CS 250/333: Frequently Asked Questions

Steven J. Zeil

August 16, 2013

Contents

This is a collection of questions (and answers!) that have arisen repeatedly in some of my past classes.

1 General Programming Questions

1.1 I'm not sure I have a correct algorithm to solve this problem. How do I check before writing and debugging the code? (Desk-Checking)

1. Research - is there a known algorithm for addressing this kind of problem? Check your text and the lecture notes. Check the web.

   Even if you find something, you will likely have to adapt it to the details of the problem at hand, but this can get you started.

2. Use a systematic procedure for design, such as stepwise refinement, to derive a new algorithm that solves your problem. (The point being that the worst thing you can do is usually to simply start throwing C++ statements around at random trying to guess at an algorithm design.

3. Whatever you come up with, use *desk checking* (a.k.a., *desk execution*) to make sure you have something reasonable before you start sweating all the fine details of coding.

   In desk checking, you sit down with a pencil and paper and "play computer".

   - Make a list of all the variables and inputs to your algorithm.
   - Make up a test case by filling in some values for the inputs.
   - Now step through your algorithm, line by line.
     - Each time a variable is assigned a value or changes value, write the value next to the variable name in your list. (For arrays, linked lists, and other "structured" data, you might want to draw a picture.)
     - Continue until you reach the end of your algorithm.

   Try this for a few different test inputs.

   Note that you don't need to have the algorithm written in a programming langauge for this to work. This procedure works just as well with pseudocode as with "real" code.

By doing this you will either discover that your algorithm works just fine, or that it breaks for some of your test cases, or you may find yourself stuck and unable to complete the simulated execution because you discover that you have left out critical details of how the algorithm is going to work. *Any* of these outcomes is valuable information to you as you design your program.

```
1: computeTheSquareRootOf (x)
2: {
3:   double root = x;
4:   double newRoot = x / 2.0;
5:   while (root and newRoot differ by more that 0.1%)
6:    {
7:      root = newRoot;
8:      newRoot = (root + (x / root)) / 2.0;
9:    }
10:  return newRoot;
11:}
```

For example, if I wanted to desk-check this pseudocode, I would make a list of the three variables:

```
x:
root:
newRoot:
```

For a test, I might choose to try and find the square root of 2. So I start off by setting x to 2:

```
x: 2
root:
newRoot:
```

Then I start through the code. At line 3, `root` is assigned:

```
x: 2
root: 2
newRoot:
```

and at line 4 `newRoot` is assigned:

```
x: 2
root: 2
newRoot: 1
```

At line 5 we check and find that the two values are different by considerably more than 0.1%, so we move inside the loop. Line 7 changes `root`:

```
x: 2
root: 2
newRoot: 1
```

and then line 8 changes `newRoot`:

```
x: 2
root: 2
newRoot: \sout{1} 1.5
```

Returning to the start of the loop, the two numbers still differ by more than 0.1%, so we rpeat the loop. Line 7:

```
x: 2
root: \sout{2} 1.5
newRoot: \sout{1} 1.5
```

Line 8:

```
x: 2
root: \sout{2} 1.5
newRoot: \sout{1} \sout{1.5} 1.416667
```

Still differ by more than 0.1%, so we go again.

Line 7:

```
x: 2
root: \sout{2} \sout{1.5} 1.416667
newRoot: \sout{1} \sout{1.5} 1.416667
```

Line 8:

```
x: 2
root: \sout{2} \sout{1.5} 1.416667
newRoot: \sout{1} \sout{1.5} \sout{1.416667} 1.414216
```

At line 5, we see these numbers differ by less than 0.1%, so we exit the loop, go to line 10 and return the output 1.414216. This is indeed close to the "true" square root of 2 (1.414213562...), so we are happy with our algorithm.

Next we might try other possibilities, such as computing the square roots of 0, 1, 0.01, 10000. We might eventually decide that we need to be a bit more precise in line 5 (0.1% of *what*?) or that the initial guess in line 4 really isn't very good when x gets large. But the point is that we can work these details out and improve our own understnading of the code by working through this process.

1.2 My program crashed. How do I find out why?

There's no easy answer to that. Here's what I do, though, when faced with a crash that I just don't understand:

1. Look at the output produced before the crash. That can give you a clue as to where in the program you were when the crash occurred.

2. Run the program from within a debugger (`gdb` if you have compiled with `g++`). Don't worry about breakpoints or single-stepping or any of that stuff at first. Just run it.

   When the crash occurs, the debugger should tell you what line of code in what file eas being executed at the moment of the crash.

   Actually, it's not quite that simple. There's a good chance that the crash will occur on some line of code you didn't actually write yourself, deep inside some system library function that was called by some other system library function that was called by some other. . . until we finally get back to your own code. That crash occurred because you are using a function but passed it some data that was incorrect or corrupt in some way.

   Your debugger should let you view the entire runtime stack of calls that were in effect at the moment of the crash. (Use the command “backtrace” or “bt” in `gdb` to view the entire stack.) So you shoud be able to determine **where** the crash occurred. That's not as good as determining **why**, but it's a start.

3. Take a good look at the data being manipulated at the location of the crash. Are you using pointers? Could some of them be null? Are you indexing into an array? Could your index value be too large or negative? Are you reading from a file? Could the file be at the end already, or might the data be in a different format than you expected?

   If you used a debugger to find the crash locations, you can probably move up and down the stack (`gdb` commands "up" and "down") and to view the values of variables within each active call. This may give a clue about what was happening.

4. Form some hypotheses (take a guess) as to what was going on at the time of the crash. Then *test your hypothesis*! You can do this a number of ways:

   (a) Add debugging output. If you think one of your variables may have a bad or unanticipated value, print it out. Rerun the program and see if the value looks OK. E.g.,

   ```
   cerr << "x = " << x << "   y = " << y << endl;
   cerr << "myPointer = " << myPointer << endl;
   cerr << "*myPointer = " << *myPointer << endl;
   ```

   (b) Add an assertion to test for an OK value. E.g.,

   ```
   assert (myPointer != 0);
   ```

   Rerun the program and see if the assertion is violated.

   (c) In the debugger, set a breakpoint shortly before the crash location. Run the program and examine the values of the variables via the debugger interface. Single step toward the crash, watching for changes in the critical variables.

   Once you have figured out what was the immediate cause of the crash, then you're ready for the really important part.

5. Try to determine the ultimate reason for the problem.

   Sometimes the actual problem is right where the crash occurs. Unfortunately, it's all to common for the real "bug" to have occurred much earlier during the execution. But once you know which data values are incorrect or corrupted, you can strart trying to reason backwards through your code to ask what could have caused that data to *become* incorrect.

   As you reason backwards, continue to form hypotheses about what the earlier cause might be, and keep testing those hypotheses as described in the prior step.

1.3 Why do compilers' error messages often give the wrong line number, or even the wrong file?

A compiler can only report where it *detected* a problem. Where you actually *committed* a mistake may be someplace entirely different. The vast majority of error messages that C++ programmers will see are

- syntax errors (missing brackets, semi-colons, etc.)
- undeclared symbols
- undefined symbols
- type errors (usually "cannot find a matching function" complaints)
- const errors

Let's look at these from the point of view of the compiler.

Syntax errors Assume that the compiler has read part, but not all, of your program. The part that has just been read contains a syntax error. For the sake of example, let's say you wrote:

```
x = y + 2 * x // missing semi-colon
```

Now, when the compiler has read only the first line, it can't tell that anything is wrong. That's because it is still possible, as far as the compiler knows, that the next line of source code will start with a ";" or some other valid expression. So the compiler will *never* complain about this line.

If the compiler reads another line, and discovers that you had written:

```
x = y + 2 * x // missing semi-colon
++i;
```

it still won't conclude that there's a missing semi-colon. For all it knows, the "real" mistake might be that you meant to type "+" instead of "++".

