_Steve Reiss
— John Ousterhout : Stanford.

Guidelines for Using C++ Effectively

C++ is a large and complex language. Parts of it are nonintuitive, other parts are poorly defined, and still others have complex and inefficient implementations. As such, it is easier than most languages to misuse. Many of its features are quite specialized and are not needed or suitable for general programming. These features are best avoided. Other features are very beneficial when used correctly and extensively, but are not encouraged by the language and are typically overlooked by programmers. The purpose of this chapter is to provide a set of guidelines on how to use the language in an effective and somewhat intuitive way so that the resultant programs are likely to work, be readable, and be maintainable.

Classes as Pointers

As we discussed in See Pointers and Objects , C++ distinguishes between objects and pointers to objects. Unlike pure object-oriented languages where an object and a pointer to the object are the same thing, in C++ they are very different. C++ lets one allocate and pass classes around as structures or to allocate storage and view things as pointers.
The advantages of viewing classes as structures, beside backward compatibility with C, are efficiency and, at times, convenience. Classes that are structures can be allocated either statically or on the stack. Classes represented as pointers should always be allocated in heap storage using the new operator. Dynamic allocation of storage can be time consuming (although today's storage allocation routines are much faster than they were ten years ago when memory allocation was a real bottleneck). It also requires that the programmer insert statements to create and delete the object, and to keep track of who owns it so it can be deleted at the proper time. All this adds to the complexity of the program.

The advantages of using structures are outweighed however by the disadvantages. Viewing objects solely as pointers tends to yield a cleaner and simpler program. One important part of simplicity and understandability of a system is consistency. If objects are sometimes viewed as pointer and sometimes viewed as structures, it makes it confusing for the reader and the programmer. Using objects in just one way lets the program be understood without having to determine what is what. Using pointers also provides a degree of portability to and consistency with other object-oriented languages such as Java.

Treating objects as pointers tends to make one's use of the C++ language a lot clearer. First, C++ provides a separate syntax for accessing fields or invoking methods of objects that are pointer versus objects that are structures. Using only pointers lets the programmer consistently use the `->` notation rather than requiring that `a.method()` in some contexts and `a->method()` in others. Second, the programmer is sure that virtual methods will be used when the object is a pointer. Objects that are viewed as structures will sometimes use virtual methods (for example when they are passed as ref parameters) and sometimes not, leading to further confusion and misunderstanding. C++ also uses a pass-by-value semantics for calling sequences. When an object represented as a structure is passed to a routine, a temporary object is created using the copy constructor of the object and this copy is passed in. If the routine makes any changes to the object, the changes are made in the temporary copy which may not be what was intended. (If the programmer means to make temporary changes, it would be better to make this obvious to the reader by making an explicit copy for this purpose.) If pointers are used consistently, then what is passed is a pointer to the object. This is both more efficient and more often meets the intent of the programmer. Finally, C++ introduces a number of implicit calls using constructors, destructors and conversion operators. If objects are viewed as pointers, these calls become explicit in the code and the code actually states what is happening in the program. If objects are viewed as structures, C++ will insert constructor and destructor calls that will be hidden from the programmer and not obvious to the reader.

Thus we introduce our first guideline:

Use pointers to represent classes.

This has several implications. First, as we noted, the pointer type and the object type should both be declared, with the pointer type being the name that is used most often. One of the header files, generally, the local header file, should contain a forward type definition that introduces the class name and defines the pointer type. For example

```
typedef class KnightMainInfo * KnightMain;
```

defines both the name `KnightMainInfo` as a class that will be declared later and the name `KnightMain` as a pointer to an instance of this class. Note that we make the pointer type shorter and thus easier to use, type, and remember to encourage its use over that of the class name. The actual class name should only be used where it is absolutely necessary such as when using the operator `new` and when calling a static method.

There are three exceptions to this general guideline, all dealing with advanced uses of C++. The

first involves the case where a class is created to represent a pointer. This can be done to provide automatic control of memory through reference counting or to further separate an external interface from the actual implementation. We will see examples of this later on. In this case, the class is meant to be treated as if it were a pointer and hence is an actual class and is passed around as such. Note, however, that such a class can be designed to be treated exactly as if it were a pointer and the fact that it is a class is hidden from both the programmer and the reader.
The second exception occurs when the system requires a simple data structure that should be treated as if it were a primitive object such as a point in a graphics program. Here the overhead of allocating, keeping track of, and freeing storage for the small objects throughout the implementation is both excessive and will make the code much more complex than it would otherwise be. The program would be more readable and simpler if the data was represented as a structure rather than as a pointer in this case. A structure should only be used in this case, however, if the object is truly simple, i.e. if it contains only simple data, is relatively small, and can be copied with no harmful effects. To distinguish such structures from standard classes, the programmer should declare it as a `struct` rather than a `class`.
The third exception involves classes that are meant to act as primitive elements of the language. For example, in order to add complex numbers to the language one wants to define a class ***Complex*** that can be used as a basic arithmetic type. This means that the programmer can mix it freely with integer and floating point numbers, can apply all the basic arithmetic operations to it in a natural way, can use it for comparisons, can input and output to it directly, and so on. Because one wants to use it with the standard operators and in expressions, it is much more natural to define such a class directly and to not use an intervening pointer. We will see an instance of such a class when we define rational numbers in See.

Class Implementations

C++ offers language features that serve both to ensure program correctness and to document the code. These include protection levels within a class and the use of const declarations. Both of these should be used to their fullest extent.

Protection

C++ offers multiple protection levels in a class. Public methods constitute the external interface of the class. These are accessible to other classes and the programmer should make few assumptions on how they are used or that parameters being passed in are valid. Private methods and data are accessible only to members of the class. These constitute the internal interface and define the implementation. The programmer presumably has sufficient control over these that documented assumptions about their parameters, calling order, etc. can be made without extra checking in the code. Protected methods are a special case that is used only in the case of inheritance. These

methods are accessible to subclasses of the class as well as the class itself.
The protection of an item in a class declaration serves both to control access to the class and as documentation on how the programmer expects the method to be used. Thus:

Use the most restricted protection level possible for each method and data element.

Moreover, since data elements directly reflect the implementation they should never be publically accessible. Here we insist:

Data elements should always be either private or protected.

Following these guidelines ensures several things. First, that the reader can understand what each method is for. Second, it should minimize the size of the external interface of a class. Since the external interface essentially defines how the class is viewed from outside, this tends to simplify the class and its use in the system. Third, it lets the compiler check that methods are only used where the programmer expects them to be use, thus avoiding potential problems later on.
If a class is very simple so that it is essentially a container for some related data items such as the X and Y coordinates of a point, then it sometimes makes sense to make its data accessible. This should be done by declaring it as a `struct` rather than a `class` to inform the reader. Note that for C++, structures and classes are the same except that the data and methods of a structure are public by default while those of a class are private. In general, a structure should include few if any methods (since methods imply an implementation that should be hidden) and should be as simple as possible. Moreover, all of a structure's methods should be public.
In addition to these levels of protection, C++ offers the notion of a friend class and friend functions. Once a class or function is declared as a friend of a given class, it has full access to the private (and protected) data and methods of the class. Because friend declarations let the program avoid the normal constraints of the protection mechanisms of the language, they introduce additional complexity by forcing the reader to be aware of exceptions to the standard rules and thus making it difficult to understand when and where methods can be called and data set. Because of this we recommend:

Avoid the use of friend declarations.

There are two instances where the use of friends is helpful in C++. Some overloaded operators can only be defined in C++ by using friend functions. This occurs when the parameter of the class type to the operation is not the first argument, for example when the output function << is overloaded. In these cases, the operator should be defined as a friend function within the class definition. Friend classes can be used when there are two classes that are very closely related. For example one might want to define a class and an iterator for the class where the workings of the iterator are closely related to the implementation of the class. Normally, the iterator would need public access to the class implementation, but making it a friend would let the implementation remain private. It is acceptable to use friend classes in such cases provided that the two classes have related names, are both declared in the same header file and are both defined in the same code file.

Const Declarations

The other self-documenting facility that C++ provides involves the use of const declarations. C++ uses the keyword `const` to represent a variety of things. Consider the method declaration:

```
const char * const memberFunction(const ClassPointer obj) const;
```

The first `const` describes the following `char`, indicating that the character is read-only. The second `const` modifies the pointer that precedes it, indicating that the pointer itself is read only and cannot be changed. The `const` inside the parameter list relates to the subsequent class pointer name. Assuming the `ClassPointer` was typedef'd to be a pointer to the class ***ClassPointerInfo***, it indicates that the parameter `obj` points to a constant object. The final `const` in the declaration refers to the method itself and indicates that the method does not modify the object and can be used if the object is constant. The parameter `obj` in this case, being constant, can only be used to invoke const methods.

