CS242N Programming Skills Workshop

Cindy Grimm **Bill Smart**
Department of Computer Science
Washington University in St. Louis

Fall 2002

Contents

Chapter 1

Overview

This document is for people who are learning C++ and want to have a better understanding of how source code becomes a running program. It's also useful for more advanced users who have specific questions about compile, memory, or semantic quirks. This is not a C++ tutorial, it's more of an explanation of how your program interacts with the compiler, the operating system, how to write code that requires a minimum of debugging, and a C++ "gotcha" guide.

This is not a manual. You are very strongly encouraged to supplement this information with `man` under Unix/Linux, and Visual Studio's help `<http link>` which is online. You can learn a lot about a compiler both by reading a compiler book and by looking at compile command options.

Both Unix-style systems and Microsoft's Visual studio are covered in this document. Despite the very different feel, they're almost identical underneath. We've indicated where they differ, what you can take advantage of in each environment, and some tips for writing portable code.

One thing to bear in mind is we've written this guide for people writing research or class code, not necessarily production code. These are projects that range from very small (a couple pages) to medium-sized (5-10 libraries, 10,000 lines of code) and that you'll spend much more time developing then running. In a typical research or class project you'll spend most of your time developing and testing algorithms. You may run your program many times while you're debugging it, but once it's working and you've processed your data the code goes on the shelf. Parts of it may see continuous re-use, and parts of it may get cannibalized for other programs, but the program itself is not continuously used like Emacs is. Because of this, it's important to minimize debugging and writing time, not running time. Making a code change that takes three days to write and debug, and saves 1/2 second of run time, when you're only going to run the program maybe 1,000 times, is a waste of time. Re-writing a simple, easy to understand algorithm into a complicated one is also bad, because you'll never be able to figure out what you did and neither will anyone else who uses it.

The document is broken into three parts. The first part talks about the compile and link process, or how your source code is turned into an executable. Some of the more common compiler and linker errors are discussed here, along with methods for fixing them. The second

section talks about memory. There's a discussion of the heap and stack, plus a laundry list of common memory and pointer problems and how to avoid or find them. There's also a brief discussion on some advanced memory techniques such as pools. The third section is the "style" section. Here we discuss how to lay out classes and source code, with an eye to reducing debugging, re-writing, and "re-understanding" time.

Important Note

This document is very much a work in progress. It is by no means complete, perfectly ordered, or all-inclusive. In its current form, it essentially represents a brain-dump by the authors. The intention of releasing it in this form is to get feedback from you, the reader, and to try to begin cataloging the various things that we cover in CS242. As such, it is only meant for distribution within Washington University in St. Louis. Please do not distribute it to anyone else without the authors' express permission.
If you have any comments or suggestions about this document, then send them to one of the authors by email:

Cindy Grimm: `cmg@cs.wustl.edu`
Bill Smart: `wds@cs.wustl.edu`

Part I

Mechanics

Chapter 2

The Mechanics of Writing Code

In this chapter we talk about how your code is converted from a set of header and source files to a single executable or library. We'll also talk about some common compiler and linker errors, how to go about adding and using an external library, and static versus dynamically linked libraries.

2.1 File Names

Before we get started talking about the actual mechanics of writing code, we'll talk about files and file-naming conventions. `C` and `C++` have specific conventions for the names of files, and what sort of thing goes into each file.

2.1.1 Header files

Header files typically end in `.h` or `.H`, although you will occasionally see `.hpp` as well. Header files ending in `.c` will correspond to either `C` or `C++` code, while `.H` and `.hpp` generally imply `C++`.

Header files typically contain (at least some of) the following:

- Macro definitions:

  ```
  #define PI 3.1415
  ```

- Inline functions:

  ```
  inline double Pi() { return 3.1415; }
  ```

- Calling conventions for functions defined in a `.cpp` file:

  ```
  int Add(int in_iNumber1, int in_iNumber2);
  ```

- class definitions:

```
class Foo {
private:
  // Static class variable
  // Needs to be allocated in corresponding .cpp file
  static int s_iClassStatic;
  int m_iMe;

public:
  // Inline function declared after class definition
  inline int Add(const int in_i);

  // Function defined in .cpp file
  int MyFunc();

  // Inline functions defined here
  Foo() { m_iMe = s_iClassStatic; }
  ~Foo() {}
};

// Need Foo:: because we're outside of class definition
// Need inline or we'll end up with multiple copies
// of the function
inline int Foo::Add(const int in_i) {
     m_iMe += in_i;
}
```

- Global variable definitions:

```
int g_intGlobal;
extern int errno;
```

- Other header files to include:

```
// Look in current directory, the include path
#include "MoreStuff.h"

// Look in include path for this file
#include <utils/MyStuff.H>
```

2.1.2 Source files

C++ source files usually end in `.cpp` or `.C`, or occasionally in `.CC` or `.cxx`. C source files end in `.c`. Source files can contain the following:

Gotcha 1:

Since the Windows file system is not case sensitive, it's usually a good idea to use the `.cpp` ending rather than the `.C` ending, even under Unix/Linux, since you may end up porting the files.

- Actual implementations for functions declared in header files

```
// Definition of Add
int Add(int in_iNumber1, int in_iNumber2)
{
  return in_iNumber1 + in_iNumber2;
}
```

- Class methods:

```
int Foo::MyFunc()
{
  return m_iMe;
}
```

- Global variables:

```
int g_intGlobal = 2;
```

- Class static variables:

```
// Similar to a global definition, but note the use of Foo::
// to indicate that it is a static variable for a class
int Foo::s_iClassStatic;
```

2.1.3 Object files

Object files usually end in `.o` or `.obj` and have the same name as their corresponding source code files. For example, the file `foo.cpp` would get compiled to the object file `foo.o`. An object file gets created for each source code file. It contains machine code corresponding to the source code in the corresponding `.cpp` file, plus a list of defined and undefined methods, functions, and variables (known as the symbol table). Any method, function, and global

or static variable that is defined in the `.cpp` file goes in the defined list. For example, `Foo::s_iClassStatic`, `Foo::MyFunc`, `Add()` and `g_intGlobal` would go in the defined list. Any method, function, or variable that is used but not defined in the source file goes into the undefined list. For example, if the `Add()` function was defined as:

```
// An externally defined function, not defined in this file
extern int BetterAdd( int in_iN1, int in_iN2 );
// Definition of Add
int Add(int in_iNumber1, int in_iNumber2)
{
  return BetterAdd( iNumber1, iNumber2 );
}
```

then `BetterAdd` would be added to the undefined list. This this means that `BetterAdd` needs to be implemented in some other object file or library.

2.1.4 Library files

Library files end in `.lib` (static, Windows), `.a` (static, Unix), `.dll` (dynamic, Windows), or `.so` (dynamic, Unix). A library is essentially one or more object files grouped together, where any defined/undefined functions or methods between pairs of object files are resolved if possible. There may still be external, unresolved names. The library groups all of the defined and undefined names into one master list. Libraries are either static or dynamic. Static libraries are linked into the executable at compile time, dynamic libraries are linked at run-time. A dynamic library is linked to the executable a run time (by a program called the linker). Dynamic libraries are generally considered to be better than static ones (for reasons we won't get into here).

2.1.5 Executables

Executable files under Windows usually end in `.exe`. Unix executables usually don't have an extension, but their file type is "executable". You can check the type of a file in Unix by running `ls -l`. See the manual page for `ls` for more details. An executable is the final program, which can be completely static, *i.e.*, stand-alone, or it may require some run-time libraries. If the executable is static that means that all of the undefined methods/functions have been resolved (usually by inclusion of some number of libraries) and the actual source code resides in the executable file. This creates a large executable, but it will run without additional libraries. Run-time executables require that the source code for the libraries that didn't get linked in at compile time be in the system somewhere. An executable also needs exactly one `main` function defined.