Now, things can be much worse. Suppose that inside a file `foo.h` you write

```
class Foo {
  ⋮
  Foo();
  int f();
  // missing };
```

and inside another file, `bar.cpp`, you write

```
#include "foo.h"

int g() {...}

void h(Foo) {...}

int main()  {...}
```

Where will the error be reported? Probably on the very last line of `bar.cpp`! Why? Because until then, it's still possible, as far as the compiler knows, for the missing "};" to come, in which case `g`, `h`, and `main` would just be additional member functions of the class `Foo`.

So, with syntax errors, you know only that the real mistake occurred on the line reported *or earlier*, possibly in an earlier-`#include`'d file.

undeclared and undefined symbols See this discussion.

type errors When you use the wrong object in an expression or try to apply the wrong operator/function to an object, the compiler may detect this as a type mismatch between the function and the expression supplied as the parameter to that function. These messages seem to cause students the most grief, and yet the compiler is usually able to give very precise descriptions of what is going wrong. The line numbers are usually correct, and the compiler will often tell you exactly what is going wrong. That explanation, however, may be quite lengthy, for three reasons:

1. Type names, especially when templates are involved, can be very long and messy-looking.
2. Because C++ allows overloading, there may be many functions with the same name. The compiler will have to look at each of these to see if any one matches the parameter types you supplied. Some compilers report on each function tried, explaining why it didn't match the parameters in the faulty call.

3. If the function call was itself produced by a template instantiation or an inline function, then the problem is detected at the function call (often inside a C++ standard library routine) but the actual problem lies at the place where the template was used/instantiated. So most compilers will list both the line where the error was detected and all the lines where templates were instantiated that led to the creation of the faulty call.

So, to deal with these, look at the error message on the faulty function call. Note what function/operator name is being complained about. Then look at the line where the faulty call occurred. If it's inside a template or inline function that is not your own code, look back through the "instantiated from" or "called from" lines until you get back into your own code. That's probably where the problem lies.

Here's an example taken from a student's code:

```
g++ -g -MMD -c testapq.cpp
/usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/../../../../include/g++-3/
stl_relops.h: In function ‘bool operator ><_Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> >(const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> &, const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> &)’:
adjpq.h:234:   instantiated from ‘adjustable_priority_queue<
 PrioritizedNames,map<PrioritizedNames,int,CompareNames,allocator<int> >,
 ComparePriorities>::percolateDown(unsigned int)’
adjpq.h:177:   instantiated from ‘adjustable_priority_queue<PrioritizedNames,
 map<PrioritizedNames,int,CompareNames,allocator<int> >,
 ComparePriorities>::makeHeap()’
adjpq.h:84:   instantiated from here
 /usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/../../../../include/
 g++-3/stl_relops.h:43: no match for ‘const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> & < const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> &'
```

Now, that may look intimidating, but that's mainly because of the long type names (due to template use) and the long path names to files from the C++ standard library. Let's strip that down to the essentials:

```
g++ -g -MMD -c testapq.cpp
stl_relops.h: In function 'bool operator >:
adjpq.h:234:   instantiated from 'percolateDown(unsigned int)'
adjpq.h:177:   instantiated from 'makeHeap()'
adjpq.h:84:   instantiated from here
stl_relops.h:43 no match for ... < ...
```

(This one is actually worse than most error messages, becuase it's easy to miss the " < " operator amist all the `<...>` template markers.)

The problem is a "no match for" a less-than operator call in line 43 of a template within the standard library file `stl_relops.h`. But that template is instantiated from the student's own code (adjpq.h) and so the thing to do is to look at those three lines (234, 177, and 84) for a data type that is supposed to support a less-than operator, but doesn't.

const errors Technically, "const"-ness is part of a type, so while sometimes these get special messages of their own, often they masquerade as ordinary type errors and must be interpreted in the same way.

1.4 What is a segmentation error? Segmentation fault? Bus error? Null reference error?

These are all various errors signaled by the underlying operating system when your program tries to retrieve from, store into, or execute instructions from an address that either doesn't exist or is reserved for another program. In practical terms, these kinds of messages almost always arise because of

- a pointer that is null, uninitialized, or pointing to an object that has already been `delete`'d, or
- array indices that are out-of-bounds, or

- iterators that are uninitialized or out of bounds.

To debug these problems, concentrate on your pointers, arrays, and iterators. Add debugging output or use a debugger to find out which one is being used when the crash occurs. Then work backwards to figure out why that pointer/index/iterator has an invalid value. Keep in mind that you might not be using a pointer, array, or iterator directly, but might be using (abusing?) a function or class that does.

1.5 What are formal and actual parameters?

The *formal parameters* of a function are the parameter names that are declared in the function header and that are used when we write the function body. For example, in the code

```
int sequentialInsert (T a[], unsigned& n, const T& x)
// insert x into sorted position within a,
//   with a already containing n items.
//   Return the position where inserted.
{
  int i = n;
  while ((i > 0) && (x < a[i-1]))
      {
       a[i] = a[i-1];
       i = i - 1;
      }
  a[i] = x;
  ++n;
  return i;
}
```

the formal parameters are named `a`, `n`, and `x`. The *actual parameters* of a *call* to a function are the values passed by the caller. So, in the call

```
k =  sequentialInsert (myArray, size-1, value);
```

the actual parameters are `myArray`, `size-1`, and `value`. Note that formal parameters are always just names. Actual parameters can be simple names or arbitrarily complicated expressions.

2 The C++ Programming Language

2.1 How do I prompt for and read keyboard input from the same line?

Use `flush` instead of `endl`.

Whenever you are doing I/O, you are dealing with hardware that is much slower than your CPU. If every time your code tried to read or write a character, a separate set of instructions had to be issued to the hardware, this would slow your codedown to a crawl. In addition, most hardware is designed to be most effifient when transfering sizable blocks of bytes at a time, not one byte at a time. So forcing a seprate hardware I/O operation every time you need to read or write a byte would slow the hardware down, further slowing your program.

Consequently, when you send output to a divice, the bytes to be output are normally saved in a *buffer* in memory. When the buffer is full, that whole block of bytes is sent to the output device. Similarly, most input devices read blocks of bytes at once, storing them in a buffer. When your program requests input, it is given the bytes from the buffer first, and a new input hardware operation occurs only when the buffer has been emptied.

For buffering to work correctly with output, special code is added to detect the end of a program's execution, and any remaining bytes in the output buffers are *flushed* to the output device before the program terminates. (That's why, when a program crashes, you sometimes get only a portion of the output that, logically,had been written up to that moment in time.)

Buffering poses a problem for interactive I/O. If we write a prompt "Please enter a number:" and those characters sit in a buffer, the person wiating at the keyboard does not get to see the prompt and does not know they are even supposed to start typing. The solution is to `flush` the output buffer:

```
cout << "Enter a number: " << flush;
int N;
cin >> N;
```

`flush` simply forces all pending output in the output buffer to be sent. It does not add any characters of its own, *not even an end of line*. Consequently any juman-typed response will appear on the same line as the prompt. The more familiar I/O manipulator, `endl` is actually equivalent to printing a newline character followed by a flush. This:

```
cout << "Hello" << endl;
```

is the same as this:

```
cout << "Hello" << "\n" << flush;
```

It's just shorter and easier to type. But unlike `flush` alone, it *does* add a new line.

2.2 What are argc and argv?

`argc` and `argv` are the most commonly used names for the input parameters to the function `main`:

```
int main (int argc, char** argv)
{
    ⋮
  return 0; // to indicate the program ran successfully
}
```

They are used for passing parameters from the command line to a program

- `argc` is the number of arguments on the command line (including the program name itself)
- `argv` is an array of C-style strings (character arrays, with a null character at the end of each string)
  - `argv[0]` is the name of the program being executed
  - `argv[1]` is the first parameter supplied on the command line
  - `argv[2]` is the second parameter supplied on the command line, and so forth.