Const declarations serve both as documentation and as a check on the programmer. Compilers will ensure that the code of a constant method does not call any non-constant methods or modify any of the data elements of the class. Moreover, the const declaration informs both the programmer and any readers of the code of this fact and lets them make the corresponding assumptions without having to check the code. Constant methods are also simpler in the sense that they will not modify the class. Thus we recommend:

Use const as much as possible in declarations.

`Const` should be used to indicate which parameters are input parameters versus in-out or output-only parameters. As many methods as possible should be declared constant. Note that it is difficult to add const declarations to existing code because a constant method can only call other constant methods and thus const declarations have to be present throughout. It is much easier to start with the design and specify everything that might possibly be declared constant as such. The initial code can then reflect these decisions. As methods are added that are not constant but were expected to be, the programmer should reevaluate the design to verify that the method should indeed be non-constant, and then should remove a minimal number of const declarations.

Parameter Types

`Const` is only one of the options that C++ offers in declaring parameters. Parameters in C are all passed by value. A const parameter is thus one that is passed but cannot be set. It is effectively an input parameter. C++ on the other hand provides two ways of defining parameters that can be used as output parameters. The first is compatible with C and uses pointers. Here a pointer to the value to be returned is passed by value to the method and the method sets the value by going indirectly through the pointer. The alternative is to use ref parameters. These are effectively implemented the same way, with the pointer to the object being passed to the called routine.

However, neither the caller nor the called routine need to be aware of pointers. The routine is called exactly as if the parameter were passed by value. Similarly, the routine itself can access and assign to the parameter, treating it as an instance of the declared type.
This means that ref parameters are more convenient to use and result in more straightforward code than parameters passed as pointers. Moreover, because the pointer usage is hidden from the programmer, errors such as references through a NULL pointer are much less likely. There are two drawbacks to using them. The first is where compatibility with C is needed. This does not apply to method calls, but might apply to external functions. Since C does not provide ref parameters, pointers must be used in this case. The second case is where the output parameter is optional. A pointer parameter lets the programmer pass a NULL value which the routine can then interpret to mean that the resultant value need not be computed or stored.
Note that output parameters are only needed when multiple values need to be returned from a single method. Methods that return multiple values are more complex than simple functional methods that return only a single value. As such they should be avoided if possible.
Because ref parameters serve both as documentation of the fact that a parameter is to be used as output and make the program simpler, we recommend:

Use ref parameters whenever a value is to be returned by a function as a parameter.

This recommends that ref parameters be used for their obvious function of returning values. The other common use of ref parameters is to provide a more efficient means for passing structures or classes. Passing a structure as a ref parameter does not create a copy of the structure first as would happen if the structure were passed by value. Instead it just passes a pointer to the structure.
Ref parameters can also be declared `const` . A constant ref parameter acts like an input variable but uses copy-by-reference semantics, that is, a pointer to the value is passed rather than the value itself being copied. Note however, that different compilers will treat const ref parameters differently in the case where an expression is passed, sometimes creating a temporary for the call and sometimes giving an error message.
Since we recommend using pointers to represent classes, we generally do not need to worry about using ref parameters for the passing structures. A class pointer can be passed to a method to let that class be modified. A constant pointer to the class can be passed to indicate a class parameter that will not be changed (directly or indirectly).
Continuing this theme, we recommend:

Use simple types for parameters wherever possible.

Simple types include all the built-in types included those introduced by the standard libraries (see See), enumeration types, typedefd pointers to simple types, and typedefd pointers to classes. Classes should always be passed using a pointer or a const pointer to the class. If the application requires structures that are not classes, it is best not to use these as parameters or to treat them similarly to classes, using pointers throughout. Arrays defined directly in C++ should be passed only where necessary since the language actually treats these as pointers and does no bounds

checking. In general, if an array is passed, an additional parameter should be passed with the bounds of the array. Pointers that are not typedefd should be passed only when they are used explicitly such as when using a pointer to allow a return value.
The programmer should also attempt to keep the types of parameters simple and consistent so as to keep the code comprehensible, enhance reuse, and avoid potential problems caused by misunderstanding. This means that the order of similar parameters sets should be the same. For example, when an array and its bounds are passed, the calling sequence should consistently be array, bounds (or consistently bounds,array). Parameter names should be provided in the declaration for any parameter whose use is not obvious based on either its type of the name of the function.

Overloading

One of the more controversial features of C++ is the ability to overload functions and methods. Overloading here can take two forms. The user first can define multiple methods with the same name but different argument types. The compiler will then consider the number and types of arguments in a given call and, using a set of little-understood rules for disambiguation, will choose the proper instance of the method as the one that will be called. Overloading also occurs where the user provides default values for parameters, thereby letting different calling sequences be used for the same method.
Overloading is controversial because, while it generally makes the code more complex, there are cases where it makes the code straightforward and easier to read. Overloading is considered harmful and omitted from many languages because it adds significant complexity to the language. This complexity shows up in the programming language definition and the compiler. The program becomes more difficult to read because it is no longer obvious what function is called from a given site. The language and compiler complexity is reflected in the complexity and obtuseness of corresponding sections of the C++ reference manual.
There are cases, however, where one wants to hide the method from the programmer. Consider a drawing program that lets an application set the color of an object using a `setColor` method. There are a variety of ways that a color can be specified -- by name, by index, or using red, green, and blue values. The application could define methods for each of these:

```
void setColorByName(const char * color);
void setColorByIndex(int pixel);
void setColorByRGB(float red,float green,float blue);
void setColorByRGBValue(unsigned r,unsigned g,unsigned b);
```

This would make it clear to the reader which method was being called. However, this also requires the programmer to remember (or lookup) the names for `setColor` for each type of call depending on the circumstances, and makes it the resultant code more difficult to read. Using an overloaded function `setColor` could make more sense here.
A related instance where overloading can simplify code occurs where values are being set inside a

library. There are cases where objects internal to a library can have extensive sets of parameters which are not exposed through standard methods as with Motif widgets or OpenGL objects. These libraries provide calling routines to set these parameters. These routines take a parameter identifier and then an associated value. Different types of values must then be passed differently. In Motif, the function is defined so that the values are not type-checked at compile or run time, leading to many potential problems. In OpenGL, a large number of different methods are defined for setting different types of parameters. For example, there are 24 different functions for specifying vertices of an object to be drawn. In these cases, overloading provides an alternative that reduces the number of names that a programmer or reader need to remember and that allows at least run-time type checking of the parameter types.

Another use of overloading occurs with constructors. Constructors are special methods that are called implicitly and hence can not be defined easily with different names. We will cover these in See Constructors .

Overloading based on the number of arguments is easier to interpret than overloading based solely on argument types because argument counting can be done without knowledge of the underlying expression types. However, even here one must be careful in how overloading will be used. Additional arguments are often added to a method as the program evolves. The addition of a new argument to one method should not cause problems due to sudden ambiguities between it and another method with the same name. Moreover, even though the resolution process is easier, overloading here still requires additional work on the part of the compiler and the reader.

Default arguments provide an alternative to overloaded functions. They let a library define functions that take a range of parameters that are rarely used but are useful in certain circumstances. These functions can then be used most of the time with their default parameters, providing the programmer with a simplified calling sequence. For example, a graphics library might have a method to specify the colors of an object. Objects, depending on their type, might have between one and three different colors, but generally one have one. The library can provide a single method to handle this case:

```
void setColors(const char *,
const char * = NULL,
const char * = NULL);
```

This method can then be called with one, two, or three colors depending on the circumstances. The alternatives to using overloading here are to always pass three colors (which adds complexity to the calling code in the cases where the additional colors make no sense), or to define three different methods based on the number of colors, which complicates the interface and requires the programmer to remember three different method names.

Default parameters can also be used to merge two different methods into a single method, thereby simplifying the interface of a class. Here, the fact that the additional parameter has a value other than its default causes one implementation to be used rather than the other. For example, the method

```
void displayInfo(const char * text,unsigned howlong = 0);
```

is used in a library package to display a message on the screen. If the second parameter is not provided, the message is the default message and is displayed continuously. If the second parameter is provided, the message is only displayed for that number of milliseconds and then the current default message is restored. The use of default parameters in this case however, is questionable. The complexity of using a single method for two different purposes combined with the need for the reader or programmer to understand the two different applications, tends to outweigh the simplicity of having fewer methods in the interface. Moreover, it is generally not a good idea to have parameters that affect control flow directly.
Based on this discussion, we recommend:

Use overloading sparingly.

While C++ lets the programmer overload global functions and operators, overloading should only be used for methods and only if it greatly simplifies the external interface of an object. It must both reduce the number of methods in the interface and should provide a natural and obvious calling sequence for its intended purpose. Default parameters are not quite as bad as overloaded methods but should also be used frugally and where the simplicity they bring to the interface far outweighs their inherent complexity. Any time you plan to use overloading or default parameters, you should justify its use using these guidelines.

