C++ Language
(for Javascript programmers)

Statically Typed Language

- **Every variable must have a type known at compile time**
  - **Allows compiler to check more rigorously**
  - **Allows compiler to produce more efficient machine code**

- **<typespec> <var_id>; // general syntax**

- **<typespec> <var_id> = <init_expr>;**

Basic Types

```
int i;                     // 32-bit signed integer
unsigned int ui;           // 32-bit unsigned integer
long long ll;              // 64-bit signed integer
unsigned long long ull;    // 64-bit unsigned integer

float f;                   // 32-bit floating-point
double d;                  // 64-bit floating-point


float f = 35.2f;           // variable with initializer
const float f = 35.2;      // can't be modified

unsigned long long ull = 0x0123456789abcdefLL;
                           // long long hex literal init value

short s;                   // 16-bit signed integer
unsigned short us;         // 16-bit unsigned integer

char c;                    // one 8-bit character/int, like 'z'
                           // c = 5     -- also legal
unsigned char c;           // unsigned 8-bit char/integer
```

std::string type

```
std::string s1 = “Bob”;
std::string s2 = “Fred”;

if ( s1 == s2 ) {              // compares string values
    ...
}

s1.charAt( 2 )                 // returns char ‘b’

std::string s3 = s1 + “ “ + s2 + “ “ + std:to_string( 5 );
                               // “Bob Fred 5”
```

Classes

```
// C1.h – header file containing class definitions
//        not implementation (though it’s legal to do so)

class C1
{
public:
    C1( int n );             // constructor, takes int param
   ~C2();                    // destructor

    void m1( void );         // method that takes no args and
                             // returns nothing
    float m2( float f );     // takes a float and returns a float

private:
    ...                      // users of C1 can’t get to these
};
```

Objects Can Be Allocated Locally

```
static void main( void )
{
    C1 obj1( 2 );            // calls C1() constructor;
                             // allocates object right here

    obj1.m1();               // calls method
    // implicitly calls ~C1() destructor when returning
}
```

Except for std::string, I generally don’t recommend allocating objects locally. It can force a lot of expensive copying.

Let’s talk pointers and dynamic allocation...

Pointers are Like References

```
C1 * obj1 = new C1( 5 ); // C1 * means pointer to C1 obj
                         // new allocates dynamic memory
                         // and calls constructor C1() on it

obj1->m1();              // -> is the dereference operator
                         // works only if obj1 is a pointer

C1 obj2 = *obj1;         // obj2 declared here (not ptr)
                         // *obj1 refers to entire object
                         // so obj2 initialized to a copy

const C1 * obj3 = obj1;  // makes a copy of pointer;
                         // const here means that *obj3
                         // can't be changed, NOT that
                         // obj3 can't be changed
obj3 = obj2;             // legal, obj3 is not const!

C1 obj4 = obj3;          // illegal!  can't turn const *
                         // into a non-const *

delete obj1;             // calls ~C1() on *obj1
                         // need not do it in same routine
```

Much cheaper to pass a pointer (64-bit) than copy an object.

Method Implementations

```
// C1.cpp - implementation of C1.h
//
#include "C1.h"

C1::C1( int n ) // remember: called by new C1()
{
    this->some_var = n;  // this is the pointer to this object
    some_var = n;        // equivalent to previous
}

C1::~C1()       // remember: called by delete
{
    // typically deallocates internal dynamic data structures
}

void C1::m1( void )
{
    // some method
}
```

Methods and Const

```
class C2
{
...
      void m1( const C2 * other ); // other is ptr to const C2

      void m2( C2 * other ) const; // *other can be modified
                                   // const here means m2()
                                   // must not try to modify
                                   // *this (in)directly
};

...

C2       * o1 = new C2();
C2       * o2 = new C2();
const C2 * o3 = new C2();   // *o3 can't be modified


o1->m1( o2 );             // legal
o1->m2( o2 );             // legal
o1->m1( o3 );             // legal
o1->m2( o3 );             // illegal: o3 is const ptr
```