So if you type the command

```
  runThis foo bar baz
```

`argc` will be 4, `argv[0]` will be "runThis"[1], `argv[1]` will be "foo", `argv[2]` will be "bar", and `argv[3]` will be "baz" Some programs allow variable numbers of command-line parameters, in which case `argc` can be used to determine how many were supplied on a given execution. When programs require a certain number of command-line parameters, however, `argc` is often checked to be sure that those parameters were supplied. If not, most programmers print a "usage" message explaining how to invoke the program. Here, for example, is a program that expects to get a single parameter, the name of a file to be read as input:

```
int main (int argc, char** argv)
{
  if (argc != 2) // 2, not 1, because the program name is included
    {
```

[1] Actually, the exact string varies. Some operating systems supply the full path, others just give the program name.

```
    cerr << "Usage: " << argv[0] << " inputFileName" << endl;
    return -1; // indicates unsuccessful execution
   }

  ifstream myInputFile (argv[1]);
   ⋮
  return 0;      // indicates successful execution
}
```

2.3 What's the odd expression with the '?' and ':' ?

You're probably looking at a *conditional expression*. It's a convenient shorthand for an if-then-else structure inside an ordinary expression. A typical example would be

```
x = (x < 0.0) ? -x : x;
```

The part before the '?' is the condition and is evaluated first. If the condition is true, then the then-part expression (between the '?' and the ':') is evaluated and the result used as the value of the entire expression. If, however, the condition is false, then the else-part (after the ':') is evaluated and that result is used as the value of the entire expression. This assignment, then, replaces `x` by its own absolute value. The whole thing, then, is equivalent to

```
double temp;
if (x < 0.0)
  temp = -x;
else
  temp = x;
x = temp;
```

A conditional expression can apear inside other expressions, anywhere a value of the same type as its then-part and else-part might appear:

```
x = sqrt((x < 0.0) ? -x : x);
```

Some things to keep in mind when using conditional expressions:

- The condition part mst be a `bool` or `int`.

- The data types of the then-part and else-part expressions must be identical, and must make sense in the context where the conditional expression is placed.

- Because this is inside an expression, not an independent statement, both the then-part and the else-part must compute *something*. There's no possible equivalent to an "if-then" statement with a missing "else".

2.4 What are #define, #ifdef, and #ifndef?

C++ lines that begin with # are preprocessor instructions. The preprocessor runs before the compiler, and its job is to "massage" the text of your source code into an easily compilable form. The most common preprocessor directive is `#include`, which is used to add the code from a .h file into the file being compiled. (See Figure 1)

You don't need to worry about doing this preprocessing yourself. When you invoke a compiler such as " `g++` ", it actually runs several programs, starting with the preprocessor, then the compiler proper, and possibly a linker at the end. `#define` is used to define symbols that the preprocessor can replace. These symbols are never seen by the compiler itself, because the preprocessor will replace them by their definition before the compiler itself ever runs. For example, in a program where we repeatedly wished to print out the author's name, we might write:

```
#define AUTHOR "John Q. Programmer"
  ⋮
cout << "This program was written by " << AUTHOR;
  ⋮
string message =  string("I'm just head over heels about ") + AUTHOR;
  ⋮

cout << "If you like this program, send " << AUTHOR
     << " lots of money!" << endl;
```

then each occurence of `AUTHOR` would be replaced by the name string before the compiler ever saw the text.= of the source code. Now, that's actually not a particularly good use of `#define`. C programmers did that sort of thing a lot, because C had no built-in facility for declaring constants. In C++, we would generally prefer doing:

```
const string AUTHOR = "John Q. Programmer";
```

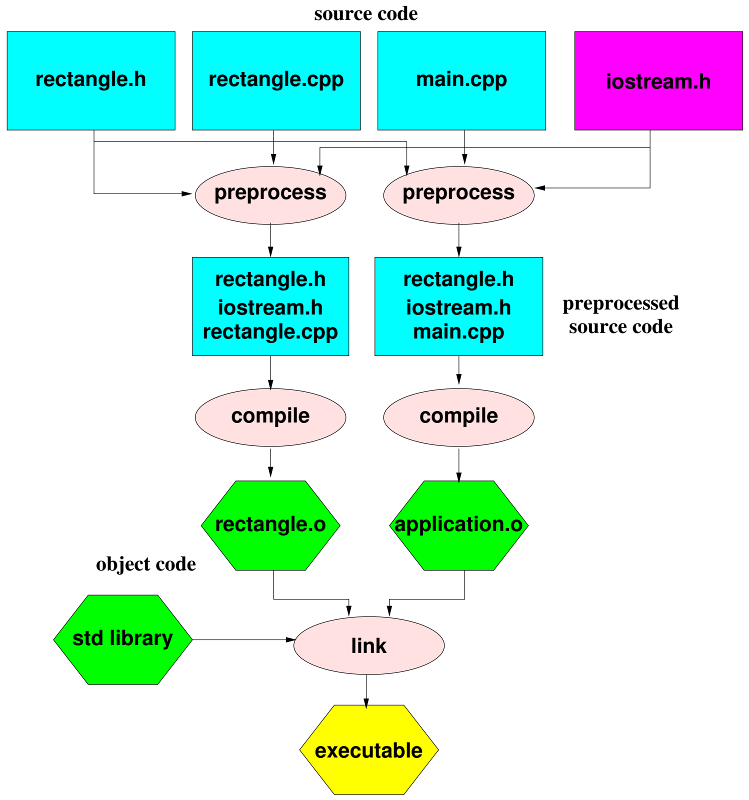


Figure 1: Building 1 program from many files

But `#define` still has two important uses in C++. One is that you can add parameters to a preprocessor symbol definition (in which case it is called a *macro*). Because macros are handled before the compiler, you can, on rare occasions, do things with these that you can't do with an ordinary C++ function. For example,

```
#define dbgout(x) cerr << #x << ": " << x << endl
```

defines a macro `dbgout` that takes a single parameter (the # inside the definition turns the parameter into a string, putting quote marks around it and, if necessary, putting backslashes in from of any special characters that normally can't appear inside strings, such as other quote marks). Then if we write

```
int i;
string s;
PersonelRecord person;
   ⋮
dbgout (i);
   ⋮
dbgout (s);
   ⋮
dbgout (person);
```

we would see, in the output, lines like:

```
i: 1
s: Hello
person: John Doe  123 Elm St   Podunk, Il
```

provided that the `PersonelRecord` class had an appropriate output function defined for it. But the major use of `#define` is in setting flags that can be used to control what text actually gets seen by the compiler. This is done in conjunction with `#ifdef...#endif` and `#ifndef...#endif` preprocessor instructions. For example, to make it easy to turn our debugging output on and off, we might write:

```
#ifdef DEBUG
#define dbgout(x) cerr << #x << ": " << x << endl
#else
#define dbgout(x)
#endif
```

In other words, if the preprocessor has a defined symbol DEBUG, then we set up dbgout so that it gets replaced by the output statement. If DEBUG is not defined, we set up dbgout so that it gets replaced by an empty line -- it “disappears” from the program entirely. So, if we want to see the debugging output, we add

```
#define DEBUG 1
```

to our code, in some location where the preprocessor will encounter it before hitting any of our dbgout’s. Alternatively, the compiler lets us define preprocessor symbols in the command line with a -D option:

```
g++ -g -c -DDEBUG myprogram.cpp
```

Some preprocessor symbols are predefined by the compiler. For example, Microsoft Visual C++ defines a symbol _MSC_VER, and GNU g++ defines one named __GNUG__. We can use these symbols to select specific code based upon the compiler that is being used.