Operator Methods

Another controversial extension that C++ provides is the ability to define or overload operators either globally or for a particular class. Global operator definitions add additional complexity to a program, generally making the program much more difficult to understand and read. They should be avoided. Operators that apply to class operands, while generally not a good idea, do have their applications.
When programmers first see that they can define new operators for a class, they tend to go all out and define lots of operators, planning to simplify the use of the class by using expressions rather than method calls. The end result of this, however, is a program that is totally incomprehensible to the reader (and to the programmer a year or so down the road). The meaning of the various operators tends to be ad hoc and hence not apparent to the reader. Combining calls and relying on operator precedence makes this even more difficult. Operators are also somewhat difficult to define and use correctly. The difference between `x++` , `++x` , and `x+1` for a class object will not be apparent to the reader (or the programmer). Moreover, getting the return type correct, especially for assignment operators, can be confusing.
The only cases where operator definitions are useful are those where their application is natural and can be understood without knowing the programmer's assumptions. There are several such instances. The first involves the use of the >> and << operators for standard I/O. These have already been defined and overloaded as part of the standard I/O library. Where it is appropriate, it

makes sense to define these operators for new classes so that the classes can be used directly for input or output. Because the context of their use has become standard, the extension of these to a class for the same purpose can be easily understood by the reader and the programmer.

A second limited instance where operator methods are acceptable occurs when defining a class that has natural operations. For example, if one were to extend C++ to support a type *RatNum* for rational numbers, defining the basic operators on this type would make sense since the reader can assume the type instances represent numbers and that the operators applied to them have their natural meanings. This is done in the standard C++ library for complex numbers. Note that if this is to be done for the class, then it should be done completely so that rational (or complex) numbers can be used in any expression context where they would make sense. This means that if one operation is defined, all the relevant operations should be defined and that operations should be defined not only for pairs of rationals, but for mixing rationals with integers (and possibly floats) as well. This use of operators is quite limited in that very few classes have natural operations.

A third instance where operator methods can be justified, involves clean extensions to the language to handle classes. One such use involves iterators. An iterator is a class that is used to step through a data structure, looking at each element in turn. Iterators can be defined for lists, trees, arrays, hash tables, as well as arbitrary user structures. An iterator class typically has a setup method where it is defined for a given structure, a test method to check if there are more items, and a next method to step to the next element. For example a list iterator might have an interface like:

```
class ListIter {
ListIter();

void setup(const List&);
int more() const;
void next();

ListItem current() const;
}
```

This iterator could be used in a C++ program using the for statement:

```
ListIter li;
List list;

for (li.setup(list); li.more(); li.next()) {
ListItem itm = li.current();
...
}
```

The normal use of a *for* statement, however, uses operators. We can extend the definition of ***ListIter*** using operators:

```
class ListIter {
ListIter();

void setup(const List&);
int more() const;
void next();

ListItem current() const;

ListIter& operator =(const List& l){ setup(l);return *this;}
operator int() const { return more(); }
ListIter& operator ++() { next(); return *this; }
}
```

Then the *for* statement could be written as

```
for (li = list; li; ++li) { ... }
```

This *for* statement uses the three operators defined above. The assignment in the first part of the *for* uses the *operator=* to set up the linked list iterator. The test in the middle part uses the conversion operator. (Since C++ expects and integer value in this context, it will attempt to convert the class object to an integer and will thus use the given conversion.) Finally the increment part of the *for* uses the *operator++* . This form for a loop construct is similar to that use for a loop that counts such as

```
for (i = 0; i < 10; ++i) { ... }
```

or a loop that goes through a string value:

```
for (p = string; *p != 0; ++p) { ... }
```

and hence might be more understandable one the reader gets used to it. In particular, it is obvious at a glance that the program is iterating through the whole list from start to finish.

Operators can be used in this case if they are used consistently for all iterators throughout the code. Other instances where operators are needed for extending the language are the definition of the -> operator when defining a class that should act like a pointer, and a definition of an assignment operator for classes that can be assigned.

Another instance where operators are useful involves treating objects as functions. The standard library, for example, has algorithm templates that require a function argument. For example, it provides a sort template that takes the relevant comparison function as its argument. Because explicit function pointers are error-prone and add significant complexity to a system, it is desirable to pass an object here rather than a function. To let the object be used as a function, the library uses *operator()* . This lets the code be read as if the passed object were a function, lets the template take only class parameters, lets the object encode any additional parameters that might be needed, and still lets the user pass an object.

This brings us to the guideline:

Define operators only inside classes and only where their use is obvious and natural.

Operators should only be defined when they help the reader of the program to understand what is happening, not when their purpose is to simply make the code shorter. Note that the use of pointers to represent classes sharply limits the utility of operator definitions and should remove much of the temptation to create operators in inappropriate circumstances.

Class Variables

C++ lets variables be declared at a variety of levels in a variety of ways. In general, however, one should restrict variable definitions to as narrow a scope as possible and strive for consistency. Variables are necessary within a routine and can be used freely there. Variable which need to exist beyond the scope of a routine are more troublesome because their domain is broader and it is more difficult for the programmer or the reader to keep track of how and where they might be used. When such variables are needed, the best approach in C++ is to make them private or protected static class members. This restricts their scope to the given class and ensures that access can only occur in methods of the class. The alternatives, making the variable file-level static data or making the variable a global program variable, should be avoided. File local variables should be implemented as static class data. General global data should be avoided where possible by modifying the program design. Where it is needed it should be implemented as a class in itself or using static methods of some known class. Thus we recommend:

Avoid non-class storage.

Memory Management

In this section we look at issues dealing with the creation and deletion of class storage in C++. We start by looking at the use of constructors and destructors. Then we provide a general discussion of techniques for safe memory management.

Constructors

C++ is a large and complex language that is not always intuitive. As an example of this, consider what the following statements might have in common:

```
Move m;
Board b = 5;
Square sq(4,5);
function(sq,m,b);
b = 6;
```

```
Square(4,5);
new Square(4,5);
: field(5)
}
```

While these represent a diverse range of statements (declarations, expressions, and an end of block), C++ lets all of them represent function calls. Some are obvious, such as the call to function. Others are not so clear. The first three represent constructor calls, `Move()` , `Board(5)` , and `Square(4,5)` . The fifth, assuming `b` is a class type, also may represent either a constructor call or a call to the assignment operator. The subsequent line is an explicit constructor call of ***Square*** . The invocation of new and the initialization of a field or subclass may also represent a constructor call. Finally, any end of block may represent destructor calls.

Implicit function calls such as these make C++ difficult to read and difficult to understand exactly what is happening. At the same time, they can make programming significantly easier by hiding many unnecessary details. One of the tasks of the programmer here is to use the language wisely so that implicit calls that are innocuous (such as those that simply initialize a class to default settings) are used to simplify the code and ensure safety, while calls that actually do something are made explicit.

One of the ways of doing this is by treating classes as pointers. A declaration of or assignment to a pointer to a class does not imply a function call. Any constructor for such an object is given explicitly using the new operator. Moreover, there are no constructor or conversion calls when passing such an object as a parameter nor are there implicit calls for the destructor. All this tends to make it clearer to the programmer and the reader exactly what the code is doing.

To understand and simplify implicit functions, one must first understand about constructors. C++ treats constructors as functions. However, constructor functions are always called implicitly. Their call sites include:

- Any declaration of a class object is a constructor call.
- Any assignment to a class object where the right hand side is not of the same type and no explicit assignment operator has been defined that can be used will be done using a constructor call.
- Any parameter that is a class object and is not a ref parameter will be passed using a constructor call.
- A subclass constructor either implicitly or explicitly (using an initializer) calls its superclass constructors.
- A constructor either implicitly or explicitly (using an initializer) calls constructors for any fields that are class objects.
- A call to new for a class represents a constructor call for that class.

In addition to being called implicitly, C++ will also create certain constructor methods for a class

without informing the programmer. If the programmer does not create a constructor for a class, then C++ will define a default constructor:

```
Class::Class()
```

that calls the default constructor for any superclasses and then calls the default constructors for any data fields that are class objects. In addition, the compiler will create a copy constructor of the form

```
Class::Class(const Class&)
```

unless the programmer has defined a constructor that can take an object of the class type (that is, can act as a copy constructor). The copy constructor will call the copy constructor or for each superclass and will call either the assignment operator or the copy constructor for each data field of the class.

These intricacies involving constructors are the source of many misunderstandings in the use of C++ and the cause of many programming problems. To simplify the code, make it more understandable, and to avoid potential pitfalls, the programmer should avoid letting C++ do a lot of hidden work and should attempt to make most things explicit. Thus we first recommend:

Define a constructor for each class.

The programmer should define at least one constructor, even if it is the default constructor and it does nothing. This makes it explicit what it going on, offers a location for setting breakpoints while debugging, and provides a place for adding appropriate initializations as the program evolves.

If a class is going to be treated as a pointer to a structure so that only the pointer will be passed around and used in expressions, this is generally sufficient. However, if a class is to be used otherwise, we strongly recommend:

Define both a copy constructor and an assignment operator for any class that will be passed as a structure.

