C++ in review

Theo Norvell. MUN

1 Introduction

C++ is a very complex language. This is no doubt part of its appeal. If you need (or merely want) some language feature, C++ probably has it in one form or another.

This paper reviews some aspects of C++. I have concentrated on those aspects that will likely be used in this course (Engr 4892—Data Structures) and that are likely to need reinforcement. In particular I have tried to warn against the kinds of misapprehensions I have seen in the past.

For the most part, I have skipped over the statement syntax and most forms of expressions. These are fairly straightforward and most books on C++ cover these aspects quite well.

To really get to know the language, read a good book or two. Bjarne Stroustrup's *The C++ Programming Language*, third edition is probably the best reference,[1] while Scott Meyer's *Effective C++* and *More Effective C++* focus on specific techniques and areas of confusion. And write lots of programs.

2 Objects or variables

A C++ *object* is a region of storage (memory). Each object will have a number of attributes: name, type, value, size, address, lifetime. Objects are often called *variables*, though sometimes the word *variable* refers to the name of an object. I will use the word variable to mean object.

2.1 Names

Most variables have *names*. The declaration

```
int i ;
```

creates a new variable and names it i. As we will see later, some variables —heap variables— do not have names.

It is important to understand that more than one variable can have the same name at the same time. For example, if we write

[1] Stroustrup is the creator of C++.

```
{
      int i = 1;
      if( x ) {
            int i;
            i = 13 ;
            ... }
      cout << i ;
}
```

then two variables both named i will exist at the same time, if the 'then' clause of the if-statement is executed. This causes no problem. The two variables occupy different location in the computer's memory, so the assignment to the inner i in no way affects the outer i. The code prints 1.

2.2 Type

Each variable has a *type*. The type limits the values that a variable may hold. For example, an int variable can hold any integer between a certain minimum and maximum.

A *class object* is a variable whose type is a class.

2.3 Value

At each point in time during the execution of a program, a variable will have a *value*. One way to change this value is of course with an assignment. The assignment

```
i = 13 ;
```

changes the value of the variable named i to 13. Please read this as i 'becomes' 13. It drives me nuts to hear people say i 'equals' 13, when they mean 'becomes'.

The following statement prints the value of the variable named i.

```
cout << i ;
```

2.4 Size

Each variable requires a certain amount of space in the computer's memory. An int variable, for example, may require 4 bytes of memory[2]. If so, 4 contiguous bytes of memory will be allocated to hold the variable value. The sizeof operator is used to compute the *size* of a variable in bytes.

The following statement prints the size of the variable named i.

```
cout << sizeof( i ) ;
```

[2] The actual number varies depending on the type of computer and sometimes the compiler used.. Two, four and eight are the usual sizes for int variables.

2.5 Address

Each variable has an *address*. This is a pointer value that locates the variable in memory. You can think of the computer's memory as a big array of bytes. The address of a variable is the index, into this array, of the first byte allocated to hold the variable's value.

For example, if our int variable is allocated bytes 100, 101, 102, and 103. Then the address of the variable is 100.

The & operator is used to compute the address of a variable. For example

```
p = &v ;
```

Assigns to p the address of the variable named v. If the type of v is T, then the type of &v will be 'pointer to T'.

The following statement prints the address of the variable named i in hexadecimal notation.

```
cout << &i ;
```

3 Lifetimes

Variables are created, used, and then destroyed. After a variable is destroyed, the space it used may be allocated to another variable.

We can classify variables according to the circumstances under which they are created and destroyed. The classifications are: static variables, stack variables, and heap variables. Each variable fits into exactly one of these categories.

3.1 Static variables.

Static variables are:

- created when the program starts (i.e. before main is called).
- destroyed when the program terminates (e.g. on returning from main).

Any variable that is declared outside of a function will be static. Any variable declared inside a function or class will be static if its declaration is proceeded with the keyword static.

```
class C {
        static int a ; // Static member variable
        char b ; // Member variable (field).
        ... } ;
static float d ; // Static variable, but only usable in this file.
double e ; // Static variable, globally linked
void a_func() {
        static C f ; // Static variable
        C g ; // Stack variable. Not static.
        ... }
```

In this course, we hardly use static variables at all.

3.2 Stack variables.

Stack variables are

- created when their declaration is encountered.
- destroyed when the end of their scope is encountered.

Stack variables are also called *auto* variables which is short for *automatic*[3]. Any variable declared within a function, that is not preceded by the keyword static will be a stack variable.

For example, if the following code is executed:

```
{
      int i = 1;
      char j = 'a' ;
      float k = 3.14 ;
      cout << (i+j+k) ;
}
```

first a stack variable i is created, then a stack variable j is created, finally a stack variable k is created. When the final } is met, all three are destroyed.

Here is a more complex example:

```
{
      int i = 1;
      if( x ) {
            char j = 'a' ;
            while( j <= 'z' ) {
                  float k = 3.14 ;
                  ....
                  if( y ) {break ;}
                  ...
            } /* k is destroyed here */
            ...
            if( z ){ return ;}
            ...
      } /* j is destroyed here */
      ...
} /* i is destroyed here */
```

Note that, if the break statement is executed, then k will be destroyed, and that, if the return statement is executed, then both i and j will be destroyed, together with all other *stack* variables that were created within the function.

The next stack variable to be destroyed is always the most recently created. This is why they are called stack variables.

[3] Although, I prefer to think it is because they have planned obsolescence.

Function parameters are also stack variables.

There is one more kind of stack variable. Its existence can often be ignored, but I include it here for completeness. These are called 'temporaries'. They hold values returned from functions. For example, suppose we have a function add with the following declaration:

```
Matrix add(Matrix a, Matrix b) ;
```

Assume x, y, and z are variables of type Matrix. If the statement

```
z = add( x, y ) ;
```

is executed, the compiler will create a temporary variable to hold the result of the function call. Assuming Matrix is a class, the compiler will construct this temporary variable using the copy-constructor of Matrix. After the temporary is created, its value is copied to z using the assignment operator of the class. At some point after that, the temporary will be destroyed using the destructor of Matrix.

3.3 Heap variables

Heap variables are

- created by the new operator.
- destroyed by the delete operator.

Heap variables are unusual in that they do not have names. They must be accessed using their addresses. Usually we use a pointer variable to hold the address of a heap variable.

