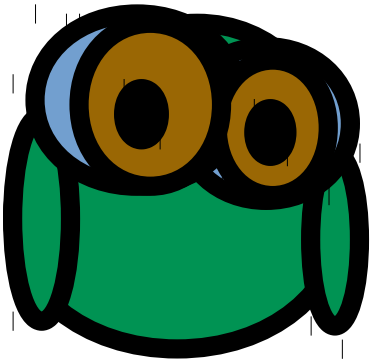
www.COOGL.org

COOGL



pronounced *see-oogl* as in:

"see ogly eyed bird ogling at you"

Concurrent Object Oriented Generic Language

1.3.027-lo-5.4.7.2

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 Error: Reference source not found), 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

1.2 Object oriented terminology

1.3 Member function invocation syntax

```
class stack promise(empty()) {
    pub lit int MAXENT = 100; // MAXENT is a literal constant
    priv int entries[MAXENT]; // an array with MAXENT ints
    priv int *sp = entries;   // sp, an int pointer initialized
    return;                   // to the 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]; }
}
```

```
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;
}
```

1.4 Hello world and type safe input and output

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

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

```
$ coogl hello.coogl
$ ./a.out
Hello, World.
$
```

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

```
#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);
}
```

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

```
on (semicolon_separated_expression_list) function_invocation_expression
```

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

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

```
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());
}
```

1.5 Compilation model

1.6 C versions and Coogl ancestry

1.7 C language schism: concurrency and 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.

"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."

> *"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)

*"**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

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

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.

1.8 Coogl syntax and language design philosophy

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

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

1.9 Programming language complexity

> *"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*

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

"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++."

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."

"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."

1.10 Book Organization

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

2.2 Comments

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

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

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

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

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

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

```
/# 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

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

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

```
int i = 10;
```

2.4 Functions and the `return` statement

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

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

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

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

2.5 Integer, floating, character, and string literals

	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	
ularge		v		v		v	v	v	v	v	v	v

2.6 Declarations and declaration contexts

```
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
```

```
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
};
```

2.7 Declaration kinds

```
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
```

```
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
```

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

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

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

```
typedef int integer;
```

```
typedef int *intptr;
```

```
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
```

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

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

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

2.8 Order of declarations

```
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
```

```
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;
}
```

2.9 Statements within functions and classes

2.10 Introduction to operators and expressions

```
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
}
```

2.11 Compound statement

{ *possibly_empty_statement_list* }

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

2.12 The `assert()` function and `...` statement

2.13 The `if` and `if else` selection statements

`if` (*expression*) *statement*

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

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

2.14 The `while` and `for` iteration statements

`while` (*expression*) *statement*

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

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

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

```
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;
}
```

2.15 Operators and expressions

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

"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”."

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

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

Expression	Result
~0xFFFF	0xFFFFFFFFFFFF0000
~0	0xFFFFFFFFFFFFFFFF

2.16 Controlling expressions, relational operators, and truth values

```
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");
}
```

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

2.17 Logical operators

2.18 Assignment and assignment-*op* operators

`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

```
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)
```

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

2.19 Increment and decrement operators

```
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
}
```

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

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

2.21 C array types, operators and expressions

```
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
```

```
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();
}
```

```
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++;
}
```

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

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

2.22 Pointers: types, operators, and expressions

```
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
```

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

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

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

```
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);
    }
}
```

```
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);
    }
}
```

```
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;
}
```

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

```
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;
}
```

2.23 Aggregate types and their operators

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

```
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
}
```

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

```
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;
}
```

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

2.24 Expressions

```
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;
}
```

2.25 Expression statements

```
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
    }
}
```

```
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
}
```

2.26 Default value returned by `main()`

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

2.27 The `if` and `if else` selection statements and indentation errors

```
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;
}
```

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

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

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

```
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);
}
```

2.28 The `goto` statement

```
size_t merge(int src[], size_t n, int src2[], size_t n2,
             int dest[], size_t nd) {
    expect(n <= src.max[0] && n2 <= src2.max[0]   // ignored
           && nd <= dest.max[0] && !(n1 ?+ n2));  // as C code
    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++;
}
```

2.29 The `switch` statement

