another thread when it completes. Note that `start()` does not return a value, nor is a value allowed to be returned through `lib.concur.thread.exit()` or can a value be obtained from a thread when it exists via `thr->join()`. Whatever value is to be communicated back when the thread exits, if any, is communicated through the object that `start()` is a delegate for, this simplifies `lib.concur.thread` significantly as it is not in the business of returning values, usually of some compromised

type anyway. If values are to be produced by the thread when it exits, they are whatever is right for the programmer, they can be obtained from the object that `start()` is a delegate for when `thr->join()` returns.

The agreement between a thread that creates another thread is that if the thread was not created detached, i.e. that it will be eventually the subject of `thr->join()`, then it is the thread that performs the `thr->join()` the thread that destroys its memory through `thr->destroy()`. If the thread was detached, then the `thr->destroy()` will be done internally by `lib.concur.thread.exit()` whether it is called explicitly or implicitly when `start()` returns. In the first case `deinit()` for the thread object occurs in the context of the thread that performed the `thr->join()` and in the second case it occurs in the context of the thread itself while it is exiting, note that when a thread is exiting it is still a thread and it might still block while it is doing its work.

A class that inherits from `lib.concur.thread`, for example `iothread`, and adds additional non-static data members can provide a static `iothread.current_iothread()` member function that returns a pointer to the `iothread` if the current `thread` is an `iothread`, `NIL` otherwise.

With `iothread.current_iothread()` it can access its non-static data members, this serves the purpose of thread local storage without the complexity of C11 `tss_t` and its `tss_dtor_t` destructors and their convoluted specification:

```
pub class iothread {
    pub inherit thread;
    pub is lib.creatable(iothread);
    priv ioqueue work;
    return;
    pub static iothread *current_iothread()
        return try_cast(iothread *, NIL)
            lib.concur.thread.current();
}
```

An `iothread` that is specialized to be a file system I/O thread, an `fsthread`, can have its own thread local storage too:

```
pub class fsthread {
    pub inherit iothread;
    pub is lib.creatable(iothread);
    priv opqueue fswork;
    return;
    pub static fsthread *current_fsthread()
        return try_cast(fsthread *, NIL)
            lib.concur.thread.current();
}
```

This scheme could be used to support a class of threads that provide the more

baroque concept of thread local storage slots that are allocated at initialization time and into which pointers to data, per thread, can be stored and fetched.

To block a thread until a condition occurs, the `mutex` that protects state changes in the condition has to be locked, the `wait()` and `timedwait()`, both specify the mutex as an argument, the mutex is atomically dropped at the same time that the thread is blocked until the condition occurs. When the thread returns from the `cv->wait()` or `cv->wait_timed()` the `mutex` has already been locked on the thread's behalf. The thread must always determine again whether the condition that it blocked for is actually true, if it isn't it must block again or do whatever other action is needed, it must not assume that the condition is true, even if the condition was indicated to a single thread through `cv->wake_one()`, because implementations are allowed to spuriously unblock threads even if `cv->wake_one()` or `cv->wake_all()` was not invoked on the condition variable.

The `cv->wake_one()` and `cv->wake_all()` member functions also have the `mutex` that protects the condition as an argument, the `require()` that the mutex be owned by the current thread, because calling these member functions without the caller having locked the mutex could lead to a incorrect synchronization and threads blocking forever. Note that because spurious unblocking can occur, there is little value in complicating the interface of `cv->wait_timed()`, if the caller is interested in knowing if `time` elapsed it can obtain the current time and determine if that was the case, thus the synchronizers are even simpler, internally they don't need to produce and communicate this information so that it can be returned as a value by `cv->wait_timed()`, it doesn't merit the extra complexity.

Various implementations of lib.concur are provided, a production one with no debugging support, a production one with additional use checks, a performance investigation one with lock performance instrumentation, and a debugging one with extensive debugging support.

15.13 Multi-threaded Sieve of Eratosthenes and thread safe `queue`

A `queue` that can be accessed concurrently with capacity `N` of value-like objects:

```
pub class queue(pub genre lang.value type) {
    priv lib.concur.mutex mutex;
    priv lib.concur.cond not_full, not_empty;
    priv lit uindex N = 100;
    priv uindex count = 0;
    priv type data[N], *getp = data, *putp = data;
    return;
```

```
    pub type get() {
        mutex.lock();
        while (count == 0) not_empty.wait(&mutex);
        type val = *getp++;
        if (getp == data.end) getp = data;
        if (count == N) not_full.wake_one(&mutex);
        --count;
        mutex.unlock();
        return val;
    }

    pub void put(type val) {
        mutex.lock();
        while (count == N) not_full.wait(&mutex);
        *putp++ = val;
        if (putp == data.end) putp = data;
        if (count == 0) not_empty.wake_one(&mutex);
        ++count;
        mutex.unlock();
    }
}
```

The following example finds all the primes smaller than `N_SIEVE` concurrently:

```
primes p;
int main() {
    p.parallel_sieve();
    p.print_primes();
}
class primes {
    pub lit size_t N_SIEVE = 1L << 24;
    pub bool sieve[N_SIEVE];
    pub queue(int) doneq;
    pub queue(int) workq;
    pub void print_primes() {
        for (int i = 2; i < N_SIEVE; ++i)
             if (!sieve[i]) on (i; '\n') print();
    }
    priv void scratch_multiples(void *vp) {
        for (;;) {
            int prime = workq.get();
            if (prime < 0) lib.concur.thread.exit();
            int mult = prime;
            while ((mult+=prime) < N_SIEVE) sieve[mult] = true;
            doneq.put(1);
        }
    }
```

```
    pub static int known[] = {2, 3, 5, 7, 11, 13, 17, 19};
    pub void parallel_sieve() {
        lit int N_THR = 4;
        for (int i = 0; i < N_THR; ++i) {
            decl lib.concur.thread *thread =
                lib.concur.thread.create(scratch_multiples);
            assert(thread);
        }

        int queued = known.max[0];
        for (int i = 0; i < queued; i++) workq.put(known[i]);
        int last_prime = known[queued - 1];

        while (queued) {
            for (int worked; queued > 0; queued -= worked)
                worked = doneq.get();
            int stop = last_prime + last_prime;
            if (stop >= N_SIEVE) stop = N_SIEVE - 1;
            for (int scan = last_prime+1; scan <= stop; ++scan)
                if (!sieve[scan]) {
                    workq.put(scan);
                    last_prime = scan;
                    ++queued;
                    if (queued >= N_THR) break;
                }
        }
        // threads exit when given a negative prime
        for (int i = 0; i < N_THR; i++) workq.put(-1);
    }
}
```

15.14 Memory model and concurrency

Concurrent programs that use `lib.concur.mutex` and `lib.concur.cond` to coordinate and implement their shared memory accesses don't need to be aware of the memory model. Programs that operate on naturally aligned scalar types (the integral and floating point types) and fetch and store the values of pointers (not what they point to, but the pointers themselves), can operate on the shared data without concern that somehow the data in individual concurrent fetches and stores will get commingled in a way that the values in memory are values that were never stored into it. If the concurrent access makes sense to the application, the compiler will not get in the way of them, the compiler will not perform actions unknown to the programmer in some misguided optimization attempt. For example, a series of 7 stores of `true` into an array of 8 `bool` that the compiler can determine that is aligned on a 64 bit boundary will not be transformed by the compiler into a 8 bit load of the 8th bool into a 64

bit register, and then setting the other 7 bytes in the register to the value `true` (i.e. one), and then store back the 8 bytes. Such an operation is reading and writing back the 8th `bool` and it might cause a concurrent store into it by another thread to disappear, as if it was never occurred. The compiler could generate code that builds the 7 byte string into a register and then perform a 7 byte string store as long as the operation does not cause the 8th bool to be stored into, i.e. if the compiler support such instructions and doing so is somehow more efficient. The compiler could produce a 32 bit store, a 16 bit store, and an 8 bit store, to implement the 7 stores, such stores are indistinguishable from individual stores in every computer system.

Similarly, systems with register pair loads and stores, or multi-register load and store instructions, those instructions can be generated by the compiler as long as the stores correspond to stores requested by the programmer.

From the programmer's perspective and the computer hardware behavior each thread of execution perceives its operations as if they occurred sequentially wherever the language specifies order relationships between them, for example, the effects an expression statement are to occur in such a way that the next expression statement perceives the first one as if it occurred in order. These are the sequence points defined in the C89 standard. Subexpression evaluation, argument expressions, and others proceed in unspecified order, unless documented to occur otherwise, thus the programmer can not depend on the evaluation order. For example: `a[i++] = i` the value stored into the array can not be determined. The compiler produces compilation errors for these expressions, but it is possible, through pointers that refer to the same memory without the compiler knowing about it, to construct expressions whose side effects are such that the results of the operation are unknown, they might be what the programmer expected, and after some unrelated code changes the results might be different when the code is recompiled or because the compiler version might have changed and it might have chosen a different optimization strategy, or maybe because the optimization levels and related parameters to it were changed. Whatever the results are, they are not undefined behavior, the results are just unspecified. For example, **memory unrelated to the data in question will not be affected arbitrarily**. Assuming that `p` and `q` point to the same `int`, which for the sake of argument we will assume has the value `7`, and that the compiler doesn't know about it: `a[*p] = (*q)++` then either `a[7]` or `a[8]` are affected, and the value stored into one of them is `8`.

The compiler and the computer hardware are allowed to reorder operations so that they occur efficiently as long as the thread that is executing them can not perceive them to have occurred out of order, other threads could perceive them to have occurred out of order. Whenever a function call that the compiler doesn't know about its code is invoked, everything needs to be in memory as the programmer programmed it to be, but the computer hardware itself might still be in the process of draining its write buffers and the operations can be perceived to occur out of order by

concurrently executing code unless the called function, for example a mutex being unlocked, performs the hardware actions required by it, for example a memory barrier instruction prior to the store that releases the mutex. In consequence, concurrent algorithms that don't coordinate their shared memory accesses with synchronizers, have to be written very carefully, in ways that force the compiler to do what the programmer requested, and that the computer hardware does so too, from the perspective of hardware concurrent execution, from the computer hardware's memory model. For portable code a weakly ordered memory model should always be assumed, code correct for it will also be correct for hardware with a strongly ordered memory model.

15.15 C11 and C++11 memory model

C11 and C++11 both specify a memory model that is meant to be the same across both standards, the memory model has the same origin, first going into C++ and from there into C. The description of the memory model is tremendously complicated, written in english prose in such a way that it attempts to be precise but doesn't include proper definitions and uses many words with meanings whose definitions are not easily found in the standard, if they are present anywhere at all.

The C11 and C++11 memory model includes notions such as data races and that data races lead to the dreaded undefined behavior, meaning there will be security holes introduced by the compiler behind the programmers back, and thus the NSA, KGB, and other spy agencies will be glad that the backdoors are being introduced by the compilers so can exploit them. Malicious hackers will do the same.

15.16 COOGL memory model

If you can decipher the C11 and C++11 memory model that is the memory model of COOGL but without the notion of data races and the undefined behavior. COOGL is a language for the real world, not a language for future mythical machines that the C11 and C++11 language standards seem to attempt to want to leave the doors open for.