For example:

```
p = new(nothrow) Matrix ;
```

tries to create a heap variable of type Matrix and assigns its address to a pointer variable p.[4]

Heap variables are only destroyed when their address is sent to the delete command.

3.4 Fields and array elements

You can think of data members and array elements as being variables too. They are created when the class object or array that they are a part of is created, and are destroyed when it is destroyed. Just like a variable, they have types, sizes, and addresses. The name of a field F in a class object o is

$$o.\mathsf{F} \quad ;$$

the name of element number i in an array a is

$$a[i] \quad .$$

[4] I say 'tries to' because there may not be enough space in the heap to fit the new variable.

You may be wondering what this '(nothrow)' thing is after new. Bear with me; it will be explained in *a later section (§4.2.3)*. Until then you can just ignore the '(nothrow)'.

3.5 Function calls

It may be helpful to look at an example of variables being created and destroyed as functions are called.

Consider the following code

```
int g( int &r ) {
      float x = 99 ;
      r = 13 ;
      return 42 ; }
int f( int i ) {
      int x ;
      if( i != 0 )
            x = g( i ) ;
      else {
            i = 99 ;
            x = 86 ; }
      return i+x ; }
void main() {
      char x = 'A' ;
      cout << f( 1 ) << endl;
      int y = 63 ;
      cout << f( 0 ) << endl; }
```

Note that main calls f and f potentially calls g. All three subroutines have a variable called x.

The following table shows how the stack evolves as the program executes; time advances from top to bottom.

About to call f	x: 'A'								
Just called f	x: 'A'	temp: ??	i: 1	x: ??					
Just called g	x: 'A'	temp: ??	i:1	x: ??	temp: ??	r: address of i	x: ??		
While returning from g	x: 'A'	temp: ??	i: 13	x: ??	temp: 42	r: address of i	x: 99.0		
While returning from f	x: 'A'	temp: 55	i: 13	x: 42					
About to call f again	x: 'A'	y: 63							
Just called f again	x: 'A'	y: 63	temp: ??	i: 0	x: ??				
While returning from f	x: 'A'	y: 63	temp: 185	i: 99	x: 86				
End of main	x: 'A'	y: 63							

Each box[5] represents the variables belonging to a single function invocation. The temporary variables (labelled temp) are created to hold the result of each function call. Note that the variables belonging to the two invocations of f are entirely distinct; that is the i and x of the second invocation are entirely different from the identically named variables of the first invocation.

[5] If you are reading the HTML version of this file, the boxes won't be visible, so you'll have to imagine where they are.

4 Pointers

4.1 Some pointers on pointers.

For each type T there is another type 'pointer to T'. The address of any variable of type T will have this type, as will a null pointer (T*)0. Pointer variables are simply variables that have a pointer type as their type.

If the last value assigned to a pointer variable p was the address of a variable v, then we say that 'p points to v' and the expression *p acts just like the variable v. Thus if p points to v,

```
*p = 13 ;
```

assigns 13 to v and

```
x = *p + 12 ;
```

assigns 12 plus the value of v to x. The expression *p is called the *dereference* of p, and the computing of *p is called *dereferencing* p.

4.2 Allocating from the Heap

4.2.1 Space leaks

Heap variables are *not* destroyed when the pointer that points to them is destroyed. For example the code

```
{ Matrix *p = new(nothrow) Matrix ; }
```

creates a (stack-allocated) pointer variable named p; then it creates a heap-allocated Matrix object; then it destroys p. Now the Matrix object will never be destroyed. It can not be, because we have destroyed the only place we kept its address. This kind of bug is called a *space leak*. The undestroyed object typically does no harm except that it uses up room in the heap that could be used for something else. Programs with space leaks often eventually run out of space and either crash or stop working.

> **The rule of equal deletion.** *Delete every heap variable you create.*

4.2.2 Heap exhaustion.

It is possible to run out of room for heap variables if you create a lot of them, or you create some very big ones, or have space leaks. The good news is that you can detect this problem because the address returned by new will be a special invalid address called 'the null pointer'. The null pointer of type 'pointer to T' can be written as (T*)0, but usually you can simply write 0.

You must never assume that new will be successful. Think of it as a polite request, rather than a command. Thus you should always check to see if it was successful or unsuccessful. E.g. rather than

```
int *p = new(nothrow) int ;   //
*p = 13 ;                     // Wrong! p may be null.
etc                           //
```

do this

```
int *p = new(nothrow) int ;    //
if( p == 0 ) {                 // Right!
   deal with the failure       //
} else {                       //
   *p = 13 ;                   //
   etc                         //
}                              //
```

> **The rule of checking new.** *Check after each* new(nothrow) *to see if you got the null pointer back.*

4.2.3 So what *is* this (nothrow) thing? Or: The new new.

The International Standards Organization (ISO) recently changed a few things about C++. new is something they changed — though why, I'll never know. Prior to the ISO, if new failed to find space, it returned a null pointer. Now, with ISO C++, if new fails to find space something happens called '*throwing an exception*' instead. See Section *Exceptions (§11.2)*, if you want to know what '*throwing an exception*' means — for now you don't have to. Luckily the ISO left a loop-hole. If you want the 'old fashioned' behaviour of getting back a null pointer, all you have to do is:

- include a standard header file called new, by putting the lines

  ```
  #include <new>
  using namespace std ;
  ```

  at the top of your source file[6], and
- write new(nothrow) instead of just new.

In this course we will use new(nothrow). The reason is simple; we want to keep up with the ISO's version of C++, but the compiler we will use does not support the new 'throwing' behaviour of new. Furthermore, until all compilers support the new 'throwing' behaviour of new, code written blithely assuming the new new behaviour will be less portable. I recommend using (nothrow) for at least the next few years.

> **The rule of nothrow.** *Include standard header* new *and use only* new(nothrow).

4.2.4 Heap allocated arrays.

You can allocate a whole array at once with statements like:

```
T *p ;
p = new(nothrow) T[i] ;
```

[6] The using namespace std; declaration is explained in *A later section (§9.1)*

where i is an integer (greater or equal to 0). This allocates an array with i elements and sets p to point to the first element of the array. Later, if p still points to the first element of the array, we can destroy the array with the statement

```
delete [] p ;
```

The C++ system will have remembered how big the array is, so there is no need to tell it again.