This will ensure that the programmer can specify the correct semantics for copying an object and not rely on whatever the compiler will generate. It again provides hooks for adding whatever code may be needed later for handling copy correctly.

In general, if objects are to be copied, it is best to define explicit methods to copy them and to not use copy constructors at all. Copying a complex object containing instances of or references to other objects presents an inherent ambiguity. The programmer must differentiate between a shallow copy where references to other objects are copied and a deep copy where new instances of referenced objects are made. Defining an explicit method here lets the program use the method name to make clear what type of copy is being done. Even for simple objects, it is much clearer that a copy is being made and a function being called when an explicit method is used.

Constructors themselves are special functions in C++, handling the initialization of superclasses and data fields implicitly. When a class object is initialized, C++ ensures that the objects from

each of its superclasses and each of its class data fields are also initialized by calling the appropriate constructors. The programmer can make these constructor calls explicit by using initialization syntax such as in

```
Class::Class(int a, int b)
: SuperClass(a), field1(b), field2()
{ ... }
```

The initializer elements represent calls to the appropriate constructors for the fields. If the field is not a class, then the elements represent assignments of the parameter to the field.

There are both advantages and disadvantages to using initializers within constructors. The advantages come from making the various constructor calls explicit. This shows the programmer and the reader that there are constructor calls being made here, hopefully indicates which constructor is being called, and lets the programmer specify a constructor other than the default constructor. Initializer syntax is actually the only way that a constructor call can evoke anything other than the default constructor on a superclass or a class field. Using initializers also can be slightly more efficient in that if they are not used the default initializer will be called in any case. Finally, initializers represent the only valid way of assigning to a const data field.

There are three disadvantages to using initializers. The first is that while they are specified in a given order, the compiler will actually implement them in the order the declarations are given. If any of the constructors have side effects or if order is important, the resultant program might not work as expected. Second, the parameters for the initializers all have to be readily available, generally computed from the parameters of the constructor itself or constants. It is difficult to pass anything complex or conditional to an initializer. Finally, initializers detract from the readability of the program. Almost everywhere else in the language, values are assigned using an assignment operator. Programmers and readers are going to look for assignment operators to see where things are set and changed. The assignment of values in an initializer is not as intuitive and straightforward as it would be using a simple assignment statement.

Based on these pros and cons and our bias towards consistency and simplicity, we recommend:

Use initializers only for superclasses and constant data members.

If the default constructor for a superclass is both simple and sufficient, we will generally omit the corresponding initializer, but it can be left in to indicate the implicit call. We do not recommend the use of initializers for data fields. Since most data fields are not class structures (especially if classes are represented as pointers), using assignments within the constructor is more readable and is more consistent with assignments and initializations in the rest of the program. If you do prefer to use initializers, they should be used completely and consistently, i.e. they should be used either for nothing as we recommend, or for everything.

Destructors

Just as constructors are called implicitly whenever a class object is created, a destructor is called

whenever the object is removed. Each implicit constructor call is paired with an implicit destructor call. Constructors called for declarations have corresponding destructors that are called at the end of the scope in which the declaration occurs. Constructors called when using the new operator have destructors called when the corresponding object is freed using the delete operator. Constructors for superclasses are called automatically when a subclass constructor is invoked. Similarly, when a subclass is deleted, the destructor for each of its superclasses will be called implicitly. Finally constructors that are called for temporary objects such as parameters have destructors that are called at the end of the statement in which the temporary occurs.

A class object is generally responsible for its own data. Just as the constructor is useful for initializing this data, the destructor is useful for cleaning up. This generally means freeing any storage that was allocated for the object and is no longer needed once the object is removed. Just as we recommend that constructors always be defined, we also recommend:

Define a destructor for every class with a constructor.

This should be done even in the normal case where no action needs to be taken if only to provide a hook for future changes to the program. Note that even if the programmer does not define an explicit destructor, C++ will define one for the class that does nothing. Note also that a destructor, whether defined explicitly or created by C++, will automatically call the destructors for any class fields of the object and any superclasses. Thus, the code within the destructor should only deal with the local, non-class fields of this object.

Safe Memory Management

Memory management is probably the one area of using C++ that is the most problematic. Memory management problems are both easy to create and difficult to detect. Whole companies have been formed just to help programmers find real or potential memory problems within programs. The best way to write working C++ code is to plan memory management as part of design and to keep it at the forefront during implementation and maintenance.

There are three types of memory management problems. Since C++ does not check array bounds problems occur when insufficient storage is allocated for an array. We recommend that this problem be dealt with using defensive coding techniques:

Always check array indices against the bounds.

This can be done in a variety of ways. One should check parameters and input values to ensure that whatever array is being accessed is valid. Whenever an array is passed to a function, the size of the array should also be passed so that such checking can be done. If output from a function is being put into an array, then the function should check that only the specified amount of output is inserted. This problem is probably most manifest in dealing with character strings. A good C++ program should always check that a string buffer has enough space to hold whatever string is being put into it and either truncate or generate an error if it does not. Again, whenever a buffer is

passed to a function, the length of that buffer should also be passed. Another technique here is to use the class `ostrstream` from the C++ library for dealing with string buffers. This class does explicit bounds checking whenever an element is inserted.

Another way of checking array bounds is to create classes to represent arrays. The classes can then define appropriate indexing operations that do bounds checking. While this can be done on an instance-by-instance basis, it is easier to create a template class to do it in general. The standard vector template class provided with C++, however, does not do bounds checking (a major drawback). In See , we illustrate how an appropriate array template could be constructed.

The other forms of memory management problems involve pointer management and arise frequently when pointers are used to represent classes. The first and most obvious of these problems involves dangling references. These occur when storage is created and then freed, but the pointer to the storage is still being used. Since C++ tends to reuse storage in order to minimize program size, the result is that the storage is reallocated and used simultaneously for multiple purposes, yielding weird and wondrous results somewhere down the road. A partial solution to this is to be sure to set any pointer to an object being deleted to `NULL` .

A second type of pointer management problem occurs when memory is allocated and never freed, even after all pointers to it have been removed. This does not generally lead to immediate problems, but tends to make the program larger than necessary at run time. Programs typically will run for much longer than the original programmer anticipated and such memory leaks tend to accumulate. For example, we wrote a programming environment that we tested on our own programs, running it for an hour or two at a time and had no memory leak problems. When we released the environment to students, we found students who stayed in the environment for twelve or more hours and memory leaks became significant.

While there are no fixed rules that will guarantee that a program will not have memory management problems, there are a few guidelines that can be followed that will minimize the problems that do arise. The first is:

A class is responsible for its own data.

This means that any non-class data that is allocated for an object such as strings or arrays should be freed when the object is freed. It also means that classes should not pass back pointers to internal data that might be freed since the validity of such pointers can not be guaranteed outside of the class. The deletion code for this data should be added to the destructor as the data is added to the class.

Managing dynamic data within a class is made more difficult if the class is not treated as a pointer and can be copied. If the default copy constructor is used in this case, then both the original class and the new copy will contain pointers to the same dynamic data. Then, when the copy is freed, the destructor for the copy will delete the storage that is still being used by the original object, or the free will be omitted and there will be memory leaks. If dynamic data is used with a class that is not treated as a pointer, then a copy constructor and assignment operator should be defined and any dynamic storage should be duplicated in the copy.

Making dynamic data local to a class and keeping it that way takes care of non-class dynamic storage, but dynamically allocated classes generally have to be treated differently since these will be passed around and have more indeterminate lifetimes. The solution we recommend to managing class storage is to assign each object an *owner* who is responsible for managing the storage of that object. The owner can be another object or it can be a function or method. Consider how we implemented the knight's tour problem. The ***KnightMainInfo*** object was owned by the function `main` which both allocated and freed it. The ***KnightBoardInfo*** object was owned by the ***KnightMainInfo*** object which created it in the `process` method and deleted it in the destructor. The various ***KnightSquareInfo*** objects were all owned by the ***KnightBoardInfo*** object which allocated them in the constructor and freed them in the destructor. The ***KnightHueristicInfo*** objects were owned by the `findRestOfTour` method of ***KnightSquareInfo*** . Finally, the ***KnightSolutionInfo*** object is owned initially by `KnightBoardInfo::findTour` which creates it, and then the ownership is passed back to `KnightMainInfo::process` which uses and frees it.

A good object-oriented design for C++ will include information on who owns each object. Objects can have many potential owners, but there can only be one actual owner at a time. For example, the ***KnightSolutionInfo*** object in our example program had two owners, the creating method and the receiving method. In a drawing package, a graphics object might have a dynamically assigned owner. If it is a top-level object on the display, it might be owned by the drawing area object; if it is a subobject in a group it might be owned by the group object; and if it was recently deleted, it might be owned by either the cut buffer or by the undo list. This information should be considered part of the design. Moreover, the programmer should track, maintain, and document this information as the code is being written. Here we recommend:

Assign an owner to each object whenever possible.

