C++ Review for COP-3530

This material is excerpted from *Data Structures and Algorithm Analysis in C++* (Second Edition) by Mark Allen Weiss and is copyrighted. All rights are reserved.

1 C++ Classes

In this course, we will write many data structures. All of the data structures will be objects that store data (usually a collection of identically typed items), and provide functions that manipulate the collection. In C++ (and other languages), this is accomplished by using a *class*. This section describes the C++ class.

```
/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    /**
     * Construct the IntCell.
     * Initial value is 0.
     */
    IntCell( )
      { storedValue = 0; }

    /**
     * Construct the IntCell.
     * Initial value is initialValue.
     */
    IntCell( int initialValue )
      { storedValue = initialValue; }

    /**
     * Return the stored value.
     */
    int read( )
      { return storedValue; }

    /**
     * Change the stored value to x.
     */
    void write( int x )
      { storedValue = x; }

  private:
    int storedValue;
};
```

Figure 1 A complete declaration of an `IntCell` class

1.1 Basic `class` Syntax

A class in C++ consists of its *members*. These members can be either data or functions. The functions are called *member functions*. Each instance of a class is an object. Each object contains the data components specified in the class (unless the data components are `static`, detail that can be safely ignored for now). A member function is used to act on an object. Sometimes member functions are called *methods*.

As an example, Figure 1 is the `IntCell` class. In the `IntCell` class, each instance of the `IntCell` — an `IntCell` object — contains a single data member named `storedValue`. Everything else in this particular class is a method. In our example, there are four methods. Two of these methods are `read` and `write`. The other two are special methods known as constructors. Let us describe some key features.

First, notice the two labels `public` and `private`. These labels determine visibility of class members. In this example, everything except the `storedValue` data member is `public`. `storedValue` is `private`. A member that is `public` may be accessed by any method in any class. A member that is `private` may only be accessed by methods in its class. Typically, data members are declared `private`, thus restricting access to internal details of the class, while methods intended for general use are made `public`. This is known as information hiding. By using `private` data members, we can change the internal representation of the object, without having an effect on other parts of the program that use the object. This is because the object is accessed through the `public` member functions, whose viewable behavior remains unchanged. The users of the class do not need to know internal details of how the class is implemented. In many cases having this access leads to trouble. For instance, in a class that stored dates using month, day, and year, by making the month, day, and year `private`, we prohibit an outsider from setting these data members to illegal dates, such as Feb 29, 2001. However, some methods may be for internal use, and can be `private`. In a class, all members are `private` by default, so the initial `public` is not optional.

Second, we see two *constructors*. A constructor is a method that describes how an instance of the class is constructed. If no constructor is explicitly defined, one that initializes the data members using language defaults is automatically generated. The `IntCell` class defines two constructors. The first is called if no parameter is specified. The second is called if an `int` parameter is provided, and uses that `int` to initialize the `storedValue` member.

1.2 Extra Constructor Syntax and Accessors

Although the class works as written, there is some extra syntax that makes for better code. Four changes are shown in Figure 2 (we omit comments for brevity). The differences are as follows:

```
        /**
         * A class for simulating an integer memory cell.
         */
        class IntCell
        {
          public:
/* 1*/      explicit IntCell( int initialValue = 0 )
/* 2*/        : storedValue( initialValue ) { }
/* 3*/      int read( ) const
/* 4*/        { return storedValue; }
/* 5*/      void write( int x )
/* 6*/        { storedValue = x; }
          private:
/* 7*/      int storedValue;
        };
```

Figure 2 `IntCell` class with revisions

Default parameters

The `IntCell` constructor illustrates the default parameter. As a result, there are still two `IntCell` constructors defined. One accepts an `initialValue`. The other is the zero-parameter constructor, which is implied because the one-parameter constructor says that `initialValue` is optional. The default value of 0 signifies that 0 is used if no parameter is provided. Default parameters can be used in any function, but they are most commonly used in constructors.

Initializer list

The `IntCell` constructor uses an *initializer list* (line 2) prior to the body of the constructor. The initializer list is used to initialize the data members directly. In the example above, there's hardly a difference, but using initializer lists instead of an assignment statement in the body saves time in the case where the data members are class types that have complex initializations. In some cases it is required. For instance, if a data member is `const` (meaning it is not changeable after the object has been constructed), then the data member's value can only be initialized in the initializer list. Also, if a data member is itself a class type that does not have a zero-parameter constructor, then it must be initialized in the initializer list. We'll see examples of mandatory use of the initializer list starting in Chapter 4.

`explicit` constructor

The `IntCell` constructor is `explicit`. You should make all one-parameter constructors `explicit` to avoid behind the scenes type-conversions. Otherwise, there are somewhat lenient rules that will allow type-conversions without explicit casting operations. Usually this is unwanted behavior, that destroys strong-typing, and can lead to hard-to-find bugs. As an example consider the following:

```
IntCell obj;     // obj is an IntCell
obj = 37;        // Should not compile: type mismatch
```

The code fragment above constructs an `IntCell` object `obj` and then performs an assignment statement. But the assignment statement should not work, because the right-hand side of the assignment operator is not another `IntCell`. `obj`'s `write` method should have been used instead. However, C++ has lenient rules. Normally a one-parameter constructor defines an *implicit type conversion*, in which a temporary object is created that makes an assignment (or parameter to a function) compatible. In this case, the compiler would attempt to convert

```
obj = 37;        // Should not compile: type mismatch
```

into

```
IntCell temporary = 37;
obj = temporary;
```

Notice that the construction of the temporary can be performed by using the one-parameter constructor. The use of explicit means that a one parameter constructor cannot be used to generate an implicit temporary. Thus, since `IntCell`'s constructor is declared explicit, the compiler will correctly complain that there is a type mismatch.

In Section 7.8, we'll see an example where the lenient rules are helpful, but this is the exception, rather than the rule.