4.3 When good pointers go bad

What happens if p does not point to a variable? I can assure you nothing good will come of executing the expression *p in this circumstance. Unlike syntax errors, the compiler will likely not give you any warning. In fact errors of this sort are very hard to catch, even through debugging. There are (at least) four ways to create a pointer variable that does not point to a variable:

1. p has not been initialized. For example duplicating a 'string':

   ```
   char *targ ;
   while( *source != 0 ) {
       *targ = *source ; // targ is not initialized!
       targ += 1 ;      source += 1 ; }
   *targ = (char)0 ;
   ```

   The correct thing to do would be to create an array (either heap or stack — as appropriate) to hold the string and initialize targ to point to the first element of that array (see next point).

2. p was obtained from pointer arithmetic that went beyond the bounds of the array. For example in fixing the previous example, we might write:

   ```
   char a[10] ; // Make space
   char *targ = & a[0] ; //Initialize the pointer this time.
   while( *source != 0 ) {
       *targ = *source ;
       targ += 1 ;      source += 1 ; }
   *targ = (char)0 ;
   ```

   If the length of the 'string' pointed to by the initial value of source is longer than 10 characters (including the 0 byte that marks its end), then the dereference of targ will eventually be invalid. And if the array source points into does not have a 0 byte in it, then the dereference of source will eventually be invalid.

3. p holds the null pointer.

4. The variable p points to has been destroyed.

   Here is an example where p pointed to a heap variable

   ```
   Matrix * q = new(nothrow) Matrix[10] ;
   Matrix *p = & q[4] ;
   ...
   delete [] q ;
   ...
   *p = a ; // Bad pointer!
   ```

The first line allocates an array of 10 Matrixes on the heap. The second line creates p and sets it to point to the 5th of them. I'll assume there are no assignments to p or q hidden in the ellipses. The delete line destroys all 10 Matrixes. The final line is in error because it makes reference to a variable that has been destroyed.

Here is an example where a stack allocated variable is referred to after it has been destroyed. Suppose there is a function that returns a pointer to a Matrix, looking something like this:

```
Matrix *f() {
    Matrix A ;
    ...
    return &A ; /* Bad idea! We are about to destroy A */
}
```

If we make a call

```
p = f() ;
```

we are assigning to p the address of a destroyed variable. If we then dereference p, it would be an error.

A pointer that points to a variable that has been previously destroyed is called a 'dangling reference'.

We can summarize these cases with a rule

> **The rule of good pointers.** *Never dereference a pointer unless you can prove that it points to a live variable. Watch out especially for uninitalized pointers, array bounds, null pointers, and dangling references.*

Violations of this rule lead to very hard to find errors. If you dereference a bad pointer, there is no guarantee your program will crash or do anything else to call attention to the bug. Often the program even behaves as you wish, and the bug only has an effect when an unrelated change is made to the program.

4.4 Pointers and arrays

Pointer variables and arrays get mixed up in the minds of many C/C++ programmers. The reason for this is that arrays sometimes act like pointers and pointers sometimes act like arrays. It is important to keep in mind the differences.

- Declaring an array allocates storage for a number of things. For example

  ```
  int a[10] ;
  ```

  allocates space for 10 ints, whereas

  ```
  int *p ;
  ```

  allocates space to hold one address, but not to hold any ints.

- Pointer variables can be changed. Arrays are not variables (in C and C++). You can write

  ```
  p = a ;
  ```

  but not

  ```
  a = p ;
  ```

  So why do they get confused?

- First, in almost every context, the name of an array is treated the same as a pointer to the first element of the array. This is why you can write 'p = a;', it is the same as

  ```
  p = & a[0] ;
  ```

  I.e. p is assigned the address of the first element of a. It is particularly important to realize that this happens in parameter passing. If you declare a function

  ```
  void f(int *p) {
      ....
  }
  ```

  You may call it with a call f(a). This is the same as f(& a[0]). This is why arrays are sometimes said to be passed 'by reference' in C and C++. Even if you declare f as

  ```
  void f(int p[10]) {
      ....
  }
  ```

  p is still a pointer, not an array! *There are no array parameters in C++ — only pointer parameters that are declared like arrays.*

- Second, pointers can be subscripted like arrays. If we have assigned our pointer with

  ```
  p = & a[0] ;
  ```

  so that it points to element zero of array a, then p[i] refers to element i of array a. p looks a lot like an array here, but that doesn't mean it is one; it isn't. The notation p[i] simply means *(p + i). The p+i part computes a pointer that points to the array element i places to the right of the one p points to. Then that pointer is dereferenced.

4.5 Pointers and 'strings'.

Pointers to characters often are often mixed up with "strings". One common way to represent a sequence of characters is to set aside some room for them in an array and refer to them by a pointer to the first character in that array. For example

```
char a[4] ;
a[0] = 'a' ; a[1] = 'b' ; a[2] = 'c' ; a[3] = (char)0 ;
char *p = & a[0] ;
```

The end of the sequence is marked with a special character (char)0, whose ASCII code is zero. (This character should not be confused with the character '0' which represents the digit 'zero';

its ASCII code is 48.) Character sequences represented in this way are commonly referred to as a *strings* or a *C-style strings*. Many library subroutines assume this representation is being used.

When the C++ compiler sees a quoted string as in

```
p = "abc";
```

it creates a special array to hold the four —yes, **four**— characters ('a', 'b', 'c', and (char)0) and the value of the expression "abc" is the address of the first member of this array.

Don't mix up the empty string and the null pointer. The statement

```
p = "";
```

sets p to be a pointer to the only character of a one character array containing the zero character ((char)0). Whereas

```
q = (char*)0 ;
```

sets q to a null pointer. After these assignments, it is legitimate to dereference p —*p is (char)0— but not to dereference q —it is a null pointer.

5 References

Two things make references confusing: (a) They are so much like pointers, you can easily forget they are different. (b) They are so unlike all other kinds of variables.

A reference variable r, is another name for some other variable. For example if I write

```
int i = 10 ;
int &r = i ;
```

then that declares r to be a second name for the variable named i. We can now write

```
r = 13 ;
cout << i ;
```

and that will print 13.

Now the above code is pretty silly because there is no advantage to calling i by any other name. A more useful use of references is when you have a long complicated name that you want to use several times:

```
{
    int &topLN = data->stack[ top_pointer ].name.last ;
    topLn is used several times
}
```

Of course you can achieve the same effect using a pointer variable:

```
{
    int ptopLN = & (data->stack[ top_pointer ].name.last ) ;
```

```
    *ptopLn is used several times
}
```

In fact that is exactly how reference variables are typically implemented by the compiler.The following three code snippets mean the same thing

{int &**r** = i ;
x = r+2 ;
r = 2*y ;
}
	{int ***p** = &i ;
	x = *p + 2 ;
	*p = 2*y ;
	}
		{
		x = i + 2 ;
		i = 2*y ;
		}

All three will likely be compiled to the same machine code. References are like pointers, but are more limited. You must initialize a reference, and you can never change which variable the reference refers to.