We will illustrate the use of object ownership for storage management as we consider further examples throughout this text.

The qualifier on this recommendation indicates that there are situations where ownership is not a viable option for memory management. Consider for example the ***BaumString*** objects created by C++ front end to Motif we used as an earlier example. The ***BaumString*** objects were created by the library but are actually managed by the application using the library. As such it is impossible within the library to assign an owner and ensure the object is freed correctly. A second exception arises where there might be multiple potential owners of a class. For example a program that maintains a symbol table may have symbol objects. The symbol objects may have one owner that is actually creating and using them and at the same time, have other owners that are the hash tables that are used for symbol table lookup.

The issue that arises is how to handle these cases cleanly and without incurring memory management problems. There are several alternatives. The easy one is to use garbage collection as is done in Java. Here the language itself does storage management, automatically freeing storage once all references to that storage have disappeared. Unfortunately, this solution is not currently

available for C++ programmers and alternatives are needed.

One alternative for the library-application problem is to put the onus of storage management on the application. Here the library should explicitly document that the storage returned from a given method or function is dynamic and must be freed by the caller. (Some libraries even provide separate free routines for each type of storage to make it clearer what is going on and to isolate the allocation storage management from library storage management.) This requires the application programmer to be more careful and to work harder, but makes writing the library easier.

An alternative that splits the work between the library and the application is to do some sort of reference counting. Here each object keeps a count of its users. When an application routine gets a handle to the object, it increments the count. If it passes the object on to other routines or objects, they also increment the count. Whenever an object or routine is done with the library object, it decrements the count. Incrementing and decrementing counters can be done by adding two methods to the library object:

```
LibObject::addRef()
LibObject::removeRef()
```

The first of these increments an internal counter on the object. The second decrements the counter and, if it reaches zero, actually deletes the object. This is shown in more detail in See Wrapper Classes . As we get into more sophisticated examples, we will see how such reference counting is actually used. Note that, with appropriate use of C++ classes that represent pointers, it is possible (but somewhat difficult) for all the reference counting to be done automatically without requiring any special calls of the application. Note also, that reference counting only works if there are no cycles of pointers among reference counted storage.

C++ Types

The typing model of C++ is complex in that it augments the already messy model used in C with classes. To use the language effectively, the programmer must avoid some of the typing features and make good use of others.

Typedefs

The syntax for defined C++ types, inherited from C, is often nonintuitive. Consider the declaration:

```
const char (* (*x)(char * const,int))(rtn);
```

This declares the variable *x* as a pointer to a routine taking two arguments and returning a pointer to a *const char*, that is a string. The variable is initialized to the actual routine *rtn*. Such declarations are impossible to both write as a programmer and to read and understand in the code.

However, C++ provides a variety of techniques whereby the type system can be used to make such code more readable.
The first language feature that is helpful is the use of typedefs to define new type names. Using typedefs we can simplify the above declaration:

```
typedef const char * ConstText;
typedef char * Text;
typedef ConstText (*RoutinePtr)(const Text,int);
RoutinePtr x = rtn;
```

The resultant code, while longer, is much easier to understand. Moreover, the definitions are more consistent. For example, the *const* for the first parameter of the routine can be put before the type name rather than afterwards, and the variable being declared is put after the type name in the actual declaration.
Typedefs are also useful for standardizing types and type names throughout a system, for ensuring that the names of built-in or library types conform to the naming conventions that are being used in the system, and for letting types change when porting or maintaining a system. As such, we recommend:

Use typedefs extensively.

Typedefs should be used, as we saw in the knight's tour example, for defining a standard set of types (*Integer* , *Boolean* , *ConstText* , *Text* , etc.), for defining the names of the object types that are actually pointers to classes, and, in addition, for defining any complex type such as the routine pointer in the above example.

Enumerations

Another very useful typing feature in the language are enumeration constants. Enumerations can be used for their intended purpose of specifying a related set of constants where only one is applicable at a time. They also can be used inside a class definition to define a single constant that is local to the class, as in:

```
class Sample {
public:
enum { EMPTY };
...
};
```

Here the constant *Sample::EMPTY* can be used and has whatever protection (private, public, or protected) is in effect when the *enum* definition occurs. While some compilers allow standard constant declarations within a class (using *static const* ...), most do not and enumerations are thus the preferred method here.
Enumerations can also be used to define sets of related flags as in:

```
enum PinSide {
```

```
PIN_SIDE_NONE = 0,
PIN_SIDE_FRONT = 0x1,
PIN_SIDE_BACK = 0x2,
PIN_SIDE_TOP = 0x4,
PIN_SIDE_BOTTOM = 0x8,
PIN_SIDE_LEFT = 0x10,
PIN_SIDE_RIGHT = 0x20,

PIN_SIDE_ALL = 0x3f
};
```

Where combinations of the flags may be passed. Because enumerations provide a logical way of associating a group of related constant names as well as define a type name that can then be used to further document parameter or variable types, we recommend:

Use enumerations to define any related set of constants and all class constants.

Naming conventions should be adhered to when defining enumeration types and constants. All enumeration constants should be viewed as constants which our basic conventions reflect as all upper case names with underscores separating words. Moreover, enumeration constants that are not defined within a class, such as those of `PinSide` above, should have a name prefix that indicates the package (`PIN` in this case) and the type (`SIDE`). Enumeration constants defined inside a class can omit part or all of this prefix since the class name needs to be used to reference it globally.

Pointer Types

While typedefs and enumerations are useful features in the language, C++ also includes typing capabilities that should be avoided. In particular, C++ extends the notion of pointers to include pointers to class fields and methods. In C pointers are used extensively. In particular they are needed to return values and to implement dynamic calls using pointers to functions. In order to provide similar features in C++, the language needed to be extended to associate a class with a pointer since class fields and class methods are treated differently than standard types. The result is some messy syntax (`::*` , `->*` , and `.*`) and generally incomprehensible code. Moreover, because C++ provides both ref parameters and virtual methods, such pointers are rarely needed and can be easily avoided. Hence we recommend:

Do not use pointers to class fields or methods.

Conversion Functions

Constructors in C++ can server multiple purposes. Their obvious application is to initialize the storage for a class. They can also be used, however, for converting data from an arbitrary type into

an object of the given class. Suppose, for example, we wanted to extend the language by adding the class *RatNum* for rational numbers. In this case it would be desirable to be able to mix integers or floating point numbers with rational expressions. This requires that we let C++ implicitly build rationals from either integers or floating point numbers and that we can generate the floating point equivalent of a rational. Conversions from arbitrary types to elements of the class *RatNum* can be handled by defining appropriate constructors:

```
RatNum::RatNum(long value)
RatNum::RatNum(double value)
```

C++ will then use these constructors in a context where an integer or floating point value is present and a *RatNum* is called for. Note that just having these two constructors is probably not sufficient. If an *int* or a *short* is used in an expression, some compilers will get confused as to whether the *int* should be converted to a long or a double before being used to call the constructor. In general, if one defines a conversion function for both integers and floating point one must define conversions for each type of integer and floating point value that might be used. The conversion from a rational number to a floating point cannot be handled using a constructor. This must be done using a separate conversion operator:

```
RatNum::operator double() const
```

This function will again be called implicitly by the compiler if a *RatNum* is used in a context where only a floating point value can occur.

Conversion functions can also be used for convenience. For example, we have defined a set of C++ classes that server as a wrapper for the Motif windowing package. Motif defines its own string type that is capable of handling extended character sets as well as font changes within a string. Most programmers writing Motif programs, however, are using simple C strings. To simplify the use of Motif, we defined a class ***BaumString*** that served as an intermediary. A ***BaumString*** object can be created from either a C string or Motif string (*XmString*):

```
BaumString::BaumString(); // Null string
BaumString::BaumString(const char *); // C string
BaumString::BaumString(XmString); // Motif string
```

Moreover, we let ***BaumString*** objects be used wherever a Motif string would be used:

```
BaumString::operator XmString() const;
```

This lets us declare methods that take a ***BaumString*** as an argument and pass it directly to motif. The programmer can call these methods with either a simple C string, with a Motif string, or with a ***BaumString*** created previously.

While conversions can be convenient, they are also the source of many potential and real problems within C++. Conversion functions, by their nature, represent implicit calls that are not obvious to the programmer and may cause the program to do something unexpected which would not be obvious from reading the code. Moreover, if too many conversions are defined, the compiler will begin to complain about ambiguous conversions. For example, if both the

conversions:

```
ClassA::ClassA(const ClassB&)
ClassB::operator ClassA() const
```

are defined, the compiler will not know which to apply when converting an object of type ***ClassB*** to one of type ***ClassA*** . Based on these potential problems, we recommend that conversions be limited to cases where a reader would understand what is happening:

Use conversions only where there is a natural mapping between the two types.