Static Values and Methods

```
class C3
{
...
     static int N = 5;                // variable assoc w/ class
     static const int P = 10;         // constant

     static void m4( float f );
};

...

C3::m4( 35.3 );                       // legal, class as a whole

C3 * o1 = new C3();

o1->m4();                             // might be legal and equiv
                                      // to C3::m4(), but
                                      // not good practice
```

Arrays

```
int a1[5] = { 5, 6, 3, 2, 1 };    // array of 5 ints

int a2[]  = { 5, 6, 3, 2, 1 };    // shorthand
```
647885

```
int a3[10];                       // uninitialized

int a3[];                         // illegal, no length

int * a4 = new int[5];            // a4 is a pointer to
                                  // array of 5 contiguous
                                  // ints

a4[0]                             // legal, equiv to *a4

int * a5 = &a4[2];                // a5 ← address of 2nd elem
int * a5 = a4 + 2;                // equivalent, this is
                                  // called pointer arith
                                  // between ptr and int

a5[10]                            // legal, will reference
                                  // garbage or segment. viol

int * a6 = new int[n];            // n need not be a constant
```

Array/Pointer Method Params

```
void m1 ( int * val )          // val is pointer to an int
{
    *val = 25;                 // legal
}

void m2 ( int a1[] );          // legal, a1 is pointer to ints
void m2 ( int * a1 );          // equivalent, but less clear
                               // (a1[0] is legal here)


void m3 ( const C1 ** c1 )     // c1 is pointer to a pointer ☺
{
    *c1 = new C1();            // way to return a pointer
}

Hint: typespecs are read right-to-left:
          pointer to a pointer to a const C1
```

PIMPL Technique
(how to keep your private parts private)

```
// C1.h - class interfaces

class C1
{
public:
     ...

private:
     class Impl;                // incomplete class definition is ok
     Impl * impl;               // pointer to implementation (PIMPL)
                                // (legal because doesn't need to
                                //  know the size of Impl)
};
```

And now in the .cpp file...

PIMPL Technique
(how to keep your private parts private)

```
class C1::Impl              // completion of definition
{
    // your private parts are safe here
};

C1::C1( void )
{
    impl = new Impl;        // equiv to: this->impl = new Impl;
    impl->foo = ...;
}

C1::~C1()
{
    delete impl;
    impl = 0;
}

C1::method( void )
{
    impl->foo = ...;
}
```

struct – carryover from C

```
struct S1
{
    int   foo;
    float goo;
};
```

is nearly equivalent to this, which is what I recommend:

```
class S1
{
public:
    int   foo;
    float goo;
}
```

To make it even more equivalent, create a new type S1:

```
typedef struct
{
    ...
} S1;       // can now use S1 as name to refer to new type
            // but I recommend using class S1 instead
```

char * - carryover from C

```
char * s1 = “Bob”;          // s1 is pointer to raw array of chars
                            // ends with \0 (null char)

const char * s2 = “Fred”; // s2 is pointer to
                            // const array of chars

s1[0]                       // returns ‘B’ in this case
s1[3]                       // returns ’\0’ in this case

std::string s3 = s1;        // copies “Bob” to std::string
s3.c_str()                  // returns const char * (internal repr)


const char * array[] = { “Bob”, “Fred”, “John” };
                            // array is also a const char **
                            // (pointer to one or more const char *)

// there are many C functions that mess with char *,
// some of the good ones are in #include ”string.h”

// however, try to use std::string rather than char *
```

Enumeration Types (shorthand)

```
enum
{
    KIND_FOO,               // numbering starts at 0
    KIND_GOO,
    KIND_MOO = 5;           // but can override
};

is basically equivalent to:

static const int KIND_FOO = 0;
static const int KIND_GOO = 1;
static const int KIND_MOO = 5;

You may also create a specific enumeration type:

typedef enum
{
    ...
} kind_t;       // can now use kind_t as name of new type
                // and kind_t is different from int when
                // compiler does type checking
```

auto type

```
C1 * o1 = new C1();
auto o1 = new C1();       // compiler assigns type: C1 *

auto f1 = 1.2f + 3.0f;   // compiler assigns type: float

auto f2 = 1.2 + 3.0;     // compiler assigns type: double

auto o2;                  // illegal: no way to know type

// I have no problems with your using auto
```