References are used almost exclusively for two purposes: parameter passing and result returning.

Reference parameters are simply reference variables that are parameters. They are initialized by the function call. The following two functions differ only in how they are called.

// Pass by reference	*// Pass by pointer*

void **r_double**(int &**r**) {
r = r+r ;
}
	void **p_double**(int ***p**) {
	*p = *p + *p ;
	}

The first is called like this '`r_double( i )`' and the second like this '`p_double( &i )`'.

Putting the word `const` in the declaration of a reference variable is a promise not to assign to the referenced variable. Consider the function

```
// Pass by const reference
float r_determinant(const Matrix &m) {
    Etc
}
```

We can tell from the declaration that it won't change the argument. The same is true if the `Matrix` is passed by value:

```
// Pass by value
float v_determanant(Matrix m) {
    Same body as r_determinant.
}
```

So why is `r_determinant` better? The call `v_determinant(a)` requires calling the copy-constructor of `Matrix` to initialize `m` with `a`'s value. But the call `r_determinant(a)` only requires the finding of `a`'s address to initialize the reference `m`. For large objects, this can be a considerable run-time saving. For integers and other small objects, passing by value is more efficient.

References are also used in return values. In this course, we will only have to do this for defining assignment operators. The usual form of an assignment operator for a class `C` is

```
C & operator=(const C &r) {
    if( the object is not being assigned itself ) {
        Finalize old value of this object and
        copy the value of r to this object. }
    return *this ;
}
```

(See section 8.1 for a description of the this pointer.)

Use references to return values with great care. Consider:

```
Matrix &f() {
    Matrix A ;
    ...
    return A ; /* Bad idea! */
}
```

Yes, this saves a call to the copy constructor. **But** it creates a *dangling reference*. The stack variable, A, to which the result reference refers will be destroyed as part of the return. The only reason you can get away with it in the assignment operator is that, if *this is a stack variable, it will have a longer life than the temporary reference used to return it. My advice is only use return by reference when you are sure the object the reference refers to will be destroyed after the temporary reference used to return it.

6 Functions

Functions are either member functions or nonmember functions. Member functions are associated with one class when defined and with one object (called the *recipient*) when called.

You provide the compiler information about functions in two ways. With *declarations* and *definitions*. The declaration tells the compiler about the type of the function. The definition tells the compiler about the implementation.

6.1 Function Declarations

Here are declarations of some nonmember functions:

```
int f(char p) ;
Matrix operator * (const Matrix &a, const Matrix &b) ; /* Binary operator*/
float operator ~(const Matrix &a) ; /* Unary operator*/
```

Declarations of member functions (a.k.a. methods) are placed within a class construct. E.g.[7]

[7] The word const at the end of the declaration of a member function indicates that the recipient object will not be changed.

```
class Matrix {
        public:
            void assign(int i, int j, float val) ;
            Matrix operator + (const Matrix &b) const; /* Binary operator*/
            Matrix operator ! () const ; /* Unary operator*/
} ;
```

6.2 Function Definitions

Function definitions supply the body of the function. Definitions of nonmember functions are straight-forward.

```
int f(char p)
{
        ...
}
Matrix operator * (const Matrix &a, const Matrix &b) /* Binary operator*/
{
        ...
}
float operator ~(const Matrix &a) /* Unary operator*/
{
        ...
}
```

Definitions of member functions require an indication of what class the function is a member of. This is done by prefixing the function's name with the class's name and a ::.

```
void Matrix::assign(int i, int j, float val)
{
        ...
}
Matrix Matrix::operator + (const Matrix &b) const /* Binary operator */
{
        ...
}
Matrix Matrix::operator ! () const /* Unary operator*/
{
        ...
}
```

7 Declaration syntax.

7.1 Declare it as you use it

People get confused by the C/C++ declaration syntax. This is not surprising as it is mad. But there is a method to its madness. The slogan is 'declare it as you use it.'

For example, if you want the expressions i, *p, a[i], (*q)[3], and *((*z[4])(5)) to be of type int. Then you declare the variables i, p, q, and z as

```
int i ;
int *p ;
int (*q)[10] ;
int *((*z[10])(int)) ;
```

What do these mean? Take q as an example. Since (*q)[i] will be an int, *q must be an array of ints, and thus q is a pointer to an array of 10 ints.

How about z? We can reason

*((*z[i])(i))	is an int, so
(*z[i])(i)	is a pointer to an int, so
*z[i]	is a function taking an integer and returning a pointer to an int, so
z[i]	is a pointer to a function taking an integer and returning a pointer to an int, so
z	is an array of 10 pointers to functions taking integers and returning pointers to ints.

The same goes for function declarations and definitions:

```
int (*(f(int i,char c)))[10] ;
```

declares f to be a function that takes an int and a char and returns a pointer to an array of 10 ints.

References are an exception to the 'declare is as you use it' rule:

```
int &r ;
int *&rp ;
```

declare r to be a reference to an int and rp to be a reference to a pointer to an int, but this does not imply that either &r or *&rp are ints.

7.2 Typedefs

It is annoying to write crazy things, like the definitions of q, f, and z above, more than you have to. This is why the typedef declaration is nice. The declaration

```
typedef int (*p_array10_int)[10] ;
```

looks like a variable declaration, except for the keyword typedef. That extra keyword tells the compiler to create a new type name representing the type the variable would have had, if it were a real variable declaration. I.e. after the typedef has been processed p_array10_int will be a type name meaning "pointer to an array of 10 ints". Thus after this declaration has been read by the compiler, you can write

```
p_array10_int q;
p_array10_int f(int i, char c) ;
```

and these will have the same effect as the previous declarations of q and f.

In this course we won't use typedefs much because they don't seem to mix well with templates.