Example 1:

An example of compiling an executable that uses one class and a C-style function. All of the files are located in the same directory. The first class is called `Point` and has the following header and source file:

```
// Point.H
class Point {
private:
  double m_dX, m_dY;
public:
  // inline functions
  double X() const { return m_dX; }
  double Y() const { return m_dY; }

  // constructor and destructor, must be
  // defined in .cpp file
  Point( const double in_dX, const double in_dY );
  ~Point();
};
```

The corresponding `.cpp` file:

```
//Point.cpp
// Point.H must be in the same directory as Point.cpp
#include "Point.H"
// Define the constructor
Point::Point( const double in_dX, const double in_dY )
{
  m_dX = in_dX;
  m_dY = in_dY;
}

// Define the destructor
Point::~Point()
{
}
```

The other header and source file. These files define a function.

```
// MyFunction.H
// We don't strictly have to include Point.H
// because we don't use any information about
// Point except that it's a class. However, it
// simplifies the life of anyone using
```

```
// MyFunction.H since they don't have
// to remember to include Point.H later.
//
// If we didn't include Point.H we would still
// have to let the compiler know about Point
// by adding the line:
// class Point;
//
#include "Point.H"

// Defines argument and return type for function
extern void MyFunction( const Point &in_pt1, const Point &in_pt2
);
```

The corresponding `.cpp` file:

```
// MyFunction.cpp
// We need Point defined. If MyFunction.H
// includes Point.H then including MyFunction.H
// will suffice. If MyFunction.H just declares
// that Point is a class, then we'd have to
// include Point.H here as well.
#include "MyFunction.H"
void MyFunction( const Point &in_pt1, const Point &in_pt2 )
{
  // do something with points
  Point ptNew( in_pt1.X(), in_pt2.Y() );
}
```

And a main file:

```
// Main.cpp
// Again, we need MyFunction.H to include
// both Point.H and the definition of MyFunction.

// MyFunction.H has to be in the same
// directory as main.cpp
#include "MyFunction.H"

// The required declaration for main
int main( int argc, char ** argv )
{
  // Make two points
```

```
    Point p1(3.0, 4.0), p2(2.0, 1.0);

    // Call my Function
    MyFunction( p1, p2 );

    // No error
    return 0;
}
```

Let's say you've decided to put `MyFunction.cpp` and `Point.cpp` into a library. To create your executable you need to do the following:

1. Compile `MyFunction.cpp` to produce `MyFunction.o`. In `MyFunction.o` is a defined label for `MyFunction`, and an undefined label for both `Point::Point(const double,`
   `const double)` and `Point:: Point`. The `Point::X` function doesn't need an undefined label because it's an inline function, so the code for it is expanded directly into the code for `MyFunction`, which the compiler can get from the header file alone.

   c++ -c -g MyFunction.cpp

   Note that MyFunction.H must either include `Point.H` or have the following forward declaration before the `MyFunction` declaration:

   Class Point;

   If you take the second route, then any `.cpp` file that includes `MyFunction.H` and also uses or declares any Point objects must also include `Point.H` themselves. It's ok to have the actual code for `Point::Point` defined somewhere else, but any code segment that declares a Point or uses any of it's methods must have the `Point.H` file included.

2. Compile `Point.cpp` to produce `Point.o`. In `Point.o` there is a defined label for `Point::Point` and `Point::~Point`. There are no undefined labels because `Point` does not depend on anything.

   c++ point.cpp

3. Link `MyFunction.o` and `Point.o` to produce `point.lib`. `point.lib` will have no undefined labels, because the undefined labels in `MyFunction.o` will be found in `Point.o`. It will have defined labels for `MyFunction`, `Point::Point`, and `Point::~Point`.

   c++ Point.o MyFunction.o -o Point.lib

4. Compile `main.cpp` to produce `main.o`. `main.o` has a declared label for the `main` function, and undeclared labels for `MyFunction` and `Point::Point`, `Point::~Point`.

c++ -c -g main.cpp

5. Link `Point.lib` and `main.o` to produce an executable. At this point the undefined labels in `main.o` will be resolved by the labels in `point.lib`.

   CC -o MyProgram main.o -L. -lPoint

Chapter 3

The Compiler

When you type c++ (or whatever your compiler command is) or hit the compile button in Visual Studio, the compiler is invoked and given the names of the source files on the command line. The first thing the compiler does is pass the files to the pre-processor. The resulting source code is then sent to the compiler. The compiler produces machine code and a list of known and unknown variable and function names.

In this chapter, we give some of the common compile options the Visual C++ compiler and for g++, discuss debug versus optimized mode, what a compiler produces, and ways to fix some common compiler errors. Most compiler errors can be fixed by understanding what the compiler is doing, so we'll start with that.

3.0.6 Compiler output

The compiler produces two things from your source code; a list of names (and their corresponding types and sizes) and machine-code. The names can broadly be categorized into data (classes, structs, variables) and functions (class methods, functions).

For simple variables (*e.g.*, a double) the compiler just needs to know where the variable lives (on the stack or heap) and its size. For more complicated variables, such as classes, the variable has a size (all of the member variable sizes added together, plus space for a virtual table if needed) and a list of offsets that say where to find member variables and functions. For class methods and functions the compiler keeps track of the data types of the inputs and the outputs. This information is usually included in the name the compiler constructs for the function. It takes the name you gave it and adds versions of the input types to it to create one long, unique string. Different compilers do this in different ways, which is why you often can't link object code from different compilers, or use a debugger from a different suite of tools. This is known as name mangling.

The compiler needs to know the size of data objects for all variable names when it creates machine code. It does not need to know this information if it's just type-checking and not generating code. Which is why this is OK:

```
class Point;
```

```
extern void MyFunction( const Point & );
```

even this is (sometimes, depends on the compiler) OK:

```
class Point;
inline void MyFunction( const Point &in_p )
{
  double d1 = 0, d2 = 0;
  double d3 = d1 + d2;
}
```

But this is not:

```
class Point;
inline void MyFunction( const Point &in_p )
{
  double d1 = in_p[0];
}
```

and neither is this:

```
class Point;
extern void MyFunction( Point in_p );
```

In the first case the compiler just needs to know that `MyFunction` takes a reference to the type `Point`, so it can perform type-checking when it finds an occurrence of `MyFunction`. In the second case `Point` is a class, which means `const Point &` is a reference to a class, which is a known size. Note that the size of the *reference* is known, but the size of the object that it refers to is not. Therefore the compiler can tell how much space to allocate on the stack, even though it has no idea of the size of the thing that `in_p` points to. In the third case `in_p` is actually used, so the compiler needs to know about it's size and the `operator[]` method. In the last case the compiler can do type-checking OK, but it can't determine how much space to allocate on the stack when calling `MyFunction`.

Every name (data or function) the compiler encounters is put into the list. In addition to the above information, the compiler also looks for the definition of the name. Here it is useful to distinguish between a declaration of a name and a definition of a name:

```
extern void MyFunction();

class MyClass {
  private:
    // declaration of static variable in class
    double d_sClassStatic;
  public:
```

```
    MyClass(); // declaration of MyClass constructor
   ~MyClass(); // declaration of MyClass destructor
};

extern double g_dGlobal;
```

The above are all declarations. They are usually put in header files but can also be located in source files.

```
void MyFunction()
{
  double d = 3;
}

double MyClass::d_sClassStatic = 3.0;
MyClass::MyClass()
{
  double d = d_sClassStatic;
}

MyClass::~MyClass()
{
}

double g_dGlobal = 3.0;
```

Are all examples of definitions. They are usually found in source files, but can occasionally be found in header files.

Gotcha 2:

A common mistake is to forget to put the extern or inline tag in front of a function, method, or global variable declaration in a header file. This turns the code in the header file into a definition. This sometimes generates a compiler error, where the compiler complains about a function being already defined. For variables and inline methods this usually generates a linker error, which complains about the method or variable being multiply defined. (Which it is, since every source file that included the header file made a definition for that name.)

In debug mode all of the names (and their de-mangled name) are put into a symbol table for use by the debugger. For both debug and optimized versions both of the latter two names are exported for the linker to use.

Variable and function names can fall into three categories:

Declared, defined and no exported These bindings are done at compile time, and the names of the variables and functions do not show up in the link table, they will be in the debug symbol table for use with the debugger. Examples of these are functions and variables that are static (this does not include static member functions), for example

```
static void MyStaticFunction() {}
static int s_iInt = 2;
```

and local variables (MyFunction is exported, but `bar` is not).

```
void MyFunction( int in_i )
{
  int bar = in_i;
}
```

(Possibly) declared in a header file, defined and exported . Local bindings are done at compile time, and the variables and function names put into the link table. Some of the methods and variable names may have been declared in a header file. Examples are:

```
// Non-static functions
void MyFunction() {}

// Class methods
MyClass::MyClass() {}

// Global variables
int g_bar = 3;

// Static member variables
int MyClass::s_iInt = 2;
```

Declared in the header or source file, but never defined These variables and functions must be included from some other object file or library.

```
// This will show up in the undefined list
extern void MyExternFunc();

void MyFunc()
{
```

```
    MyExternFunc();
}
```

3.0.7 Compile modes: debug, optimized/release, profiling

When source code is compiled with the debugging flags on more information is put into the symbol table. This lets debuggers know the correspondence between the machine code and source code, and between addresses and variables. Many programmers take advantage of `assert()` and error printing code which is complied when the compiler is in debug mode, but not in optimized mode. This produces a program which is easy to debug, but when compiled with the optimize flags on is much faster.