The memory model of COOGL is a weakly ordered memory model, it mandates that loads and stores of the fundamental types are indivisible if they are done to addresses that are multiples of the data type size. On a 32 bit system concurrent loads and stores of 64 bit integral types (`ularge` and `large`) might not be performed atomically, because they are usually implemented with individual 32 bit loads and stores because the 32 bit instruction sets usually don't have 64 bit loads and stores. This of course, is just a reflexion of normal hardware. The programmer is supposed to know what he is doing, the compiler will not penalize him by saying that these are data races and that the whole program will misbehave, the dreaded undefined behavior will not be triggered. Note that loads and stores of 64 bit floating point values are

atomic on 32 bit computer systems, other than some stone age ones that might implement floating point by emulating it with integer instructions, those are not interesting, and they certainly don't tend to support multiple CPUs in an SMP configuration.

Portable functions that provide access to the underlying hardware operations required to work with a weakly ordered system are provided by the language libraries, see §L.4.

15.17 Atomic memory operations

Atomic memory operations supported by the hardware are provided through portable libraries. If an operation on a specific data size or a specific operation is not supported, then the corresponding library function is not provided and the compilation will fail, the programmer will have to deal with it by implementing an alternative operation, the supported interfaces are provided in `lib.atomic`. Where possible the missing operations, implemented in software, possibly at considerable expense, are provided in `lib.atomic.missing`. A program can import lib.atomic and lib.atomic.missing into its own address space, say `app.atomic` and access them from there when it is initially being ported, the programmers can later investigate the impact of the user of the slower missing ones and figure out what they want to do.

15.18 Exception handlers

If an exception handling function returns, the instruction that caused the exception to be raised is retried, causing the exception to occur again, unless the handler performs some action that causes the exception to no longer occur, for example, by mapping memory into an area where memory was not previously mapped, or changing the protection of the memory area that caused the exception.

XXX Exception handlers are per thread, arguments, etc. Alternate stacks. Kind of like per-cpu hardware interrupt stacks. Should exception handlers be allowed to block, etc? Then can not share pool of exception handling stacks, but if they can block then they could block to acquire an exception handling stack, but that could lead to resource exhaustion related deadlocks, then again exception handling is supposed to be exceptional, not the bread and butter of a software design, at least not in the exception handling paths which should be brief and non-blocking. Memory footprint costs, sharing of exception handlers, between threads (to reduce cost), etc. Need the minimalist mechanism which is UNIX signals with alternate stacks, etc. Need alternate stacks for when the thread stack overflows. In Windows could use the Vectored Exception Handling features. See also `SetConsoleCtrlHandler()` for Windows console processes.

XXX Should a proposed standard C library extension that is OS neutral be designed? In Windows stack space exhaustion might need to be carefully managed to

ensure that at least there is some space to switch to an alternate stack when required, so minimal stack space might need to be guaranteed. Language level specification vs C library signals setjmp/longjmp to an outer layer. Initially implement POSIX signals on Windows and divorce the design of a new API from this language, only adopt it if it becomes part of C in the future.

Appendix 1L – Libraries `lang`, `lib`, and `libc`

> *"C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays considered as a whole. There is no analog, for example, of the PL/1 operations which manipulate an entire array or string. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions: there is no heap or garbage collection like that provided by Algol 68. Finally, C itself provides no input-output facilities..."*
>
> *"Although the absence of some of these features may seem like a grave deficiency ("You mean I have to call a function to compare two strings?"), keeping the language down to modest dimensions has brought real benefits."*
>
> -- Brian W. Kernighan and Dennis M. Ritchie

The language run-time support is described in this appendix, programmer accessible aspects of it are exposed in the `lang` namespace. The language library, `lib`, and a safe subset of the standard C library, `libc`, are described.

1L.1 Generic function `lang.on_array()`

XXX use `argsof` and arguments now have to be in a the table which has to be augmented with a place for argument values, but if the evaluation of the arguments is non-trivial and contradicts the sequential execution model, then it can not be done, usually it can, it is just passing a pointer argument, e.g. the `FILE` pointer to be used.

```
extend namespace lang {
    on_array.type on_array(pub genre lang.integral type,
                           on_array.delegate delegates[])
                               require(delegates.max[0] > 0) {
        pub typedef type (*delegate)() deleg;
        delegate *dp = delegates;
        delegate d = *dp;
        type n = d();
        if (n > 0)
            while (++dp < delegates.end) {
                d = *dp;
                type c = d();
                if (c <= 0) break;
                n += c;
            }
        return n;
    }
}
```

1L.2 Obtaining the object that contains a field `field_to_obj()`

The function `field_to_obj()`, field to object, is used to obtain a pointer to an object from a pointer to a field within the object, is a language provided function, its `fieldp` argument is a pointer to `field_type`, `type` has a member of `field_type`, its name is specified in the `field` argument. If `fieldp` points to the `field` member of `type`, then a pointer to the containing object is returned, otherwise `NIL` is returned.

```
extend namespace lang {
    priv field_to_obj.type
        *field_to_obj(pub genre void type,
                      genre void field_type,
                      fieldof type field_type field,
                      field_type *fieldp) { ... }
}
```

1L.3 Atomic array descriptor fetching and copying

Atomically fetching an array descriptor that is not a local variable or argument to a function is lock free and not more than 2 times as expensive than the underlying loads to obtain the data from the array descriptor. Atomically updating the array descriptor is also lock free where the hardware has the correct instructions to atomically update two pointer sized words, ignoring 32 bit environments because they are not

really interesting anymore for multi-threaded code on SMP systems. Single thread code has similar issues on a single CPU system, but they are easier to solve, 32 bit on SMP is becoming extinct, we look forwards with the language not backwards. Implementations exist for 32 bit SMP, they are not worth discussing here (for example, just use a generation count as in the more than one dimension case below).

ARM64, x86/64, and POWER all have 16 byte (data must be 16 byte aligned) atomic memory load_linked/store_conditional or x86/64 cmpexch instructions. Unidimensional array descriptors implemented as two 64 bit words: `start` and `max[0]`. (`end` is computed at runtime in this case). The value of end can be assumed to exist and be valid in array descriptors that exist in the run-time stack as part of the calling convention so they don't need to be recomputed when they are passed around as arguments to functions. The calling convention on systems with enough registers might pass the 3 values, other architectures might just pass the two values instead. Note that having `max[0]` instead of `end` be the better choice of the value to represent in array descriptors stored in global memory is based on the fact that multiplication to compute `end` is much faster than division to compute `max[0]`, and that quite often the multiplication is against a small literal and the compiler can perform those frequently without hardware multiplication.

Atomically fetching the 16 byte value, without performing stores, makes use of a register pair load, and a generation count. A register pair load, even though obviously not atomic, is an indivisible instruction, meaning that a thread can not be preempted in the middle of it, so there is no concern for having to hook the scheduler to restart the instruction sequence. To fetch the two values a register pair load is used, but because the values might not really be a logical pair but a broken pair a generation number stored in both would have to be used. Using a relatively narrow generation number is good enough because there are no unbounded long lived preemptions between the two values being read, the cache line could conceivably be bounced around between CPUs for a period of time in between the two hardware issued fetches for the register pair load, but for a large enough generation number it is inconceivable that two words that exist in the same cache line that makes it to the L1 cache of a CPU would continuously be removed from the cache as the two words are repeatedly updated in other CPUs and eventually the L1 coming back to have its second word fetched and have the generation number coincide with the generation number fetched from the first word. For example with 64 bit words, if 16 bits are dedicated to the generation number, it would have required 64K updates of the array descriptor while a CPU stalls between consecutive loads of two words that coexisted in the L1 while the first word was satisfied. An extra safety net could be built that counts the number of retries from mismatched generation numbers, if the number of mismatches is larger than a reasonable value an exception could be thrown as the repeated storing

into the array descriptor could be part of an attempt by malicious code to defeat the language safety.

Atomically updating with 16 byte atomic update requires reading a consistent prior value against which to perform the atomic update because the prior version of the generation count has to be updated correctly, thus the atomic fetch is first performed and the values obtained are the basis for the 16 byte atomic update after incrementing the generation count.

If a register pair load is not available then a different implementation approach would have to be used because a thread preemption between the two word loads could allow for the generation count to wrap and produce an incorrect generation count mismatch. The thread preemption could could honor an instruction restart region and simply backup the program counter appropriately so that when it resumes it retries the two word fetches.

Threads could have a preemption count which they could fetch prior and after to see if they have been preempted. This is a much cleaner interface than restartable PC backing by the kernel. The kernel just has to increment the per-thread preemption count and not be aware of special address ranges for restartable instruction sequences. Per-thread preemption value would be part of thread context and when the thread is running it could be exposed (read-write! For user mode thread multiplexing!) at a well known address per-core. So apart from validating gen was the same it would also ensure that preemption did not happen. Note that running an interrupt handler and resuming from the kernel to the same thread wold be considered a preemption because arbitrary code ran while the thread was running. If the kernel is updating the preemption count, and user mode is updating it too, then they have to be done with a ll/sc pair in user mode (and kernel mode too if the hardware architecture requires it), note that with user level thread switching on a core, the per-thread sequence numbers are being context switched too to the well known per CPU location.

Array descriptors with more than one dimension would require 3 or more words, so they might as well also store end in memory, their non run-time incarnations would have a generation count in them. Their atomic fetch would fetch the generation count, use the value fetched as a data dependency to compute the address of the other values and a final re-fetch of the generation count. If the generation count did not change then the atomic fetch succeeded. A special generation count, say zero, would mean that an update is in progress and the code fetching the values atomically would have to spin until a stable generation count is obtained. A thread updating an array descriptor would use the low values that are CPU numbers as the generation count and establish it atomically if it was not a CPU number already. A thread that has thus locked the generation count and was preempted would require its preemption to go through a

release sequence prior to the preemption to restore the generation count and its program counter would have to be reset to the start of its instruction restartable sequence for when it resumes. Communication to the thread preemption code might involve a per CPU location that stores the address of the generation count, the value to be restored, and the program counter to back up to, note that because the values of the array descriptor are in the middle of an update their prior values would have to be rewound, but this all amounts to a way of making backwards possible, it would be just as easy to have the preemption code to complete the update instead of undoing it, instead of storing the pre-values, what would be stored in the preemption per CPU area would be the post-values and new generation count desired and the starting address where the stores should be performed, the read would then atomically fetch the generation count, compute the new one, all the new values, and last store the address of where the new values go, this final store which is perceived by the CPU in program order tells a future preemption that if the generation count in the data is the CPU number then the values are to be used for the update, if the generation count is not the CPU number, it means that the thread not yet locked it, so the program counter can be safely backed up to the point where the thread refetches the generation count and is about to store it again in the per-CPU memory, note that for this preemption the pointer to the data would to updated to zero to indicate when it resumes and if it is repreempted nothing needs to be done. Note that an immediate preemption after fetching the generation would be caught by the compare-and-swap like operation that ensures that the storing of the CPU number in the generation is not done over a newer generation number or some other thread having stored their CPU number there.

In the discussions above intimate interactions with the scheduler through user mode accessible and documented shared memory that is per CPU is implied, depending on the environment thread id could be used instead of CPU number and per thread memory instead of per CPU memory, the details are similar but more intricate and vary from system to system.

Note that memory barriers are not required as assignment into or fetching from global array descriptors can still be weakly ordered with respect to other data.

1L.4 Weakly ordered memory control

Portable functions that provide access to the underlying hardware operations required to work with a weakly ordered system are provided by the language libraries.

1L.5 Standard input output

XXX

1L.6 String literals and the `str` string type

XXX

Appendix 2S – Identifier mapping and calling convention

"C is peculiar in a lot of ways, but it, like many successful things, has a certain unity of approach that stems from development in a small group."