Standard Output Prints

```
#include “iostream.h”
...
cout << “Hello there i=“ << i << “\n”;
cerr << “ERROR: WTF\n”;   // prints to standard error


// however, I prefer that you use our Sys.h static class:

#include “Sys.h”
...
Sys::print( “Hello there i=” + std::to_string( i ) + “\n” );
Sys::die( “ERROR: WTF” );
Sys::assert( x == 5, “x should be 5 at this point” );
```

Raw Data Init and Copy

```
#include “Sys.h”

...

float * f1 = new float[n];  // allocated memory uninitialized!

Sys::memzero( f1, n * sizeof( float ) );    // zero-out bytes

float * f2 = new float[n];

Sys::memcpy( f2, f1, n * sizeof( float ) ); // copy raw bytes
                                            // from f1 to f2
```

These routines are faster than for loops.

Building and Running

Manual Compile and Link

```
// myprog.cpp
//
#include “C1.h”

static int main( int argc, const char * argv[] )
{
    // argc == arg count + 1
    // argv[0] holds program name, arg[1] holds first arg
    //
    C1 * o1 = new C1();
    ...
    return 0;             // normal/success return
}

$ g++ -o C1.o –c C1.cpp       // compile C1.cpp -> C1.o (object)
$ g++ -o myprog.o –c myprog.cpp // compile myprog.cpp -> .o

$ g++ -o myprog myprog.o C1.o  // link executable myprog
                               // often we say myprog.exe instead
```

This is fine for tiny ”play” programs, but for something serious it is better to use “make” which figures out what needs to get rebuilt and in what order...

Makefile

```
DEPS = \                        # all .o files depend on all these
    C1.h \                      # don't need to list .cpp files

OBJS = \                        # .o files to get built
    C1.o \
    myprog.o \

LIBS = \                        # other .o files not rebuilt here
    ../lib/*.o \

PROGS = \                       # executables to get built
    myprog.exe \

include ../make/common.mk       # all the disgusting make rules
                                # (written by Bob)
```

```
$ make                          # makes all executables (if needed)
$ make myprog.o                 # builds just myprog.o  (if needed)
$ make myprog.run               # equiv to: ./myprog.exe
$ make myprog.drun              # equiv to: gdb -tui myprog.exe
                                # (debugging discussed next)
$ make clean                    # delete all .o and .exe files
```

gdb debugger

[I'll give a live demo]

```
$ g++ -g ...                    # on both compile and link lines
$ gdb -tui myprog.exe
$ make myprog.drun              # equivalent, in our environment
```

- `b foo` — # set breakpoint at function foo
- `R` — # run prog (or rerun it)
- `R arg1 arg2` — # (re)run with arguments
- `c` — # continue from breakpoint
- `bt` — # show function call frames
- `fr 2` — # focus on frame 2
- `p x` — # print value of x
- `p/x x` — # print value of x in hex
- `p x->foo` — # can print any expression
- `p x->method( 2 )` — # can even call functions
- `n` — # next statement (step over)
- `s` — # next statement (step into)
- `list foo.cpp:25` — # list source at foo.cpp line 25
- `help commands` — # show gdb commands

./.gdbinit file may contain gdb commands that will get executed when gdb starts up, such as setting breakpoints, R arg1, etc.

gprof Profiler

[I'll give a live demo]

1) Edit application Makefiles and add -pg option to compile and link lines. Different apps have different Makefile orgs.

2) Rebuild the app from scratch so that -pg takes effect.

3) Run the app as it's normally run. Try to arrange things so that the program runs for at least a minute.

```
./myprog arg1 ...
```

It will generate a ./gmon.out binary file with raw stats.

4) Run gprof to produce human-readable output from gmon.out:

```
gprof myprog gmon.out > analysis.txt
```

analysis.txt will show histogram of top routines and those are the ones that should be accelerated

Note: it's important to choose a representative set of apps!