One other difference between debug code and optimized code is the machine code itself. When debug machine code is created its flow of operations is nearly identical to the source code. Optimizing compilers can, and usually do, eliminate variables, re-arrange statement execution orders, and unwind loops. This means that the flow of execution through the machine code often bears little resemblance to the original source code. There is a third

---

Gotcha 3:

> Never try to look at optimized code in the debugger. Most debuggers will have a really hard time associating the generated code with the lines in the source.

---

way to compile your code; with profiling on. Profiling adds a bit to every function which keeps track of how many times the function is called and total execution time within that function. When you run a profiled program it spits out a file with these numbers in it. This can be invaluable for figuring out which routine is using the most resources. However, since this document deals with code that usually isn't optimized, we won't go into details of profiling code here.

Switching between modes is accomplished by changing compile flags in Unix, and using the build->settings menu option in Visual Studio (which just creates a different set of flags to be passed to the compiler and linker).

The compiler uses the `-D` define command to indicate which of the builds it is doing. The `-DDEBUG` command is equivalent to putting

```
#define DEBUG
```

at the top of each source file. Debug mode has `DEBUG` (or sometimes `DEBUG_` defined, optimized mode has `NDEBUG` or `NDEBUG_` defined. The latter causes all `assert()` macros to become empty statements. In Unix and Linux systems, debugging is turned on if `NDEBUG` is not defined (i.e. it is on by default).

3.0.8 Compiler flags

The behavior or Unix and Linux compilers is controlled through the flags that are passed to the compiler. You should be aware that different compilers will have different flags. Always check the compiler documentation before applying any flags to your compile. You will typically use only a small number of compiler flags when writing code.

As an example of the sorts of flags that you might use, here are some for `gcc`. Note that `gcc` (the GNU C compiler) and `g++` (the GNU C++ compiler) are actually the same compiler, with different front-ends. As a result, they share a lot of flags. For Visual Studio these flags are set under the project->settings dialog.

`-I` This specifies an additional include path for the compiler. When it comes to an `#include <foo.h>` directive, the compiler looks for the file `foo.h` in a number of standard directories (such as `/usr/include`). By giving the compiler a `-I` flag, you can tell it to look in another directory *before* looking in the standard ones. For example, `-I.` will look in the current directory first, and `-Iblah` will look in `./blah`. You need a separate `-I` for *each* additional directory. The order in which they are searched is the order in which they appear on the compile line.

Visual Studio has two include path mechanisms, one per project and one for the entire program (irrespective of the workspace). The first is found on the C++ - preprocessor tab, the second is under tools->options directories.

`-L` Similar to the `-I` flag, this lets you change where the compiler looks for things. This time, it's libraries. The linker looks in a number of standard places for the libraries that you need (such as `/usr/lib`). By passing `-L` flags, you can get it to look somewhere else first. For example, `-Lfoo` will look in `./foo` first, before the system library directories.

Be aware that, if you are using shared libraries, your `LD_LIBRARY_PATH` environment variable will also have to include the directories from which you're using libraries. If it doesn't, the loader will not be able to find them, and your code will not run, since this information is not stored in the executable.

Visual Studio has to path mechanisms, one per executable project and one for the entire program. The first is found on the link - input tab, the second is under tools->options directories.

`-l` This specifies which libraries the compiler should use when assembling your code into an executable. All of the core functions that you will use are contained in standard libraries (`libc` for C and `libstdc++` for C++) that get included every time, without you knowing about. However, other if you use other functions, then you will probably have to include other libraries. A common example is math functions, defined in `math.h`. It's not enough to have `#include <math.h>` in your code. You also have to tell the compiler what library you want to use to supply the implementation for these declarations. You can do this with the `-l` flag. When you put `-lblah` on the compile-line, the compiler first looks for a library called `libblah.so` in the directories

specified by any preceding `-L` flags, then in the standard system directories. If it fails to find it, the compiler then looks for `libblah.a` in the same places. DIFFERENCE BETWEEN LIBRARY TYPES?

In Visual Studio the libraries are specified in the link tab.

`-g` Generate debug information. In Visual Studio a debug and an optimized (release) version are automatically made when you create the project.

`-c` compile but don't try to link (no `main()` function). This is used when you only want to create object files.

`-o` specify output executable name (defaults to a.out). This is set in the link tab in Visual Studio.

`-O` Optimize the generated code. Different levels of optimization are available (`-O1`, `-O2`, and `-O3` are the commonly used ones). In Visual Studio the release version has optimize flags which are available under the C++ ->optimize tab.

`-D` define - most common use is -DNDEBUG. In Visual Studio these are found under the C++ preprocessor tab.

`-W` warnings - more common use if `-Wall` to turn on all warnings. In Visual Studio there are three warning levels, available from the C++ tab.

3.0.9 Common compiler problems

Make the distinction between pre-processor errors, compiler errors, and link errors.

Can't open file ... This is sometimes a typo or a case mis-match, but often it's an include path problem.

Undefined ...blah.. Usually these are caused by a typo or a missing header file. Sometimes this can be caused by having header files in the wrong order. Try putting an include for the missing header file as the first line in your source file - if this fixes the problem, then some header file is using that class before you've explicitly included it.

Sometimes you've locked out the header file; usually encountered when you've made a new header file and forgotten to change the `#ifndef` line at the top.

If you're including code for the first time, or compiling somewhere new, then you may need to add some include paths.

If you're using new code or you can't remember where you defined the object, the easiest way to find it is `fgrep`, or the find-in-files option in Visual Studio (the binoculars with the flag). Look through the .H files for "class foo" or even just "foo".

Local definitions not allowed Usually means you forgot a closing brace.

Type not compatible errors Usually these are obvious. One less obvious one is when you've named a variable in a class with the same as name as a method in that class. Sometimes there is a global function that has the same name as a method or a local function. Remember that global functions can always be specified by `::MyFunction` to distinguish them from a method or class variable `MyClass::MyFunction`.

Can't convert to... these are usually variations of the above type errors when there's some sort of cast that's available but doesn't work. Usually shows up with respect to const - you can't call a non-const method from a const one.

Errors with stream and it's derivatives in Visual Studio There are two ways to include the stream classes, either as a templated STL version (`#include <stream>`) or as a traditional class (`#include <stream.H>`). The two can't be mixed - if any include file has the .H version, all of the others must as well. If you're using the templated version you also need to put the line

```
using namespace std;
```

after the include line.

A variety errors, usually lots of them, at the start of a source file This is often the indication of a missing close bracket in a header file. Remember that the pre-processor takes all of the header files and strings them into one long text stream with the source file. So an error in a header file may not manifest itself until the source file.

In Visual Studio the compiler gets an internal compiler error If you are using `sin()` and `cos()`, there's a bug in VC5.0 that causes the compiler to die if you're passing anything but a double variable to `sin()` or `cos()`. The fix for this is to pull the parameters out and assign them to a variable:

```
const double dAngle = 3.0 * dFoo...;
const double dRes = cos( dAngle );
```

Multiply defined ... errors Most common cause is forgetting an extern or an inline on a function, variable, or method in a header file. If it's not immediately obvious, try doing a search through the files for the offending name.

An error in a template function/method that has compiled fine before Templates are compiled only as-needed, and act a lot like a macro. So if a templated method is found but not used, the compiler just checks that the parenthesis line up (syntactic check), more or less. It's not until the method is called (and the template parameters resolved) that the code is actually semantically checked (e.g., type-checking). And the semantics are dependent upon the template parameters, so compiling with a new type may produce errors because the passed-in class doesn't have the necessary methods.

Chapter 4

The C Preprocessor

The pre-processor is part of the compiler and is run before any code gets generated. It's main job is to expand out anything that starts with a `#`, i.e., `#include` and `#define`, and to strip out comments. The output is a text stream that is sent to the compiler.

There are three common compile problems related to the pre-processors. The first problem has to do with header files and how to make sure they're included exactly once. The second relates to the life-span of `#define` macros and how to make sure they expand correctly. The last problem is peculiar to Visual Studio, and involves pre-compiled headers and the `StdAfx.H` file.

4.0.10 Header file inclusion

In the example given at the beginning of this section the `main` program includes `MyFunction.H` and `Point.H`. `MyFunction.H` in turn includes `Point.H` as well. How do you make sure that you only get one copy of `Point.H`? This is the common solution:

```
#ifndef _POINT_H_DEFS
#define _POINT_H_DEFS

//... previous point.H declaration

#endif
```

The first time the preprocessor encounters `Point.H` the name `_POINT_H_DEFS` will be undefined, so it will continue processing the remainder of the file. As it's processing the file it will add `_POINT_H_DEFS` to its list of defined labels. So the next time it opens up `Point.H` it will skip the entire file. The trick here is to make sure that `_POINT_H_DEFS` is unique. If you create a class using Visual Studio's add class mechanism, it will generate an id that is based on the file name and a long, randomly generated number. If you're doing this by hand, don't chose something like `POINT`!