--Dennis Ritchie

The memory layout of a `class` is specified by the language. Binary compatibility between completely separately compiled modules is supported through interfaces exposed through classes, even if different compilers are used. The layout of classes that have not been published (see §8.4) is not dictated by the language and are subject to heavy optimization.

Compilation of COOGL source code into C11 requires that identifiers be mapped to C11 identifiers in a way that accounts for their accessibility modifier and the scope where they were declared. This chapter explains this identifier mapping. In other languages this process is referred to as *mangling*.

2S.1 Introduction to the calling convention

A calling convention, in programming languages, is a description of the argument passing and value returning convention followed by the language to implement functions and function calls. Calling conventions are usually specified in a computer architecture specific way, for example what registers are used to pass integral arguments, what registers are used to pass pointer arguments, what registers are used to pass floating point arguments, how are structures passed by value, what is done when there are not enough registers to pass all the arguments via registers, for example how is the run-time stack call frame laid out to pass additional arguments that can not be passed in registers. Additionally, how values returned by the function returned, for example in which registers the values are returned, if possible, and if they can not be returned fully in registers how are the values returned in memory. An aspect of the calling convention is which registers are to be preserved by the function and which registers can be changed without preserving their values for the calling function. The calling convention also includes details about how the run-time stack is organized, its stack frame alignment, what memory can be used, etc. If these conventions are followed properly then other tools such as debuggers can examine the run-time stack

and reliably show the various call frames and the data of the functions.

Modern calling conventions pass quite a few arguments in registers and usually follow a similar register assignment convention to return values from the function. Older calling conventions might be limited in the number of registers used to return values. Machines with dedicated address and integer registers are not in common use anymore, all modern systems pass pointers and integers in general purpose registers and floating point values in floating point registers.

COOGL is compiled into C, its calling convention is specified in C, to understand its calling convention at the assembly level you have to understand the calling convention of the underlying C compiler. Understanding the calling convention at the C level is more important because it will help you understand how to call C code and how it calls into COOGL code.

2S.2 Hidden arguments: `this` and `on`

Hidden arguments to functions are passed by the compiler on behalf of the programmer, for example the `this` object pointer when invoking a non-static member function, e.g. `stack->push(10)`, the function at the C level receives two arguments, `this` and `v`, the value to be pushed, the first argument is `this` followed by the arguments of the function.

Another example is when the class constructor is invoked it receives a hidden argument, named `on`, that points to the memory on which the object is to be constructed, note that that argument is not the `this` argument, the on argument is specified after the arguments to the constructor. Constructors don't have a this argument unless they themselves are a non-static member function of another class, see the `iterator` class that is a non-static member function of class `stack` in §4.14. This means that a class constructor that is also a non-static member function of another class receives both of these hidden arguments: `this` and `on`.

2S.3 Tuple arguments and return value

Functions that have tuple arguments receive those arguments as individual arguments, not as a structure. Functions that return tuple values return the tuple value in a structure where each member of the tuple has a corresponding structure member with the same name and type. If a future version of C includes support for tuples, and COOGL is adapted to support that version of C as its target language, then, as an evolutionary option, the compiler can be directed to return tuple values natively as native C tuples instead of returning them through structures.

Note that modern calling conventions, for example 64 bit ARM, are able to pass and return structures by value through calling convention registers designated for argument passing, at least when the structure fits completely in those registers. 64 bit little-endian POWER calling convention also allows for this. The RISC-V calling convention doesn't. Some older calling conventions only allow one value to be returned in registers, others allow up to two values. The most common tuples have two pointer sized (or smaller) values, most calling conventions are pretty efficient already in this case, for example both 32 bit and 64 bit x86 allow structures with two words, 2 x 32 or 2 x 64 bits respectively, to be returned in two registers.

2S.4 Arguments that are a value object

An object passed as a value to a function is constructed by the calling function from another object, by either calling `init()` or `init_deinit()` on the source value. Only the calling function knows which of these calls is appropriate, if the source object doesn't need to be deinitialized, then `init()` will be invoked, but if the source object will be deinitialized immediately, then `init_deinit()` is invoked. The memory for the object that is being passed by value must be allocated by the calling function (for example as a local C variable), in consequence, when objects are passed by value the address of the object is what is actually passed as an argument to the function being called. An obvious optimization performed by the compiler, is that if the source object is to be deinitialized, after being used to initialize the value object, then instead of even invoking `init_deinit()` it can instead just pass the address of the value object.

Very small objects, a few words, might be more efficiently passed in registers than creating the object in memory and passing a pointer to it. For example an object that completely fits in a single register would be passed more efficiently in a register instead of creating it in memory just to pass a pointer in a register to it. All objects that fit in a register, are constructed in a C variable and passed by value at the C level.

It is enticing to pass objects by value when they use more than a single register, say two registers, and when the underlying C calling convention for passing structures by value would pass the structure in registers. The problem with doing this is that such a structure might internally have pointers into itself and passing that in registers and then having the receiving function put those registers in memory to dereference the memory would cause the pointers to point to the data on the calling side and not on the receiving side. By restricting this optimization to register sized objects, the object even if it is internally just a pointer, could not point to itself, so it is safe to do so.

It is also tempting to do other similar optimizations if the object is just plain data.

For example an object that has not been published within the module that defines it, this optimization could be done for it globally within the module, for example if the compiler detects that it is passed by value significantly. The current compiler doesn't implement this, but it might in the future if there is demand for it. An alternative solution, for the programmer, for such plain-data objects would be to implement them in structures and depend on the efficient passing of structures by value from the underlying C compiler. A future enhancement to the language might allow the programmer to specify that objects of a specific class type should be passed by value following the structure passing convention.

Allowing the calling convention to be specialized for value objects depending on their size (i.e. if they fit in a register or not) does not impose additional coupling between separately compiled modules because those modules already are exposing the size of their objects to each other when they objects are allowed to be declared on the stack, assigned to each other, and initialized from each other. The size of the objects become part of the *application binary interface* (ABI) exposed by the module. The only way to decouple knowledge about the size of specific objects between modules is to have their constructor be inaccessible, i.e. to be priv, and requiring that the objects of that type to only be created dynamically, or if pre-allocated, to be pre-allocated by the module that defines their class which can then pass pointers to them to code in other modules. Objects not meant to be members of other objects can also benefit from this technique. Classes without a priv constructor also expose their size as part of an ABI when declared to be members of, or inherited by, other objects.

2S.5 Return values that are a value object

Objects returned by value, that fit in a register, are returned as value. Objects larger than that, that are the single value returned by a function, or a value returned as part of a tuple, are optimized better if they can have memory for them in the called function (instead of the calling function). The calling convention for receiving object values returned by a function passes the address of where the objects values should be placed as additional arguments passed to the function through the normal calling convention. There are two possible cases for the receiving objects, they either already exist as objects and are being assigned to, or are newly declared objects being initialized, its either one of these two, not a combination of these cases.

An additional argument that indicates either that the values returned are being assigned to existing objects, or that they are used to initialize objects being declared. By passing an extra argument with this information, the most optimal function can be used, i.e. `reinit_deinit()` or `init_deinit()`, respectively, but in some cases it might need to use `reinit()` or `init()` instead, for example when the value object

being returned is not about to be deinitialized because it will continue to exist after the function returns, for example because it exists in a globally reachable data structure. If the function just returns the value of another function that it calls, then it just has to pass the extra arguments with this information and it is no longer involved in the decision making process, supporting tail recursive functions efficiently.

The calling convention for functions that return object values passes a hidden argument, `bool return__init`, followed by one or more addresses for the receiving value objects that have already been constructed (when `return__init` is false) for example because they are being assigned into; or the raw memory where the receiving value objects are to be initialized (when `return__init` is true). Note that when a function is inlined, the return__init argument, usually a constant argument, and any generated if statements based on it, are removed by the compiler.

Values returned by functions that return a tuple, that are not objects, are returned as if all those values were part of a C structure, that structure is returned by value and subject to whatever is done for those by the underlying C compiler's convention.

2S.6 Unidimensional array descriptor arguments

Unidimensional array descriptors, a `lang.vecdesc`, are a type implemented by the language, their value fits in two registers, its two members to hold the values of `start` and `max[0]`, their argument passing convention doesn't follow the convention of objects of user defined classes. The `start` member value is passed in the argument for the array descriptor, and the `max[0]` member value is passed in the next argument. For example:

```
void scale(double vec[], double factor) {
    for (index i = 0; i < vec.max[0]; i++) vec[i] *= factor;
}
```

Is compiled into this C code:

```
void scale(double *vec, size_t vec__max0, double factor) {
    for (index i = 0; i < vec__max0; i++) vec[i] *= factor;
}
```

The prototype produced for this functions forces calls to `scale()` from C to provide the `vec__max0` argument.

2S.7 Array descriptor return value

Array descriptors are returned as a value as if they were a C structure.

2S.8 Multidimensional array descriptor arguments

Multidimensional array descriptors are passed as a value as if they were an object of a user defined class, the calling function ensures that the array descriptor, if its value is coming from a data structure, i.e. not a local variable or an argument, that it is atomically copied into memory in the calling function, the calling function passes a pointer to that memory, even though a pointer is passed, if that function subsequently passes the array descriptor to other functions it simply passes the pointer it received.

If a function that receives an array descriptor as an argument assigns to the array descriptor, then the compiler makes a local copy of the array descriptor for the function to manipulate and change. In general functions don't assign to array descriptors received as arguments, the compiled code can depend on the invariant that they are not addressable from any other thread and that their values don't change when passed as an argument to another function, the only code that can change it is the function function itself, so its compilation is fully aware of the only source of changes to it. For example the function can cache the `start` and various `max[]` values into registers without concern about these values changing, it can still pass the pointer to the array descriptor to other functions without any concern about it changing.

2S.9 Internal and external identifiers

Assuming these global declarations:

```
pub int func(int arg) { return arg + 1; }
pub int var;
pub enum state { free = 0, valid = 1, io = 2 };
pub struct huge { large data[8]; };
```

When compiled into C11 code their names, at the C level don't change.

The identifiers `func` and `var` are external identifiers from the perspective of C. External identifiers correspond to executable code or data declared in the global scope or data that is declared `static`, such code and data exist at fixed memory address locations while the module is loaded in memory and able to be used at runtime.

The opposite of an external identifier is an internal identifier. Internal identifiers are associated with non-global and non-static data, for example a local variable, a function argument, or a non-static data member. Internal identifiers are also associated with entities that don't exist in memory during execution of the compiled code, for example an enumeration type and its value, or a literal declaration, or the type declared by a struct declaration. Above `arg`, `state`, `free`, `valid`, `io`, `huge`, and `data` are all internal identifiers.

An external identifier length limit might be imposed by the static or the run-time linker, and not just by the underlying C11 compiler. The internal identifier length limit is imposed, usually, only by the C11 compiler, certain environmental factors such as field length limits on debugging information formats, or limitations of debuggers might cause the internal identifier length limit to be smaller than what the compiler might support internally. For example, the same compiler on two different platforms might have different internal identifier length limits.

The C11 standard requires that external identifiers of up to 31 characters and internal identifiers of up to 63 characters be supported. To ensure maximum COOGL source code portability, the programmer can choose that these be the limits used instead of the actual limits imposed by the underlying compiler. Additionally, programmers can choose their own limits, presumably the most restrictive limits across the platforms supported by their software. Limits larger than supported by the target platform for which code is being compiled are not allowed.