Constructor-based conversions should be used in preference to operator based conversions wherever possible. Moreover, as we noted, if a conversion from an integer or floating point type into a class is defined, then conversions should be defined for all other numeric types. Note also that the use of pointers to represent classes avoids most of the cases where C++ will use conversions implicitly. Also, the more recent C++ standards have introduced the notion of an explicit constructor (defined using the `explicit` keyword in front of the constructor definition). Such a constructor is guaranteed not to be called implicitly by the compiler. Note, however, that many compilers do not yet support this feature.

Comments

We end our discussion of using C++ effectively by considering comments. C++ provides two types of comments, inline comments beginning with `//` and C-style comments delineated by `/*` and `*/` . Both of these should be used extensively in coding along with blank lines to enhance the readability and understandability of the program text. Throughout our discussion of implementation and the use of C++ we have emphasized;

Write code to be read as well as executed.

To be useful, comments should be both appropriate and meaningful. One should assume that the reader can understand the code. Hence the comments should not just duplicate what the code says, but should go beyond it. Comments should provide additional information relating to design or a higher-level view of what a particular section of code is doing. Any code that is not straightforward and hence self-documenting, should have an associated comment.
Comments should also provide a formatting structure that makes the program look good. Each file, section, class, and routine should start with a block comment that identifies the item and provides enough information so that a reader can know whether to read the item in more detail or skip it for a particular issue. The block comment should be separated with white space and should stand out. For inline comments describing declarations or code fragments, we generally use C++ style comments.

In any case, commenting code is something that programmers need to get in the habit of doing.

Many programmers, for expediency, will write the code first, get it working, and comment it afterwards (for example, just before handing it in as homework). This has the advantage that one only needs to comment code that works and that code can be written with less typing since comments aren't needed initially. It has several disadvantages however. First, comments written after the fact tend to be inaccurate. Programmers may or may not recall every detail of the code and, when trying to add comments in a hurry later on, will not always be correct. Such comments, for similar reasons, also tend to be more duplicative of the code rather than providing the reader with auxiliary information. Finally, if comments are not inserted until the end, they are not available to help the programmer while debugging and getting the program to work in the first place. Adding comments as you write the code also has its own advantages. In particular, the comments can serve as placeholders for what other things need or can be done, serving as stubs that let the programmer come back later and easily fill in the details. Hence we strongly recommend:

Comment code as it is entered.

One other application of comments is as placeholders for code that has not been written or to remind the programmer of potential problems with the code. In both of these cases, the programmer wants to be sure to get back to the code at some later time. Here the programmer should adopt some commenting convention that makes these types of comments look different from normal informative comments and that lets all such comments be readily located. For example, formatting the comment as

```
// ########## Check for special case of x = 0 here
```

not only makes the comment stand out, but also lets the programmer find all such comments (and hence what is left to be done) by searching all source files for the text string `"##########"`.

Summary

Throughout this chapter we describe how to make effective use of C++ using a set of guidelines. These include:

- Use pointers to represent classes.
- Use the most restricted protection levels possible.
- Data elements should always be private or protected.
- Avoid the use of friend declarations.
- Use `const` as much as possible in declarations.
- Use `ref` parameters whenever a value is to be returned.
- Use simple types for parameters wherever possible.

- Use overloading sparingly.
- Define operators only where their use is obvious and natural.
- Avoid non-class storage.
- Define a constructor for each class.
- Define both a copy constructor and an assignment operator for any class that will passed as a structure.
- Use initializers only for superclasses and constant data members.
- Define a destructor for every class with a constructor.
- Always check array indices against their bounds.
- A class is responsible for its own data.
- Assign an owner to each object whenever possible.
- Use typedefs extensively.
- Use enumerations to define any related set of constants and all class constants.
- Do not use pointers to class field or methods.
- Use conversions only where there is a natural mapping between the two types.
- Write code to be read as well as executed.
- Comment code as it is entered.

To these should be added the recommendations in the previous and subsequent chapters.

Exercises

1. The best way of testing program readability is to try to read a program. This is difficult for ones own code, but much easier when one is reading someone else's code. Take a program that was done for one of the earlier exercises or elsewhere and exchange programs with someone else. Analyze the program you are given and come up with a list of suggestions as to how it could be made more readable.

2. None of the guidelines given here is absolute or agreed to by all programmers. Take one or two of the guidelines that you disagree with and write up a justification as to when and why this guideline should be violated.

computer science II

cmsc 214
spring 2002

Writing Test Drivers

Revised February 23, 2002.

Background

Even though many of you have had several semesters of experience programming, you may be surprised to learn that writing code in the "real world" (or at least, some parts of the real world) involves a lot more planning and testing than you think.

For example, most beginning programmers believe "I must spend all my time coding, because time spent planning and testing code is a waste of time". Yet, in many software companies, you may spend more than half your time planning how to write code. This involves sitting down and determing what you want the code to do. You may think about the "actors" (objects that do things) and what they do ("actions"). Actors translate to classes and actions to methods in those classes.

Once those decisions have been made, and coding has begun, you need to think of how to test the code. Companies often have separate groups of people: those who write code (developers) and those who test code (testers).

At this point, you aren't ready to work in teams, so some of that kind of testing is postponed to a course in software engineering.

Nevertheless, it's a good idea to test your code. The question is: how?

Primary Input and Primary Output

To make sure your code passes minimum criteria, the introductory programming courses have developed a concept known as primary input. The primary input is often a file which is read in from your program either as a legitimate file or more often than not, through input redirection. Your program must then produce an output which matches the primary output. The primary output is the "correct" solution to the primary input.

Primary Outputs: Are They a Crutch?

It's common in the first few programming courses to use primary inputs and outputs. However, some students use primary inputs/ouputs too heavily as a "crutch" (this is what people with, say, broken feet use to help them move around). A "crutch" is an aid.

Students seem fearful of writing their own test files *before* using primary inputs and outputs. They literally do not trust themselves to write their own test files successfully, and want to see an "official version". But think about it this way: if you're writing your own code, who's going to give you a primary input? Doesn't it make sense to learn how to write your own test code? This means not only "feeding" your program with *your* input, but also determining what output *ought* to be produced by your input.

Don't underestimate the importance of learning to write your own test files. Doing so forces you to determine what a reasonable output should be. Many CMSC 106 students are, obviously, inexperienced programmers. Some of them try to match primary output. When they fail to match primary output, they scratch their heads and head to a TA.

A typical encounter goes like this: "TA, TA! My output doesn't match primary". "Student, what's wrong? Why doesn't it match?" "TA, TA! I don't know! I looked at my output, and it's different from the primary." "Stduent, if I took away the primary output, could you tell me what output the primary input would produce?" "TA, TA! Don't touch my primary output! How else am I supposed to know what's going on?"

OK, so the scenario is artificial, but not too far from what happens. Students often come to office hours, and have no idea how the primary output is produced. All they know is that their own output and the primary output are different. In effect, these students have become human "diff" machines, unaware of *why* their output is different.

You may believe that matching the primary output is the quickest way to get finished. Perhaps it is. However, you may be hurting yourself in the long run, if you don't learn how to write your own test files. And you may also be hurting yourself if you don't test your classes before writing test files.

Testing Classes

How do you write classes? Do you write one method, then test, then a second method and test? Do you write all the code to the entire class and test the class afterwards? Or do you write all your classes, and test afterwards?

Some people believe in writing as much code as possible, without a thought to testing the code. They do so even as they've been told to test after each method. Such people believe in the "code first, test (much) later" approach to code writing. There is some virtue to this approach. You learn to write lots of code. You learn to deal with hundreds of errors messages at a time, without flinching. You also may be spending more time debugging than you need to.

There's an approach you can take that may seem rather radical. You can actually write a "test driver" prior to coding the class. Yes, *before* coding the class. Of course, you might

wonder how it's possible to do this. This doesn't happen until you plan the class ahead of time. This means writing down methods, data members, etc.

We'll look at how to try this idea out.

You can still even write such test drivers after the fact. You've written your class, and now you want to test it. After all, you should check to see if your class works, right?

`runTestDriver()`--Organizing the Test Driver Code

We're going to begin by talking about how to test a class once it's been written. This will help you think about the order you should test your classes even before it's written.

Where should you write the testing code? One possibility is to create a file, such as **FooTester.cpp**, have a **main()** and put all the code there.

However, there's a nicer place to put the testing code (to be called *test driver*)---in the class itself.

In the class that you plan to test, write a *static* method called **runTestDriver()**. Why *static*? A static method is one that doesn't belong to an object. It's basically like a standalone function, but resides inside a class. Thus, you are forced to declare variables of the class, etc., inside this method.

Let's look at the details:

1. First, in your .h file, write

   ```
   static void runTestDriver();
   ```

   in the *public* section of your class. This will allow the test driver to be run in any function (say, **main()**) as follows:

   ```
   int main() {
       Foo::runTestDriver();  // runs test driver for Foo object
       return 0;
   }
   ```

   Since the method is static, you need to call it by prefacing the method call with **Foo::** or the name of the class you are testing, followed by two colons.