Gotcha 4:

A common gotcha is to copy a header file and edit it to produce a new class, without editing the `#ifndef` line.

Note 1: define

The biggest problem with `#define` is that is has no real scope and no sense of type. Once the pre-processor encounters the above definition (unless there's an `#undef`) *every* occurrence of `SIMPLEDEFINE1` gets replaced. Which means only one header file can define the name `SIMPLEDEFINE1`. Which is not a problem until one piece of code tries to use `#define FALSE 1` and another piece of code wants to use `#define FALSE 10`.

There's also no type checking with `#define`. In the example above, the compiler will complain because it can't add 13.0 to "hello", but it won't say that you've passed inappropriate values to `MACRO`.

4.0.11 #define macros

The use of `#define` has decreased greatly in C++ because there are many other preferred methods for performing the same tasks. The typical `#define` declaration looks like:

```
#define SIMPLEDEFINE1 13.50
#define SIMPLEDEFINE2 "Hello"
#define MACRO1(INPUT1,INPUT2) ((INPUT1)+(INPUT2))
```

The pre-processor simply takes every occurrence that matches what's on the left and replaces it with what's on the right. So the compiler sees

```
double a = (13.50)+("hello");
```

in the input text stream instead of

```
double a = MACRO1( SIMPLEDEFINE1, SIMPLEDEFINE2 );
```

Here are some common uses of `#define` in C and corresponding C++ approaches.

- Definition of a set of constants which together represent the state of a variable. Using a typedef codeenum class encapsulates the terms in all caps, so you can have multiple `NEGATIVE` labels in multiple enumerators without any conflict.

```
#define NEGATIVE -1
#define POSITIVE 1
#define ZERO 0

int iState = NEGATIVE;
```

versus

```
typedef enum {
  NEGATIVE,
  POSITIVE,
  ZERO
} IntegerState;

IntegerState iState = IntegerState::NEGATIVE;
```

- Definition of a number that is constant within a given compile but may be changed in different applications using the same code. Using a `const` variable doesn't slow down the execution, but it does allow for type checking.

```
#define MAX_NUM_ITERATIONS 10
#define PI 3.1415
```

versus

```
const int g_iMaxNumInterations = 10;
const double g_dPi = acos(-1.0);
```

- Macros. The advantage of macros is they're expanded at compile time (so they don't incur run-time cost) but you can encapsulate common actions.

```
#define MULTIPLYFOURNUMBERS(a,b,c,d) ((a)*(b)*(c)*(d))
```

versus

```
inline double MultiplyFourNumbers( const double in_d1,
                                   const double in_d2,
                                   const double in_d3,
                                   const double in_d4 )
{
```

```
  return in_d1 * in_d2 * in_d3 * in_d4;
}

// Works for any type that has the add operator
template<class CoordType>
inline CoordType MultiplyFourItems(
                    const CoordType in_d1,
                    const CoordType in_d2,
                    const CoordType in_d3,
                    const CoordType in_d4 )
{
  return in_d1 * in_d2 * in_d3 * in_d4;
}
```

Gotcha 5:

There are two reasons to use inline functions instead of macros; type-checking and correct expansion. You may be wondering about all of the parenthesis in the macro definition. If you don't have them, this can happen:

```
double dRes = MULTIPLYFOURNUMBERS( 3+2, 5+6, 10+2,
MULTIPLYFOURNUMBERS(1,2,3,4));

// Which is expanded to this with parenthesis:
double dRes = ( (3+2)*(5+6)*(10+2)*( (1)*(2)*(3)*(4) );

// But is expanded to this without them:
double dRes = 3+2*5+6*10+2*1*2*3*4;
```

And it only gets worse with macros that expand pointer members.

4.0.12 Pre-compiled headers in Visual Studio

Visual studio can use the pre-processor (and the compiler) to pre-process the header files for a project and store them in a file. This can greatly speed up compile time for projects that include a lot of header files. This is accomplished through the use of a header and source file, usually called `StdAfx.H` and `StdAfx.cpp`. The `StdAfx.cpp` file just includes the `StdAfx.H` file. The `StdAfx.H` file has all of the header files the project uses. At the start of every source file in the project is

```
#include "StdAfx.H"
```

Gotcha 6: StdAfx

Whenever you make a new source file for a project that's using pre-compiled headers, make sure you add this include as the first line in the file or you'll get an end of file found without finding a pre-compiled header error.

You can also use the project->settings->C++ pre-compiled dialog to say a particular file is not using the pre-compiled header. It's usually a better idea just to add the include in.

Chapter 5

The Linker and the Loader

The linker is the last step in the compile chain. You should never get to the point of linking your program until everything has been successfully compiled. The linker's job is to match all of the undefined names with defined ones. Some linkers insist that all of the declared names have definitions, others only insist that the undefined names be matched up.

Gotcha 7:

> Sometimes you could swear you made a particular change (usually in a library), but when you run the program the change isn't there. (One reason to always walk through new changes in the debugger at least once.) This can happen if the dependency tree is out of date or if you grab the wrong library. There are a couple ways this can happen:
>
> - You forgot to add a header file to the dependency list. In this case the file won't be compiled, so it's a quick check to look back at the makefile output and see if this is the case (or check the date on the object file).
> - You made a change to a library, but the program's Makefile doesn't have a dependency on that library, so the program didn't re-link.
> - The name you used for the library output (or `.o` files) is not the same as the name you used for the program's link line (or library build line).
> - Sometimes time-stamps go wrong, and it's worth doing a clean build.

Visual Studio has problems with `MFCLib`. Visual Studio has several different types of the same basic libraries. They are differentiated on whether the code was compiled static/dynamic, single thread/multiple thread, and using MFC/not using MFC. The for-

mer flags are set under project->settings->C++ code generation, the latter under project->settings->general. All of the object files and all of the projects must have the same settings.

5.0.13 Fixing link problems

Linker errors tend to be caused by missing variables or functions, missing libraries, or libraries listed in the wrong order in the compile command. Linkers can be one-pass or two-pass. In a one-pass linker the libraries are read in the order they appear on the command line. It only keeps a list of the undefined names, and matches them up. So if a name is defined before there's an undefined reference to that name, the linker won't link them. This is why the most general libraries should be last in the list.

Example 2:

> Suppose that we have a library, `foo`, that depends on some functions in the math library. A compile command like this
>
> g++ bar.c -lm -lfoo
>
> will fail, because the linker will not look at the math library (`-lm`) again after it looks at `foo` and and realizes that there are unresolved symbols. The correct version of the compile command is
>
> g++ bar.c -lfoo -lm
>
> Why aren't all linkers two-pass? No idea.

A two-pass compiler will make a second pass through with its undefined list.

Missing libraries usually produce a lot of undefined names. If the library is designed properly the names should give an indication of the name of the needed library. For instance, all of the OpenGL Library calls are prefixed with `gl`. If the library name is not immediately obvious, try an `fgrep` or file search for that name in suspected source files. You can also use `nm` to list the names of the symbols in a library (pipe it to `grep`). Windows, unfortunately, doesn't have `nm`, but it's search in files will work for non-text files.

If you've determined where that library is and are sure it's included on the command line, try moving it to the end of the library list and see if it fixes the undefined errors (it may generate a slew of new ones, but at least you know you have the right library, just in the wrong place in the list). A really drastic fix is to list all libraries twice in the compile command (which essentially forces one-pass compilers to do passes). Doing this is OK to track down problems, but you should work out the actual correct order for the libraries.

Single undefined names can be harder to find. Sometimes they simply aren't defined, in which case you need to go back and add that method. Remember that global variables and static class variables need to have exactly one source file where they're defined.

Multiply defined names are usually an indication of a missing inline in a header file. Sometimes it's because a library is included more than once.

Remember that adding a header file for a library is not sufficient — you need to add the library to the link line.

Gotcha 8:

> A function with the same name and the same parameters will not have the same name when compiled with the C compiler and with the C++ compiler. If you are linking to a library built with a C compiler, you must tell the compiler that the function is named in C mode, not C++: `extern "C" void MyCFunction();`

Chapter 6

`make` and Makefiles

This is a breakdown of the more important features of the `make` command and Makefiles. Makefiles are recipes for building your program, and can range from simple and straightforward to bizarrely complex. The most basic Makefile is simply a list of source files, the header files they depend on (i.e., include) and any system libraries. The Makefile lists commands to compile the source files into object files, and a command to link them all together with the required libraries. Only the files with dependencies that have changed get re-compiled.

This chapter is a *very* brief introduction to `make`. It can do many, many more things that we touch on here. You should seriously consider learning more about `make` and Makefiles, since they will save you a *lot* of time in the long run.