The `explicit` keyword is new, and not all compilers support it. However, the preprocessor can be used to replace all occurrences of `explicit` with white space[1], so there's no reason not to put `explicit` in your code.

Constant member function

A member function that examines but does not change the state of its object is an *accessor*. A member function that changes the state is a *mutator* (because it mutates the state of the object). In the typical collection class, for instance, `isEmpty` is an accessor, while `makeEmpty` is a mutator.

In C++, we can mark each member function as being an accessor or a mutator. Doing so is an important part of the design process, and should not be viewed as simply a comment. Indeed, there are important semantic consequences. For instance, mutators cannot be applied to constant objects. By default, all member functions are mutators. To make a member function an accessor, we must add the keyword `const` after the closing parenthesis that ends the parameter type list. The const-ness is part of the signature. `const` can be used with many different meanings. The function declaration can have `const` in three different contexts.

---

1. Use the following statement:
`#define explicit`

Only the `const` after a closing parenthesis signifies an accessor. Other uses are described in Sections 2.2 and 2.3.

In the `IntCell` class, `read` is clearly an accessor: it does not change the state of the `IntCell`. Thus it is made a constant member function at line 3. If a member function is marked as an accessor but has an implementation that changes the value of any data member, a compiler error is generated.[2]

```
#ifndef _IntCell_H_
#define _IntCell_H_

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read( ) const;
    void write( int x );
  private:
    int storedValue;
};

#endif
```

Figure 3 `IntCell` class interface, in file ***IntCell.h***

1.3 Separation of Interface and Implementation

The class in Figure 2 contains all the correct syntactic constructs. However, in C++ it is more common to separate the class interface from its implementation. The interface lists the class and its members (data and functions). The implementation provides implementations of the functions.

Figure 3 shows the class interface for `IntCell`, Figure 4 shows the implementation, and Figure 5 shows a `main` routine that uses the `IntCell`. Some important points follow:

Preprocessor commands

The interface is typically placed in a file that ends with `.h`. Source code that requires knowledge of the interface must `#include` the interface file. In our case, this is both the implementation file and the file that contains `main`. Occasionally, a complicated project will have files including other files, and there is

2. Data members can be marked `mutable` to indicate that const-ness should not apply to them. This is a new feature, that is not supported on all compilers.

the danger that an interface might be read twice in the course of compiling a file. This can be illegal. To guard against this, each header file uses the preprocessor to define a symbol when the class interface is read. This is shown on the first two lines in Figure 3. The symbol name, `_IntCell_H_`, should not appear in any other file; usually we construct it from the filename. The first line of the interface file tests if the symbol is undefined. If so, we can process the file. Otherwise, we do not process the file (by skipping to the `#endif`), because we know that we have already read the file.

Scoping operator

In the implementation file, which typically ends in `.cpp`, `.cc`, or `.C`, each member function must identify the class that it is part of. Otherwise, it would be assumed that the function is in global scope (and zillions of errors would result). The syntax is `ClassName::member`. The `::` is called the *scoping operator*.

```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */

IntCell::IntCell( int initialValue ) : storedValue( initialValue )
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
void IntCell::write( int x )
{
    storedValue = x;
}
```

Figure 4 `IntCell` class implementation in file ***IntCell.cpp***

```
#include "IntCell.h"

int main( )
{
    IntCell m;   // Or, IntCell m( 0 ); but not IntCell m( );

    m.write( 5 );
    cout << "Cell contents: " << m.read( ) << endl;

    return 0;
}
```

Figure 5 Program that uses `IntCell` in file ***TestIntCell.cpp***

Signatures must match exactly

The signature of an implemented member function must match exactly the signature listed in the class interface. Recall that whether a member function is an accessor (via the `const` at the end) or a mutator is part of the signature. Thus an error would result if, for example, the `const` was omitted from exactly one of the `read` signatures in Figures 3 and 4. Note that default parameters are specified in the interface only. They are omitted in the implementation.

Objects are declared like primitive types

In C++, an object is declared just like a primitive type. Thus, the following are legal declarations of an `IntCell` object:

```
IntCell obj1;        // Zero parameter constructor
IntCell obj2( 12 );  // One parameter constructor
```

On the other hand, the following are incorrect:

```
IntCell obj3 = 37;   // Constructor is explicit
IntCell obj4( );     // Function declaration
```

The declaration of `obj3` is illegal because the one-parameter constructor is `explicit`. It would be legal otherwise. (In other words, a declaration that uses the one parameter constructor must use the parentheses to signify the initial value.) The declaration for `obj4` states that it is a function (defined elsewhere) that takes no parameters and returns an `IntCell`.

1.4 `vector` and `string`

The new C++ standard defines two classes: the `vector` and `string`. `vector` is intended to replace the built-in C++ array that causes no end of trouble. The

problem with the built-in C++ array is that it does not behave like a first-class object. For instance, built-in arrays cannot be copied with =, a built-in array does not remember how many items it can store, and its indexing operator does not check that the index is valid. The built-in string is simply an array of characters, and thus has the liabilities of arrays plus a few more. For instance == does not correctly compare two strings.

The `vector` and `string` classes in the STL treat arrays and strings as first-class objects. A `vector` knows how large it is. Two `string` objects can be compared with ==, <, and so on. Both `vector` and `string` can be copied with =. If possible, you should avoid using the built-in C++ array and string. Because this is not always possible, we discuss the built-in array and string in Section 2.6.

Unfortunately, the `vector` does not come with index-range checking, and is also not available on all compilers. Fortunately, it is easy to write a `vector` class with bounds-checks, and a reasonable subset of `vector` features is provided in Appendix B. We use that class throughout. Likewise, the `string` class is not universally available; we provide a simple version in Appendix B.