```
switch (expression) compound_statement
```

```
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;
    }
}
```

```
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;
    }
}
```

```
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);
}
```

```
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;
}
```

2.30 The `do-while` iteration statement

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

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

2.31 The `break` and `continue` statements

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

}
```

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

```
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: ;
}
```

```
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

```
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]);
}
```

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

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

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

```
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
}
```

```
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);
}
```

3.2 Multi dimensional array descriptors

```
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
}
```

```
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]
}
```

```
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
```

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

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

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

3.3 Array descriptor access restrictions

3.4 Restricted array descriptors

3.5 Restricted array descriptor accesses are atomic

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

3.6 Array and array descriptor indexing is checked

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

3.7 Arrays of arrays vs multidimensional arrays

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

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

3.8 Array descriptor use in expressions

```
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

```
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;
}
```

```
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;
}
```

```
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;
}
```

3.10 Use of pointers based on array descriptors is always safe

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

3.11 Functions that return array descriptors

```
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];
}
```

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

3.12 Implicit array descriptor for string literals

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

3.13 Tuples

```
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
}
```

```
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

```
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 = 3.14159265358979323846264338327;
```

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()`

4.2 Class declarations are function declarations

```
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

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

```
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 ...
}
```

4.4 Object declarations and `decl`

```
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

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

4.6 Introduction to inheritance and member function redefinition

```
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
    }
}
```

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

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

4.7 Access to redefined member functions

```
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

4.9 Restrictions on constructor calls to non-static member functions

4.10 Constructor organization

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

```
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

```
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 ...
}
```

```
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 ...
}
```

```
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);
}
```

```
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`

```
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 ...
}
```

```
stack *const this
```

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

```
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); }
    }
}
```

```
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);
}
```

4.15 Functions as degenerate types and nested member functions

```
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);
}
```

4.16 Functions with default argument expressions

```
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)[] {...}
```

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

4.17 The stringify operator

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

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

```
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);
}
```

```
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

5.3 Destructor, the `deinit()` member function

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

5.4 Destructor can not call non-static member functions

5.5 Brief introduction to namespaces

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

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

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

```
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();
        }
    }
}
```

```
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()`

5.9 The `lang.value` interface

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

5.10 Member functions specified by `lang.value`

```
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;
}
```

5.11 A `string` class example

```
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()`

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

5.13 Some `string` operations

```
    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;
    }                                                // continued below
```

```
    pub index find(char c, index start = 0) {          // string
        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
```

5.14 The initialization constructor: `init()`

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

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

5.15 Brief preview of strings of generic value types

```
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
}
```

5.16 Object slicing along incorrect type boundaries is not allowed

5.17 Pseudo constructors

```
    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
```

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

5.18 Default construction

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

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

5.19 Object reinitialization: `reinit()`

```
    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
```

```
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)
}
```

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

```
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); }
```

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

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

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

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

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

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

```
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()));
        }
    }
}
```

5.23 Object arguments and return values

5.24 Literal members

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

6.2 Interfaces

6.3 Single inheritance and multiple interfaces

```
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
}
```

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

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

6.4 The `defer` and `redef` function modifiers

```
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;
    }
}
```

```
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

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

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
}
```

6.6 Redefining static member functions

6.7 Accessibility modifiers

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

```
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);
    }
}
```

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

6.9 Member access aliases

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

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

```
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;
    }
}
```

```
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;
}
```

```
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

```
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
}
```

```
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 ...
}
```

```
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);
}
```

```
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 ...
}
```

```
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);
}
```

6.11 Single inheritance example

```
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;
}
```

```
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
```

```
    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
```

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

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

```
    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); }
}
```

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

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

6.12 Pointers and inheritance

```
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;
}
```

```
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

```
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;
}
```

6.14 Constructor and destructor restrictions and contracts

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`

7.2 Class declaration continuation: `continue class`

```
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); }
}
```

7.3 Class of pointers and array descriptors implicit location

7.4 Pointer arithmetic

7.5 The `sizeof` and `sizeofex` operators

7.6 Layout control of class objects: `class struct`

```
class struct foo { pub char c; pub large l; pub short s; }
void test() { assert(sizeof(foo) == sizeof(large) * 3); }
```

7.7 Only global declarations can be hidden

7.8 Name lookup relative to the scope of a function