7.3 const

The keyword const can be confusing. At least it always confuses me. It declares that a variable is constant and hence will not change. Thus

```
const float pi=3.14159 ;
```

declares a variable pi that will always have this one value. If you later write

```
pi = 3.0 ;
```

the compiler will spot your mistake. The declaration

```
int *const p = & i ;
```

declares a pointer variable p that will always point to i. Note that it is the pointer, not the integer that will not change. Sometimes it is necessary to promise that a pointer will be used to read the value of a variable, but never to change it. Such a pointer is declared as

```
const int *p ;
```

7.4 A Confusing Convention

Most C++ coders write

```
int* p ;
```

rather than

```
int *p ;
```

Of course these mean the same thing. But what does

```
int* p, q ;
```

mean? Don't be fooled by where the * is. This declares q to be an int not a pointer to int.

You also get some choice about where you write const. You can write

```
const int num_letter=26, num_vowel=5 ;
```

or

```
int const num_letter=26, num_vowel=5 ;
```

They mean the same thing — both variables are const.

8 Classes

8.1 Members

Classes give the programmer the opportunity to define his/her own types. Each class object[8] consists of a number of *fields* (a.k.a. *data members*) and *methods* (a.k.a. *function members*). The fields contain data, and the methods access and change the fields.

Each member (whether data or function) is categorized as either public, protected, or private. In this course, we won't need protected.

The public members of an object named o may be accessed in any code that o can be accessed in. The name of a member m of an object o is o.m. Sometimes the class object will not have a name (e.g. if it is a heap variable). As long as there is a pointer p to it, we can still write (*p).m. But this is awkward, so C++ provides an alternate syntax: p->m.

The private members of a class object can only be accessed from the code that is in the bodies of member functions of the class.

When a function member f of an object o is called (e.g. with a call like o.f()), o is termed the *recipient*. Within the body of the function f you will likely want to refer to the members of the recipient. Any mention of a member name m will refer to a member of the recipient, o. Occasionally you will want to refer to the recipient, o, itself. The keyword this is always a pointer to the recipient, so you can refer to the recipient as *this.[9]

Generally speaking you should declare all fields as private or protected. Those methods you declare as public will constitute the *public interface* of the class.

8.2 Structs

Structs are just classes where members are public unless declared otherwise[10]. The following declarations are the same

```
struct listElem {
    int data ;
    listElem *next ;
} ;
```

```
class listElem {
    public :
        int data ;
        listElem *next ;
} ;
```

8.3 Special methods

8.3.1 Constructors

When a class object is created, one of its constructor methods will be used to initialize its fields. The name of a constructor method is the same as the class and it is declared with no return type at all (not even void!). For example:

[8] A *class object* is just an object that has a class as its type.

[9] As a result you can refer to a member m as this->m. This is long-winded, but it emphasizes that m is a member of the recipient, rather than a local variable of the function.

[10] Structs are hold-over from C, which does not have classes. C also does not have private members, inheritance, or function members.

```
class point {
      public:
           point() ; /* The default constructor */
           point( const point &p ) ; /* The copy constructor */
           point(float a, float b) ; /* Another constructor */
           ...
      private:
           float x, y ;
}
```

declares a class with three constructors. Which one gets called depends on how the object is created:

```
point w ; /* Default constructor */
point x = w ; /* Copy constructor. Not the assignment operator. */
point y( x ) ; /* Copy constructor*/
point z(1.0, 2.7) ; /* The other constructor */
```

By the way, the declaration

```
point f() ; /* Don't do this by mistake */
```

does **not** invoke the default constructor. In fact it declares f to be a function that takes no parameters and returns a point.

One problem with constructors is what value the fields will have when they begin. Normally a default constructor is first used on each field whose type is a class. For example, if we define:

```
class line {
      public:
           line( const point &a, const point &b ) ;
      private:
           point p0, p1 ;
}
line::line( const point &a, const point &b ) {
      p0 = a ;
      p1 = b ;
}
```

Point's default constructor will be invoked for both p0 and p1 before the beginning of line's constructor, which promptly overwrites the values just constructed. It would be slightly more efficient to use point's copy constructor. Indeed, C++ provides a way to do exactly that. We define

```
line::line( const point &a, const point &b ) :
      p0(a),
      p1(b) {
}
```

The stuff between the colon and the left brace is a list of fields to be constructed and argument lists for their constructors. Any fields left out of the list will still be constructed by their default constructor.

8.3.2 Destructors

Each class has exactly one destructor. This method is called when the object is about to be destroyed (e.g. when a stack object goes out of scope, or a heap object is deleted). Immediately after the destructor is finished, each field of the object is destroyed. The destructor's name is ~C where C is the class name. Like a constructor, the destructor should be declared with no return type.

8.3.3 Compiler generated methods

There are certain methods that every class will have, whether or not you define them. If you don't define them, the compiler will supply definitions.

- **The default constructor.** This is what gets called when a class object is created, but there are no arguments given to the constructor. For example

  ```
  Matrix a ;
  p = new(nothrow) Matrix;
  q = new(nothrow) Matrix[10] ;
  ```

  The default constructor is called to construct the stack variable a, the heap variable *p, and the ten heap variables q[0],.., q[9].

- **The copy constructor**. This is what gets called when a class object is created and initialized with a value of the same type. Consider for example

  ```
  Matrix a = x ;
  Matrix b(x) ;
  p = new(nothrow) Matrix(x);
  ```

  Assuming x has type Matrix, the copy constructor is called in each of these cases.

  The copy constructor is also used for parameter passing and returning a value from a function. Consider a function declared as

  ```
  Matrix transpose(Matrix a) ;
  ```

  and the function call:

  ```
  x = transpose( y ) ;
  ```

  The copy constructor is called to copy the value of y into the newly created parameter a. When the return in the function is executed, the copy constructor is called again to initialize

a temporary variable with the returned value. The assignment operator is then used to copy from the temporary variable to x.

- **The assignment operator.** This is called when an assignment (with =) is made to a previously created class object.
- **The destructor.** This is called when the class object is destroyed.

If you do not declare these methods, the compiler will create them for you.[11] It is good to know what the compiler generated methods will do, since it is often not appropriate.