`vector` and `string` are easy to use. The code in Figure 6 reads a bunch of `string`s into a `vector<string>` (notice that we specify the type of `vector`) and then outputs them in reverse order. We use the `resize` method to double the `vector`'s capacity if it is full. Notice also, that `size` is a method that returns the size of the `vector`. Without a `vector` and `string` class, this code would be much more complex.

```
#include <iostream.h>
#include "vector.h"      // vector (our version, in Appendix B)
#include "mystring.h"    // string (our version, in Appendix B)

int main( )
{
    vector<string> v( 5 );
    int itemsRead = 0;
    string x;

    while( cin >> x )
    {
        if( itemsRead == v.size( ) )
            v.resize( v.size( ) * 2 );
        v[ itemsRead++ ] = x;
    }

    for( int i = itemsRead - 1; i >= 0; i-- )
        cout << v[ i ] << endl;
    return 0;
}
```

Figure 6 Using the `vector` class: Read some strings and output them in reverse order.

`string` is also easy to use, and has all the relational and equality operators to compare the states of two strings. Thus `str1==str2` is `true` if the value of the strings are the same. It also has a `length` method that returns the string length.

2 C++ Details

Like any language, C++ has its share of details and language features. Some of these are discussed in this section.

2.1 Pointers

A pointer variable is a variable that stores the address where another object resides. It is the fundamental mechanism used in many data structures. For instance, to store a list of items, we could use a contiguous array, but insertion into the middle of the contiguous array requires relocation of many items. Rather than store the collection in an array, it is common to store each item in a separate, non-contiguous piece of memory, that is allocated as the program runs. Along with each object is a link to the next object. This link is a pointer variable, because it stores a memory location of another object. This is the classic linked list that is discussed in more detail in Chapter 16.

To illustrate the operations that apply to pointers, we rewrite Figure 5 to dynamically allocate the `IntCell`. It must be emphasized that for a simple `IntCell` class there is no good reason to write the C++ code this way. We do it only to illustrate dynamic memory allocation in a simple context. Later in the text, we will see more complicated classes where this technique is useful and necessary. The new version is shown in Figure 7.

```
        int main( )
        {
/* 1*/      IntCell *m;

/* 2*/      m = new IntCell( 0 );
/* 3*/      m->write( 5 );
/* 4*/      cout << "Cell contents: " << m->read( ) << endl;

/* 5*/      delete m;
/* 6*/      return 0;
        }
```

Figure 7 Program that uses pointers to `IntCell` (there is no compelling reason to do this)

Declaration

Line 1 illustrates the declaration of `m`. The `*` indicates that `m` is a pointer variable; it is allowed to point at an `IntCell` object. The *value* of `m` is the address of the object that it points at. `m` is uninitialized at this point. In C++, no such check is performed to verify that `m` is assigned a value prior to being used (however, several vendors make products that do additional checks, including this one). Use of uninitialized pointers typically crash programs because they result in access of memory locations that do not exist. In general, it is a good idea to provide an initial value, either by combining lines 1 and 2, or by initializing `m` to the `NULL` pointer.

Dynamic object creation

Line 2 illustrates how objects can be created dynamically. In C++ `new` returns a pointer to the newly created object. In C++, there are two ways to create an object using its zero-parameter constructor. Both of the following would be legal:

```
m = new IntCell( );    // OK
m = new IntCell;       // Preferred in this text
```

We generally use the second form because of the problem illustrated by `obj4` in Section 1.2.

Garbage collection and `delete`

In some languages, when an object is no longer referenced, it is subject to automatic garbage collection. The programmer does not have to worry about it. C++ does not have garbage collection. When an object that **is allocated by new** is no longer referenced, the `delete` operation must be applied to the object (through a pointer). Otherwise, the memory that it consumes is lost (until the program terminates). This is known as a *memory leak*. Memory leaks are, unfortunately, common occurrences in many C++ programs. Fortunately, many sources of memory leaks can be automatically removed with care. One important rule is to not use `new` when an *automatic variable* can be used instead. In the original program, the `IntCell` was not allocated by `new`, but instead was allocated as a local variable. Its memory is automatically reclaimed when the function in which it was declared returns. The `delete` operator is illustrated at line 5.

Assignment and comparison of pointers

Assignment and comparison of pointer variables in C++ is based on the value of the pointer, meaning the memory address that it stores. Thus two pointer variables are equal if they point at the same object. If they point at different objects, the pointer variables are not equal, even if the objects being pointed at are themselves equal. If `lhs` and `rhs` are pointer variables (of compatible types), then `lhs=rhs` makes `lhs` point at the same object that `rhs` points at.

Accessing members of an object through a pointer

If a pointer variable points at a class type, then a member of the object being pointed at can be accessed via the `->` operator. This is illustrated at line 3.

Other pointer operations

C++ allows all sorts of bizarre operations on pointers that are occasionally useful. For instance, < is defined. For pointers `lhs` and `rhs`, `lhs<rhs` is true if the object pointed at by `lhs` is stored at a lower memory location than the object pointed at by `rhs`. There is rarely a good reason to use this construct. However, one example of an equally unusual operation is illustrated in Section 7.8.

One important operator is the *address-of operator* `&`. This operator returns the memory location where an object resides and is useful for implementing an alias test that is discussed in Section 2.5.

2.2 Parameter Passing

Many languages, C and Java included, pass all parameters using call by value: the actual argument is copied into the formal parameter. However, parameters in C++ could be large complex objects, for which copying is inefficient. Additionally, sometimes it is desirable to be able to alter the value being passed in. As a result of this, C++ has three different ways to pass parameters. However, there is a simple rule to decide which method to use.

The three parameter passing mechanisms are illustrated in the following function declaration that returns the average of the first `n` integers in `arr`, and sets `errorFlag` to `true` if `n` is larger then `arr.size()` or smaller than 1.

```
double avg( const vector<int> & arr, int n, bool & errorFlag );
```

Here `arr` is of type `vector<int>` and is passed using *call by constant reference*, `n` is of type `int` and is passed using *call by value*, and `errorFlag` is of type `bool` and is passed using *call by reference*. The parameter passing mechanism can generally be decided by a two-part test.