   Convention Call the function you want to run tests on **runTestDriver()**.

   If you need to have helper functions, you should preface them with **testDriver**. For example, suppose you need to do some operation such as "count", then you can write a static (yes, static) method called **testDriverCount()**. This method, like all helper functions, can be placed in the PRIVATE section of the class header file.

2. Second, write the method in your .cpp file, as in:

   ```
   void Foo::runTestDriver() {
      // put your testing code here
   ```

```
}
```

Notice that they word `static` is no longer here. Just some C++ rule.

OK, now you know where to put the code, let's look at an example.

Suppose you're starting to write a class. You want to test a method at a time. Which methods should be implemented first, so that it makes testing easier? Here's how I would do it:

1. **Write the default constructor and `print()` method first (which can be used to implement the output operator).**

   You want to start simple, and the simplest place to start is the default constructor. You also need a way of determing whether the object is "correct" and one way to do this is to be able to print it out.

   Here's how I suggest you start out (this is in **Fract.cpp**):

```
#include <iostream>
#include "Fract.h"

using namespace std;

void FractDriver::runTestDriver()
{
   Fract f1;  // default fraction

   cout << "Testing default constructor" << endl;
   cout << "---------------------------" << endl;
   cout << f1 << endl;
   cout << "[ANSWER] 0 / 0" << endl << endl;
}
```

   This can be called in **main()** in, say, **main.cpp**.

```
#include <iostream>
#include "Fract.h"

using namespace std;

int main()
{
   FractDriver::runTestDriver();  // calls driver
}
```

   It's easy, it's simple. But there are important features. First, there are some output lines to indicate what kind of test is being run. In this case, it's the default constructor being run.

   Then, the object is printed. Then, the "expected answer" is printed. It's useful to print the expected answer, so you can anticipate what *you* think the output is. It's often a good idea to know what you think the output is and see if it is indeed the

output.

2. **You may wish to test other constructors (except the copy constructor, which is tested later)**

```
#include <iostream>
#include "Fract.h"

using namespace std;

void Fract::runTestDriver()
{
   Fract f1( 2, 6 );  // fraction 2/6

   cout << "Testing next constructor" << endl;
   cout << "------------------------" << endl;
   cout << f1 << endl;
   cout << "[ANSWER] 1 / 3" << endl << endl;
}
```

3. **If you've written the copy constructor and assignment operator, you may want to test that next.**

   If you haven't written it because you intend to use the implicit version created by the compiler, then testing the copy constructor and assignment operator may be silly. Still, it doesn't hurt. When you test these methods, create an "original" object, make a copy (one copy using the copy construtor and another using assignment operator), modify the original, and see if the copy is modified. If it's a deep copy, then the copy should be the copy of the original PRIOR to the original being modified. Here's a sample test driver.

```
#include <iostream>
#include "Fract.h"

using namespace std;

void FractDriver()
{
   Fract orig( 2, 5 );
   Fract copyOfOrig = orig; // copy constructor
   Fract secondCopy;

   secondCopy = orig; // assignment operator

   orig.add( 1, 5 ); // modify orig

   cout << "Testing copy constructor" << endl;
   cout << "------------------------" << endl;
   cout << orig << endl;
   cout << "[ANSWER] 3 / 5" << endl;
   cout << copyOfOrig << endl;
   cout << "[ANSWER] 2 / 5" << endl << endl;

   cout << "Testing operator=" << endl;
```

```
    cout << "-----------------" << endl;
    cout << orig << endl;
    cout << "[ANSWER] 3 / 5" << endl;
    cout << secondCopy << endl;
    cout << "[ANSWER] 2 / 5" << endl << endl;
}
```

As you may notice, it may *not* be a good idea to test the copy constructor and assignment operator right away. The reason? You need methods to modify the original object. If you only have a default constructor, the object won't have changed much. Especially for objects that store dynamic memory (linked lists, trees), you need methods, such as **add()** so the object will be complex enough to fully test the copy constructor/assignment operator.

> **Rule of Thumb** For complex objects, test copy constructor and assignment operator later. Get simpler methods to work first.

This is especially true if you are testing linked lists. You need a way to create something besides an empty linked list to properly test the copy constructor/assignment operator.

4. **Test the other methods.**

   There should be several other methods, and you should write tests for them.

How We Can Check What Your Test Driver Does

While you can testing as throughly or as leniently as you want, we should be able to see the results of your test by calling your test driver. Thus, we should be able to make a call to **Foo::runTestDriver()** (where **Foo** is replaced by the class you are testing), and it should print information.

Drawbacks of Printing

Other than using a debugger, the most common way to debug code is to print. Usually, one uses a debugger to locate an existing bug. Writing test drivers is a preventive measure. You try to find the bugs, before they find you.

The advantage of printing is that it's easy to write the code. However, printing output has drawbacks. It's hard to automate. You, the programmer, must sit down and determine whether your code passed the test or not by visually inspecting line after line of output.

Wouldn't it be nice if you could somehow automate the steps so that it only prints when something has failed? That would be convenient.

Alas, it takes more time to write automated tests, than simply printing everything. But that time may be well worth it (if you have lots of time). You can often run test after test, with very little effort. You don't have to recompile code.

To write such tests, you have to be reasonably clever to come up with tests to fit whatever class you are testing. Let's look at an example. Suppose you wish to test whether a sorted linked list is sorting correctly or not. You can create an input file that looks like:

```
add 4
add 3
add 10
add 8
add 5
answer 4 3 5 8 10
```

Your test driver processes the commands on each line. When the test driver sees "add", it adds the number as a new element of a linked list. When your test driver sees "answer", it will double check to see if the answer is correct. The "answer", in this case, is 4 (indicating the size of the list), followed by the four numbers in sorted order. In effect, it's what's supposed to be in the linked list. The test driver will check the expected answer (from the input file) to the values in the linked list, and see if they are the same.

For example, here's a way to to write a test driver that can handle the above input file.

```
#include <iostream>
#include "List.h"

using namespace std;

void List::runTestDriver()
{
   List list;
   string cmmd;

   while ( cin >> cmmd )
    {
      if ( cmmd == "add" )
        {
            int num;
            cin >> num;
            list.add( num );
        }
      else if ( cmmd == "answer" )
        {
            Iterator iter = list.iterator();
            int size;
            cin >> size;
            if ( size != list.size() )
               cout << "Error: sizes do not match" << endl;

            int *arr = new int[ size ];
            int index = 0;
            for ( iter.goFirst(); iter.inList(); iter.goNext() )
               {
                   if ( iter.getCurrent() != arr[ index ] )
                     {
```

```
                        cout << "Error at index " << index << endl;
                        cout << "Expected " << arr[ index ]
                             << " but see " << iter.getCurrent() << endl;
                    }
                index++;
            }
        }
      else
         cout << "Invalid command: " << cmmd << endl;
    }

}
```

This isn't the cleanest test driver (there could be many improvements). However, it gives you some idea how you could write a test driver which can "automatically" check if the answers are correct, and only print errors when something is wrong.

Admittedly, the input file requires that you write the solution down, but once you do, it's easy to create many variations of the input, without having to recompile the code.

Writing Debugging Methods

Sometimes, it's even useful to write methods whose sole purpose is to help you debug the class. For example, suppose you have a sorted linked list. You might want to write a `verify()` method.

This method might have a prototype that looks like:

```
bool debugVerify( int arr[], int size );
```

The word "debug" is placed in front so that it's easy for the person looking at the class that it's a "debug" method, and not really meant for use by the user of the class.

This method takes an array and the array's size as input. It assumes the array is sorted with exactly the same contents as the linked list. Again, you should be able to see how this method can be used with the driver in the previous method (which checked for the "answer").

While this method is not useful to the "user" of the class, it is useful to you, the developer of the class.

Figuring out Test Cases

For complicated classes, like linked lists, you may need to create very thorough tests. For example, to properly test a linked list, you should create all possible ways to insert three values (e.g., largest, middle, smallest, OR smallest, largest, middle, etc). If you can get them all correct, then it probably works on larger inputs sizes.

The key is to make simple tests first, which test all possible situations, then work your way up to more complicated tests. Some programmers pick very difficult cases first (often relying on the sample input or output provided), but refuse to try simple cases

(complaining it takes too much time to write their own test cases). Start with simple input cases, debug that first, then work your way to more complicated cases. You'll discover that fixing problems in small test cases often means avoiding problems in large test cases.

Code Coverage

Although we won't use this method much, another way to test code, is to create tests based on code coverage. There are two rules for writing test cases based on code coverage: one for "if-else" statements and one for loops.

if-else statements

Suppose you have an if-else statement like the one below.

```
if ( <cond 1> )
  {
    cout << "In body 1" << endl;
    // body 1
  }
else if ( <cond 2> )
  {
    cout << "In body 2" << endl;
    // body 2
  }
else if ( <cond 3> )
  {
    cout << "In body 3" << endl;
    // body 3
  }
```

You will need to write four test cases. The first test case should cause "In body 1" to print. The second should cause "In body 2" to print. The third should cause "In body 3" to print. The fourth should cause nothing to print (since there is no final else case).