2S.10 Identifier mapping from COOGL to C

When COOGL is compiled into C most identifiers have their names adjusted so that they can exist within the single global identifier name space of C code, both global internal and global external identifiers exist in a single global name space in C.

Only pub declarations at the outermost scope have their names unchanged when compiled into C code. All other names require some amount of adjustment of their names when translated into C, the mapping into C is described in this chapter.

The language reserves double underscore for identifier mapping, user defined identifiers can not contain double underscore, nor can they start or end with underscore. Sometimes triple underscore is used, in those cases, the triple underscore is shown in red bold, i.e. ___,to make it easier to discern from double underscore.

When file names are included as part of an identifier mapping, the extension of the file name is always ignored, and every slash is replaced by double underscore, for example `src/file.cog` is mapped to `src__file`. Note that all filenames are always relative to the top of a hierarchy where they exist, file names are not absolute, they can't start with slash, walking up the directory hierarchy via the parent directory is not allowed, i.e. `..`, it is also not allowed to use the current directory, i.e. `.` in filenames. For example, these are not allowed: `/s/f.cog`, `../s/f.cog`, `s/../f.cog`, and `./f.cog`.

The general rule for use of triple underscore vs double underscore is that triple underscore separates entities of these different kinds from each other: modules, filenames, and identifiers. Whereas double underscore separates entities of the same kind

(components of a pathname from each other, or identifiers from each other).

Module names and filenames must be proper COOGL identifiers with the exception that the ASCII minus character, '-', can be used too, it is a synonym for underscore and is often preferred by programmers than underscore in their file names.

2S.11 Identifier mapping: global declarations outside of lexical scope

If the identifier declaration is global, and not within a lexical scope, for example not inside a namespace or a class declaration, then: `pub` declarations remain unchanged; `priv` declarations are prefixed by `priv___module___file___`; and `prot` declarations are prefixed by `module___`. The identifier space within which global `priv` declarations outside a namespace exist is the identifier space within a single file, because file names only have to be unique within a module, the identifier mapping for them has to also include the module name. The identifier space within which prot declarations outside a namespace exist is the identifier space within a module, in consequence these identifiers only need to be prefixed by the module name. The default module name is `mn`, short for the main module. For example the source file `src/file.cog`:

```
pub  int a;
priv int b;
prot int c;
int c2;  // default for globals is prot
```

is compiled into this C code:

```
int a;
int priv___mn___src__file___b;
int mn___c;
int mn___c2;  // default for globals is prot
```

Note that when you are debugging a program, for example in a command line debugger you are always within some context, when you are examining code, setting breakpoints, single stepping, etc. you are always working in a specific location, inside a function, inside a file, and inside a module. When you specify names to the debugger, your names are interpreted relative to your current location, as if it were your current directory, so you refer to them relative to the current scope. Also you specify them in their unmapped form, using their original names.

2S.12 Identifier mapping: global declarations inside a lexical scope

If the declarations were within a language lexical scope, for example declared within a namespace or members of class, then the module and file that contains them

are completely irrelevant, their accessibility modifiers dictate their identifier mapping. Note that in this case the identifier that provides the lexical scope for the declaration has to be mapped too, if that identifier is a global declaration that is not inside lexical scope, that identifier has its name mapped as described above, otherwise it is mapped by the same means as identifiers are mapped when inside a lexical scope. For the following example we will use a global `pub namespace prog`, because the mapping for `prog` doesn't affect its identifier.

When declarations are inside a lexical scope the `pub` accessibility qualifier doesn't affect the identifier mapping; `priv` causes a single underscore to be appended to the identifier as part of its mapping; and `prot` causes a double underscore to be appended. This code:

```
pub namespace prog {
    pub  int a;
    priv int b;
    prot int c;
    // int invalid;   // error: accessibility modifier required
}
```

Is compiled into this C code:

```
int prog__a;
int prog__b_;
int prog__c__;
```

If `prog` was declared `prot`, i.e.: `prot namespace prog {...}`:

```
int mn___prog__a;
int mn___prog__b_;
int mn___prog__c__;
```

If `prog` was declared `priv`, i.e.: `priv namespace prog {...}`:

```
int priv___mn___src__file___prog__a;
int priv___mn___src__file___prog__b_;
int priv___mn___src__file___prog__c__;
```

2S.13 Exceeding the external identifier length limit

Module name and file name components in the identifier mapping can be encoded differently on platforms that have a restrictive external identifier length limit, e.g. when a module's filename's are too long and cause the identifiers to become too long for the target platform. In the alternative encoding the module name and all the file name components are replaced by a number whose first digit is 0-9 (which makes it different than a module name which must be a valid COOGL identifier), followed by digits in base 62 using the characters 0-9A-Za-z, each digit representing values in the

range 0-61 (0-9 is 10 digits, and A-Z and a-z are 26 digits each, 62 = 10 + 26 x 2). Four digits constructed this way are used to encode the module and filename mapping. The first two digits represent the module name, allowing 10 x 62 = 620 possible module names, the last two digits represent the file name within the module, allowing 62 x 62 = 3844 files per module. The mapping between file numbers and file names is kept in a file managed by the compiler, the mapping between module numbers and module names is managed by the compiler and its tools. The encoding is actually variable length. A minimum of 4 digits must be present. If there are 2 x N digits, the first N belong to the module number and the second N belong to the file number. If there are 2 x N + 1 digits, the first N + 1 belong to the module name, and the other N to the file number. For example an operating system kernel with a particularly large set of device drivers, more than 620, and where unique symbolic information for all of them is required to be produced even though they won't all ever be loaded into a single running kernel at the same time.

Note that this is mostly a legacy C compiler problem, most compilers and linkers support very large symbols to support the C++ name mangling done by most implementations of that language. The programmer can choose to use these shorter identifiers for other reasons, by using the compiler option `--base62-names`, for example if other unrelated tools have trouble with these long identifiers.

2S.14 Identifier mapping for a class and its members

A class declaration declares both a function and an underlying C struct used to implement the data layout of objects of the class type. Assuming the `stack` declaration in §1.3, but assuming it was declared `pub`, so we can ignore the identifier mapping for `stack` itself, and that it was in the source code file `stack.cog`. Its compilation results follow (comments and `#line` directives were removed, the code was reformatted to fit, and automatic inlining was disabled), the two resulting files `stack.h`:

```
typedef struct stack__struct stack;
enum { stack__MAXENT = 100 };
struct stack__struct {
    int *sp_;
    int entries_[stack__MAXENT];
};
void stack__promise(stack *on);
void stack__on(stack *on);
```

```
void stack__class(stack *on);
bool stack__empty(stack *this);
bool stack__full(stack *this);
void stack__push(stack *this, int v);
pub int stack__pop(stack *this);
pub int stack__top(stack *this);
```

The second file produced is `stack.c`, below. Note that implicit `this` maps to `this` and the memory were an object is to be constructed maps to `that`. Remember that the class constructor does not have a `this` argument (unless it is itself a non-static member function of another class, see §4.14), instead the class constructor receives an extra hidden argument after the class arguments (see §XXX calling convention), which is named `on` in the compiled code, the `raw` pointer that refers to the memory that is to be constructed to be used as a `stack`.

```
#include <lang.h>
#include "stack.h"
void stack__promise(stack *on) {
    promise(stack__empty(on));
}
void stack__on(stack *on) {
    on->sp_ = on->entries_;
}
void stack__class(stack *on) {
    stack__on(on);
    stack__promise(on);
}

bool stack__empty(stack *this) {
    return this->sp_ == this->entries_;
}
bool stack__full(stack *this) {
    return this->sp_ == this->entries_ + stack__MAXENT;
}
void stack__push(stack *this, int v) {
    require(!stack__full(this));
    *(this->sp_)++ = v;
}
int stack__pop(stack *this) {
    require(!empty(this));
    int retval = *--(this->sp_);
    promise(!full(this));
    return retval;
}
```

```
int stack__top(stack *_) {
    require(!empty(this));
    return this->sp[-1];
}
```

Note that the constructor `stack__class()` uses two supporting functions `stack__on()` and `stack__promise()`, this relates to the promises of a class only being checked after an object is fully built, for example if another class inherits from it, then `stack__promise()` is only invoked after the class that inherited from `stack` has been constructed, the descendant uses `stack__on()` to construct the inherited stack, not `stack_class()`, and after the descendant constructor is done (all implemented in its own `xxx__on()` function) then its constructor, for example `opstack` from §6.7, `opstack__class()` calls both `opstack__on()` and `stack__promise()`, remember that in an inheritance chain only one constructor can `promise` a post-construction invariant, all descendants must satisfy it, just as `opstack` does, which requires that the `promise` be checked only after the outermost object is constructed.

Note that certain language keywords are used as part of the identifier mapping, for example `on` and `class` were used above, they are not valid user defined identifiers, so their use by the name mapping convention doesn't cause problems. The `on` and `this` keywords are also used to name function arguments The `promise()` in the compiled C code is similar to an `assert()`, it is provided in `<lang.h>`.

2S.15 Identifier mapping of array descriptor declarations

The declaration of a unidimensional array descriptor is equivalent, to and compiled as if the declaration was made using the `lang.vecdesc` (§Error: Reference source not found) generic class. Similarly, the declaration of a multidimensional array descriptor is as if the declaration was made using the `lang.arraydesc` (§Error: Reference source not found) generic class. See §S.16. For example:

```
int v[];          // compiled as: decl lang.vecdesc(int);
float matrix[][]; // compiled as: decl lang.arraydesc(int, 2);
```

2S.16 Identifier mapping and generic code

A generic function or class has one or more generic arguments, i.e. declared with `genre`, each use of such a class or function with a unique combination of generic argument types causes code for the function or class to be specialized for those specific types. The identifier mapping for generic classes uses the type name for the generic arguments to map the function's or class' name so that their specialized form be

uniquely identified.

A related aspect of the generic specialization of code is that for some type combinations the code that is generated, at the instruction level, could be identical between specializations with unrelated types. The compiler ensures only a single copy of such common specialized code is generated. For example a the generic `stack` class from §11.3, specialized to implement a stack of `int *` or a stack of `float *` is shared because the stack at its lowest levels ends up just being a stack pointers and what those pointers point to is not important at that level. Note that a stack of ularge and double could not share the same code because functions such as `push()`, `pop()`, and `top()` require different underlying calling conventions on most modern systems for their argument passing and value returning that is not the same at the instruction level, different registers, integer or floating point, are used.