1. If the formal parameter should be able to change the value of the actual argument, then you **must use call by reference**.
2. Otherwise, the value of the actual argument cannot be changed by the formal parameter. If the type is a primitive type, use call by value. Otherwise, the type is a class type and would generally be passed using call by constant reference.[3]

---

3. However, class types that are small (for instance, those that store only a single built-in type) can be passed using call by value instead of call by constant reference.

```
const string & findMax( const vector<string> & a )
{
    int maxIndex = 0;

    for( int i = 1; i < a.size( ); i++ )
        if( a[ maxIndex ] < a[ i ] )
            maxIndex = i;

    return a[ maxIndex ];
}

const string & findMaxWrong( const vector<string> & a )
{
    string maxValue = a[ 0 ];

    for( int i = 1; i < a.size( ); i++ )
        if( maxValue < a[ i ] )
            maxValue = a[ i ];

    return maxValue;
}
```

Figure 8 Two versions to find the maximum string. Only the first is correct.

In the declaration of `avg`, `errorFlag` is passed by reference, so that the new value of `errorFlag` will be reflected in the actual argument. `arr` and `n` will not be changed by `avg`. `arr` is passed by constant reference because it is a class type, and making a copy would be too expensive. `n` is passed by value because it is a primitive type and is cheaply copied.

To summarize the parameter-passing options:

- Call by value is appropriate for small objects that should not be altered by the function.
- Call by constant reference is appropriate for large objects that should not be altered by the function.
- Call by reference is appropriate for all objects that may be altered by the function.

2.3 Return Passing

Objects can also be returned using return by value, return by constant reference, and occasionally, return by reference. For the most part, do not use return by reference. In Section 4.3 we will see one example where it is useful, but this is rare.

It is always safe to use return by value. However, if the object being returned is a class type, it may be better to use return by constant reference, to avoid the

overhead of a copy.[4] However, this is only possible if it is guaranteed that the expression in the return statement has lifetime that extends past the return of the function. This is a very tricky part of C++, and many compilers will fail to give a warning message for incorrect use.

As an example, consider the code in Figure 8, which contains two nearly identical functions to find the largest (alphabetically) `string` in an array. Both attempt to return the value by constant reference. The first version, `findMax`, shows acceptable use: the expression `a[maxIndex]` indexes a `vector` that already exists outside of `findMax`, and will exist long after the call returns. The second version is wrong. `maxValue` is a local variable that does not exist when the function returns. Thus it is improper to return without making a copy of it. If the compiler fails to complain, then the return value may or may not contain useful information, depending on how quickly the compiler decides to reclaim the memory that was used by `maxValue`. This makes for a difficult debugging job.

2.4 Reference Variables

References and constant reference variables are commonly used for parameter passing. But they can also be used as local variables or as class data members. In these cases, they variable names become synonyms for the objects that they reference (much like the formal parameters become synonyms for actual arguments in call by reference). As local variables, they avoid the cost of a copy and thus are useful when querying a data structure that contains a collection of class types. Thus, in many cases, client code such as

```
string x = findMax( a );
  ...
cout << x << endl;
```

is better written as

```
const string & x = findMax( a );
  ...
cout << x << endl;
```

A second use, that we will see in Chapter 5, is to use a local reference variable solely for the purpose of renaming an object that is known by a complicated expression. The code we will see is similar to the following:

```
List<T> & whichList = theLists[ hash( x,theLists.size() ) ];
ListItr<T> itr = whichList.find( x );
if( itr.isPastEnd( ) )
    whichList.insert( x, whichList.zeroth( ) );
```

4. The `const` here means that the object being returned cannot itself be modified later on. It is different from the `const` in the parameter list and the `const` that signifies an accessor.

A reference variable is used so that the considerably more complex expression, `theLists[hash(x,theLists.size())]`, does not have to be written three times.

Reference variables can be used as class data members, though we do not do this in the text (however, an Exercise in Chapter 3 suggests a design that uses a reference variable as a data member). References must be initialized by the constructor to the object that they will reference.

2.5 The Big Three: destructor, copy constructor, `operator=`

In C++, classes come with three special functions that are already written for you. These are the *destructor*, *copy constructor*, and `operator=`. In many cases you can accept the default behavior provided by the compiler. Sometimes you cannot.

Destructor

The destructor is called whenever an object goes out of scope or is subjected to a `delete`. Typically, the only responsibility of the destructor is to free up any resources that were allocated during the use of the object. This includes calling `delete` for any corresponding `new`s, closing any files that were opened, and so on. The default simply applies the destructor on each data member.

Copy constructor

There is a special constructor that is required to construct a new object, initialized to a copy of the same type of object. This is the *copy constructor*. For any object, such as an `IntCell` object, a copy constructor is called in the following instances:

- a declaration with initialization, such as

  ```
  IntCell B = C;
  IntCell B( C );
  ```

  but not

  ```
  B = C;      // Assignment operator, discussed later
  ```

- an object passed using call by value (instead of by `&` or `const &`), which, as mentioned earlier, should never be done anyway.
- an object returned by value (instead of by `&` or `const &`)

The first case is the simplest to understand because the constructed objects were explicitly requested. The second and third cases construct temporary objects that are never seen by the user. Even so, a construction is a construction, and in both cases we are copying an object into a newly created object.

By default the copy constructor is implemented by applying copy constructors to each data member in turn. For data members that are primitive types (for instance, `int`, `double`, or pointers), simple assignment is done. This would be the case for the `storedValue` data member in our `IntCell` class. For data members that are themselves class objects, the copy constructor for each data member's class is applied to that data member.