The following description is somewhat specific to GNU `make`. Be sure that this is the version of make that you are using, since other versions can have different functionality. We also recommend asking someone who's been coding for a long time for their Makefile. Good Makefiles are like currency, and once you find one that you like, you'll keep it for years.

Some common tasks performed using `make` include (ranging from simple to more complex):

- Use the compiler to automatically generate a list of header file dependencies for each source code file. The list of dependencies is usually stored in a .depend file in the directory, and is either automatically updated every time or when the user says to. This is much less error prone then writing the dependency list by hand.

- Compile the object files to someplace other than the current directory.

- Debug versus optimized/release versus profiling mode. This lets you say `make debug` or `make optimized`, and it will use the same dependency list, but different compile options and usually will put the output in a different place or use a DB versus R extension to distinguish them.

- Creating a library from your source code. This will compile all of your source code, package it up into one or more libraries, and install these and the required header files into the appropriate places.

- For large projects it gets very tedious to make new Makefile for each library and program. In this case the majority of the Makefile (what compiler options to use, include paths, library paths, what linker to use, where to compile to, etc.) is pulled out into a "meta" or "stub" Makefile. The makefiles in the individual directories all include this meta Makefile plus a list of the object files and library name for this directory.

6.0.14 Makefiles in Visual Studio

Underneath all of Visual Studio's visual interface is a Makefile (of sorts). In fact, you can ask Visual Studio to generate a Makefile (the format is similar, although not exactly the same, as Unix style Makefiles). Usually, though, most of the Makefile information is stored in the `.dsp` (project) and `.dsw` (workspace) files.

Project files: There is one of these for each of the projects listed in the Workspace window. The project file lists the header and source code files for the project, where to put the object and library or executable files, and compile and link options. The compile and link options are dependent upon the type of project — when you ask for a new project, it gives you this list to chose from. Some of the items are libraries (either static or dynamic), some are executables (either command prompt or a full windows program). In addition, you can specify what type of build (debug or optimized/release) you want to do.

Workspace files: These combine several projects together and specify dependencies between the projects. Most common usage is to have a single executable project and zero or more library projects in the workspace.

It's possible to edit project and workspace files in a text editor, but this is usually a bad idea. The proper way to edit files is through the project->settings dialog.

Gotcha 9:

> There's one main gotcha with the project->settings dialog. If you don't change or touch an individual file's settings then it will inherit whatever changes you make to the project as a whole (for example, changing the output directory). The same is true for the library/executable output name and location. There is also a slot for the name of the executable to run when debugging (project->settings->debug). Usually this is the same as the executable name (project->settings->link) but if you edit the debug slot then they may get out of sync. If all else fails, delete the item in the slot and let Visual Studio fill it in for you.

Here are some common operations and how to do them.

- Adding/deleting files from a project. This is done through the workspace window (or project->add files). I usually add files by right-clicking on the project and picking add files. If you do project->add files it will default to adding files in the selected directory. The files can be anywhere - they don't have to be in the current directory. You can also add both a `.cpp` and a `.h` file at the same time by choosing class view (instead of file view), right clicking on the project and saying "add class". Deleting files is just a matter of selecting the file(s) and clicking on the scissors.

  It's always a good idea to add any header files as well as source files, since Visual Studio will save all of those files for you before compiling. It won't save any files that are not in the dependency list.

- Libraries for executables. There are two ways to add libraries to an executable. The first is to type the library name into the project->settings->link dialog. The second is to add that project to the workspace and a dependency saying the executable depends on that project (project->dependencies). You can (either accidentally or intentionally) add the library both ways.

  There are advantages and disadvantages to including the project in the workspace. The advantage is that the library will be re-compiled if you change it, and you have access to all of the files and classes. It can get a bit tedious adding all of the necessary projects, however.

- Pre-processor directives (-Ddefine and -Ipath in Unix). Use project->settings->C/C++ and pick the Preprocessor under Category. The first line is a list of preprocessor definitions. The second is additional include directories. The entire list of compiler options shows up in the window at the bottom - you can edit this by hand although you really shouldn't have to under normal circumstances.

- Object output location: project->settings->general, make sure the project you're interested in is selected on the left. There's two output directories, one for intermediate files and one for output files. See above note on project->settings change order.

- Executable and Library output location and name: project->settings->library or link. Select the project of interest on the left. See above note on project->settings change order.

6.0.15 Simple Makefile example

Here is a simple makefile for the example given earlier.

Example 3: makefile

```
CC = g++
CPPFLAGS = -g +w

MyProgram: main.o Point.o MyFunction.o
        ${CC} -o MyProgram main.o MyFunction.o Point.o

main.o: main.cpp MyFunction.H Point.H
        ${CC} ${CCFLAGS} -c main.cpp

MyFunction.o: MyFunction.cpp MyFunction.H Point.H
        ${CC} ${CCFLAGS} -c MyFunction.cpp

Point.o: Point.cpp Point.H
        ${CC} ${CCFLAGS} -c Point.cpp
```

The first line says use `g++` for the compiler. The second line provides the compile flags for `g++`. To learn about the possible flags available see the `g++` man page using `man g++`. Some useful flags are `-g` to add necessary information to be used by the debugger, `+w` to have the compiler give extra warning messages about questionable constructs, `-o filename` to provide a filename for the output (versus the default of `a.out`), and `-c` to indicate that no linking should be performed (and thus this is used to generate the object files).

The remaining pairs of lines each have the form:

```
target : prerequisites
        the command that generates the target
```

where the prerequisites are all the files that are used in generating the target. Note that it is essential that the second line begins with a tab.

In our example Makefile above, the first pair of lines says that to create the executable `MyProgram` it will use `main.o`, `Point.o` and `MyFunction.o`. Thus if either of these files have been updated more recently than `MyFunction` then the command

```
${CCC} ${CCFLAGS} -o MyProgram main.o MyFunction.o Point.o
```

should be run. This command links the given object files to create the executable `MyProgram`. Then to run the program you can type `MyProgram` in your Unix shell. The following three pairs of lines in our example Makefile are the commands and dependencies for generating the object files. For example, if `Point.H`, `main.cpp`, or `MyFunction.H` change then `main.o` needs to be rebuilt.

```
${CCC} ${CCFLAGS} -c main.cpp
```

Finally, to use the Makefile simply type `make MyProgram`. As a default if you just type `make` it will make the first target given.

Part II

Memory

Chapter 7

Memory

We assume that the reader has a basic understanding of what memory is (an indexed array of boxes of a set size) and the difference between physical and virtual memory. Here we discuss how the memory for a program is allocated and freed as it is executing. We break the memory allocation description into explicit allocation (new and delete on the heap) and implicitly (local variables on the stack).

It is important to remember that memory ownership is a convention, not a rigidly enforced law. As an analogy, consider the lines painted on the street that divide traffic flows. Only convention prevents you from driving over the line in the middle and going the "wrong way". Even if you are driving on the wrong side of the street it isn't a problem until a truck comes the other way and insists upon being in the same lane as you... Memory problems are often of this nature; they only show up when two routines finally access the same piece of memory. The original transgression may have occurred much earlier.

We first discuss the stack and the heap, what they are and when they get used. We then describe some of the more common memory problems, how to recognize them, and how to prevent them. This is followed by a section on mechanisms in Visual Studio (and other memory management libraries) that can be used to track memory leaks. (For most class-sized projects these systems are over-kill, but they are invaluable in large programs.) Finally, we close with comments on memory overhead, both time and space, and when it's worth your time to optimize.

7.0.16 Heap and stack

The memory allocated for your program is broken into four segments, the text segment (executable code), the data segment (globals and statics), the heap (`new` and `malloc`), and the stack (local variables, stack frame). The text segment size is determined at link time, and consists of the code in your object files plus any static libraries. The data segment is also determined at link time, and consists of global and static variables. They are separated to reduce the chance of overwriting code when changing the global and static variables. The heap contains dynamically allocated data. The stack contains local variables and data for function calls. The stack grows and shrinks dynamically at run-time, but the *size* it grows

by when you make a function call is determined at compile time. You are not allowed to write to the text segment; if you do, you will get a segmentation fault (invalid address) error.

7.0.17 The heap

The heap got its name because memory allocate on the heap is jumbled together, as if you dumped a bag of memory blocks on the floor. Anytime you do a `new` or `malloc` the space for that data comes from the heap. Each memory allocation marks out a section of memory. That section of memory consists of a header plus a piece of memory at least the size of what you asked for (it may pad out the end to make sure it ends on a word boundary). The header says how big the following piece of memory is and also contains a pointer to the top of the heap. The heap also keeps a pointer to the start of every one of these memory blocks.

When you subsequently free a piece of memory, the heap marks that memory block as free in its own list, but does nothing other than that. So that piece of memory will still look valid, at least until the heap recycles it for use in another `new`/`malloc` call.