The identifier mapping for the generic `stack` from §11.3, which is parameterized with the type of its elements and the maximum number of elements, assuming it was declared `pub`. Note that entries[] is implemented by the generic class `lang.vecdesc`:

```
pub class stack(genre lang.value type,
            size_t max, int *error) promise(empty()) {
    priv type entries[];
    entries.create(max);
    priv type *sp = entries.start;
    *error = !sp = libc.ENOMEM : 0;
    ...
}
```

When used as a `stack(int)`, results in `stack__genre__int.h`:

```
typedef struct lang__vecdesc__genre__int__struct
                lang__vecdesc__genre__int;
struct lang__vecdesc__genre__int__struct {
    int *start;
    size_t max[0];
};
void lang__vecdesc__genre__int__create(
         lang__vecdesc__genre__int *on, size_t n);
void lang__vecdesc__genre__int__destroy(
         lang__vecdesc__genre__int *on);
typedef struct stack__genre__int__struct stack__genre__int;
struct stack__genre__int__struct {
    lang__vecdesc__genre__int entries_;
    int *sp_;
};
```

```
void stack__genre__int__promise(stack *on);
void stack__genre__int__on(size_t max, int *error , stack *on);
void stack__genre__int__class(size_t max, int *error, stack*on);
void stack__genre__int__deinit(stack *this);
bool stack__genre__int__empty(stack *this);
bool stack__genre__int__full(stack *this);
void stack__genre__int__push(stack *this, int v);
pub int stack__genre__int__pop(stack *this);
pub int stack__genre__int__top(stack *this);
```

The second file produced is `stack__genre__int.c`:

```
#include <lang.h>
#include "stack.h"
void stack__promise(stack *on) {
    promise(stack__empty(on));
}
void stack__genre__int__on(size_t max, int *error , stack *on){
    lang__vecdesc__genre__int__create(&on->entries_, max);
    on->sp_ = on->entries_.start;
    *error = !sp = libc__ENOMEM : 0;
}
void stack__genre__int__class(size_t max, int *error, stack*on){
    stack__on(max, error, on);
    stack__promise(on);
}
void stack__genre__int__deinit(stack *this) {
    lang__vecdesc__genre__int__destroy(&this→entries_);
}
bool stack___genre__int__empty(stack *this) {
    return this->sp_ == this->entries_.start;
}
bool stack__full(stack *this) {
    // COOGL: atomic fetch not needed, immutable: entries_
    return this->sp_ == this->entries_.start
                      + this->entries_.max[0];
}
void stack__push(stack *this, int v) {
    require(!stack__full(this));
    *(this->sp_)++ = v;
}
```

```
int stack__pop(stack *this) {
    require(!empty(this));
    int retval = *--(this->sp_);
    promise(!full(this));
    return retval;
}
int stack__top(stack *_) {
    require(!empty(this));
    return this->sp_[-1];
}
```

2S.17 Functions with default argument expressions

Functions with default argument expressions are implemented by having a different version of the function for each argument that has a default value. Each implementation computes its default argument, and invokes the next version with that additional argument. For example, for `memget()` from §4.16 these additional functions are:

```
void *memget__1(size_t size) inline {
    return memget__2(size, true);
}
void *memget__2(size_t size, bool cached) inline {
    return memget(size, cached,
                  size >= sizeof(large) ? sizeof(large) :
                  size >= sizeof(int)   ? sizeof(int)   :
                  size >= sizeof(short) ? sizeof(short) : 1);
}
```

Their names are formed by appending a double underscore to the function name, in this case just `memget`, followed by their number of arguments. These functions then have their names mapped into C when compiled into C.

Note that the UNIX `open()` system call function is used with either 2 or 3 arguments, when specified as COOGL code the third argument is specified with a default value. Note that in some operating systems the C prototype for `open()` is specified as a variable argument function, only declaring the first two arguments, which causes the type checking of its third argument to be bypassed in the calling location, fundamentally this kind of function was never addressed by C89 or later versions of C.

```
int open(char *path, int flag, mode_t mode = 0) {...}
```

2S.18 Identifier mapping of functions risky to caller

As described in §14.9 functions such as `strchr()` are risky for the caller, the compiler must verify that the value returned by it, if its first argument is a local variable,

is not misused in a way that leads to unsafe code. The identifier mapping for it is different because of this:

```
char *strchr(char str[], int c) promise(retval == NULL ||
                                        str.start <= retval &&
                                        retval <= str.end) {
    char v;
    char *s = str;
    char *send = str.end;
    for (; s < send; ++s) {
        if ((v = *s) == c)
            return s;  // address of c in s, even if c == 0
        if (!v) break;
    }
    return NULL;       // return NULL otherwise
}
```

Is compiled into this code:

```
char *strchr__return__str(char *str, size_t str_max0, int c) {
    char *str__end = str + str__max0;
    char v;
    char *s = str;
    char *send = str__end;
    for (; s < send; ++s) {
        if ((v = *s) == c) {
            promise(str <= s && s <= str__end);
            return s;  // address of c in s, even if c == 0
        }
        if (!v) break;
    }
    return NULL;       // return NULL otherwise
}
```

XXX other cases need to be described.

Appendix 3D – Differences between C and CLEAN

"#if is almost always followed by a variable like "pdp11." What it means is that the programmer has buried some old code that will no longer compile. Another common usage is to write "portable" code by expanding all possibilities in a jumble of left-justified chicken scratches."

--Ken Thompson

CLEAN is a subset of COOGL that is C compatible, the most significant change is the removal of the C preprocessor and the addition of safe programming. Other changes are small, they simplify and improve the language, without silently changing the meaning of C code.

3D.1 Summary of differences between CLEAN and C

This appendix can be skipped by programmers that are not familiar with the C programming language.

COOGL is an evolution of a subset of C, this common subset is CLEAN. COOGL is not a superset of C. Language evolution requires change, if change is restricted to additions, i.e. if removal is not allowed, the accumulation of features leads to too much complexity.

Evolution of the C base of COOGL involves a few changes to the base C language. These changes are not silent changes. This means that these changes don't allow CLEAN code to behave different when used as C code, it means that any use of a feature of C that has been removed, causes a compilation error instead of a silent change in its meaning. This language design principle restricts the changes to the removal of arcane, useless, or problematic C features (those that usually lead to incorrect programs or needless complexity). This design principle allows some of those removed features to be brought back at a later time if backwards compatibility with some of them is later required, for example by a hypothetical transitional compiler that might fully or partially support both C and COOGL languages at the same time.

These evolutionary changes to the C language in COOGL and a few of the simpler additions to COOGL are described below. A detailed description of them is presented throughout the rest of this chapter.

A fundamental difference between C and COOGL is that C is not a context free

language. Determining what specific language constructs are being used in C is sometimes not possible without examining code that is elsewhere. A *context free language* is one that can be parsed purely based on its syntax, without the aid of a symbol table, context free languages make the writing of tools that are language aware easier to implement because the syntax of the language, its parsing, does not require the use of information from declarations found elsewhere to resolve syntactical ambiguities at parse time. C requires the use of a symbol table to resolve ambiguities in its parsing, it also requires that entities be declared prior to their use. Language aware editors and source code searching are much easier to implement when a language is context free because parsing with the aid of a symbol table is not required to be able to determine what a certain syntactical construct means.

Some examples of ambiguous C constructs:

- `a(*b)` which, if `a` is a type, is a declaration of `b` as a pointer to objects of type `a`; otherwise it is a function invocation of `a` with argument `*b`.
- `a(*b)(c)` which if `a` is a type, is a declaration of `b` as a function pointer that has an argument of type `c` and returns a value of type `a`; otherwise it is an invocation of a function `a` with argument `*b` and the returned value, presumably a function pointer, then being invoked with argument `c`.
- `(a)(b)` which can be interpreted as cast `b` to the type `a`, or as a function call of `a` with argument `b`.
- `(a)(b)(c)` which can be parsed as `((a)(b))(c)` or as `(a)((b)(c))` depending on whether `a` is a type or not.

When using languages that are parsed in a single pass, such as C, the programmer has to provide redundant information, for example function prototypes, and other prior declarations to ensure that the information required to disambiguate the parsing, or even to ensure proper code generation, is available at the appropriate times. COOGL is globally compiled in a single pass, but it doesn't require prototypes, removing a common source of programming errors in C. No form of forward declarations such as prototypes, `extern`, or `struct tag;` declarations are required, or allowed. C might generate incorrect code if the function signatures is unknown when compiling a function call, for example:

- C assumes that functions are `int`, unless declared otherwise, thus a function that returns float or a structure type will have the wrong code generated for it in the calling location, causing strange errors at run time, which might even include invalid memory references.
- C assumes that integer function arguments are passed with argument passing

conventions associated with `int` arguments. A function whose declaration has not been seen seen by the compiler is assumed to receive integral arguments as if they were `int`. If the argument is actually meant to be `long long` (the C99 type for a 64 bit integer) and assuming that `int` is 32 bits, the stack layout of arguments might be wrong, or the argument calling convention might be confused enough to the point were the values received by other arguments might be shifted between arguments in unexpected ways.

Even though use of prototypes is a well known solution to these problems, a missing include header file can easily cause these problems not to be caught. Editing of header files, and nested header files, sometimes causes a prototype that has been moved between header files to no longer be visible from some C code. This problem is always a possibility in C because prototypes are not mandatory. Some compilers have options that cause warnings or errors if a function is invoked without its signature being known, the language itself doesn't require any such checking.

Two simple changes to COOGL's subset of the C language were made to make the language context free, which facilitated the removal of C prototypes and forward declarations. These changes also make programs easier to read when declarations are combined with other COOGL enhancements such as generic types. The two changes are:

- When the C operator `sizeof` is to be applied to an expression, instead of a type, the `sizeofex` operator should be used instead.
- Cast expressions in COOGL use the `cast(type)` operator instead of the C `(type)` operator, for example `cast(int)` instead of `(int)`.

C features removed in COOGL's C subset are:

- The C preprocessor was removed. The `#include`, `#define`, `#if`, `#ifdef`, `#ifndef`, `#error`, `#pragma`, and `#line` preprocessor directives are not available. Language level facilities in COOGL exist to accommodate or remove the underlying needs for these features, with the exception of the arbitrary unstructured code mutations that are possible in C through unstructured uses of `#define`, `#ifdef`, and `#include`, see §D.3.
- Legacy K&R C function declaration syntax was removed. Only the C++ derived syntax introduced in C89 is allowed, see §D.4. Functions without arguments can be declared with or without `void` in the function's argument list.
- Variable argument functions are not allowed, this reduces language complexity while removing a feature that is unsafe and not required. In consequence `printf()` and related functions are not supported either. See §D.5.

- Declaration of variables of a `struct`, `union` or `enum` type can not occur in the same declaration that is the `struct`, `union` or `enum` type declaration itself. No variable or function declarations can follow the closing curly brace of any of these declarations, see §D.7 and §D.8.
- Every declaration is local to its scope, except labels, see §D.9.
- Nested `struct` or `union` within another `struct` or `union` are allowed but must be without a tag, see §D.10.
- Forward declarations are not needed nor allowed.
- Declarations in nested scopes are not allowed to hide function arguments, local declarations, or members, see §D.11.
- Members of a `union` must all be plain data, see §14.5 and §14.7.

COOGL does not mandate the size of any type, various common data type size combinations can be supported, as dictated by underlying hardware and operating systems (such as ILP32, LP64, P64, and ILP64). Programming languages that mandate the size of native types, for example Java and C#, result in programs that are hard to port to wider or narrower word machines. For example, even today, with 64 bit machines, Java mandates that `int` be a 32 bit type, that means that an array with more than 2^{31} entries (one bit is consumed to represent negative numbers) can not be used, because indexes are typically declared `int`, and Java mandates that `int` types be 32 bits. So code that works today, needs to be changed as the systems evolve instead of the language adapting appropriately to the system. The widening of the C `int` type from 16 bits to 32 bits is an example of such adaptation when C was moved to the DEC VAX from the DEC PDP-11. The types `index` and `uindex` are meant to be used as index types instead of `int` and `uint`.

The ALGOL 68 inherited syntax that uses `short`, `long`, `long long`, `long double`, `double double`, etc. to specify native type sizes was removed, this simplifies the language in the areas of generic programming and constructor arguments to native types:

- The use of `short` and `long` together with `int` in declarations is invalid. Thus forms such as: `short int` and `int long` are invalid, plain `short` and `long` should be used instead. Use the type `large` which is at least 64 bits instead of `long long`, for example in a ILP32 environment.
- The `unsigned` and `signed` modifiers were removed from COOGL, they are reserved keywords. A complement of unsigned types was added: `byte`, `ubyte`, `schar` (signed `char`), `uchar`, `ushort`, `ulong`, and `ularge`.
- All declarations must have an explicit type. The `int` type is not an optional

implicit type assumed when an explicit type is missing, see §D.21. Because `short` and `long` are types, instead of type width modifiers, the most common uses of optional `int` are preserved, for example: `long l` or `short s`.