- **The default constructor.** The compiler-generated default constructor will invoke a default constructor on each field that has a class as its type. Fields that do not have a class as their types (e.g. int fields or pointer fields) are left uninitialized.
- **The copy constructor.** The compiler-generated copy constructor copies each field of the old object to the corresponding field of the new object. For fields that have a class for their type, this copying is done with a copy constructor. For other fields, it is done with assignment.
- **The assignment operator**. The compiler-generated assignment operator copies each field of the old object to the corresponding field of the new object. This copying is done with assignment operators.
- **The destructor.** The compiler-generated destructor does nothing.[12]

Here is an example class

```
class example {
    public:
        int i_ ;
        Matrix mat_ ; } ;
```

If I were to be explicit and not leave the generation of any methods up to the compiler, the class would look like this:

[11] Here is the whole truth: The compiler will generate definitions for these methods *only if* your code actually uses the methods. And the compiler will generate the default constructor, *only if* you did not declare any constructors at all. Finally, there are two more methods the compiler will generate for you, these a const and nonconst versions of the 'address-of" operator &.

[12] But remember after the destructor is done, each field that has a destructor is destructed.

```
class example {
    public:
        Matrix mat_ ;
        int i_ ;
        example() ; // Default constructor
        example( const example &r ) ; // Copy constructor
        example& operator= ( const example &r ) ; // Assignment operator
        ~example() ; // Destructor
} ;
```

```
example::example() // Default Constructor
    : mat_ () // Construct the Matrix field
    { } // The int field is not initialized
```

```
example::example( const example &r ) // Copy constructor
    : mat_ ( r.mat_ ) // Copy construct the Matrix
    {
        i_ = r.i_ ; // Copy the int
    }
```

```
example& example::operator= ( const example &r ) // Assignment operator
    {
        mat_ = r.mat_ ; // Assign using the assignment operator of Matrix class
        i_ = r.i_ ; // Copy the int
    }
```

```
example:: ~example() // Destructor
    { } // Do nothing, but mat_ will be destructed after.
```

When the implementation of an object involves heap-allocated storage, these compiler-generated methods are usually wrong and harmful, as is shown in the following case study.

8.3.4 Case study for the compiler-generated methods

Consider this class for representing character strings

```
class String {
      public:
            ⋮
      private:
            int len ;
            char *c ;
} ;
```

The idea is to allocate just as many bytes from the heap as are needed to represent the string. The number of characters in the string will be len and c will point to the first in an array of len bytes on the heap.

Let's start with the destructor. The compiler generated destructor will do nothing. When a String is destroyed, the last pointer to the heap allocated array (i.e. the c field) will be destroyed. This is a **space leak**. The solution is to declare the destructor by adding the line

```
            ~String() ;
```

to the public part of the class and defining the destructor function as

```
String:: ~String() {
      delete c[] ;
}
```

But there is a problem. Consider

```
b already exists
{ String a = b ;
      use a for a while
} /* a is destroyed here */
use b here
```

a will be created using the copy constructor. The compiler generated copy constructor will simply copy over each field.

```
a.len = b.len ;
a.c = b.c ;
```

Thus after the constructor is done, a.c and b.c will both point to the (first element of the) same array. This may seem like a big space saving, but there is a problem. When a is destroyed, according to the destructor we just wrote, the array that a.c points to will be deleted. But this is also the array that b.c points to! That is, b.c, will become a **dangling reference**! We can not accept the compiler generated copy constructor and must write our own. Add

```
        String (const String &b) ;
```

to the public part of the class and define the copy constructor

```
String:: String(const String &b) {
    len = b.len ;
    c = new(nothrow) char[len] ;
    for(int i=0 ; i < len ; ++i) { c[i] = b.c[i] ; }
}
```

Suppose we assign

```
a = b ;
```

The compiler generated assignment operator amounts to

```
a.len = b.len ;
a.c = b.c ;
```

We have the same problem as with the copy constructor. The solution is to declare and define an assignment operator for the class. Add

```
        const String & String::operator = (const String &b) ;
```

to the public part of the class and define the assignment operator

```
String & operator = (const String &b) {
    if( &b != this ) {
        delete c[] ;
        len = b.len ;
        c = new(nothrow) char[len] ;
        for(int i=0 ; i < len ; ++i) { c[i] = b.c[i] ; } }
    return *this ;
}
```

Why is there a delete? Well, if c already has a value, then it would be bad to simply overwrite it; that would again be a space leak. (Note that the assignment operator does nothing when an object is assigned to itself. Why did I do this?)

Finally consider the sequence

```
String a ;
a = b ;
```

It looks innocuous, but look at what happens in the assignment. The first thing that happens is equivalent to

```
delete (a.c)[] ;
```

But the compiler-generated default constructor will not have initialized a.c. We are asking the C++ run-time system to delete an arbitrary variable. If we are lucky, this will crash immediately, but we might have a bug that will be latent for a long time. We must ensure that the c field is initialized to something, so we must write our own default constructor. We might as well initialize to the empty string. Add

```
        String() ;
```

to the public part of the class and define

```
String:: String() {
        len = 0 ;
        c = (char*)0;
}
```

We should ask whether delete ((char*)0) [] is allowed. It turns out that it is, and is guaranteed to cause no harm.

> **The rule of compiler-generated methods.** *If your class contains pointers to heap variables, you probably need to declare and define all four compiler-generated methods.*

(The alert reader will have noticed that in this example, I repeatedly broke "the rule of checking new". If I were writing code to use in a product, rather than to illustrate the above points, I'd have been careful to check for null pointers.).

8.3.5 Avoiding compiler generated methods

There is a sneaky trick you can play on the compiler, if you want to avoid writing a special method and you don't want the compiler to generate one for you either: Declare the method, but don't define it! There is no obligation to define any method, if it is never called. But what if someone does call the method? Well they will get a nearly incomprehensible error message when their program is linked. To give them a better error message and to make it come at compile time, simply declare the method as private.

Here is an example:

```
class Matrix {
      public:
           Matrix( int rowNum, int colNum ) ;
           Matrix(const Matrix &b) ;
           const Matrix & Matrix::operator = (const Matrix &b) ;
           ~Matrix() ;
           double get( int i, int j) ;
           void set( double val, int i, int j ) ;
      private:
           Matrix() ; // Never defined!!
           int rNum, cNum ;
           double *data ;
} ;
```

In this case, I did not want to write a default constructor for my class because there is no obvious default size for a matrix.

8.4 Inheritance

In this course we will not be using inheritance very much, so I'll only cover as much as we need. There is much more that I won't mention.

8.4.1 Deriving classes.

Inheritance is used to create new classes (*derived classes*) from old ones (*base classes*). The derived class will have all the fields and methods of the base class and also whatever new fields and methods it adds.