2.5 Why do .h files begin with #ifndef and #define?

First, it helps to know just what these statements mean. Suppose you had written a program composed of 3 files:

foo.cpp

```
#include <iostream>
#include <string>
#include "myADT1.h"
#include "myADT2.h"

int main() {
  ⋮
```

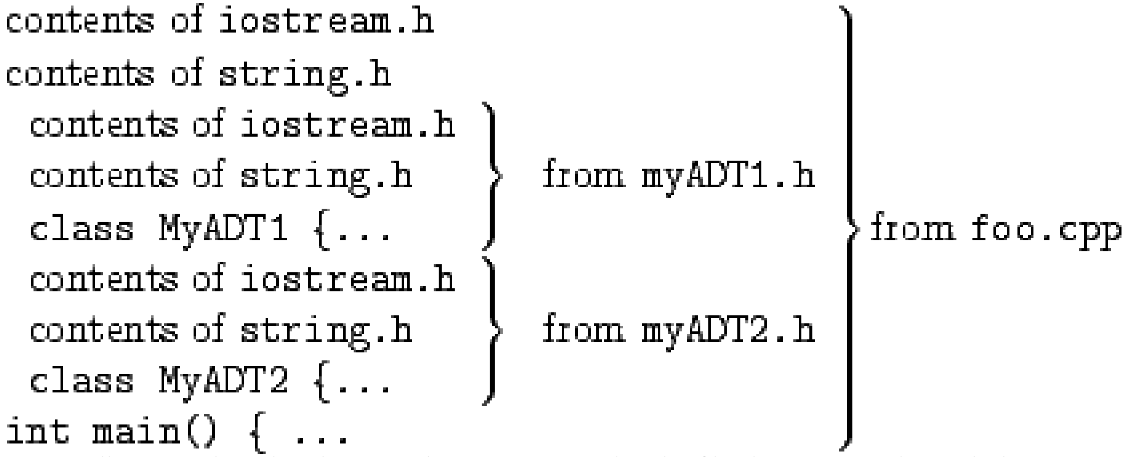
myADT1.h

```
#include <iostream>
#include <string>

class MyADT1 {
  ⋮
```

myADT2.h

```
#include <iostream>
#include <string>

class MyADT2 \{
  ⋮
```

There’s a real danger that when you compiled foo.cpp, your code after preprocessing would look like:



```
contents of iostream.h
contents of string.h
  contents of iostream.h
  contents of string.h
  class MyADT1 {...
  contents of iostream.h
  contents of string.h
  class MyADT2 {...
int main() { ...
```

Actually, it's potentially worse than that, because the system string header file almost certainly #includes iostream, so add another copy of iostream for each use of the string header. These two header files are large and complicated, and you may be compiling the same statements from them 6-12 times even in this simple example. Multiply this by all the other system headers that get used, and you can see that a lot of your time could be wasted waiting for statements that were already compiled to be compiled again and again and ... To avoid this problem, C++ programmers have a simple convention:

> *Every* header file gets enclosed in a structure like this:
>
> ```
> #ifndef SOMESYMBOLNAME
> #define SOMESYMBOLNAME
>    ⋮
> #endif
> ```

The actual symbol name isn't important, but needs to be unique for each header file, so most programmers base it on the header file name, replacing dots and other non-alphanumeric characters by underscores. So we can safely assume that the iostream header file looks something like:

```
#ifndef IOSTREAM_H
#define IOSTREAM_H
```

```
    ⋮
#endif
```

and the string header file looks something like:

```
#ifndef STRING_H
#define STRING_H
    ⋮
#endif
```

So in our example, when we compile

```
#include <iostream>
#include <string>
#include "myADT1.h"
#include "myADT2.h"

int main() {
   ⋮
```

the preprocessor hits the first #include and begins reading the iostream header file. The first thing it sees there is

```
#ifndef IOSTREAM_H
```

i.e., include the following code only if `IOSTREAM_H` is *not* defined. That symbol is not defined yet, so the preprocessor will include all code from there to the `#endif`. While it is doing so, it hits the next line:

```
#define IOSTREAM_H
```

and so, for the remainder of the compilation, the symbol `IOSTREAM_H` is defined. That means that, the next time we hit an

```
#include <iostream>
```

either in our own code or in some other header like `<string>`, the preprocessor will come to

```
#ifndef IOSTREAM_H
```

and will omit all the following code because now `IOSTREAM_H` is defined. So our whole program layout becomes:

```
con
con
 cl
 cl
int
```

There are no repeated copies of the code from inside any header file, the code seen by the compiler is shorter, your compilations take less time, and life is good!

2.6 What is a namespace?

At an informal level, a *namespace* is the set of all names that have been declared at some point in a program.

Often, programmers will `#include` a header file in order to use just one or two names decalred in that header. But because the header file actually declares dozens or maybe hunders of other items, all of those names become part of the programmer's namespace. This effect is sometimes called *namespace pollution*.

The problem with namespace pollution is that it increases the odds that the programmer might accidentally choose a variable or function a name that is already declared in a header somewhere and has been, unbeknownst to the programmer, found its way into the namespace. The result of such a *namespace collision* is usually an error message, and often a fairly unintelligable one at that.

Even worse, in large programs a programmer might be trying to use headers from two different commercial libraries. If these two libraries try to use the same name for anything, the resulting namesapce collision would force the programmer to choose between the two libraries.

For these reasons, the C++ standardization committee introduced the C++ `namespace` as a way to control namespace pollution. A `namespace` is a "container" of names. Every library is supposed to use its own namespace and only to declare things within that namespace. This includes the C++ standard library, hwere everything is declared within the namesapce `std`. Declaring things within a namespace is easy:

```
namespace MyCS361Library {
   int foo ();
```

```
    const int bar = 21;
};
```

There are three ways to get access to a name that has been declared within a namespace:

1. Put the namespace name in front, connected by " :: ". E.g.,

   ```
   void baz(int& i)
   {
      i = i + MyCS361Library::bar - MyCS361Library::foo();
   }
   ```

2. Use a "using declaration" to make a single name from the namespace visible:

   ```
   void baz2(int& i)
   {
      using MyCS361Library::bar;   // makes bar visible
      i = i + bar - MyCS361Library::foo();
   }
   ```

3. Use a "using directive" to make *everything* declared in a namespace visible:

   ```
   void baz2(int& i)
   {
      using namespace MyCS361Library;   // makes foo and bar visible
      i = i + bar - foo();
   }
   ```

2.7 What's the difference between <string>, <string.h>, and <cstring>?

The C programming language does not really include a character string data type. Instead, C programmers store strings in arrays of characters, with the convention that a character of value 0 is used at to indicate the end of the string.

The file `<string.h>` is used *in C programs* to get access to a variety of functions for manipulating these arrays of characters.

The file `<cstring>` is used *in C++ programs* to get access to these same functions (following the usual convention of adding “c” to the front of C-language standard header names).

C++ does have a character string type as part of its standard. The type named “ `string` ” and is declared in `<string>`.

What do you get in a C++ program if you try to `#include <string.h>`? Well, in fact, the C++ standard doesn’t say, so anything could happen. Most likely, you will get the C library header, the same as if you had asked for `<cstring>`. Of course, there’s no guarantee that a `<string.h>` from the C library will even compile in C++. On very rare occasions, you might get something related to `<string>`.

So the upshot is, a C++ program should avoid using `<string.h>`. There’s just no telling what it might do.