```
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);
    ...
}
```

```
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

```
struct person {
    char *name;
    int age;
};
person j = {"Jill", 29};
person k = {.name = "Ken", .age = 31};
```

```
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
```

```
    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`

```
extend class stack {
    pub void iterate(void work(int val) deleg) {
        for (int *p = sp; --p >= ent; ) work(*p);
    }
}
```

```
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();
}
```

```
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();
}
```

```
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

```
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

8.2 Global declarations in C

```
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;
}
```

```
extern int random_prev;                        // C code in file use.c
int main() { random_prev = 0; }
```

8.3 Modules and accessibility modifiers

8.4 Publicized and published declarations

8.5 Module specification

```
$ cooglc file1.coogl file2.coogl file3.coogl -o name
$ cooglc name    # equivalent, name.spec lists the source files
$ cat name.spec
file1.coogl
file2.coogl
file3.coogl
$
```

8.6 Controlling access to class as type vs as constructor

```
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

```
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 ...;
}
```

```
pub namespace lang { ... }// COOGL compile and run time support
pub namespace lib { ... } // COOGL library
```

```
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);
}
```

```
int cat() {
    int c;
    while ((c = libc.getchar()) != libc.EOF)
        if (libc.putchar(c) == libc.EOF) return libc.EOF;
    return 0;
}
```

```
int cat() {
    import libc;
    int c;
    while ((c = getchar()) != EOF)
        if (putchar(c) == EOF) return EOF;
    return 0;
}
```

```
namespace C99_standard_library { ... };
```

```
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;
}
```

```
import libc;
int cat() {
    int c;
    while ((c = getchar()) != EOF)
        if (putchar(c) == EOF) return EOF;
    return 0;
}
```

```
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

8.9 Class initialization

```
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

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()`

```
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 The on statement

```
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);
    }
}
```

```
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();
}
```

9.3 The on expression

```
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);
}
```

```
n = on (a; " "; b; '\n') scan();
```

```
{
    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;
            }
        }
    }
}
```

```
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;
}
```

```
{
    typedef int (*scandfp)() deleg;
    scanfp scantab[4] = {&a.scan, &" ".scan,
                         &c = b.scan, &'\n'.scan};
    n = lang.on_array(int, scantab);
}
```

```
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;
    }
}
```

```
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
    }
}
```

```
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;
    }
}
```

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

```
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

```
void print7primes(FILE *f) {
    on (2; 3; 5; 7; 11; 13; 17) fprint(f ? f : stdout);
}
```

9.5 Byte count vs operation count `on` value convention

9.6 Compile time and run time enabled traces with `on`

```
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;
}
```

```
import tr;
int doit(int n, char name[]) {
    on (tr.go; "n="; n; "name="; name; tr.end) trace();
}
```

9.7 Optional argument expression evaluation

```
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

9.9 Use of `return` *`expression;`* in `void` functions

```
void f() {...}
void g() {
    if (a()) return f();
    ... complicated code follows ...
}
```

```
void g() {
    if (a()) {
        f();
        return;
    }
    ... complicated code follows ...
}
```

9.10 Function values that are `vital`

```
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);
}
```

9.11 Classes whose objects are `vital`

```
class trace() vital { ... }
```

9.12 Jump statements cause object destruction

9.13 Loop-member functions and the `loop` 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
    }
}
```

```
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);
}
```

```
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;
}
```

```
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;
}
```

9.14 No structured exception handling

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

Group	Operator				Associativity	Description		Example	
1	. -> [] () ()inline				left	dot arrow array subscript function call inline call		obj.member ptr->field array[index] func(a,b) f() inline	
2	++ -- - ~ .. &		?++ ?-- ?- ! ^ *		right	inc dec negative bit not global addressof	checked++ checked-- checked- logic not member pointedby	++i i-- -i ~i ..g &i	i?++ ?--i ?-i !i f(^m) *ip

	sizeof()
	#
	cast()
	up_cast()
			sizeofex()
			try_cast()
			uptr_cast()
			unsafe_cast()
					right	void stringfy(t a, char m[] = #a){}
						t *p = uptr_cast(t, NIL) u;
						int *up = try_cast(int*,a[],NIL)cp;
						derived *d = up_cast(base *, NIL)b;			