For example consider

```
class Shape
{
      public: Shape() ;
      public: void move( int delx, int dely ) ;
      public: int getX() ;
      public: int getY() ;
      private: int xcoord, ycoord ;
} ;
class Circle : public Shape
{
      public: Circle() ;
      public: void expand( float ratio ) ;
      private: int radius ;
} ;
```

This declares two classes. The Circle class has private fields xcoord, ycoord, and radius, it also has public methods move, getX, getY, and expand, as well as a default constructor, copy constructor, destructor, and assignment operator. Similarly, we could create classes for rectangles, triangles, text shapes, etc. The move, getX, and getY functions need only be written once.

Inheritance is more than just a way to create classes quickly, it is also a way to write "generic code", i.e. code that can work on data of many types. Consider a function

```
void up( Shape &sh ) {
      sh.move( 1, 0 ) ;
}
```

We can call this function with any object whose type is Shape or is derived from Shape. E.g.

```
up( aCircle )
```

8.4.2 Slicing

One error that it is possible to make with inheritence is called slicing. When a variable of type Shape is created there is only enough space set aside for the fields of the Shape class. Suppose you try to store a Circle in a shape variable using either Shape's copy constructor or its assignment operator:

```
Circle circ ;
...
Shape sh( circ ) ; // Copy constructor.
...
sh = circ ; // Assignment operator.
```

What will happen? The radius field will be completely lost. Furthermore, the compiler will give no warning that information is being lost in the copy.

8.4.3 Virtual Functions

The implementation of the move function is the same regardless of the shape. However, sometimes you need to declare a function in the base class that will be implemented differently for each derived class. In our example, a function that draws the shape on a computer screen is such a function. The algorithm for drawing a circle is quite different from the algorithm for drawing a rectangle. We add declarations of a method called draw:

```
class Shape
{
        ...
        public: virtual void draw() = 0 ; // draw is declared in the base class.
        ...
} ;
class Circle : public Shape
{
        ...
        public: virtual void draw() ; // draw is declared in the derived class.
        ...
} ;
```

The draw method is defined for the Circle class:

```
void Circle::draw()
{
        ...
}
```

and for rectangle, triangle, etc., but not for the base class Shape.

You can now call the draw routine of any Circle, Rectangle, Triangle etc., without even knowing what the exact type of the object is. Consider this routine

```
void drawAllShapes( Shape* shapeArray[], int arraySize ) {
        for( int i=0 ; i < arraySize ; ++i )
                shapeArray[i]->draw() ;
}
```

Now you can fill up an array with pointers to Circles, Rectangles, Triangles, etc., and call the above routine to draw them all. For each object, the appropriate definition of draw will be executed. This is the beauty of virtual functions.

8.4.4 Abstract Classes

The declaration of draw in the Shape class looks a bit odd. The =0 at its end indicates that you do not intend to define draw for the Shape class; the compiler will complain if you try. Such a function declaration is called a *pure virtual function declaration*. The presence of at least one pure virtual function declaration marks a class as an *abstract class*. You can not create objects that belong to abstract classes. This is what I want, in this case, because it does not make sense to have an object that is simply a shape and not some particular kind of shape. You can not write

```
Shape sh ; // Create a shape on the stack. NOT ALLOWED
```

Nor can you write

```
Shape *shP = new(nothrow) Shape ; // Create a shape on the heap. NOT ALLOWED
```

But you can write

```
Shape *shP = new(nothrow) Circle ; // Create a circle on the heap. ALLOWED
```

I recommend using virtual functions unless you are 100% sure that all derived classes will share the same implementation of a function[13]. Use pure virtual function declarations when there is no sensible implementation of a function for the base class.

9 Namespaces and the standard library

9.1 Namespaces

When a group of people come together there is sometimes a name-clash: does the name 'John' refer to John Quaicoe or John Robinson? We can resolve the clash by using family names. Likewise in a large program the same name may be used for several different purposes. *Namespaces* provide a way to group related things (classes, functions, static variables, typedefs) under a single name —the namespace name— that serves as a sort of family name. Here is an example:

```
namespace Bird {
        ...
        class Duck { ... } ;
        ...
}
namespace Action {
        ...
        class Duck { ... } ;
        ...
}
```

After these declarations we can write either

```
Bird::Duck
```

or

```
Action::Duck
```

there is no name-clash.

9.2 The Standard Library

The ISO has defined a standard library full of useful definitions. All functions and classes defined in the ISO standard library are defined in a name space called std. So the classic hello-world program that used to be written:

[13] A special case is when you are 100% sure that a class will not be used as a base class.

```
#include <iostream.h>
int main( ) {
      cout << "Hello world\n" ;
      return 0 ; }
```

is now written:

```
#include <iostream>
int main( ) {
      std::cout << "Hello world\n" ;
      return 0 ; }
```

There are two differences. Where input/output used to be declared in "iostream.h", the ISO library name does not have the ".h" part. Second is the use of namespace std in std::cout. These are minor differences, but in this course we will use the ISO library because it is more standard and has more stuff in it.

The dropping of the ".h" is consistent throughout the standard library, as is the use of namespace std. Some of the headers available are

<iostream>	Input/output
<string>	A string class
<complex>	Complex numbers
<new>	Declares nothrow
<cmath>	The C math library
<cstring>	Functions supporting C-style strings.

Writing std:: in front of all these standard names gets to be a pain. There is a solution. We can declare that we want to be able to use all the names in a namespace without mentioning the namespace. The declaration is

```
using namespace X ;
```

where X is a namespace's name. For example, the hello-world program can be written:

```
#include <iostream>
using namespace std ;
int main( ) {
      cout << "Hello world\n" ;
      return 0 ; }
```

10 Templates

10.1 Template classes

Sometimes you want a number of classes that differ only in the types of certain fields. It is tedious to write the same definitions over and over again with only minor variations. It is better to write one definition that works for all types.[14] Such a definition is called *generic*. This is what C++'s template construct is for.

A template class definition is like a function from types to classes.

The syntax is

```
template <class T> class C ... ;
```

where T (called the parameter type) could be any identifier. After this declaration has been processed by the compiler you can declare a variable

```
C<X> v ;
```

where X is any type. The type of v is a class whose definition is obtained by substituting X for T throughout the definition of C. In general, you can use C<X> anywhere you can use a type name: for declaring variables, function parameters, function return types and so on. The type C<X> is called 'C instantiated with X'.