Minor C language adjustments in COOGL are:

- Parenthesis are mandatory in certain expressions that usually cause confusion among enough C programmers. This does not mean that everything has to have parenthesis, it means that certain uses that usually have them, or they would be a bug in most cases, must have them, see §D.16.
- C89 changed the original K&R C behavior of expressions that involve both signed and unsigned operands. COOGL follows all the rules of C99 for expression evaluation and integer and floating point promotions. With the exception that relational comparisons between signed and unsigned types, which behave different in K&R C and C89. COOGL does not allow such intermixing in comparisons, see §D.17.
- The syntax `d3[i,j,k]` does not correspond to the 3 dimensional array indexing, it is an invalid expression in COOGL, though it is valid in C, but its meaning in C does not correspond to the tridimensional array indexing that programmers used to other programming languages might expect.

Minor C related features added to COOGL are:

- The COOGL `/# comment #/` replaces the need for the C preprocessor when `#if 0`-ing out code so that it not be compiled. Also BCPL style comments, were added, e.g. `// comment until end of line`, see §D.2.
- The use of `struct` and `union` is preserved for interfacing with C programs and data. Programming in COOGL revolves around `class` and `interface` declarations, not around `struct` or `union` declarations. C style programming with just `struct` and `union` declarations is supported with some minor backwards compatible simplifications, see §D.6, §D.7, §D.8, and §D.10.
- COOGL provides the ability for its functions to be invoked from C code, and vice versa, see §D.15.
- The COOGL `bool` type is a boolean type with literal values `true` and `false` the same conversion rules as the C11 `_Bool_` type, see §D.18.
- An integer type, a floating point type, or a pointer type, can be used as a modifier in a `enum` declaration, thus dictating the type associated with its values and the storage requirements for variables of that `enum` type, see §D.19.

- Given that `#define` C preprocessor mechanism was removed, constant declarations in COOGL use the `lit` keyword, alternatively, for a family of related constants, an `enum` declaration can be used. Values declared in an `enum` or in a `lit` declaration are compile time constants, they can be used to size arrays or as the number of bits in a bit field declaration, both of which are invalid uses in C for an `enum` value, see §D.20.
- Function expansion in the invocation location, also known as function in line expansion. The use of `inline` modifiers in function declaration and in function invocation locations provides fine grained control over this facility, see §Error: Reference source not found.
- Support for zero length and variable length arrays, see §D.22.
- Indentation induced mismatched `if else` programming errors cause compilation errors, see §2.28.

The C keyword `const` and `volatile` remain in the language begrudgingly mostly to interface with C code and have a larger common subset with it. Dennis Ritchie was right: *"I'm not convinced that `const' and `volatile' carry their weight; I suspect that what they add to the cost of learning and using the language is not repaid in greater expressiveness."*

Their complexity was indeed not merited by the language. They might be deprecated in the future, `volatile` particularly. Some intrinsics for device register load stores would have been enough just like x86 compilers had `in()` and `out()` intrinsics to produce the x86 `in` and `out` instructions. Some architectures, with load store accessible device hardware registers, require specialized instructions to be used (such as the POWER `eieio`, enforce in order execution of input output) when using load and store instructions to access device registers. Touching hardware device registers might require other delicate actions such as setting up very low level exception handling in case the device stops working, hangs, or misbehaves in some other way, thus actual device hardware register access is hardly ever done from C through loads and stores of device registers declared `volatile`.

3D.2 Comments

There are 3 types of comments in COOGL. C style comments which are bracketed by the `/*` and `*/` tokens:

```
/* C style comment, continues irrespective of line
   boundaries until the end of comment token */
```

BCPL style comments, which start with the `//` token and end at the end of the line:

```
// BCPL style comment, ends at the end of the line
```

Comments bracketed by the `/#` and `#/` tokens:

```
/# Comment form used to comment out code that has the
   other two styles of comments, continues irrespective
   of line boundaries until the end of comment token #/
```

To avoid programming errors that result from a missing closing comment token, the following restrictions apply:

```
/# invalid because of this-> /# extra comment start token #/
/* invalid because of this-> /* extra comment start token */
/* invalid because of this-> /# COOGL comment start token */
/* invalid because of this-> // BCPL comment token */
// invalid because of this-> /* C comment start token
// invalid because of this-> */ C comment end token
// invalid because of this-> /# COOGL comment start token
// invalid because of this-> #/ COOGL comment end token
```

The purpose of COOGL style comments is to allow code with comments to be commented out, so this is valid, see §2.2 for rationale and examples:

```
/# ok to have these tokens here: */ /* // #/
```

3D.3 No C preprocessor

The C preprocessor langugage is not part of the COOGL language. The idiomatic uses of the preprocessor are addressed by native COOGL language facilities and code organization conventions:

- The use of `#define` macros to implement `assert(expr)` and similar macros is supported through the `#identifier` operator, see §4.17 for its use to implement `assert(expr)` and §D.17 for another example `puts_if(e)`.
- The temporary commenting out of code by placing it between a `#if 0` or an `#ifdef notdef` and a `#endif` is addressed by the `/# comment #/` as described above.
- Uses of `#if`, `#ifdef` and `#ifndef` in function bodies can be replaced by `if` statements with constant expressions. This mandates that all the code compiles irrespective of the constant expression values. This is both a restriction and a long term benefit. Platform dependencies (operating system, window systems, hardware, etc.) that would otherwise be dealt with conditional compilation are either so small that they can be easily addressed through `if` statements or are misplaced and ought to be factored out anyway into environment dependent code in a platform dependent file. Any modern compiler

is capable of removing dead code, for example dead code subordinate to a compile time constant expression in an `if` statement, thus there is no run-time cost associated with replacing `#ifdef` with `if` statements.

- Uses of `#ifdef` outside of functions to cause different versions of a function to be chosen depending on platform considerations are addressed by segregating the functions for each version into a separate file and choosing the appropriate platform dependent file when building the program.
- Uses of `#ifdef` to cause structure declaration mutations in a platform dependent manner can be replaced with platform dependent files and the platform dependent substructure included in what would have been the structure with `#ifdef` in its declaration. For example the C code:

```
struct io {
#ifdef UNIX_IO
    int file;
#endif
#ifdef LIBC_IO
    FILE *file;
#endif
    /* common to both */
};
```

Can be dealt with 3 source files. Source file `io.cog` contains:

```
class io {
    pub inherit iox;
    // common to both
}
```

file `unix_io.cog` contains:

```
class iox { pub int file; }
```

and file `libc_io.cog` contains:

```
class iox { pub FILE *file; }
```

Only one of these last two files is used as part of the program compilation. There are other ways to deal with this example when common code with C is required. See XXX (empty structures and fields of their type).

- The use of `#define` to introduce named constants, as in this C code:

```
#define MAXENT 100
struct stack {
    elem *sp;
    elem entries[MAXENT];
};
#define STACK_INIT(stackp) \
    ((stackp)->sp = (stackp)->entries)
```

is replaced in COOGL by the use of `lit` or `enum` declarations:

```
class stack {
    pub lit int MAXENT = 100;
    pub elem entries[MAXENT];
    pub elem *sp = entries;
}
```

A COOGL `lit` is a constant expression. It can be used wherever a constant is required, e.g. to size an array, as a bit field, or as the value of a `switch()` `case` label. The C `const` does not work this way, it cannot be used as a constant expression in any of those cases. The benefit of symbolic debugging and the prevention of arbitrary language mutation outweighs any benefits of preserving `#define`. Additionally, COOGL constants can be declared within a scope as done above for `class stack`, i.e. where they are relevant, they don't pollute the global name space as `#define` does. The `MAXENT` declaration could have been global if that is what was really desired. The C `#define NULL` declaration is a `lit` declaration in COOGL, see §D.23.

- In COOGL the underlying integer base type of an `enum` can be declared, thus enumerations are not restricted to `int` values. For example:

```
ularge enum {
    MSB64 = 1uLL << 63
}
```

- Uses of `#define` to declare named macros with arguments that can be used to inline arbitrary text in the macro invocation location, without regards for scope, is replaced by well formed function inline support. Support for inline expansion of: functions, member functions, nested functions and delegates completely removes the non language mutating needs for this form of `#define`. The unification of structures, classes and functions also removes the need for `#define`s similar to the `STACK_INIT()` above that are usually located near the structure declaration. In COOGL the `class` declaration is the constructor as shown in the `class stack` above, see §4.2.

- The use of `#define` to make believe that fields within an inner structure are part of an outer structure that contains them is addressed by inheritance or

the `alias` field aliasing syntax, for example the C code:

```
struct inner {
    int in_count, in_total;
};
struct outer {
    struct inner in;
    int out_free;
};

/* bad: these have global scope! */
#define out_count in.in_count
#define out_total in.in_total

int main() {
    struct outer o;
    o.out_count = 3;
}
```

is written this way in COOGL:

```
class inner {
    pub int in_count, in_total;
}
class outer {
    pub inner in;
    pub int out_free;
    // good: these have class scope
    pub alias out_count = in.in_count;
    pub alias out_total = in.in_total;
}
int main() {
    outer o;
    o.out_count = 3;
}
```

- Use of `#define` to implement macros such as `OFFSETOF()`:

```
#define OFFSETOF(type, field) \
    ((size_t) &(type *)0)->field)
```

These are addressed by the generic programming facility which allows for type arguments with `genre` and field name arguments with `fieldof`, see §11.3 and §11.13 for their respective descriptions and an implementation of the `offsetof()` function.

- The need for the `#include` mechanism and the notion of header files in general is removed. Declarations are extracted from the source files without any

programmer intervention. The set of files that makes a program is known at compile time. It is provided by the programmer, the list is supplied in a file or as part of the compiler command line argument list. The compiler caches information required for separate compilation into files that it manages transparently to the user. Structure layouts, function signatures, etc. are cached by the compiler and the information is not re-extracted unless the source files that contained them have changed. The compiler also remembers the list of files, if the set of files changes, or their contents change, it adjusts the cached information.

- Optimized compilation for production always recompiles every file so that it has an opportunity to inline functions and perform global optimization. Optimized compilation for development can be done without inlining thus obtaining the benefits of incremental compilation. Code that requires strict separate compilation doesn't take full advantage of global optimization, for those cases, usually libraries, the compiler knows where to find the cached extracted information for the library interfaces. Even in the case of libraries, inlining can be chosen for selected sets of functions, thus `getchar()`, `putchar()`, and `isalpha()` equivalent functions don't have to be any slower than their C `#defined` implementations. COOGL files that contain only the class and structure data layouts, signatures and other declarations required or relied upon for strict separate compilation can be produced by the compiler, i.e. everything is extracted other than constructs that directly cause instructions to be emitted by the compiler. These compiler produced files are similar to the header files that are maintained explicitly by C programmers.

3D.4 No K&R C function declarations

The K&R, ALGOL style, function declarations are removed. Function declarations are only in the non-K&R form introduced by C++ and later adopted by C89. For example, `abs()`'s declaration is invalid COOGL code, this is a K&R style declaration, the declaration of `max()` is valid:

```
int abs(i) int i; { return i >= 0 ? i : -i; }   /* K&R style */
int max(int a, int b) { return a > b ? a : b; }
```

A function declared without arguments is synonymous with a function taking no arguments, for example, `void f() {}` is equivalent to `void f(void) {}`.