2.8 How do I convert a string to a character array (or a character array to string)?

To get a character array from a `std::string`, use the string’s `c_str()` function:

```
std::string str;
   ⋮
char* cstr = str.c_str();
```

You will often see this done with older libtrary functions that were originally designed to work with character arrays and noy yet updated to work with strings:

```
void copyFile (string fileName)
{
   ifstream input (fileName .c_str() ); // open an input file;
   ofstream output ("output.txt"); // No c_str() required - "output.txt" is a
                                   //    character array, not a string.
   string line;
   getline (input, line);
   while (input)
   {
      output << line << endl;
      getline (input, line);
   }
}
```

Going in the opposite direction is even simpler. The `string` class has a constructor for building strings from character arrays, so this takes palce automatically most of the time:

```
void copyFile (string fileName);
   ⋮
copyFile ("input.dat");   // automatically converted to std::string
```

2.9 How do I convert a string to a number (or a number to a string)?

There are some quick-and-dirty functions for doing this in `<cstdlib>` for converting to numbers:

```
#include <cstdlib>
using namespace std;
   ⋮
int i = atoi("123");
double f = atof("3.14159");
```

Now, these functions actually convert from character arrays, not strings, so if you have a string you need to convert to a character array:

```
#include <cstdio>
#include <string>
using namespace std;
   ⋮
string s1 = "123";
string s2 = "3.14159";
   ⋮
int i = atoi(s1.c_str());
double f = atof(s2.c_str());
```

If you need to go in the other direction, or if you need to deal with strings with unusual formatting, then there is a more general technique. In fact, you can convert *any* datatype that has << and >> I/O operators to and from strings by reading from and writing into a string. The `istringstream` is an input stream that reads from a string:

```
#include <sstream> // provides istringstream and ostringstream
```

```
#include <string>
using namespace std;
   ⋮
string s1 = "123 3.14159";
   ⋮
istringstream in (s1); // create a stream reading from s1
int i;
double f;
in >> i >> f;  // i will contain 123 and f will contain 3.14159
```

The `ostringstream` is an output stream that writes into a string:

```
#include <sstream> // provides istringstream and ostringstream
#include <string>
using namespace std;
   ⋮
string s1;
ostringstream out (); // create a stream writing into a string
int i = 245;
double f = 1.2;
out << i << ":" << f;
string s = out.str();  // Retrieve the string we have written
cout << s << endl;     // Prints "245:1.2"
```

Again, let me point out that this `stringstream` approach can be used to convert between `string` and *any* data type that you can read and write.

2.10 How do you tell when a string search has failed?

The standard string class contains a number of functions for searching strings. For example

```
    pos = string1.find("abc");
```

sets `pos` to the position where “abc” first occurs within `string1`, and

```
    pos = string1.find_first_of("abc");
```

sets `pos` to the position where any one of the characters 'a', 'b', or 'c' first occurs within `string1`, and

```
    pos = string1.find_first_not_of("abc");
```

sets `pos` to the position where any characters *except* 'a', 'b', or 'c' first occurs within `string1`. There are variations on these searching functions that allow you to begin searching from any position within the string, or that search for the last occurrence of a string or character instead of the first.

But what happens when the thing you're looking for isn't anywhere in the string? Many books say that the search functions will, in that case, return "some value larger than the length of the string". That's actually true, but it's just a bit imprecise and more than a little misleading. In particular, it suggests that the following code is probably appropriate:

```
int k = string1.find(string2);
if (k < string1.length())
   cout << "Found " << string1.substr(k) << endl;
```

when, in fact this code is actually likely to behave strangely and may even be flagged by some compilers as dangerous.

More exactly, when any of these search functions fail, they return `string::npos`, a special value declared in the string class. "npos" stands for "n^th position", which in a string of n characters would be past the end of the string. But `npos` is actually defined like this:

```
 class string {
    :
       typedef ... size_type;
    :
  static const size_type npos = -1;
    :
};
```

What's going on here with `npos`? It's `static`, meaning that this single entity is shared by all string objects rather than replicated inside each string. It's `const`, meaning that its value never changes. Its data type is `size_type`, which is declared earlier in the string class, and is guaranteed by the C++ language standard to be some unsigned integer type.[2] Now the initial value, -1, is a bit of a trick. Viewed as a pattern of bits, -1 consists entirely of 1 bits. But that pattern of bits consititutes a -1 only if we treat that block of bits as a signed value. Take that same number of 1 bits and treat them as an unsigned integer, and it becomes the

[2] Whether it's unsigned int, or unsigned long, or something larger depends on the compiler, but it will be *some* unsigned integer type.

largest possible unsigned value that can be held in number of bits. So `string::npos` is actually the largest possible unsigned integer that can be held in a `string::size_type` value.

Now, what happens when you do

```
int k = string1.find(string2);
if (k < string1.length())
```

and `string2` does not actually occur anywhere inside `string1`? The function call returns `string::npos`, which we store in `k`. Because `k` was declared as `int` (a signed type), it becomes equal to -1. Now when we compare it to `string1.length()`„ the compiler will convert `string1.length()` to an `int` and compare the two `int`s. `string1.length()` is probably a moderate-sized positive number, but `k` is negative, so the test succeeds just as if the `find` had actually been successful. That's *not* what we had intended!

Note that if `k` had been declared as `unsigned`, things would have been just fine. But I don't think most people really expect there to be such a difference in behavior depending on whether `k` was declared `int` or `unsigned`. For that reason, many compilers will issue warning messages when they detect comparisons between `int`s and `unsigned`s.

To be safe, you should do either or both of these things when checking to see if a string search has failed:

- Make sure that all the variables used to hold string positions are declared as `string::size_type`, and/or
- Don't compare the result against the string length. Compare against `string::npos`.

In other words, the safest way to write the above code would be:

```
string::size_type k = string1.find(string2);
if (k != string::npos)
   cout << "Found " << string1.substr(k) << endl;
```

2.11 What does the `assert` function do?

It helps programmers protect themselves from their own mistakes!

An `assert` statement takes a single argument, a boolean condition that we believe should be true unless someone somewhere has made a programming mistake or our data has somehow been corrupted.

```
#include <cassert>
   ⋮
void printRecord
        (PersonnelRecord[] people,
         int numPeople,
         int personToPrint)
{
  assert (personToPrint >= 0 && personToPrint < numPeople);
  cout << people[personToPrint];
}
```

assert is declared in the sysem header file <cassert>. Under most circumstances, the assert statement shown here would be translated into something like:

```
if (!(personToPrint >= 0 && personToPrint < numPeople))
  {
   cerr << "assertion violated in line 8 of foo.cpp" << endl;
   abort(-1);
  }
```

The exact details will vary from one compiler to another, but the general idea is that if the assertion condition is, as expected, true, then nothing happens. If, however, the assertion is false, then a run-time error message is generated and the program aborts. This is entirely consistent with the idea that a failed assertion is definite evidence of a programming error that has already corrupted the state of the program.

That's if you compile "under most circumstances". But, if you compile the code with the symbol NDEBUG defined to some (any) value[3], then all assert statements are translated as comments, effectively removing those tests from the code.

So, when the code is ready for release, you have a choice. You can leave the assert statements active, on the theory that having a program abort is worse than having it quietly produce incorrect results and pretend that nothing is wrong. Alternatively, you can remove the assert tests by compiling with NDEBUG set, on the theory that the end customer would rather see, (and

[3] e.g. by adding a statement

```
#define NDEBUG
```

somewhere before the assert statement or, when compiling with g++, by adding the compilation flag -DNDEBUG to the compilation command might, if you get lucky, fail to see) some incorrect answers than be served with an unintelligible error message and then have the program abort.