Heap fragmentation: A memory block can be recycled in one of two ways; it can be re-used when the program requests a block of identical size, or it can be combined with any adjacent blocks to create a larger block. This only happens if there are adjacent free blocks. If there is no re-usable block of a big enough size, the new block is added at the top of the heap. After enough multiple, random-sized news and deletes the heap begins to look like swiss cheese. This is really only a problem with programs that run a long time (several hours to days) and are constantly doing news and deletes. The symptoms look a lot like a memory leak; the longer the program runs, the bigger its size. This problem is best addressed by using memory pools; this is rarely necessary for class-sized projects.

7.0.18 The Stack

The stack is a stack of data for the current procedure or function calls and their local variables. Ever time a function call is made the data for the function, called the stack frame (variables that are passed in, local variable declarations, register values and some extra book-keeping information) is allocated space on the top of the stack. When the function exists the current function the stack pointer is moved back by an equal amount. That memory is now considered "free" as far as the stack is concerned. Any returned value from the function actually lives in this freed memory; normally this would be bad, except the returned data is guaranteed to be copied into its new home before the next function is called.

In the debugger you can walk up and down the stack (called stack crawling), looking at the values of all the local variables. This corresponds to moving a debugging pointer, which knows the size and name of all function stack frames, up and down the stack. The size of the stack frame is known at compile time, so the compiler can give this information to the debugger.

Chapter 8

Memory Problems

8.0.19 Common memory problems

The stack: It's rare to have a memory allocation problem directly related to the stack, but there are several secondary effects that show up, for example, when memory that was freed is referenced.

- Endless/bottomless recursion: The stack can not grow forever. Symptoms of this are a program that runs forever, gets slower and slower, and bigger and bigger. The easiest way to verify this problem is when your program crashes (or you stop it), look at the stack list. If there's hundreds of functions on the stack, you're recursion is not bottoming out. On some systems everything (including the cursor) will freeze as your program eats up all of the available real and virtual memory.

- Local variables suddenly changing values without an explicit access. This can be a symptom of walking off the end of an earlier variable. For example, if you had the following declared:

  double a[3], b;

  and you mistakingly wrote

  a[3] = 7.0;

  The effect of this would be to set `b` to 7.0. This is because local variables are allocated one after the other on the stack. So `a[3]` actually occupies the same memory location as `b`. The order the compiler puts the variables in the stack frame is usually (but not always) the same order as they appear in the source code. An optimizing compiler is more likely to shuffle variables around and to re-use existing variable space, so this is another source of problems that show up only in optimized mode.

  The same effect can be seen with class variables, since they are also allocated as a chunk of contiguous memory.

A note to the experts: Because of word boundaries one variable may not follow immediately after the previous one, but appear on the next word boundary. For example:

```
char c[3];
double d;
```

will result in 3 bytes for `c`, an empty padded byte, then the double d. So setting `c[3]` will not affect `d`, but `c[4]` will.

- Returning the address of a local variable.

```
double *foo()
{
  double a;
  return &a;
}
```

Most compilers issue warnings if you try to do this — listen to them! Remember that after the function exits the local variable space is now considered "free", although the contents of it may not be overwritten for awhile.

Symptoms of this are variables that change after a function call, even if that function does not touch that variable. It can also show up when you switch from debugging to optimized code, since the compiler may change the stack size.

- Illegal operation on function return: Also, arbitrarily ending up in some other piece of code. This happens when you overwrite the slot in the stack frame which points to the location to pop back to. You have to "under run" a variable to do this:

```
double a;

* (&a - sizeof(double *)) = 3.0;
```

- Passing by copy constructor by "accident". This can happen when you accidentally type

```
foo( bar in_bar );
```

instead of

```
foo( bar &in_bar );
```

The symptoms of this are a variable that does not have the correct value coming into a function, or a variable that was suppose to be set in the function but wasn't.

Common heap problems

- Wild pointers: It is perfectly valid syntactically to do the following:

  double ∗s;

  ∗s = 10.0;

  `s` is called a wild pointer because its value may point anywhere. There are three common methods for generating wild pointers. One is forgetting to allocate space or assign a valid memory location to the pointer variable (as above). The second method is to delete the data the pointer points to, without "telling" the pointer by setting it to `NULL`. The third method is to write over the pointer address by accident, as discussed above.

  The effects of a wild pointer can range from subtle to obvious. On the more obvious side is a segmentation fault when you try to write to the pointer. This happens when the pointer points to data in the text segment. Visual Studio (in debug mode) initializes pointers if you don't do so explicitly to be `0x7d7d7d7d` so that any references to them will cause an exception. Slightly less obvious is when you access the pointer and get obvious garbage out.

  It is, unfortunately, perfectly possible for your pointer to point to a (now free) piece of memory that originally contained data from the same type. In this case you may not notice the problem for a long time, and it will usually manifest itself as data in a pointer (not necessarily the wild pointer) suddenly changing values. If a wild pointer is pointing to some place in memory legitimately owned by a valid pointer, one or both of the pointers can exhibit odd behavior.

  Remember that `new` or `malloc` (which may occur with an array class, without an explicit call by you, when you add or subtract an element) can move your memory somewhere else entirely. This is a bad thing to do:

  // Make 10 items of foo
  Array<foo> aoFoo(10);

  // pointer to the third element of foo
  foo ∗myFoo = &aoFoo[3];

  // add an element to foo
  aoFoo += foo(. . .constructor data. . .);

  The pointer myFoo may now point to memory that has been freed because of a reallocation of the memory for aoFoo.

Chapter 9

Memory Do's and Dont's

Here are some good memory management practices. The key here is to apply these practices all the time. It's like brushing your teeth - it's a lot easier to remember to brush your teeth at nights if you brush them *every* night.

- Use an array class (stl, CS342, or other) whenever you're allocating arrays of data. You'll take a small space and time hit, but the amount of time used will be far less than just one debug cycle looking for an off-the-end array error. Array classes are also "smart" in that they perform bounds checks in debug mode but not optimized mode, so the time hit goes away. Most array classes are also somewhat smart about reducing the number of calls to the heap (`new`/`malloc`) by allocating slightly more space then requested, so the next request or object addition will not cause another `malloc` call. If you're doing something that requires a lot of memory, these classes also have a mechanism for shrink-wrapping the amount of memory allocated if need be.

- Even if you aren't using an array class (because you know you only have three items) use accessor functions with bounds checks. And use them everywhere. It does not cost a single cycle more in optimized mode.

```
class foo {
private:
  int foo[3];
public:

inline int GetFoo(const int in_i) const
{
  ASSERT( ( in_i >= 0 ) && ( in_i < 3 ) );
  return foo[in_i];
}
inline int &SetFoo(const int in_i)
{
  ASSERT( ( in_i >= 0 ) && ( in_i < 3 ) );
```

```
    return foo[in_i];
  }
};
```

Even in this case using an array would be better - about the only time you might want to do this approach is if you know you have a fixed size for your data and you'll be making a lot of them and don't want to take the memory and allocation hit.

- Always initialize data. Don't ever write `int *foo`; or even `int foo`. Same thing goes for class constructors - your constructor should set a default value for every variable in the class. It takes 10 seconds to write and can save you hours looking for corrupt data. Remember that if your constructor initializes all of the data then you will reduce the chance of a pointer masquerading as valid when it points to a re-used piece of code.

- Same thing goes for destructors. It may seem odd, but if you have pointers in your class, set them to `NULL` in your destructor. Again, this can reduce data masquerading.

- Beware taking addresses of objects in dynamically allocated arrays and expecting those addresses to stay valid. If that array is resized the pointer will become invalid. Remember that adding or removing objects from array classes can cause a memory reallocation. The same thing can happen with the following:

```
int *foo = malloc(10 * sizeof(int) );
int *myFoo = foo[3];
realloc(foo, 20 * sizeof(int) );

//myFoo may now be invalid
```

- Use `const` when keeping pointers to other classes that you're not responsible for allocating, deleting, or editing.

```
class Foo {
  const Bar *m_opBar;

  void SetBar( const Bar *in_opBar ) { m_opBar = in_opBar; }

  Foo() : m_opBar(NULL) {}
  ~Foo() { m_opBar = NULL; }
};
```

You can now access all of the `const` members in `Bar` and whoever is looking at your code knows that you are not responsible for allocating or deleting the pointer.

- Always declare `operator=` and copy constructors (`constructor( const class & )`). If you aren't planning on having any copy constructors, put them in the private section of the class and don't bother writing bodies for them (or if you do, put an `ASSERT(FALSE)` in the body). This will cause the compiler to throw a warning if you accidentally type `foo( class bar )` instead of `foo( class &bar)` for classes with no copy constructor, or if the compiler sneakily tries to use them itself.