`operator=`

The *copy assignment operator*, `operator=`, is called when `=` is applied to two objects, after they have both been previously constructed. `lhs=rhs` is intended to copy that state of `rhs` into `lhs`. By default the `operator=` is implemented by applying `operator=` to each data member in turn.

Problems with the defaults

If we examine the `IntCell` class, we see that the defaults are perfectly acceptable, and so we do not have to do anything. This is often the case. If a class consists of data members that are exclusively primitive types and objects for which the defaults make sense, the class defaults will usually make sense. Thus, a class whose data members are `int`, `double`, `vector<int>`, `string`, and even `vector<string>` can accept the defaults.

The main problem occurs in a class that contains a data member that is a pointer. We will describe the problem and solutions in detail in Chapter 3; for now we can sketch the problem. Suppose the class contains a single data member that is a pointer. This pointer points at a dynamically allocated object. The default destructor for pointers does nothing (for good reason — recall that we must `delete` ourselves). Furthermore, the copy constructor and `operator=` both copy not the objects being pointed at, but simply the value of the pointer. Thus we will simply have two class instances that contain pointers that point to the same object. This is a so-called *shallow copy*. Typically, we would expect a *deep copy*, in which a clone of the entire object is made. Thus, when a class contains pointers as data members, and deep semantics are important, we typically must implement the destructor, `operator=`, and copy constructor ourselves.

For `IntCell`, the signatures of these operations are:

```
~IntCell( );                       // destructor
IntCell( const IntCell & rhs );    // copy constructor
const IntCell & operator=( const IntCell & rhs );
```

Although the defaults for `IntCell` are acceptable, we can write the implementations anyway, as shown in Figure 9. For the destructor, after the body is executed, the destructors are called for the data members. So the default is an empty body. For the copy constructor, the default is an initializer list of copy constructors, followed by execution of the body.

`operator=` is the most interesting. Line 1 is an alias test, to make sure we are not copying to ourselves. Assuming we are not, we apply `operator=` to each data member (at line 2). We then return a reference to the current object, at line 3, so assignments can be chained, as in `a=b=c`.

In the routines that we write, if the defaults make sense, we will always accept them. However, if the defaults do not make sense, we will need to implement the destructor, and `operator=`, and the copy constructor. When the default does not work, the copy constructor can generally be implemented by mimicing normal construction and then calling `operator=`. Another often-used option is to give a reasonable working implementation of the copy constructor, but then place it in the `private` section, to disallow call by value.

When The Defaults Do Not Work

The most common situation in which the defaults do not work occurs when a data member is a pointer type, and the pointee is allocated by some object member function (such as the constructor). As an example, suppose we implement the `IntCell` by dynamically allocating an `int`, as shown in Figure 10. For simplicity, we do not separate the interface and implementation.

```
        IntCell::~IntCell( )
        {
            // Does nothing since IntCell contains only an int data
            // member. If IntCell contained any class objects their
            // destructors would be called.
        }

        IntCell::IntCell( const IntCell & rhs )
        {
        }

        const IntCell & IntCell::operator=( const IntCell & rhs )
        {
/* 1*/      if( this != &rhs )   // Standard alias test
/* 2*/          storedValue = rhs.storedValue;
/* 3*/      return *this;
        }
```

Figure 9 The defaults for the big three

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 )
      { storedValue = new int( initialValue ); }

    int read( ) const;
      { return *storedValue; }
    void write( int x );
      { *storedValue = x; }
  private:
    int *storedValue;
};
```

Figure 10 Data member is a pointer; default are no good

There are now numerous problems that are exposed in Figure 11. First, the output is three 4s, even though logically only `a` should be 4. The problem is that the default `operator=` and copy constructor copy the pointer `storedValue`. Thus `a.storedValue`, `b.storedValue`, and `c.storedValue` all point at the same `int` value. These copies are *shallow*: the pointers, rather than the pointees are copied. A second less-obvious problem is a memory leak. The `int` initially allocated by `a`'s constructor remains allocated and needs to be reclaimed. The `int` allocated by `b` and `c`'s constructor is no longer referenced by any pointer variable. They also need to be reclaimed, but we no longer have a pointer to it.

To fix these problems, we implement the big three. The result (with the interface and implementation separated) is shown in Figure 12. Generally speaking, if a destructor is necessary to reclaim memory, then the defaults for copy assignment and copy construction are not acceptable.

If the class contains data members that do not have the ability to copy themselves, then the default `operator=` will not work. We will see some examples of this later in the text.

```
int f( )
{
    IntCell a( 2 );
    IntCell b = a;
    IntCell c;

    c = b;
    a.write( 4 );
    cout << a.read( ) << endl << b.read( ) << endl << c.read( ) << endl;
    return 0;
}
```

Figure 11 Simple function that exposes problems in Figure 10

```
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );

    IntCell( const IntCell & rhs );
    ~IntCell( );
    const IntCell & operator=( const IntCell & rhs );

    int read( ) const;
    void write( int x );
  private:
    int *storedValue;
};

IntCell::IntCell( int initialValue )
{
    storedValue = new int( initialValue );
}

IntCell::IntCell( const IntCell & rhs )
{
    storedValue = new int( *rhs.storedValue );
}

IntCell::~IntCell( )
{
    delete storedValue;
}

const IntCell & IntCell::operator=( const IntCell & rhs )
{
    if( this != &rhs )
        *storedValue = *rhs.storedValue;
    return *this;
}

int IntCell::read( ) const
{
    return *storedValue;
}

void IntCell::write( int x )
{
    *storedValue = x;
}
```

Figure 12 Data member is a pointer; big three needs to be written

2.6 The World of C

C++ inherits its basic syntax from C. Some C-style constructs are occasionally seen in C++, even though C++ provides alternatives. We list a few of these.

`struct`s

In C++, a `struct` is exactly like a `class` except that by default, all members are `public`. There is no other semantic difference. As a result, it is easy to write a C++ program that never uses `struct`. Even so, a `struct` is commonly used to signal a `class` that contains only `public` data and constructors, since such a `class` behaves like a C-style `struct`.

`typedef`

The `typedef` is used to indicate that a symbol should be a synonym for an existing type. For instance,