3	* / %		?* ?/ ?%		left	multiply divide modulo	checked * checked / checked %	i * j i / j i % j	i ?* j i ?/ j i ?% j
4	+ -		?+ ?-		left	plus minus	checked + checked -	i + j i - j	i ?+ j i ?- j
5	<< >>				left	shift left shift right		i << j i >> j	
6	< <= > >=				left	less than less or equal than greater than greater or equal		i < j i <= j i > j i >= j	
7	== !=				left	equal not equal		i == j i != j	
8	&				left	bitwise and		i & j	
9	^				left	bitwise xor		i ^ j	
10	|				left	bitwise or		i | j	
11	&&				left	logical and		i && j	
12	||				left	logical or		i || j	
13	? :				right	conditional		i ? j : k	
14	= ?= |= &= ^=	<<= >>= *= ?*=	+= ?+= /= ?/=	-= ?-= %= ?%=	right	assign, op assign, checked assign, and checked op assign		i=j i<<=j i-=j i/=j	s?=u i+=j i?*=j i?%=j
15	,				left	comma		i, j	

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`

```
if (min <= n & n <= max)
```

```
/* 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 */
```

```
(a ^ b) == ((~a & b) | (a & ~b))
```

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`

```
if (x & 1 && b == 0) go();
```

```
if ((x & 1) && (b == 0)) go();
```

10.2 The member lookup operator

10.3 Fine grained function inline control

10.4 Checked arithmetic operators

```
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];
}
```

```
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

C keywords affected by Coogl	
removed	**enhanced**
auto const extern long long register signed unsigned volatile	enum union static void

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 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

10.6 Removed keywords

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 identifier 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 Error: Reference source not found), 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 §SError: Reference source not found.

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
```

```
int main() {
    int a = 2000111222;
    int b = a + 1000333444; // b == -1294522630 (overflowed)
    int c = b - 1000000000; // c == 2000444666 (underflowed)
    assert(a + 333444 == c);
}
```

"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'; }
```

```
int shift32(int a) { return a << 32; }
```

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

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

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

```
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

```
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
```

```
char *cp = "abc";             // valid
decl char *("abc") cp;        // invalid
```

```
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

```
void swap(genre lang.value type, type *a, type *b) {
    type temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void test() { int i = 1, j = 2; swap(int, &i, &j); }
```

```
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
}
```

```
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]; }
}
```

11.4 Restrictions on type arguments

```
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
}
```

```
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
}
```

```
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

```
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))];
}
```

```
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;
}
```

```
decl stuff(1, cast(double *) NIL, 3.141593, "hi") s;
```

```
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();
}
```

11.6 Specialization of generic classes and functions

```
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
}
```

```
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
}
```

```
class bitmaplit(genre lang.whole type = ularge,
                  pub 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;
}
```

```
void x() { decl bitmaplit(256) b; assert(sizeof(b) == 32); }
```

11.7 Type variables must be initialized, never assigned

11.8 Function names vs class names

11.9 The `argsof` tuple type member

```
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

```
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
    }
}
```

```
class ratio(pub int numerator, pub int denominator) {
    pub is lib.creatable(ratio);
}
```

```
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
}
```

11.11 Public static member functions that can't be inherited

`pub !inherit`, which causes `create()` not to be inherited by `derived`.

11.12 Literal arguments to generic classes

11.13 Field name argument declarations with `fieldof`

```
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(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);
}
```

11.14 Generic intrusive lists

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

11.15 Generic doubly linked list: `list`

```
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);
}
```

11.16 Use of `list`

```
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);
}
```

```
decl list(entry, link1) list1;
decl list(entry, link2) list2;
decl list(entry, link3) list3;
```

```
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() {
    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

12.2 Indexing types

12.3 Floating point, complex, and imaginary types

```
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

```
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 =  3.14159265358979323846264433 };
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
```

```
pub kind(^USERDEF) u;
```

```
enum ularge page {
    SIZE   = 4096,
    OFFSET = SIZE - 1,
    MASK   = ~OFFSET,
    ...                         // other values are valid
};
page pg = 0;                    // integer type: ularge
```

```
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
}
```

```
enum class mode {
    r = 4,
    w = 2,
    x = 1,
    ...    // other values are valid
}
mode rw = mode.r | mode.w;
mode rwx = rw | mode.x;
```