- Let me say this again – use an array class instead of `int *foo = new int[10];`

- Avoid `new`ing explicitly unless you have to. For instance, if you have a class foo with a member variable bar, and bar is used throughout the life-cycle of foo, do

```
class foo {
  bar m_bar;
};
```

not

```
class foo {
  bar *m_opBar;

  foo() { m_opBar = new bar; }
 ~foo() { delete m_opBar; m_opBar = NULL; }
};
```

Sometimes you may feel like you have to new bar because bar needs some construction information that isn't available in `foo`'s constructor. You can usually get around this; see the section on construction .

This also holds true for arrays of data. You're better off doing:

```
Array<foo> aoFoo;

for (int i = 0; i < 10; i++ )
  aoFoo += foo( i, ... other data );
```

and writing a copy constructor for `foo` then doing

```
Array<foo *> aopFoo;
for ( int i = 0; i < 10; i++ )
  aopFoo += new foo( i, ..., other data );
```

since if aopFoo goes out of scope and you haven't deleted the data, you now have a memory leak. You also have a potential wild pointer problem. The only reason you might want to do the latter is if the copy constructor of foo is extremely expensive and

you are doing lots of adds and deletes from the array (on the order of 1000 operations). Or if you need to keep track of the pointers themselves because the pointers in aopFoo will be passed to other methods individually.

Function returns should also avoid news and use the copy constructor if possible:

```
R2Pt Foo(... some data ... )
{ return R2Pt(3.0, 4.0); }
```

not

```
R2Pt *Foo(... some data ... )
{
  R2Pt *pptPtRet = new R2Pt(3.0, 4.0);
  return pptPtRet;
}
```

If you do have to return data that is expensive to copy you can use the pass by reference:

```
void Foo( R2Pt &out_pt )
{
  out_pt = ...; // set out_pt
```

Valid reasons for explicitly `new`ing (instead of using an array class or declaring the variable on the stack):

- The data are large or expensive to copy and you need it to persist for longer than the function call or the calling function.
- You need to let more than one class/function have access to the data. In this case it is important to make it very explicit who is in charge of allocating and deleting data. If a pointer may be allocated or deleted in more than one place, and it's not blazingly obvious how the matched pairs of allocates and deletes work, use reference counters.

Chapter 10

Debugging Memory Problems

10.0.20 Debugging strategies

As always, stepping through your code, one line at a time, and watching the values of variables is the best way to find most errors. These are assuming you've already gone through and initialized all local variables and have default initializations for all your classes. Here are some tricks specific to memory management:

- If you suspect you're running off the end of a variable somewhere, rearrange the declaration order of the local variables (or the class members). If you are overstepping boundaries, a different variable should be affected.
- If you have a pointer that's going bad somewhere and it's only after many calls (so watching it step by step is prohibitive) try adding an `is-ok` variable that you check everywhere you can using `assert()`. Sort of a mini magic id. Put those `assert()` in every part of the code where you can get at the pointer, even if you don't think that piece of code is the problem. Remember that you're trying to find the first time you cross the line in the middle of the road, not when you get hit by the truck.
- If you suspect you're running off the end of an array, switch to an array class. Which you should have been using in the first place. Even if you're not (for some reason) create an accessor function for that variable that has a bounds check in it, and substitute it everywhere, even within the class. If the variable is not a class member but a local variable, write a small inline function:

  void Check(**double** *foo, **int** i) { ASSERT((i) >= 0 && (i) < 3), foo[(i)];
  }

- Chasing down memory leaks. This is a fine art and best approached by matching up allocations and deletions, putting break points and seeing if they get called when expected. `grep`ing (file searching) for new is another good approach. This can help you find a new that you forgot to get rid of, or didn't know you were still using. Reducing your use of new and delete is really the easiest way to prevent memory leaks.

Chapter 11

Advanced Memory Issues

The following are some advanced memory management tricks and tools. There are standard packages out there that over-ride new and delete for classes and implement one or more of the following.

Magic IDs Each class type has it's own "magic" id - a unique number, one for each class. This is stored as an `int` as the first data element of the class. On construction the int is set to the unique number, and at destruction it's set to a not-a-class id. Every access to the pointer then checks to see if the id is correct. (This is the `Valid()` call on most Window MFC classes.) This helps in preventing wild pointers because it's very unlikely that a piece of memory will just happen to have the correct magic id. It still won't prevent two pointers of the same class type thinking they own the same piece of memory.

Reference counting This is a useful technique when you have multiple references to a pointer and you want to know when it's safe to delete it (when no one else still has a pointer to that data). The class has an integer count variable that is the number of classes that are using that pointer. Whenever a class wants to use the data in the pointer it increases the reference counter. When it's done, it decreases the reference counter. The class that allocated the data agrees not to free it until the reference counter goes to zero.

Instrumented memory manager This is like reference counting but for everything that gets allocated and de-allocated on the heap. Visual Studio has this built-in in debug mode (see `AfxMemory`...) and Java does this one better in order to keep track of when things can be freed and garbaged collected. This is essentially an addition to the heap routine which keeps track of matching allocates and deletes and time stamps them. This lets you ask "what's been allocated but not freed since the last time I asked?". This is invaluable for chasing down memory leaks.

Memory pools If you're allocating and deallocating lots of small objects that are all of the same size, a memory pool can reduce the time hit of going to the heap for each

allocation. Essentially, a memory pool is a mini heap itself, that knows all of the objects are the same size. This means it can represent the used/free status of each object as a single bit. The memory pool requests a large chunk of memory from the heap and doles out pieces of it as needed (by over-riding new and delete of those objects).

Memory pools may seem cool and you may think you absolutely have to have one, but think again. Unless you're staring at the screen waiting minutes for your program to run, and it's spending 80 percent of its time doing allocation (which you found out by profiling), don't bother. If this is a piece of code that is doing lots of allocation and deletion and it will be used for several years by several different programs, then it might be worth it. Maybe.

Part III

Style

Chapter 12

Proper Coding

12.1 Writing code

This is a set of guidelines for writing fairly stable, robust code for small to medium sized projects that have changing guidelines. It isn't meant for production-level code, although many of the same principles apply. Our assumptions are:

- The programmer will spend as much (or more) time debugging the program than any user will running it. This means we optimize for debugging time, not run time. It takes a long time for a savings of a second to add up to two hours spent fixing the bug caused by saving that second.
- The correctness of the algorithm has not been proven. So incorrect behaviour may be an implementation error or a bad assumption in the algorithm itself.
- Some parts of the code may be re-used or recycled, but others may be one-time only. This designation may also change over time.

The goal is to spend as much time as possible exploring algorithms and as little time as possible chasing "stupid" bugs and to ensure (as much as possible) that the code is actually computing what you think it's computing. Counter-intuitive as it may seem, writing more code (assertions, debug and test code, class encapsulation) usually results in faster overall development time.

The following is a summary of our suggestions. Above all, remember to *think*, not react.

- Don't ignore compiler warnings.
- Encapsulate data and algorithms as much as possible. Avoid secondary effects — algorithms should do exactly what they say and nothing more (don't bury computations within computations). Don't change data unless that's the purpose of the algorithm. Don't use data that isn't obviously accessible from where you are.
- When writing a new piece of code do one or more of the following:

  - Step through the code, checking values of ALL variables to see if they make sense. Make sure all branches are visited, or at least put a break point in the un-visited ones.
  - Is there a verification routine you can write, which you can then call at any time to check if your data is still valid or your algorithm did what it was suppose to? Make this nice and easy to use so you use it, and also wrap it up in an assert and leave it in, unless it's an extremely expensive test.
  - Write a test routine with known inputs and outputs. Make sure this test case hits all of the "bad" inputs.
  - If you make any assumptions, especially about input, put them in the code as asserts.

- Never assume that you'll remember anything about the code you write — write it for an audience. If you do anything odd or different, or your computations are complicated, put a comment in.
- Avoid pointer operations and new and delete if at all possible. Memory problems caused by pointers going where they shouldn't are horrible to chase down. By the same token, ALWAYS use arrays with index bounds checks.
- Do not "double use" or re-use variables within a routine. Make a new one.
- Use a naming convention to distinguish between member variables, global variables, function variables, and local variables. Prefix your class and function names (or use namespaces) to prevent eventual name conflicts.
- Use the static keyword on any local functions to prevent name conflicts.
- Do not expose the actual data of your classes; write accessor functions.
- Pick a layout scheme and stick to it. White space is free, so use it.

12.2 Structure

12.2.1 Library Structure

This section talks about high to mid-level decisions regarding code structure. See any object-oriented text book on how to break up your code into manageable pieces. The focus of this section is more about when to create a library, what should be in a library, etc.