2.12 What is ptrdiff_t?

`ptrdiff_t` is the name of a special integer type declared in `<cstddef>`. `ptrdiff_t` is the kind of integer that you get when subtracting one pointer/address from another. For example,

```
#include <cstddefs>
  ⋮
Folderol array[26];
Folderol* pointer1 = &(array[1]);   // pointer1 is address of array[1]
Folderol* pointer2 = &(array[5]);   // pointer2 is address of array[5]
ptrdiff_t distance = pointer2 - pointer1;
```

At the end of this code, `distance` will be 4, the number of `Folderol`-sized chunks of memory between the two addresses in `pointer1` and `pointer2`. Could we have just said:

```
int distance = pointer2 - pointer1;
```

instead of using `ptrdiff_t`? Well, yes, in this case, but only because *we know that the answer (4) is small enough to fit in an `int`*. And that's the catch. If we want to write portable code, we don't know how many bits are in an int, or how many are in a pointer. (On some platforms, the number of bits in a pointer varies depending on how far away the pointer points!). So the use of `int` or `long` or any other conventional integer type would be risky in general.

That's why we have `ptrdiff_t` — on any platform `ptrdiff_t` will be an integer type big enough to hold the distance between two pointers.

2.13 What is the difference between a pointer and a reference?

In a sense both `T*` and `T&` of these represent an address to something of the `T`. But they are still considered different types, and we use them differently.

	Pointers	**References**
Type:	`T*` is the data type for pointers to objects of type `T`.	`T&` is the data type for references to objects of type `T`.

	Pointers	**References**
Size:	A pointer is stored in as many bytes as required to hold an address on the computer. This often makes pointers much smaller than the things they point to. We take advantage of this small size when storing data and when passing parameters to functions. It's much faster and memory-efficient to copy a pointer than to copy many of the things a pointer is likely to point to. [4]	A reference is stored in as many bytes as required to hold an address on the computer. This often makes reference much smaller than the things they refer to. We take advantage of this small size when storing data and when passing parameters to functions. It's much faster and memory-efficient to copy a reference than to copy many of the things a reference is likely to refer to.

	Pointers	**References**

Destination:	Pointers almost always point to data allocated on the heap. Pointers can be (and frequently are) null. If you forget to initialize a pointer, it contains a random pattern of bits (whatever happened to be left over in the block of memory from earlier use) and trying to follow that pointer to its destination is a *bad* thing.
	`T* ptr = new T; // ptr holds the address`
	`                // of something on the heap`
	`T* ptr2 = 0;`
		`T& ref = x; // ref holds the address of x`
		References may refer to data on the heap but are just as likely to refer to local variables in the current function or in a function that called it. References cannot be null. The only way to declare a reference is in an expression that initializes it, or to declare it as a function parameter. Either way, you are guaranteed that references will have been properly initialized.

Dereferencing:

Pointers	References

To get access to the entire object pointed to by a pointer, use the unary ∗ operator. If the thing pointed to is a class or struct, you get access to a member via the -> operator.
 	
`T value = *ptr;`
`int dataMember = ptr->memberName;`
	To get access to the entire object referred to by a reference, just use it like you would an ordinary variable. If the thing referred to is a class or struct, you get access to a member via the . operator (just as if you had an ordinary variable and not a reference).
	
	`T value = ref;`
	`int dataMember = ref.memberName;`
	
	The fact we use references just like ordinary variables makes them convenient as function parameters. In the language C, which lacked reference types, parameters that we would pass as T& were always passed as T∗ instead. The result was that most C functions where an ugly (IMO) mess of ∗ and -> operations.

Mutability:

Pointers	References
You can change where a pointer point to by simply assigning it a different address.	You cannot change where a reference refers to. Once initialized, a reference holds the same address until it is destroyed.

	Pointers	**References**
Deletion:	Because most pointers point to objects allocated on the heap, the programmer who uses pointers incurs an obligation to delete those objects when they are no longer needed. Failure to delete unneeded objects is a programming error (memory leak) that slows programs down and may lead to crashes if the available memory space is exhausted. Deleting something that is still needed (via other pointers) is a programming error (dangling pointers) that leads to incorrect output or crashes. Deleting the same object twice is a programming error that leads to incorrect output or crashes.	References cannot be used to delete objects. Because many references refer to objects not allocated on the heap, that's appropriate.

It is possible to convert between references and pointers.

```
T t;
T* ptr = new T;
T& ref = t;
ptr = &t; // & gets the address of t as a pointer
ptr = &ref; // & gets the address stored in ref as a pointer
T& ref2 = *ptr; // * gets the address in ptr as a reference
```

Technically, the value returned by unary * is a reference, and the value returned by unary & is a pointer. However, just because it's possible to do this doesn't mean its a good thing. Converting pointers to references is generally OK. In fact, it happens all the time in C++ without our noticing it. Going in the other direction is dangerous however. References often do not refer to things on the heap, and when we store those addresses in a pointers there is a great temptation elsewhere in the program to delete them. That's a quick way to corrupt your program's data.

2.14 Can I have 2 functions with the same name?

Yes, as long as they take different numbers or different types of parameters. This is called *overloading* and is a common element of C++ style.

2.15 What is the difference between a declaration and a definition?

Pretty much everything that has a "name" in C++ must be declared before you can use it. Many of these things must also be defined, but that can generally be done at a much later time. You *declare* a name by saying what kind of thing it is:

```
const int MaxSize;           // declares a constant
extern int v;                // declares a variable
void foo (int formalParam); // declares a function (and a formal parameter)
class Bar{...};              // declares a class
typedef Bar* BarPointer;     // declares a type name
```

In most cases, once you have declared a name, you can write code that uses it. Furthermore, a program may declare the same thing any number of times, as long as it does so consistently. That's why a single .h file can be included by several different .cpp files that make up a program --- most .h files contain only declarations. You *define* constants, variables, and functions as follows:

```
const int MaxSize = 1000;                  // defines a constant
int v;                                     // defines a variable
void foo (int formalParam) {++formalParam;} // defines a function
```

A definition must be seen by the compiler once and only once in all the compilations that get linked together to form the final program. A definition is itself also a declaration (i.e., if you define something that hasn't been declared yet, that's OK. The definition will serve double duty as declaration and definition.).

2.16 What goes in a .h file? What goes in a .cpp file?

The short answer is that a .h file contains shared *declarations*, a .cpp file contains *definitions* and local declarations.

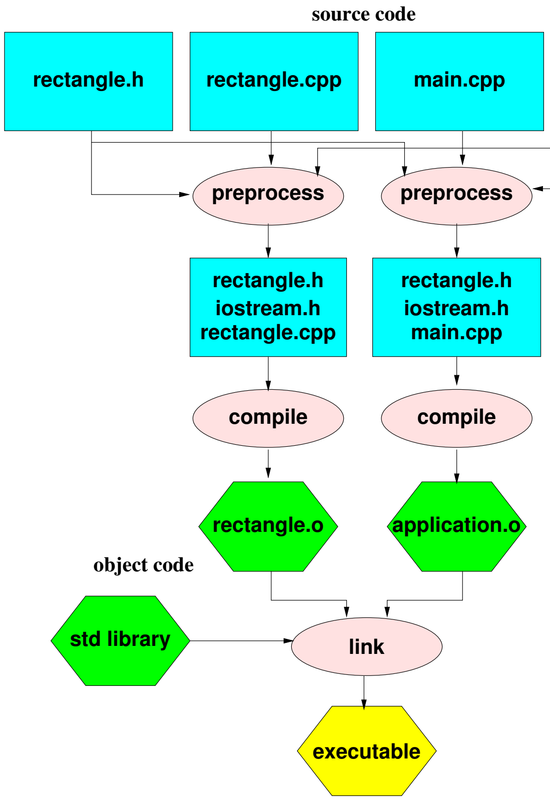
It's important that you understand the difference between declarations and definitions.