```
decl mode(^r | ^w) rw;
decl mode(^r | ^w | ^x) rwx;
```

```
(db) print rwx
mode.r | mode.w | mode.x
(db) print cast(int) rwx
7
(db)
```

12.5 Bit fields

```
struct ieee64 { uint sign:1; uint exponent:11; fraction:52; };
```

```
struct ieee64 { fraction:52; uint exponent:11; uint sign:1; };
```

```
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();
}
```

```
0000000f000703010000000000000001f000000000000003f
```

```
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)
```

12.6 Unicode characters

```
unic u = U'x';               // 32 bit Unicode character literal
unic up[] = U"32 bit Unicode literal";
```

12.7 Unicode 16 bit characters

```
char16_t u = u'x';           // 16 bit Unicode character literal
unic up[] = u"16 bit Unicode literal";
```

12.8 Character and string literal

```
void invalid() {
    int c = 'abcd'; // error: multi-character literal
}
```

```
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();
}
```

12.9 Incompatible and global types

```
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
```

12.10 Types and literal dimensions

```
typeglob float {m} meter_t;
typeglob float {min} minutes_t;
```

```
typeglob float {km = 1000`meter} km_t;
typeglob float {hr = 60`min} hour_t;
typeglob float {kph = 1`km / 1`hr} kph_t;
```

```
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
}
```

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`

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`

12.15 Base class of all dynamically sized arrays: `lang.dynarray`

12.16 Construction and destruction of `lang.carray` and `lang.dynarray`

```
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
}
```

```
class building {
    pub apartment apt[10];
    pub void deinit() { apt.deinit(); }
}
```

12.17 `lang.arraydesc` and `lang.vecdesc` array descriptors

```
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];
    }
}
```

12.18 Number type interface hierarchy: `lang.number`

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 *`

```
class void * {
    pub is lang.value(genre void *);
}
```

```
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.
}
```

12.20 Smart pointers and their `priv` member: `ptr`

```
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(); }
}
```

```
continue class stk * {
    pub is nilable(class stk *);                    // See §12.21.
    pub is equalable(class stk *);                  // See §12.21.
    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
}
```

```
void use() {
    int err;
    stk *sp = stk.create(20, &err);
    sp->push(1);
    sp->push(2);
    sp->pop();
    sp->pop();
}
```

12.21 Explicitly declared classes and smart pointer restrictions

```
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());
    }
}
```

```
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?
    }
}
```

```
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

13.2 `The v[]` declaration syntax in C

```
/* 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 */
```

```
/* Argument declaration, equivalent to: void f(int *v) {...} */
void f(int v[]) {...}           /* Valid in K&R-C/C89/C99/C11 */
```

```
/* 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[]; }
```

```
/* Declaration of a[] not as the last structure member: */
struct s { int v[]; int b; } /* Invalid in: K&R-C/C89/C99/C11*/
```

```
/* Local variable declaration: */
void function() { int v[]; } /* Invalid in: K&R-C/C89/C99/C11*/
```

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 The `type v[][]` declarations are always invalid in C

```
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

```
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);
}
```

```
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
}
```

```
float sum(int n, float vector[]) { ... }
float sum(int n, float *vector) { ... }
```

```
/* error: array type has incomplete element type  */
int f(int m[][]) { ... }
```

```
int f(int m[][20]) { ... }
```

```
int f(int (*m)[20]) { ... }
```

```
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(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);
}
```

13.6 Restrictions on array descriptors and variable length arrays

13.7 Array memory reinterpretation

```
void use() {
    int a[10][21];
    int b[][][][] = lib.array.make(int, {2,3,5,7}, a.start);
}
```

13.8 Dynamic creation and destruction of arrays

```
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;
}
```

```
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

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.

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 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

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.