The material in a library should be fairly focused and related. Dependencies on other libraries should be documented. As much as possible, keep dependencies minimal. To do this, you may need to break up functions that might otherwise be in the same library. For example, suppose you have a curve class. Now suppose you write some curve fitting functions

that require a lot of extra libraries, such as LAPACK and its dependencies. Most applications don't use the fitting routines, so it might make sense to pull the fitting routines out into a separate library. That way you don't force any application that wants to use curves to also include the entire LAPACK dependency tree.

Separate, as much as possible, user interface code from functionality. That way you can easily put your classes into a library which can be used for a multitude of applications (command line, interactive, etc.). Include test routines with the library — and make an easy-to-call test function so that anytime you want you can make sure nothing has been broken in the library.

Sometimes you know in advance that the code you're writing will fit nicely into a library. If this is the case, create the library and go to it. Other times you'll be developing code that evolves into a library. It's fine to do this, just stay away from things that make it hard to extract code into a library:

- Header file dependencies. Keep them clean, and use a header file to keep lists of (grouped) global functions and global variables. Use a pre-fix on class and function names, or namespaces.
- Bundle data and functions whenever possible and it makes sense. That way you're not searching through code trying to find that missing reference.
- Take the time to thoroughly name or comment variables and methods. You'll forget what they are in 2 months.

It's time to pull out code into a library when it's stabilized and you're not making lots of changes. At this point it's usually worth making a pass through, commenting and cleaning up now-unused variables and methods. Anything that's written, but not tested, should be labeled as such.

12.3 Low-level do's and don'ts

The following is a list of low-level coding habits you should get into. Don't skimp.

12.3.1 Avoiding pointers

Allocating memory with `new` and `delete` (or `malloc` and `free`) is one of the leading causes of memory leaks and bad pointers. Pointer manipulation code is also notoriously prone to bugs. While you can't avoid memory allocation, you can allocate memory in less dangerous ways. One way is to use the stack whenever possible. The second is to use an array class for allocating groups of objects. This also has the advantage of avoiding bad indexing errors, provided your array class has a bounds check.

There are two basic classes of arrays — one-time allocation and dynamic. Most dynamic allocation classes employ some clever method to avoid constantly doing reallocations, usually

by allocating slightly more memory than needed. So if you aren't planning on re-sizing the array then a one-time array will be more space efficient. One drawback to array classes is that they use more space. This is greatly outweighed by the reduction of errors and built-in run-time bounds checking on access. (Optimized versions take the bounds check out.)

Example 4:

Here are two methods to return data without doing a new memory allocation. One is for use with small data types, one for large (a class you don't want to copy unless you have to).

```
class SmallFoo {
public:
    // methods needed for copy construction
    SmallFoo &operator=( const SmallFoo & );
    SmallFoo( const SmallFoo &in_foo ) { *this = in_foo; }

    // methods needed for dynamic allocation array classes
    bool operator==( const SmallFoo & ) const;
    SmallFoo();
};

SmallFoo MakeFooFunc()
{
    SmallFoo foo;
      ...
    return foo;
}

void CallFoo()
{
    const SmallFoo foo1( MakeFooFunc() );
    const SmallFoo foo2 = MakeFooFunc();

    // Make a bunch of foo
    Array<Foo> afoo(10);

    ...
    // foo1, foo2, and afoo will all be deleted for you
}

class BigFoo {
```

```
private:
    // Put these up here (and don't define them
    // in the .cpp file) to prevent the compiler
    // from making one for you.
    BigFoo &operator=( const BigFoo & );
    BigFoo( const BigFoo & );
public:
    BigFoo( ... );
};

void MakeFooFunc( BigFoo &out_foo )
{
    .... // construction
}

void CallFoo()
{
    BigFoo foo;
    // check that allocation went ok
    verify( MakeFooFunc( foo ) );

    ...
    // foo will be deleted for you
}
```

Example 5:

Sometimes you need to allocate a class object and pass a pointer to that object into another class object. First, think if you really need to do this. For example, if a and b both need c, and a needs b, then does it make sense to put c into b, and let a get at c through b? You can often use delayed initialization to get around some mutual reference problems. In this case, the default constructor does nothing but set the member variables to "unknown" values. There is a separate initialization call to actually set the values.

The key to making these activities safe is to make it very clear where (and when) objects are allocated and de-allocated. If the new and delete can go in the same routine, then allocate on the stack instead of doing a new and delete pair.

```
void MyFunc()
```

```
{
    Foo myFoo; // allocate myFoo on the stack, not the heap

    Foo *myFooAlloc = new Foo; // allocates on the heap

    ...

    delete myFooAlloc; // Must remember to delete

} // compiler deletes foo on exit
```

12.3.2 Notation and formatting

People get into religious wars over whether the curly bracket should go on the end of the `if` or on the start of the next line. It doesn't really matter, so long as you are self-consistent. The following are some things most people agree on:

- Use white space, both between values and blank lines.
- Split calculations up if they get too long (clever use of const can make equations run faster if they're split up).
- Keep sections of code to one page, if possible. Replace sections of code with function calls if it makes it clearer.
- Name variables, functions, classes, and methods as descriptively as possible. Use either namespaces or prefix your class and function names with a reasonably unique identifier. You probably aren't the only one who's made a class called named "Point". Most editors have some form of tab completion, so long names don't always take a long time to type.

Most big software houses have specific formatting rules for the code their employees write. If you're working with someone else's code, stick with their formatting rules. One example of widely-used coding standards is available at `http://www.gnu.org`.

I'm going to make a brief plug here for a modified version of Hungarian notation. It has two purposes. One, it's a visual guide for determining where variables come from. Two, it's a form of type-checking. The idea is to add a prefix to the variable name that describes where the variable is declared (local, global, class member, etc.) and what type it is (e.g., array of integers). I use a subset of the type prefixes. The standard prefixes are:

- m_
  member variable.

- g_
  global variable.
- s_
  static variable.
- in_
  input into a function or method.
- out_
  a parameter to the function that's set within the function (the input values don't matter)
- io_
  a parameter to the function that's used as both an input and an output.
- No prefix means it's a local variable.

The standard type prefixes are:

- i integer
- c char
- f float
- d double
- p pointer
- a array
- o other

plus you can make up other letter combinations for commonly used types, such as pt for point. Some examples:

- **int** m_aiIndices[10];
  An member variable that is an array of integers.
- **const** Point &in_ptStart
  An input point to a function.
- **double** &out_dLength
  A double parameter to a function. After the function returns the variable will be set to the length.
- **double** g_dEpsilon = 1e-6
  A global variable.

12.3.3 Global and static designations

A global variable is one that's declared in a `.cpp` file but not within a function or method. A static variable is defined similarly, but with the `static` keyword added. The difference between the two is that the static variable can not be used outside of the `.cpp` file it's declared in. A `static` variable can also be declared within a class, in which case the class defines the scope.

```
// define global variable in this file
double g_dGlobalEpsilon = 1e30;

// define static variable in this file
static double s_dStaticEpsilon = 1e30;


// For .H files:
// declare global variable defined elsewhere
extern double g_dGlobalEpsilon;

class Foo {
private:
  // static variable only accessible in this class
  static double m_dMyEpsilon;
};

// must have this line in the corresponding .cpp file
double Foo::m_dMyEpsilon = 1e-30;
```

Global and static variables should be used with great caution. Global variables can be set from anywhere, so it's very hard to keep track of what happens to them. Static designations help with this problem by restricting who can access the variable, but you can still get unexpected behaviour because the variables persist.

The most common use of global variables is to keep track of some state or data that's active and valid throughout the entire life-span of the program. This is a perfectly legitimate use. To prevent problems, always perform checks on global variables (if possible) before using them, such as making sure they're in valid, expected ranges. If the global variable is a pointer, add in a magic ID number and check it's validity every time.

Some people use global variables to avoid passing data into functions (they don't want to add the extra 30 bytes to each function call). Bad idea. If your argument lists are getting long, bundle up the data into a class and pass the class around. If you always explicitly pass the data that you are editing and using, then it is very clear to the reader what can and can't happen inside of a given function call. If every function edits or uses some mysterious global variable then it's hard to know what's safe to change.

Static variables, both defined in files and in classes, are somewhat safer. The most common use for static variables is to keep data that is expensive to compute and multiple instantiations of a class (or multiple function calls) can use. It's best to wrap this up with a couple access methods or functions.

Gotcha 10:

There are a couple of main gotchas with static variables. The first is in initialization order. Every comiler is different. Some create and initialize static variables when the library is loaded, some when the variable is accessed for the first time, and some when the program starts up. Given a set of static variables in different files, you can't guarantee the order of initialization.

The second common problem is forgetting that you have a static variable involved in a recursive call. Any static variable used in a class passed into a recursive call will not be copied when the rest of the class is copied onto the stack. This means that keeping a recursive count in a static member variable is a very bad idea.