```
typedef string * ptr_to_string;
```

says that `ptr_to_string` is a synonym for the `string*` type. `typedef` is less-often used in C++ than C because in many cases it is better to define a new class that encapsulates the behavior of this type that to use a `typedef`.

There are two common uses of the `typedef`. One is to define system-dependent information. Thus, the type `int32`, representing a thirty-two bit integer, could be a `typedef` defined in a header file. On some machines it would be an `int`, on others it could be a `short`, and on others it could be a `long`. A second use is to provide a synonym for a long type name. Long type names are common when templates (especially in the STL) are instantiated. An example of this is in Appendix A.

Parameter Passing: C-Style

In C, all parameters are passed using call-by-value. However, C programmers often need to pass using call-by-reference. Since this is not possible in C, a commonly used trick is used: a pointer to the object is passed instead of the object. Call-by-value means that the value of the pointer (where it points) cannot change, but does not disallow changing the pointee. To illustrate the idiom, we show how an integer is passed by reference. The function `zero` will change the object being pointed at to 0. `zero` declares:

```
void zero( int *val ) { *val = 0; }
```

The function call is made by passing the address of x to function zero:

```
int x = 5;      // Object x has value 5
zero( &x );     // Object x will have value 0
```

Passing using C++ call by reference is preferable to this idiom. However, many libraries are written to work with both C and C++, and thus pass variables using the C-style. Thus you may need to use this idiom. We do not use elsewhere in the text.

C-Style Arrays and Strings

The C++ language provides a built-in C-style array type. To declare an array, `arr`, of ten integers, one writes:

```
int arr1[10];
```

`arr1` is actually a pointer to memory that is large enough to store 10 `int`s, rather than a first-class array type. Applying `=` to arrays is thus an attempt to copy two pointer values, rather than the entire array, and with the declaration above is illegal because `arr1` is a constant pointer. When `arr1` is passed to a function, only the value of the pointer is passed; information about the size of the array is lost. Thus the size must be passed as an additional parameter. There is no index range checking, since the size is unknown.

In the declaration above, the size of the array must be known at compile time. `10` cannot be replaced by a variable. If the size is unknown, we must explicitly declare a pointer and allocate memory via `new[]`. For instance,

```
int *arr2 = new int[ n ];
```

Now `arr2` behaves like `arr1` except that it is not a constant pointer. Thus it can be made to point at a larger block of memory. However, because memory has been dynamically allocated, at some point it must be freed with `delete[]`:

```
delete [ ] arr2;
```

Otherwise, a memory leak would result, and the leak could be significant, if the array is large.

Built-in C-style strings are implemented as an array of characters. To avoid having to pass the length of the string, the special null-terminator `'\0'` is used as a character that signals the logical end of the string. Strings are copied by `strcpy`, compared with `strcmp`, and their length can be determined by `strlen`. Individual characters can be accessed by the array indexing operator. These strings have all the problems associated with arrays, including difficult memory management, compounded by the fact that when strings are copied, it is assumed that the target array is large enough to hold the result. When it is not, difficult debugging ensues, especially when room has not be left for the null terminator.

Appendix B describes a `vector` class and a `string` class, that are implemented by hiding the behavior of the built-in C-style array and string. By studying that class, you can see how C-style arrays and strings are manipulated. It is

almost always better to use the `vector` and `string` class in Appendix B (or the ones defined in the C++ library, if your compiler is current), but you may be forced to use the C-style when interacting with library routines that are designed to work with both C and C++. It also is occasionally necessary (but this is rare) to use the C-style in a section of code that must be optimized for speed.

3 Templates

Consider the problem of finding the largest item in an array of items. A simple algorithm is the sequential scan, in which we examine each item in order, keeping track of the maximum. As is typical of many algorithms, the sequential scan algorithm is type-independent. By type-independent, we mean that the logic of this algorithm does not depend on the type of items that are stored in the array. The same logic works for an array of integers, floating-point numbers, or any type for which comparison can be meaningfully defined.

Throughout this text, we will describe algorithms and data structures that are type independent. When we write C++ code for a type-independent algorithm or data structure, we would prefer to write the code once, rather than recode it for each different type.

In this section we will describe how type-independent algorithms (also known as generic algorithms) are written in C++. C++ provides the *template*. We begin by discussing function templates. Then we examine class templates.

3.1 Function templates

Function templates are generally very easy to write. A *function template* is not an actual function, but instead is a pattern for what could become a function. Figure 13 illustrates a function template `findMax` that is virtually identical to the routine for `string` shown in Figure 8. The line containing the `template` declaration indicates that `Comparable` is the template argument: it can be replaced by any type to generate a function. For instance, if a call to `findMax` is made with a `vector<string>` as parameter, then a function will be generated by replacing `Comparable` with `string`.

```
        /**
         * Return the maximum item in array a.
         * Assumes a.size( ) > 0.
         * Comparable objects must provide
         *   copy constructor, operator<, operator=
         */
        template <class Comparable>
        const Comparable & findMax( const vector<Comparable> & a )
        {
/* 1*/      int maxIndex = 0;

/* 2*/      for( int i = 1; i < a.size( ); i++ )
/* 3*/          if( a[ maxIndex ] < a[ i ] )
/* 4*/              maxIndex = i;
/* 5*/      return a[ maxIndex ];
        }
```

Figure 13 `findMax` function template

```
        int main( )
        {
            vector<int>     v1( 37 );
            vector<double>  v2( 40 );
            vector<string>  v3( 80 );
            vector<IntCell> v4( 75 );