3D.5 Variable argument functions are not allowed

Variable argument functions are not allowed, this feature originated in BCPL and

got into C via B, this mechanism is not type safe, nor extensible, and notoriously error prone in its use. In consequence `printf()`, `scanf()`, and related functions are not supported either. The `on` syntax provides a type safe, extensible, and general purpose mechanism to implement formatted input output and traces. The traces can be compile time removable traces when not required for debugging, and very efficient, both when they are run-time disabled or enabled. See §9.2 and §9.3.

Leftover historical variable argument functions in C that were never meant to be supported with variable argument lists, but have now been contorted into them, such as the `open()` UNIX system call:

```
int open(const char *path, int oflag, ...);
```

Which is meant to be used in one of the following two ways, the 2nd one providing `mode` when `oflag` includes the `O_CREAT` flag:

```
int open(const char *path, int oflag);
int open(const char *path, int oflag, mode_t mode);
```

Note that the use of the C variable argument list declaration syntax `...` removes information from the declaration of `open()` and the type of its optional 3rd argument, allowing incorrect arguments to be used without compilation warnings or errors.

In COOGL the declaration of `open()` follows, the type of its 3rd argument is correct and if the user doesn't specify it, its value defaults to zero.

```
int open(const char path[], int oflag, mode_t mode = 0) { ... }
```

3D.6 Forward `struct` and `union` declarations are invalid

Mutually referring structures in C require these forward declarations:

```
struct node;
struct tree {     /* C code */
    struct node *root;
    int count;
};
struct node {
    struct tree *top;
    struct node *left, *right;
};
```

COOGL is globally compiled, it doesn't need types to be declared before they are used. The forward `struct` and `union` forms are not needed, they are invalid in COOGL:

```
struct node;                    // error: syntax error
union united;                   // error: syntax error
```

The global compilation model allows these forms of cross dependencies to be expressed without order considerations. All declarations, with the exception of local and global variables, are order independent. Forward declarations such as the ones shown above for structures or extern declarations for data types are invalid, and not required in COOGL.

The equivalent COOGL code is:

```
struct tree {
    struct node *root;
    int count;
};
struct node {
    struct tree *top;
    struct node *left, *right;
};
```

The reason that order matters in local declarations is the obvious one, construction occurs at declaration time, there is a well defined life time for a local variable, furthermore there is a well understood execution flow. The actual order of declarations of types, classes and functions does not correspond to an execution order, it only has to do with a compilation order. Order of global variable declarations matters as a feature that can be depended upon. Single pass compilation used to matter a very long time ago, i.e. when the program being compiled resided on tape and two passes meant reading the tape twice (because the program source code could not be assumed to fit in memory with the compiler and the compilation data). It is, of course, very safe to say that those days are long gone.

3D.7 Variable declarations in type declarations are invalid

C allows for a `struct` or `union` declaration together with variable declarations and `typedef` declarations. For example this valid C code is invalid COOGL code:

```
struct node {
    struct node *parent, *left, *right;
    struct info *data;
} root_node, *root = &root_node;        // error: syntax error
typedef struct { int i; } other_t;      // error: syntax error
```

The syntax that in the code above allows for `struct node`, `root_node`, and `root` to be declared together has been removed from the COOGL language. The code above has to be written in COOGL as:

```
struct node {
    struct node *parent, *left, *right;
    struct info *data;
};          // mandatory semicolon required
struct node root_node, *root = &root_node;
struct other { int i; };
typedef struct other other_t;
```

3D.8 Semicolon after closing curly brace in `struct` and `union`

In COOGL a semicolon is mandatory after the closing curly brace of a `struct` or a `union` declaration (unless the `struct` or `union` is nested within another `struct` or `union`). The COOGL syntax for `struct` and `union` is a strict subset of the C syntax. As seen above, not all valid C `struct` and `union` declarations are valid COOGL declarations, the only declarations that are valid are the ones in which the semicolon follows the closing curly brace (ignoring intervening whitespace and comments). In COOGL a declaration based on the type just introduced by a struct or union declaration is never followed by declarations after the closing curly brace.

3D.9 Every declaration is local to its scope

Declarations in COOGL are local to the scope that contains them. In C, when a function whose return value or complete signature is needed for appropriate compilation sometimes the function is declared (with or without `extern`) in the scope of the function that needs the declaration. For example:

```
int main() {  /* C code */
    void *malloc(size_t);
    char *p = malloc(100);
    ...
}
```

In COOGL, such a declaration would be an incomplete, i.e. invalid, declaration of a nested function of `main()`, i.e `main.malloc()`, whose code was not provided.

In COOGL, `lit`, `enum`, `struct`, `union`, `class` and function declarations within a scope introduce a name only within that scope. Every declaration introduces a name within the scope where it occurs. If the name is to be referred to from outside of the scope, a qualified name must be used. Actual access to the entity is subject to the accessibility determined by the entity declaration.

3D.10 Invalid nested typeless `struct` and `union` declarations

A `struct` and `union` declaration must specify a tag name, for example:

```
struct {       // error: missing struct tag name
    int i, j;
};
```

The rationale for this is simple, no variables can be declared at `struct` or `union` declaration time in COOGL, thus if no tag name is given, the declaration would serve no purpose, it could not be used for anything.

Nested `struct` and `union` declarations within other `struct` or `union` declarations are allowed but they must not contain a tag, for example:

```
struct outer {
    int out;
    struct inner {  // error: nested struct has tag: inner
        int in;
    };
};
```

The meaning of the similar C code is actually compiler dependent, some compilers ignore the inner structure and allocate no space to it. Other compiler's treat it as an anonymous `struct` that does allocate space in the structure, a form of poor man's inheritance. Another reason to disallow these forms in COOGL is that every declaration in COOGL is relative to its scope, which would not be the case in C. In C the `struct inner` is introduced as a global type, there are no nested types in C. To avoid silently changing underlying legacy C behavior these nested forms with a tag are invalid.

An easy, C compatible work around in COOGL is:

```
struct inner {
    int in;
};
struct outer {
    int out;
};
```

A typical use of nested structure declarations in C is valid in COOGL too:

```
struct foo {
    int i;
    struct {
        int a, b;
    } table[10];
};
```

3D.11 Non global names cannot be hidden

Non global names cannot be hidden, for example by other declarations within a block, function, `class` or `enum`. For example:

```
void func(int i) {
    int i;              // error: hides i
    int b;
    if (i) { int b; }   // error: hides b
}
```

3D.12 Struct and union For C interoperability

COOGL `struct` and `union` declarations exactly follow the structure layout rules of the native C compiler. In COOGL fields within a `class` must be explicitly declared with an accessibility modifier (i.e.: `pub`, `priv`, `prot`, etc). Entities declared within a `class` without an explicit accessibility modifier are not actual fields or members of the `class`, they are simply local variables of the constructor of the `class`. This enables the unification of classes and functions in COOGL, see §4.2.

For `struct` and `union` the accessibility modifier of every field is implicitly `pub`. It is not possible to declare constants, enumerations or functions within a `struct` or `union`. It is invalid to use an accessibility modifier within a `struct` or `union`. `Structs` or `unions` can not have executable code of any kind in them, they are not constructors.

Neither a `struct` nor a `union` can contain or refer to a type that is not either: a fundamental type, a function pointer, a `struct`, an `union`, or an array of these types. Function pointers in them must not have as their signature a type outside of this restricted set of types, nor can they be delegate function pointers, see §7.10. Anything that appears in a COOGL `struct` or `union` can be extracted (together with its dependent declarations) and used to interface with C code without the risk of COOGL constructs being part of them.

Arrays within a `struct` or `union` can make use of `lit` or `enum` declarations to express their dimensions. Shared source code files between C and COOGL for those would require `#defines` that are equivalent to the `lit` and `enum` values, for example:

```
/* file common to C and COOGL */
struct foo {
    int i;
    int b[NUMB];
};
```

Some other COOGL file contains:

```
lit int NUMB = 17;
```

Usually a simple COOGL program can be written to produce the `#define` for `NUMB`:

```
// file gen.cog
int main() { on ("#define NUMB "; NUMB) print(); }
```

Alternatively, the values might be C `#define`s and the COOGL file with the `lit` declarations can be generated by a C program.

3D.13 Function invocation from C Code

A bridge between C and COOGL is that both languages use the same calling convention, the identifier name mapping rules to map a COOGL identifier into a C identifier are very simple, and are part of the language definition. Any COOGL function can be called from C code, and vice-versa, non-static member functions receive as their first argument the address of the object that the member function is being invoked on. See §Appendix 2S – Identifier mapping and calling convention.

3D.14 Global declarations are by default `prot`

The default behavior of `prot` was chosen for all global declarations in COOGL because it leaves COOGL quite near C in this area. Particularly, when one considers that `typedef`, `enum` and `struct` declarations in C are almost always in header files, which are easily included from C files.

Complementing, with accessibility modifiers, the C use of `static` with global variable and function declarations was required in COOGL because without header files, control of the visibility of `typedef`, `enum`, `struct`, `class`, and other declarations needed a syntax. Use of `static` with them would be strange, and is not allowed, use of `priv`, `prot`, and `pub` is natural.

3D.15 Interfacing with other languages

The issues involved in language interfacing are well understood. For example, to invoke Fortran code from C, many languages turn the function names to all lowercase and append an underscore. Data type issues are more complicated. Particularly array indexing (starts at 1 in Fortran) and order of dimensions of multi-dimensional arrays in memory. The FORTRAN array access `a(i,j)` is `a[j-1][i-1]` in C. Interoperability with other languages is not a high priority goal for COOGL, system level

solutions, at the translated C level are used for this purpose.

3D.16 Mandatory parenthesis in a few troublesome cases

COOGL includes all of the operators in C, with the same precedence, associativity and semantics. COOGL requires parenthesis to be used in a few cases. The cases where parenthesis are needed are fully described in §10.1. The cases involved are the ones where if parenthesis are omitted it is very likely a programming error, or likely to cause confusion when read by most C programmers.

For example: `if (x & 2 == 2)` means this in C: `if (x & (2 == 2))` which further means: `if (x & 1)`. In COOGL it results in a compilation error, the programmer must write: `if ((x & 2) == 2)` if that is what he meant, or: `if (x & (2 == 2))` if that is what he surprisingly actually wanted.

Parenthesis are required when certain operators are used together and when the parsing of the expression doesn't already imply an interpretation that doesn't lead to confusion. In the example shown above, the specific use of `&` and `==` mandates the need for parenthesis. Parenthesis are not required in the large majority of cases, for example: `x = y & 2;` or `if (x & 2 && x != 0xff)`.

3D.17 Errors with signed and unsigned: `< <= >= >`

Anomalous mixed sign comparisons are not supported, neither the C89 nor the K&R C behaviors are supported, the programmer must address the compilation error explicitly, for example through a cast. Turning one type to the other is inherently incorrect for some cases.