14.4 Bad memory accesses in C

```
void wrong() { char x[1]; x[200] = 'x'; }
```

```
void store_1_at_index_100(int *p) { p[100] = 1; }
void wrong() { int v = 1; store_1_at_index_100(&v); }
```

```
void stomp() { char *p = malloc(10); free(p); *p = 'x'; }
```

```
char s[] = {"hi"};
char d[3];
void stomp() {
    s[2] = 'x';
    strcpy(d, s);
}
```

```
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 */
}
```

```
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); }
```

14.5 Plain and non-plain data and types

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

14.7 Unions can't contain indexes, pointers, or array descriptors

14.8 Global memory can't refer to memory on the run-time stack

14.9 Returning addresses of run-time stack allocated memory

```
char *set(char *p) { *p = ' '; return p; }
char *bad(char c) { return set(&c); } // error: returns address
                                      // of local: c
```

```
char buf[] = {"hello"};
char *set(char *p) { *p = ' '; return buf; }
```

```
char *bad(char c) { return set(&c); } // no error produced
                                      // at compile time
```

```
char *set(char *p) { *p = ' '; return p; }
```

```
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
}
```

14.10 Run-time stack allocated memory and execution contexts

14.11 Run-time stack growth is checked

14.12 Casts and safety: `cast()` and `try_cast()`

```
try_cast(type *, mem, value) addr
```

```
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);
}
```

```
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; };
```

```
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;
}
```

```
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;
}
```

```
    if (data.max[0] < size)
        return make_message_slow(data, h, pre, b, post, f);
```

14.13 Restrictions on class members whose type is a plain data type

14.14 Implicit pointer conversions without casts

```
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
}
```

```
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_up_cast_example(nfp, nnp, fp, rwatp, rwp);
}
```

14.15 Pointer to base up cast to pointer to derived: `up_cast()`

```
up_cast(type *, value) ptr
```

```
nfs_pointer_up_cast_example(nfs_file *nfp, nfs_node *nnp,
                            file *fp, rdwrat *rwatp, rdwr *rwp){
    nfp = up_cast(nfs_file *, NIL) nnp;          assert(nfp);
    nfp = up_cast(nfs_file *, NIL) fp;           assert(nfp);
    rwatp = up_cast(rdwrat *, NIL) rwp;          assert(rwatp);
    nfp = up_cast(nfs_file *, NIL) rwp;          assert(nfp);
    nnp = up_cast(nfs_node *, NIL) rwp;          assert(nnp);
    fp = up_cast(file *, NIL) rwatp;             assert(fp);

    // this fails, the rwp did not come from a nfs_dir object
    nfs_dir *ndp = up_cast(nfs_dir *, NIL) rwp;  assert(!ndp);
}
```

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. Oth-

erwise, 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.

14.17 Trapping pointer value interface and implementation

14.18 Use of NULL and zero as pointers is deprecated

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; }
```

14.19 Addresses of members based on NULL or trapping addresses

```
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

```
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.

14.21 Deconstructed values and uninitialized variables

```
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
```

14.22 Permanent association of heap virtual addresses and types

14.23 Array walking through pointer ranges is always valid

```
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;
}
```

14.24 Invalid pointer value computation

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;
}
```

```
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;
}
```

14.25 Use of objects at `start-1` and at `end`

```
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]
}
```

```
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
}
```

14.26 Out of bounds indexing causes an exception

```
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;
}
```

14.27 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 §14.29) 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 carefully 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.

14.28 Dynamically unloaded modules and safety

14.29 Hardware and software exceptions and exception handlers

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

15.2 Language design considerations

15.3 Allowing concurrency support through libraries

```
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;
}
```

```
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;
}
```

```
pub void set_low_wsb(wsbx *d) {
    ularge u = *(ularge *) d;
    u |= 0x300020100uLL;
    *(ularge *) d = u;
}
```

```
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];
}
```

```
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];
}
```

15.4 Concurrency support in libraries is optional

15.5 Weakly ordered concurrent memory accesses

15.6 Concurrency support in C

```
void remove(node *p) {
    p->prev->next = p->prev;
    p->next->prev = p->next;
}
```

15.7 Language design dilemma

15.8 Concurrent programming building blocks

15.9 Execution contexts

15.10 Threads, mutexes and condition variables

15.11 Weaknesses and complexity in C11 `<threads.h>`

15.12 Concurrency support in Coogl `lib.concur`

```
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() {...}
        }
    }
}
```

```
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();
}
```

```
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();
}
```

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();
    }
}
```

```
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

15.15 C11 and C++11 memory model

15.16 Coogl memory model

15.17 Atomic memory operations

15.18 Exception handlers