            // Additional code to fill in the vectors

            cout << findMax( v1 ) << endl;  // OK: Comparable = int
            cout << findMax( v2 ) << endl;  // OK: Comparable = double
            cout << findMax( v3 ) << endl;  // OK: Comparable = string
            cout << findMax( v4 ) << endl;  // Illegal; operator< undefined

            return 0;
        };
```

Figure 14 Using `findMax` function template

Figure 14 illustrates that function templates are expanded automatically as needed. It should be noted that an expansion for each new type generates additional code; this is known as *code bloat*, when it occurs in large projects. Note also, that the call `findMax(v4)` will result in a compile-time error. This is because when `Comparable` is replaced by `IntCell`, line 3 in Figure 13 becomes illegal: there is no `<` function defined for `IntCell`. Thus it is customary to include, prior to any template, comments that explain what assumptions are made about the template argument(s). This includes assumptions about what kinds of constructors are required. Also note that `findMax` does not work with C-style strings, because `operator<` for two `char*` compares pointer values.

Because template arguments can assume any class type, when deciding on parameter passing and return passing conventions, it should be assumed that template arguments are not primitive types. That is why we have returned by constant reference.

Not surprisingly, there are many arcane rules that deal with function templates. Most of the problems occur when the template cannot provide an exact match for the parameters, but can come close (through implicit type conversions). There must be ways to resolve ambiguities and the rules are quite complex. Note that if there is a non-template and a template, and both match, then the non-template gets priority. Also note that if there are two equally close approximate matches, then the code is illegal and the compiler will declare an ambiguity.

It is important to note that for most compilers, function templates cannot be separately compiled. Generally their entire definition will be placed in `.h` files that are included by anyone that might need them.

3.2 Class Templates

In the simplest version, a class template works much like a function template. Figure 15 shows the `MemoryCell` template. `MemoryCell` is like the `IntCell` class but works for any type, `Object`, provided that `Object` has a zero-parameter constructor, a copy constructor, and a copy assignment operator.

Notice that `Object` is passed by constant reference. Also, notice that the default parameter for the constructor is not 0, because 0 might not be a valid `Object`. Instead the default parameter is the result of constructing an `Object` with its zero-parameter constructor.

```
/**
 * A class for simulating a memory cell.
 */
template <class Object>
class MemoryCell
{
  public:
    explicit MemoryCell( const Object & initialValue = Object( ) )
      : storedValue( initialValue ) { }
    const Object & read( ) const
      { return storedValue; }
    void write( const Object & x )
      { storedValue = x; }
  private:
    Object storedValue;
};
```

Figure 15 `MemoryCell` template class without separation

```
int main( )
{
    MemoryCell<int>    m1;
    MemoryCell<string> m2( "hello" );

    m1.write( 37 );
    m2.write( m2.read( ) + " world" );
    cout << m1.read( ) << endl << m2.read( ) << endl;

    return 0;
}
```

Figure 16 Program that uses `MemoryCell` template class

```
/**
 * A class for simulating a memory cell.
 */
template <class Object>
class MemoryCell
{
  public:
    explicit MemoryCell( const Object & initialValue = Object( ) );
    const Object & read( ) const;
    void write( const Object & x );
  private:
    Object storedValue;
};
```

Figure 17 `MemoryCell` template class interface

Figure 16 shows how the `MemoryCell` can be used to store objects of both primitive and class types. Notice that `MemoryCell` is not a class; it is only a class template. `MemoryCell<int>` and `MemoryCell<string>` are the actual classes.

If we implement class templates as a single unit, then there is very little syntax baggage. Many class templates are, in fact, implemented this way because currently, separate compilation of templates does not work well on many platforms. Therefore, in many cases, the entire class, with its implementation must be placed in a `.h` file. Popular implementations of the STL follow this strategy.

However, eventually, separate compilation will work, and it will be better to separate the class templates interface and implementation in the same way that is done for classes. Unfortunately, this does add some syntax baggage.

Figure 17 shows the interface for the template class. That part is, of course, simple enough, since it is just a subset of the entire class that we have already seen.

```
#include "MemoryCell.h"

/**
 * Construct the MemoryCell with initialValue
 */
template <class Object>
MemoryCell<Object>::MemoryCell( const Object & initialValue )
  : storedValue( initialValue )
{
}

/**
 * Return the stored value.
 */
template <class Object>
const Object & MemoryCell<Object>::read( ) const
{
    return storedValue;
}

/**
 * Store x.
 */
template <class Object>
void MemoryCell<Object>::write( const Object & x )
{
    storedValue = x;
}
```

Figure 18 `MemoryCell` template class implementation

For the implementation, we have a collection of function templates. This means that each function must include the template line, and when using the scope operator, the name of the class must be instantiated with the template argument. Thus in Figure 18, the name of the class is `MemoryCell<Object>`. Although the syntax seem innocuous enough, it can get fairly substantial. For instance, to define `operator=` in the interface requires no extra baggage. In the implementation, we would have:

```
template <class Object>
const MemoryCell<Object> &
MemoryCell<Object>::operator=( const MemoryCell<Object> & rhs )
{
    if( this != &rhs )
        storedValue = rhs.storedValue;
    return *this;
}
```

Typically, the declaration part of the more complex functions will no longer fit on one line, and will need splitting as done above.

Even if the interface and implementation of the class template are separated, few compilers will automatically handle separate compilation correctly. The simplest, most portable solution, is to add an `#include` directive at the end of the interface file, to import the implementation. This is done in the online code. Alternative solutions involve adding explicit instantiations for each type as a separate `.cpp` file in the project. Since these details will change rapidly, it's best to consult local documentation to find the proper alternative.

3.3 `Object`, `Comparable`, and an Example

In this text, we repeatedly use `Object` and `Comparable` as generic types. `Object` is assumed to have a zero parameter constructor, an `operator=`, and a copy constructor. `Comparable`, as suggested in the `findMax` example, has additional functionality in the form of `operator<` that can be used to provide a total order.[5]

Figure 19 shows an example of a class type that implements the functionality required of `Comparable`. The `Employee` class contains a `name` and a `salary`, and defines `operator<` on the basis of `salary`. A more complicated `operator<` is possible; for instance, we could break a tie in `salary` by using the `name` data member. The `Employee` class also provides a zero-parameter constructor, `operator=`, and copy constructor (all by default). Thus it has enough to be used as a `Comparable` in `findMax`.

To have practical utility, either its data members must be public, or we must provide additional accessors and mutators. Figure 19 shows a `setValue` member function, and also illustrates the widely-used idiom for providing an output function for a new class type. The idiom is to provide a `public` member function, named `print` that takes an `ostream` as a parameter. That `public` member function can then be called by a global, non-class function `operator<<`, that accepts an `ostream` and an object to output.[6]

---

5. Some of the data structures in Chapter 12 use `operator==` in addition to `operator<`. Note that for the purpose of providing a total order, `a==b` if both `a<b` and `b<a` are `false`; thus the use of `operator==` is simply for convenience.

6. An alternative to this idiom is to have `operator<<` directly implement the logic in `print`. Because `operator<<` is not a class member, it would need to be made a `friend` function of the `Employee` class, requiring the introduction of even more C++ syntax. This alternative has the additional disadvantage of not working on older compilers that do not correctly mix `friend` declarations with global template functions. It also has the disadvantage of not working correctly in more complex contexts involving inheritance, that are beyond the scope of this text.

```
class Employee
{
  public:
    void setValue( const string & n, double s )
      { name = n; salary = s; }

    void print( ostream & out ) const
      { out << name << " (" << salary << ")"; }
    bool operator< ( const Employee & rhs ) const
      { return salary < rhs.salary; }

    // Other general accessors and mutators, not shown
  private:
    string name;
    double salary;
};

  // Define an output operator for Employee
ostream & operator<< ( ostream & out, const Employee & rhs )
{
    rhs.print( out );
    return out;
}

int main( )
{
    vector<Employee> v( 3 );

    v[0].setValue( "Bill Clinton", 200000.00 );
    v[1].setValue( "Bill Gates", 2000000000.00 );
    v[2].setValue( "Billy the Marlin", 60000.00 );
    cout << findMax( v ) << endl;

    return 0;
}
```

Figure 19 `Comparable` can be a class type, such as `Employee`

4 Using Matrices

Several algorithms in Chapter 10 use two-dimensional arrays, which are popularly known as matrices. The C++ library does not provide a `matrix` class. However, a reasonable `matrix` class can be quickly written. The basic idea is to use a vector of vectors. Doing this requires knowledge of *operator overloading*. Operator overloading allows us to define the meaning of a built-in operator. Actually, we've already done this when we define `operator=`. For the `matrix`, we define `operator[]`, namely the array-indexing operator. The `matrix` class is in Figure 20.

4.1 The data members, constructor, and basic accessors

The matrix is represented by an `array` data member that is declared to be a `vector` of `vector<Object>`. The constructor first constructs `array`, as having `rows` entries each of type `vector<Object>` that is constructed with the zero-parameter constructor. Thus we have `rows` zero-length vectors of `Object`.

```
template <class Object>
class matrix
{
  public:
    matrix( int rows, int cols ) : array( rows )
    {
        for( int i = 0; i < rows; i++ )
            array[ i ].resize( cols );
    }
    const vector<Object> & operator[]( int row ) const
      { return array[ row ]; }
    vector<Object> & operator[]( int row )
      { return array[ row ]; }
    int numrows( ) const
      { return array.size( ); }
    int numcols( ) const
      { return numrows( ) ? array[ 0 ].size( ) : 0; }
  private:
    vector< vector<Object> > array;
};
```

Figure 20 A complete `matrix` class

The body of the constructor is then entered and each row is resized to have `cols` columns. Thus the constructor terminates with what appears to be a two-dimensional array. The `numrows` and `numcols` accessors are then easily implemented as shown.

4.2 `operator[]`

The idea of `operator[]` is that if we have a `matrix` `m`, then `m[i]` should return a vector corresponding to row `i` of `matrix` `m`. If this is done, then `m[i][j]` will give the entry in position `j` for vector `m[i]`, using the normal `vector` indexing operator. Thus the `matrix` `operator[]` is to return not an `Object`, but instead a `vector<Object>`.

We now know that `operator[]` should return an entity of type `vector<Object>`. Should we use return by value, by reference, or by constant

reference? Immediately we eliminate return by value, because the returned entity is large, but guaranteed to exist after the call. Thus we are down to return by reference or by constant reference. Consider the following method (ignore the possibility of aliasing or incompatible sizes, neither of which affects the algorithm).

```
void copy( const matrix<int> & from, matrix<int> & to )
{
    for( int i = 0; i < to.numrows( ); i++ )
        to[ i ] = from[ i ];
}
```

In the `copy` function, we attempt to copy each row in `matrix from` into the corresponding row in `matrix to`. Clearly, if `operator[]` returns a constant reference, then `to[i]` cannot appear on the left side of the assignment statement. Thus it appears that `operator[]` should return a reference. But then an expression such as `from[i]=to[i]` would compile, since `from[i]` would not be a constant vector, even though `from` was a constant matrix. Oops!

So what we really need is for `operator[]` to return a constant reference for `from`, but a plain reference for `to`. In other words, we need two versions of `operator[]`, that differ only in their return types. That is not allowed. However, there is a loophole: since member function constness (that is, whether a function is an accessor or a mutator) is part of the signature, we can have the accessor version of `operator[]` return a constant reference, and have the mutator version return the simple reference. Then all is well. This is shown in Figure 20.

4.3 Destructor, copy assignment, copy constructor

These are all taken care of automatically because the `vector` has taken care of it. Thus this is all the code needed for a fully functioning `matrix` class.