C89 dictates that `-1 > 1u` evaluates to true. The comparison between `-1` (signed) and `1u` (unsigned) causes the `-1` to be converted to an unsigned value with the same bit pattern, which is a very large unsigned value. As a result `-1 > 1u` is a true condition in C89, which is contrary to intuition. COOGL addresses these issues by causing a compilation error and forcing the programmer to be aware of it and address it.

```
#define PUTS_IF(e) do if (e) puts(#e); while (0)   /* C code */
int main() {
    PUTS_IF(-1 > (unsigned long long) 1);
    PUTS_IF(-1 > (unsigned int) 1);
    PUTS_IF(-1 > (unsigned short) 1);
    PUTS_IF(-1 > (unsigned char) 1);
}
```

The output of the C program above is:

```
-1 > (unsigned long long) 1
-1 > (unsigned int) 1
```

The promotion to `int` during expression evaluation, the C89 dictated result occurs for `unsigned int` and `unsigned long long` but it does not occur for `unsigned char` or `unsigned short`, this is assuming that `sizeof(short)` `<` `sizeof(int)`, otherwise the unexpected result also occurs for `unsigned short`. C89 adopted value preserving semantics, thus when an `unsigned char` or an `unsigned short` are evaluated in an expression that involves an `int`, their values can be fully represented as `int` and the result of the comparison has the expected behavior.

The equivalent COOGL comparisons below, results in compilation errors for the first two, the unreasonable ones that imply that a negative number is greater than a positive number. This approach is within the COOGL design principle of not making silent changes to the C language, i.e. changes that would make the code behave different between C and COOGL.

This program uses the special `#` argument stringifying operator, see §4.17, to cause the compiler, not a preprocessor, to turn the expression used for argument `b` into a string and use its value for the default value of another argument:

```
void puts_if(bool b, char msg[] = #b "\n"){ if(b)msg.print());}
int main() {
    puts_if(-1 > cast(ularge) 1)); // error: int > ularge
    puts_if(-1 > cast(uint) 1));   // error: int > uint
    puts_if(-1 > cast(ushort) 1));
    puts_if(-1 > cast(ubyte) 1));
    puts_if(true, "override default msg[] with this\n");
}
```

Most relational comparison between sub-expressions of type `int` and `uint` , or between `int` and `ularge`, result in a compilation error. The mathematical correct thing to do for these comparisons would be to ensure that negative numbers are always smaller than unsigned values, and only comparing their values if the signed value was not negative. Mixed signed and unsigned comparisons of this nature are not directly supported by computer hardware comparison instructions, thus it is not supported by the COOGL language either, though a higher level language could adopt such a strategy. Comparison between an unsigned value and a literal signed value that is non-negative are allowed, the literal value is considered unsigned and the comparison proceeds as a comparison between unsigned values.

3D.18 Hardware dependent types and `bool`

COOGL has a `bool` type together with `true` and `false` literal values. The

`sizeof(bool)` is `1`. Any non-zero expression assigned to a `bool` variable causes the variable to have the value `true`. A zero expression causes the variable to have the value `false`. When `true` is converted to an integer type, its numeric value is `1`, when `false` is converted to an integer type its value is `0`.

COOGL defines a set of hardware independent types (`int:8`, `int:16`, `int:32`, and `int:64`, and their `uint` counterparts), COOGL doesn't have to worry about the class of machines with 18 bit words and 36 bit double words from a few decades ago, or the ones with 27, 29, or 31 bit words. Long gone are the days of 6, 7 and 9 bit byte machines! COOGL simply assumes that machines are byte addressable and that they have native support for arithmetic all the way up to 64 bits, even if multiple instructions are required to implement 64 bit arithmetic in a few legacy systems, i.e. 32 bit embedded processors and Intel/AMD x86 which is now in full transition to x86-64 which has full 64 bit support.

These are the COOGL hardware dependent types and `typedef` defined types, and their sizes on modern systems:

size in bytes	number of bits	signed type	unsigned type
1	8	byte	ubyte
2	16	short	ushort
4	32	int	uint
8	64	large	ularge

3D.19 Type specifiers in `enum`

COOGL allows `enum` declarations to have an integer type, a floating point type, or a pointer type, as a modifier, see §Error: Reference source not found.

3D.20 `lit` modifier introduces a compile time constant

The `lit` modifier introduces a compile time constant. It can be used to size an array, as the number of bits in a bit field declaration, or as the value of the `case` label of `switch()` statement. The C `const` is a weak notion of a runtime constant that should not be modified, it is not a compile time constant.

3D.21 Declarations must have an explicit type

The last vestiges of C's untyped BCPL and B genome are removed. All declarations must explicitly indicate the type involved, `int` is not a default type for when a type is missing. This valid C code is not valid COOGL code:

```
i;

main() { i=3; }
```

The types must be explicit:

```
int i;
int main() { i=3; }
```

3D.22 Variable length arrays

COOGL allows multidimensional variable length arrays, i.e. with the number of entries in the array determined at run time, in a better way than C99 does, see chapter §13.

3D.23 `NULL` pointer

Traditionally C programs make use of `NULL` as the value associated with an invalid pointer, for example the value returned by `malloc()` when memory allocation fails, or the value that terminates a `NULL` terminated list. `NULL` in C is a `#define`, these two forms are used in various compilation environments:

```
#define NULL   0
#define NULL   ((void *)0)
```

The definition of `NULL` in COOGL is:

```
lit void *NULL = cast(void *) 0;
```

Because of the undefined behavior of NULL pointer dereferencing and the unsafety that might arise from the address space layout of most operating systems, the declaration of `NULL` and the ability to use 0 as a pointer value are only allowed if the compiler option `--NULL` is used, `NIL` should be used instead, see §14.16.

```
COOGL --NULL p.cog
```

3D.24 Name mapping, double underscore, and underscore retrictions

There is minimal name mapping from COOGL to C to support generic programming and for member function names to be qualified by the name of the `class` or `interface` where they are defined. The name mapping rules are part of the language definition, allowing COOGL code compiled with different COOGL compilers to be linked together, see §Appendix 2S – Identifier mapping and calling convention.

Use of double underscore, i.e __, in identifiers is invalid. Double underscore is used by COOGL to separate user defined identifiers into compound identifiers. For

example the `pop()` function of the `stack` class has its name mangled into `stack__pop()` when translated into C. The use of double underscore is uncommon enough that making their use in identifiers and thus allowing the name mapping to be simpler is a worthwhile tradeoff.

Use of underscore at the start or end of an identifier is also invalid. This prevents a class `stack_` and a member function `pop()` into being mangled as `stack___pop()`, and class `stack` with member function `_pop()` being mangled the same way (note the 3 underscores, not two in `stack___pop()`).

3D.25 Deceiving indentation causes compilation errors

To aid in the discovery of visual indentation induced bugs, such as the one shown in §2.28, COOGL produces a compilation error if the indentation levels of an `if` and its corresponding `else` could cause confusion.

[This page left blank to work around issue in LibreOffice that is causing an empty blank page to be created in the next chapter in between its pages]

Appendix 4C – Sharing Code and Using C Code

"Independently, we went on and tried to rewrite Unix in this higher-level language that was evolving simultaneously. It's hard to say who was pushing whom —whether Unix was pushing C or C was pushing Unix. These rewrites failed twice in the space of six months, I believe, because of problems with the language. There would be a major change in the language and we'd rewrite Unix."

"The third rewrite—I took the OS proper, the kernel, and Dennis took the block I/O, the disk—was successful; it turned into version 5 in the labs ..."

-- Ken Thompson

The mechanism used by COOGL programs to interface with C libraries and their header file exposed interfaces is important for programs that need to make use of such libraries or coexist with binary C code. For example an operating system kernel level file system implementation written in COOGL that needs to function in an operating system written in C. Or an operating system kernel written in COOGL that makes use of large bodies of preexisting code written in C, device drivers, file systems, networking, for example a reengineered version of the Linux kernel in a multi-year effort to rewrite the Linux kernel into COOGL.

Source code that needs to be shared and used both as C and COOGL is written in the CLEAN subset of COOGL. A few programming conventions are followed to allow the code to be used in both languages. The conventions are presented together with examples.

4C.1 genassym lesson

When incompatible languages need to share data structure information, for example C and assembly code, there are two ways of doing it. The wrong way, involves calculating the structure offsets and maintaining a series of #defined values (or some other assembly language macros) and maintaining both the C and assembly definitions carefully adjusting the offsets whenever the C structures change. Similarly for enu-

meration values. The right way is to have a C program that includes the header file and when run produces the definitions to be used by the assembly code, those programs have been historically called genassym because the generated the assembly symbols from the C headers. They make use of offsetof() like macros and they are easy to maintain. They use sizeof() to produce structure sizes, etc. Only the symbols required to be accessed from assembly need to have their offsets produced. The width of the data types themselves might change over time, some genassym generated code might include macros that expand to the correct load or store instruction for the underlying CPU architecture, the genassym can choose the correct instruction based on the sizeof of the individual fields.

The same technique can be used when C headers contain interfaces that need to be used from COOGL code. The programmer can write a small program that maps `#define`d values into `lit` declarations, or vice versa. Because the `struct`. `union`, `enum`, `typedef`, and function declaration syntax is largely the same in both languages and because there are no silent differences between them, it is also easy to maintain civilized C header files that can be preprocessed to produce a COOGL source file against which the COOGL code can be compiled in a way that it is made believe that the code that it will be linked against is COOGL code but at link time the C binary will be specified instead, or that the code lives in another module and that the symbols will be resolved at dynamic linking time at program startup time, or when the COOGL written module is dynamically loaded.

XXX show example of shared source code and header required on the C side to cover minor language differences, e.g. #define sizeofex sizeof, #define cast(x) (x).

Appendix 5R – Language and Compiler Manual

"xxx"

by XXX

The grammar for the language is specified in a simple format, in the file spec/grammar.spec, the keywords and operators for the language are specified in another simple file, spec/tokens.spec. From these two files a `bison` grammar with automatically generated code is produced by the script `tool/mk`, including code to build the parse tree, create symbol tables for each scope, and insert the symbols into the symbol table, it also produces a `flex` file, and together with pre-existing code the compiler is built. The transformation of the grammar into a compiler includes automatic transformation of lists into tables. Additionally nodes that are not needed in the parse tree are pruned, which makes the parse tree more compact and easier to examine. The compiler has various options to dump the parse tree, dump declarations, among others.

The result of all this is that the grammar is very easy to modify and the compiler is re-generated automatically without any effort.

5R.1 Introduction to the COOGL Compiler

XXX some of the descriptions are in the module specification section.

5R.2 Compiler Options

XXX some of the descriptions are in the module specification section.

5R.3 Compiler Option Specification

XXX feature turning on/off: through a compiler option, through an environment variable that specifies flags, or locally or globally for the system through a configuration file whose names is specified in a variable, if set for local modification, or globally if not set (for system wide specification).

5R.4 Enabling `...` Statement

Make `...` a valid statement only with the `--...` compiler flag.

5R.5 Enabling `NULL` Support

Use `--NULL` to enable use of `NULL` or `0` as pointer values.

5R.6 Compilation of Concurrent Code

Use `--concurrent` to compile a program or a modules to work in a concurrent environment, this option causes the choosing of modules that implement facilities required by the module that also support concurrency.

5R.7 Add `--clean` to `gcc`

Add a `--clean` flag to `gcc` to ensure that the common subset between C and COOGL is compiled and that causes an initial hidden header file to be included, `clean.h`, so that `cast()` and other things that need to be mapped are mapped, including basic types, etc.

5R.8 Compiler Specification Files

A source code repository with many COOGL programs, libraries, and loadable modules can be browsed with a code browser that is language aware by using the COOGL compilation description file. With C, makefiles and complicated build environments and scripts would have to be examined or interpreted to determine what files belong to what modules, programs, etc. The compilation description file needs a specification.