- A .h file is intended to be `#included` from many different .cpp files that make up a single program. In fact, the earliest stage of compilation, the *preprocessor*, actually replaces each `#include` by the full contents of the included file. Consequently, a .h file may be processed many times during the compilation of a single program, and should contain should contain only *declarations*.

- A .cpp file is intended to be compiled once for any given build of the program. So the .cpp file can have any declarations that it doesn't need to share with other parts of the program, and it can have definitions. But the main purpose of a `.cpp` file is to contain definitions that must only be compiled once. The most common example of this would be function bodies.

Never, ever, ever name a `.cpp` file in an `#include`. That defeats the whole purpose of a C++ program structure. `.h` files are `#included`; `.cpp` files are compiled.

A typical program will consist of many .cpp files.



Usually, each class or group of utility functions will have their definitions in a separate .cpp file that defines everything declared in the corresponding .h file. The .h file can then be `#included` by many different parts of the program that use those classes or functions, and the .cpp file can be separately compiled once, then linked together with the output of other .cpp files to form the complete program.

Splitting the program into pieces like this helps, among other things, divide the responsibility for who can change what and reduces the amount of compilation that must take place after a change to a function body.

2.17 What is an inline function?

When we define a function, it is usually compiled into a self-contained unit of code. For example, a function

```
int foo(int a, int b)
{
  return a+b-1;
}
```

would compile into a block of code equivalent to

```
stack[1] = stack[3] + stack[2] - 1;
jump to address in stack[0]
```

where the "stack" is the *runtime stack* a.k.a. the *activation stack* used to track function calls at the system level, stack[0] is the top value on the stack, stack[1] the value just under that one, and so on. A function call like

```
  x = foo(y,z+1);
```

would be compiled into a code sequence along the lines of

```
push y onto the runtime stack;
evaluate z+1;
push the result onto the runtime stack
push (space for the return value) onto the runtime stack
save all CPU registers
push address RET onto the runtime stack
jump to start of foo's body
RET: x = stack[1]
pop runtime stack 4 times
restore all CPU registers
```

As you can see, there's a fair amount of overhead involved in passing parameters and return address information to a function when making a call. The amount of time spent on this overhead is really all that large. If the function body contains several statements of any kind of loop, then the overhead is probably a negligable fraction of the total time spent on the call.

```
class Foo {
private:
   int bar;
public:
   int getBar ();
   void setBar (int);
};

int Foo::getBar ()  {return bar;}

void Foo::setBar (int b) {bar = b;}
```

But many ADTs have member functions that are only one or two lines long, and often trivial lines at that. For these functions, the overhead associated with each call may exceed the time required to do the function body itself. Furthermore, because these functions are often the primary means of accessing the ADT's contents, sometimes these functions get called thousands of times or more inside the application's loops.

For these kinds of trivial functions, C++ offers the option of declaring them as *inline*.

```
class Foo {
private:
   int bar;
public:
   int getBar () {return bar;}
   void setBar (int);
};

inline
void Foo::setBar (int b) {bar = b;}
```

An inline function can be written one of two ways. First, it can be written inside the class declaration. Second, we can place the reserved word `inline` in front of the function definition written in its usual place outside the class declaration. When we make a call to an inline function, the compiler simply replaces the call by a compiled copy of the function body (with some appropriate renaming of variables to avoid conflicts). So, if we have

```
inline int foo(int a, int b)
{
  return a+b-1;
}
```

and we later make a call

```
  x = foo(y,z+1);
```

This would be compiled into a code sequence along the lines of

```
evaluate z+1, storing result in tempB
evaluate y + tempB - 1, storing result in x
```

Most of the overhead of making a function call hs been eliminated.

Inline functions can reduce the run time of a program by removing unnecessary function calls, but, used unwisely, may also cause the size of the program to explode. Consequently, they should be used only by frequently-called functions with bodies that take only 1 or 2 lines of code. For larger functions, the times savings would be negligable (as a fraction of the total time) while the memory penalty is more severe, and for infrequently used functions, who cares?

Inlining is only a recommendation from the programmer to the compiler. The compiler may ignore an inline declaration and continue treating it as a conventional function if it prefers. In particular, note that inlining of functions with recursive calls is impossible, as is inlining of most virtual function calls. Many compilers will refuse to inline any function whose body contains a loop. Others may have their own peculiar limitations.

2.18 Why is operator++ sometimes declared with an int parameter?

The ++ and -- operators are unusual in that they can be written in either prefix (`++x`, `--x`) or postfix (`x++`, `x--`) form. Whether you write `++x` or `x++`, the value of `x` is increased by 1. But when you write a ++ expression inside another expression, its return value depends on whether you used the prefix or postfix form.

- The prefix form returns the value `after` the increment or decerement was performed.
- The postfix form returns the value `before` the increment or decerement was performed.

So, for example, the code

```
int i = 0;
int j = 0;
cout << ++i << ' ' << j++ << endl;
cout << i << ' ' << j << endl;
```

would print

```
1 0
1 1
```

Now that's all very well for the builtin ++ for `int`, but what happens when we write ++ for our own classes? Like most operators, ++ can be thought of as a shorthand for a function named `operator++`, so it's not too hard to see that we can say:

```
class SomethingWeCanIncrement
{
  :
  SomethingWeCanIncrement& operator++ ();
  :
};
```

but, somewhat late in the game, the designers of C++ realized that they had neglected to provide a syntax for indicating whether a unary operator was prefix or postfix. This is only a problem for ++ and `--`, because these are the only operators that can be written in both forms. The solution they came up with is a complete kludge. If you declare

```
const MyIncrementableClass operator++();
const MyIncrementableClass operator--();
```

you are declaring the *prefix* operators. If you declare

```
MyIncrementableClass operator++(int);
MyIncrementableClass operator--(int);
```

you are declaring the *postfix* operators. What do you do with the `int` parameter for the postfix operators? *Absolutely nothing!* It's just a dummy parameter used to distinguish the prefix and postfix forms. Finally, note that the prefix forms return a reference. The postfix forms return a non-reference value. That's because the prefix forms are returning the value that has been incremented/decremented. That value exists, so it's easy to return:

```
class MyIncrementableClass {
   ⋮
  const MyIncrementableClass operator++() {
         ⋮
     // do what you need to do to increment it
         ⋮
     return *this;
  }
   ⋮
```

On the other hand, the postfix form returns the value before the increment/decrement takes place. Usually the only way to do that is to make a copy of that value first, then do the increment, then return the copied value.

```
class MyIncrementableClass {
   ⋮
  const MyIncrementableClass operator++(int) {
        MyIncrementableClass clone = *this; // save old value
        operator++();                       // increment this
        return clone;                       // return the old value
  }
   ⋮
```

2.19 Why do many functions come in pairs that are nearly identical except for a `const`?

```
class Point {
  int xcoord;
  int ycoord;
public:
  Point (int x, int y) :
    xcoord(x), yxoord(y)
  {}

  int& x() {return xcoord;}
  int& y() {return ycoord;}
};
```

Let's suppose that we have a simple ADT: Note that, because the `x()` and `y()` functions return a reference (an address), we can both *look at* and *assign to* the components of a point:

```
aPoint.x() = aPoint.y();
```

```
void foo (Point& p1, const Point& p2)
{
   cout << "(" << p1.x()  <<
  "," << p1.y() << ")" << endl;
   p1.x() = 14;

   cout << "(" << p2.x()  <<
  "," << p2.y() << ")" << endl;
   p2.x() = 14;
```

Now suppose that we try to use our ADT in a function. This code has a real error in it. We have received `p2` as a const reference. That means that `foo` promises not to do anything to `p2` that would change its value. The final assignment statment clearly attpents to change `p2`. Obviously the programmer is confused and has forgotten that promise. When we compile this function, we would get error messages from the compiler on all our uses of `p2` (not just on the assignment). Why? Because `foo` promises not to do anything to `p2` that would change its value, and the compiler is going to enforce that promise. Obviously the assignment of 14 to `p2.x()` breaks that promise. But, in general, the compiler has no way of knowing that *any* call to `x()` or `y()` would not change the `Point` they were applied to. The fact that the current bodies of those don't make any such changes is irrelevant - the designer of the `Point` ADT could come along and change those at any time. So the compiler will not permit us to apply *any* function to `p2` that does not promise to leave `p2` unchanged.