For example, consider this template class definition:

```
template <class L, class R>
class Pair {
    public:
        Pair(const Pair &p) ;
        Pair(L leftInit, R rightInit) ;
        L getLeft() ;
        R getRight() ;
    private:
        L left;
        R right ;
} ;
```

We must define the four member functions. Notice that we must instantiate Pair everywhere except when it is being used as the name of a constructor function.

```
template <class L, class R>
Pair<L, R>::Pair( const Pair<L,R> &p ) :
    left(p.left),
    right(p.right)
{
}
```

[14] Or at least for several types.

```
template <class L, class R>
Pair<L, R>::Pair( L leftInit, R rightInit ) :
        left(leftInit),
        right(rightInit)
{
}
```

```
template <class L, class R>
L Pair<L,R>::getLeft() {
        return left ;
}
```

```
template <class L, class R>
R Pair<L,R>::getRight() {
        return right ;
}
```

Now if there is a declaration

```
Pair<int,float> q(13, 3.1459 ) ;
```

it is as if q were defined to have a type PairIF with the following definition

```
class PairIF {
        public:
                PairIF(const PairIF &p) ;
                PairIF(int leftInit, float rightInit) ;
                int getLeft() ;
                float getRight() ;
        private:
                int left;
                float right ;
} ;
```

```
PairIF::PairIF( const PairIF &p ) :
        left(p.left),
        right(p.right)
        {
}
```

```
PairIF::PairIF( int leftInit, float rightInit ) :
        left(leftInit),
        right(rightInit)
        {
}
```

```
int PairIF::getLeft() {
        return left ;
}
```

```
float PairIF::getRight() {
        return right ;
}
```

Since instantiations of a template class can be used anywhere types can be used, they can be used as arguments to instantiate another template class, or even the same one:

```
Pair<Pair<int,char>,Pair<float,Matrix> > weird(a,b) ;
```

Note that I had to put a space between the two successive >'s so the compiler wouldn't think I was writing the >> operator.

10.2 Template Functions

One can also write generic functions using a similar syntax. In fact, we've already seen that the member functions of a template class must be written as template functions.

11 Undesired events.

Sometimes something happens that requires the program to radically alter what it is doing. Two examples that we will encounter in this course are: running out of memory, and the detection of programmer error.

There are various things you can do when such an undesired event occurs. Which is chosen depends a lot on what the undesired event was and what the nature of the application is. In some cases it is reasonable to stop and print an informative message, for other programs stopping should never be an option.[15]

[15] Consider a rocket guidance system. There are two computers —a primary and a backup. In the primary, shutting down might be a reasonable response to some undesired events. This is because the backup should take over when the primary shuts down. In the backup, shutting down should never be an option. The first launch of an Ariane 5 rocket ended in a crash, costing over 650 million $, when both the primary and backup guidance computers shut down in response to a detected programmer error.

In this course we will use two different mechanisms for dealing with undesired events. For programmer error, we will use the "assert" macro, and for running out of memory, we will use exceptions.

11.1 The "assert" macro.

The assert macro is called with a boolean argument.

```
assert( x );
```

is roughly equivalent to

```
if( x ) {
    /* do nothing */ }
else {
    Print a message and stop the program. }
```

It is usually used to document the beliefs of the programmer. For example, if a programmer is writing a function to compute the integer part of the square root of a number, she/he might write.

```
#include <assert.h>
:
/* nat_root( i ) The integer part of the square root of i.
     Precondition: Should only be called with a nonnegative argument.
     Postcondition: Returns the integer part of the square root of i */
int nat_root( int i) {
      assert( i >= 0 ) ;
      int r ;
      compute the integer part of the square root of i into r.
      assert( r*r <= i && i < (r+1)*(r+1) ) ;
      return r ;
}
```

The first assert detects the error of calling the routine with a negative argument. The second detects any errors in computing the square root. Both these potential errors are programmer errors — either on the part of the programmer who called the function, or the programmer who wrote it.

Asserts are best used as an executable complement to the documentation of preconditions, postconditions, and invariants.

Asserts should not be used to detect undesired events that can be anticipated to occur in production use of a program (e.g. after it is sold). Examples are running out of memory and incorrect user input.

11.2 Exceptions

Throwing an exception causes the program to stop what it is doing and jump to a piece of code called an 'exception handler'. Exception handlers are declared and used like this:

```
void g() {
      Matrix foo ;
      Some code g0
      if( i==1 ) { throw 10 ; }
      Some code g1
}

void f() {
      try {
            some code f0
            g() ;
            some code f1 }
      catch (float &f) {
            cout << "Caught a float "<< f << '\n'; }
      catch (int &i) {
            cout << "Caught an int "<< i <<'\n'; }
      some code f2
}
```

Assuming no throws are executed, the following sequence can be expected to happen during the execution of f:

f0 g0 g1 f1 f2

Note that the output statements in the catch clauses are not executed. But what if i==1 after *f0 g0*? Then the statement

```
throw 10 ;
```

is executed. This causes the subroutine to be returned from immediately (thus *g1* is skipped and all auto variables created in g are destroyed); furthermore *f1* is skipped and the C++ system starts looking for a catch clause that matches the exception. The first catch is skipped because the exception value, 10, is not a float. The second catch clause catches the exception; i is initialized with 10 and the output statement is executed. After that the normal flow of control resumes with *f2*.

11.2.1 Where's the catch?

Exceptions are a form of 'goto' statement. As such, they are very tricky and should be used as a programming technique of last resort. I am only using them in this course because they seem to solve a problem I haven't solved to my satisfaction in any other way.

If not used with extreme care, they will cause more problems then they solve. Consider the following subroutine (using the C subroutines for file opening and closing).

```
void bad() {
        int *p = new(nothrow) int[10000] ;
        FILE *f = fopen("afile", "r") ;
        g() ;
        fclose( f ) ;
        delete p[] ;
}
```

If an exception is thrown from g(), the file will not be closed and the large array will never be deleted from the heap. There are ways to fix the subroutine, but the point is that if exceptions were not a possibility, there would be no problem to fix.

If we were to fix the above code, we might write:

```
void good() {
        int *p = new(nothrow) int[10000] ;
        FILE *f = fopen("afile", "r") ;
        try {
                g() ;
                fclose( f ) ;
                delete p[] ; }
        catch( ... ) { // Catch any exception
                fclose( f ) ; // Clean up
                delete [ ] p ; // Clean up more
                throw ; } // Rethrow the exception
}
```

The documentation of good should explain that it may throw any exception that g may throw.