How does a member function promise to leave its object unchanged? By having the word `const` after the parameter list.

So we could change our ADT as shown here, which would allow our `foo` function to compile.

```
class Point {
  int xcoord;
  int ycoord;
public:
  Point (int x, int y) :
    xcoord(x), yxoord(y)
  {}

  int& x() const {return xcoord;}
  int& y() const {return ycoord;}
};
```

But we're really lying to the compiler here. We are saying that these two functions are safe to use with a value we don't want changed. The simple fact that we can usethem to write

```
p2.x() = 14;
```

shows this to be a lie.

With this change

```
class Point {
  int xcoord;
  int ycoord;
public:
  Point (int x, int y) :
    xcoord(x), yxoord(y)
  {}

  int x() const {return xcoord;}
  int y() const {return ycoord;}
};
```

We can live up to our promise by changing the output type:but now when we compile

```
void foo (Point& p1, const Point& p2)
{
   cout << "(" << p1.x()  <<
  "," << p1.y() << ")" << endl;
   p1.x() = 14;
   cout << "(" << p2.x()  <<
  "," << p2.y() << ")" << endl;
   p2.x() = 14;
```

both assignments get flagged with errors. That's OK for the assignment to p2, which we should not be allowed to do (since p2 is declared as const) but there's no logical reason to forbid this operation on the non-const p1.

We get the best of both worlds be including both the const and the original non-const forms of these functions:

```
class Point {
  int xcoord;
  int ycoord;
public:
  Point (int x, int y) :
    xcoord(x), yxoord(y)
  {}

  int& x() {return xcoord;}
  int x() const {return xcoord;}


  int& y() {return ycoord;}
  const int& y() const {return ycoord;}
};
```

Now when we compile

```
void foo (Point& p1, const Point& p2)
{
   cout << "(" << p1.x()  <<
  "," << p1.y() << ")" << endl;
   p1.x() = 14;
   cout << "(" << p2.x()  <<
  "," << p2.y() << ")" << endl;
   p2.x() = 14;
```

the compiler chooses, for each call to `x()` and `y()` the “best fitting” version of the function. When applied to the non-const `p1`, the compiler uses the non-const versions of `x()` and `y()`. When applied to the const `p2`, the the compiler uses the const versions of `x()` and `y()`. That menas that only the line with assignment to `p2`gets flagged, as is appropriate.

2.20 I have other C++ questions.

Many questions about C++ can be answered at the C++ FAQ LITE. Information about the class and function templates in the C++ standard library can be found at SGI’s site.

3 C++ Compilers and Error Messages

3.1 How do I fix errors involving undeclared/undefined names?

First, look *very* closely at the error messages. Does it say “undeclared” or “undefined”? These are two very different things, and understanding the difference is the key to fixing the problem.

So, if the compiler says that a function is *undeclared*, it means that you tried to use it before presenting its declaration, or forgot to declare it at all.

The *compiler* never complains about definitions, because an apparently missing definition might just be in some other file you are going to compile as part of the program.

But when you try to produce the executable program by linking all the compiled .o or .obj files produced by the compiler, the *linker* may complain that a symbol is *undefined* (none of the compiled files provided a definition) or is *multiply defined* (you provided two definitions for one name, or somehow compiled the same definition into more than one .o or .obj file).

For example, if you forget a function body, the linker will eventually complain that the function is undefined (but the name of the function may be mangled in the error message, see below). If you put a variable or function definition in a .h file and include that file from more than one place, the linker will complain that the name is multiply defined.

3.2 What should I do about the error message about "warning: ... is implicitly a typename" and "warning: implicit typename is deprecated"?

Actually, you could just ignore these, because they are only warnings, not errors. Your code will compile properly even with these. This error arises in certain uses of template parameters (or of names that are `typedef`'d in terms of a template parameter. For example,

```
template <class Container, class T>
void fillContainer (Container& c, T value)
{
   Container::iterator b = c.begin();
   Container::iterator e = c.end();
   fill (b, e, value);
}
```

will probably get this warning from g++ complaining about the mentions of "`Container::iterator`". The fix is

```
template <class Container, class T>
void fillContainer (Container& c, T value)
{
   typename Container::iterator b = c.begin();
   typename Container::iterator e = c.end();
   fill (b, e, value);
}
```

3.3 Why do I get a message about discarding qualifiers?

The message

```
In some-function-name, passing const some-type-name ...
        discards qualifiers
```

occurs when you try to pass a const object to a function that might try to change the object's value. For example, if you have a class `C`:

```
class C {
public:
   C();
   int foo ();
   void bar (std::string& s);
   ⋮
}
```

and you try to compile the following code:

```
void baz (const C& c1, C& c2, const std::string& str)
{
  int i = c1.foo();     // error!
  int j = c2.foo();
  c2.bar(str);          // error!
    ⋮
```

then a C++ compiler should flag the 1st and 3rd line indicated above. The g++ compiler will say something along the lines of

```
In function 'void baz(const C\&, C\&, const std::string\&)':
   passing 'const C' as 'this' argument of 'int C::foo()' discards qualifiers

In function 'void baz(const C\&, C\&, const std::string\&)':
  passing 'const std::string' as argument 1 of
  'void C::bar(std::string\&)' discards qualifiers
```

The first message complains that you have passed a const object as the left-hand parameter (implicitly named `this`) to the function `foo`, which has not promised to leave that parameter unchanged. You have, in effect, tried to discard the "qualifier" (the word "`const`") in the `const C&` datatype. The second message makes a similar complaint about the string parameter being passed to `bar`. Again, the object being passed is marked as `const`, but the declaration of `bar` suggests that `bar` is allowed to change the string it receives as a parameter. To get rid of this message, you must examine what it is you are trying to do and determine whether:

- The declarations of the functions you are trying to call are incorrect. Perhaps `foo` and `bar` really *should* promise to leave those parameters unchanged. I.e., perhaps they should have been declared like this:

```
class C {
public:
   C();
int foo () const ; // don't change the object it's applied to
void bar ( const std::string& s);  // don't change s
   ⋮
}
```

- or, perhaps the application code is declaring things as `const` that it really *does* want to change:

```
void baz ( C& c1, C& c2, std::string& str)
{
  int i = c1.foo();   // ok
  int j = c2.foo();
  c2.bar(str);        // ok
    ⋮
```

- or, perhaps the application really is not supposed to change those parameters, and you will need to find some other way to accomplish what you need to do. For example,

```
void baz (const C& c1, C& c2, const std::string& str)
{
  C c1Copy = c1;
  int i = c1Copy.foo();   // ok
  int j = c2.foo();
    ⋮
```

3.4 How do I fix errors "No match for..."

This is a variation on the messages saying a symbol is undeclared. In particular, you will get this message when you call a function, and there are one or more functions with that name, but your set of actual parameters' data types do not match up with the formal parameter list of any of the declared functions. Either you are correctly calling a function that you have not declared, or you are trying to call a declared function with the wrong kind of parameters. This can be one of the longer error messages you will ever get, as g++ tries to list out *all* the functions with the same name that it knows, with the data types of all their parameters, as a way of showing you all your existing options. Although the list can be a bit daunting, if you are running in a support environment (e.g., emacs) that let's you step from message to message while displaying the relevant line of code, this list can actually be quite helpful.