Loops

```
while( <cond> )
  {
    cout << "In while body" << endl;
    // body
  }
```

For a loop, you need to write one test that causes the body to print, and one test for it not to print at all.

Combining Loops and if-else

Using the two rules above, you should be able to combine them to determine all possible paths through your method, and generate a test for each possible path. Unfortunately, if your method is very long, this may require many, many tests. This is one reason to keep your methods short (by calling helper functions)---it makes it easier

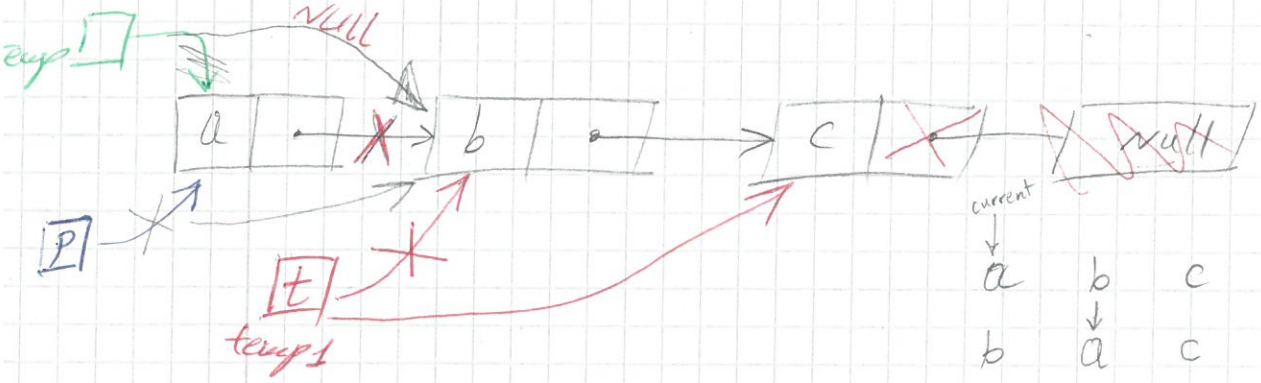
to test.

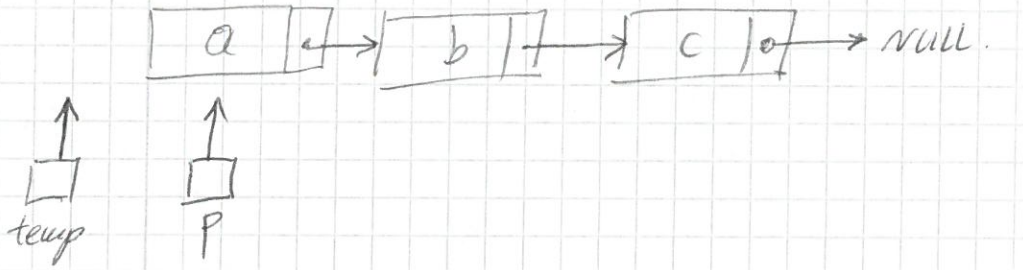
There are some software tools that check for test coverage. Basically, you use the tool to "instrument" your executable (which modifies the executable with special testing code). Then, you run a series of test files. The tool checks to see which lines of code are run and which are not run.

If a certain line of code is not run, then there are two possibilities. First, you did not create a test case that would make the code run (and thus, that piece of code was untested), or second, it's impossible to get to that piece of code (because of various conditions that are impossible to satisfy), in which case, the code is not important and shouldn't be there.

Unfortunately, it's hard to tell which of the two cases has occurred. All you know is that some code wasn't run, so it wasn't tested. Based on that information, you need to figure out if it's possible to write a test case to make the code that didn't run, run.

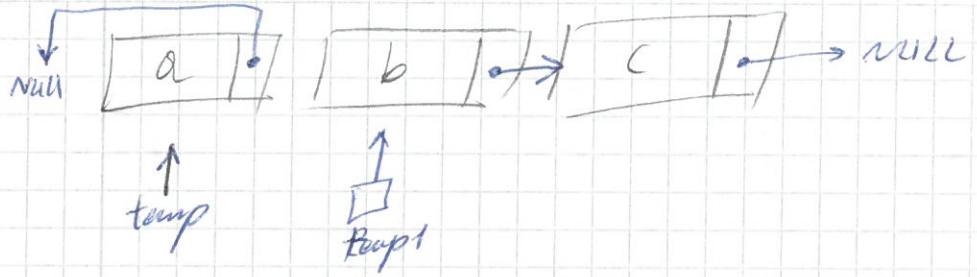
We won't be using such tools in this course, but it's useful to know that they exist.

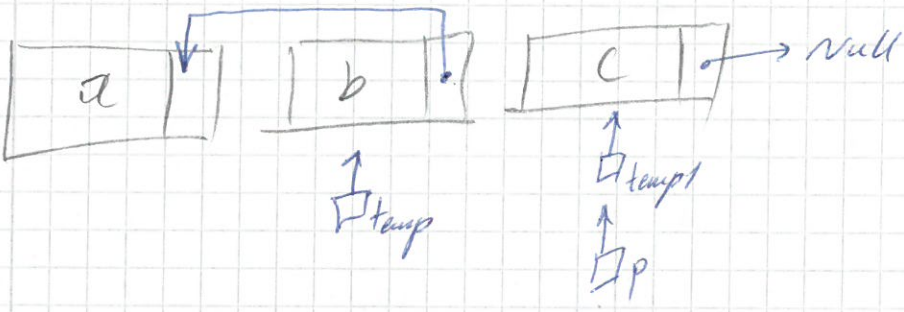


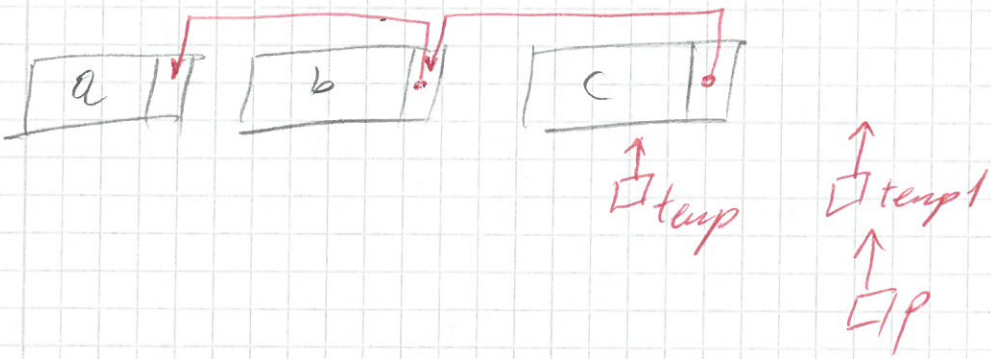


Iterative method for reverse linked-list

P 64







Valgrind is a memory mismanagement detector.

Memory leak messages: There are several kinds of leaks; The 2 most important categories are:

- "Definitely lost": your program is leaking memory Fix it!
- "Probably lost": your program is leaking memory, unless you're doing funny things with pointers (such as moving them to point to the middle of a heap block).

Memcheck also reports uses of uninitialised values, most commonly with the message "conditional jump or move depends on uninitialised value(s)". it can be difficult to determine the root cause of these errors. Try using the

--track-origins=yes

to get extra information.

it is worth fixing errors in the order they are reported, as later errors can be caused by earlier errors.

"still reachable" category is not a memory leak. it means that your program still has reference to memory that will be freed later.

Valgrind can detect the following (the popular) memory related problems:

- Use of uninitialized memory.
- Reading/writing memory after it has been freed.
- Reading/writing "off the end" of malloc'd blocks.
- Memory leaks - where pointers to malloc'd blocks are lost forever.
- Mismatched use of malloc/new/new[] vs free/delete/delete[]
- Doubly freed memory.
- Reading/writing inappropriate areas on the stack.

To check for memory leaks during the execution of program try:

```
valgrind --tool=memcheck --leak-check=yes
         --show-reachable=yes
         --num-callers=20
         --track-fds=yes ./camelot
```

with no others arguments, valgrind presents a summary of calls to free and malloc.

Finding Memory Leak with Valgrind

Memory leaks are among the most difficult bugs to detect because they don't cause any outward problems until you have run out of memory and your call to malloc suddenly fails. In fact, when working with a language like C or C++ that does not have GARBAGE COLLECTION, almost half your time might be spent handling correctly freeing memory.

If you have a memory leak, then the number of allocs and the number of frees will differ. Notice that some errors might be suppressed -- this is because some errors will be from standard library routines rather than your code.

If the number of alloc differs from the number of frees, you'll want to rerun your program again with the leak-check option. This will show you all of the calls to malloc/new/new[] that don't have a matching free.

AVOIDING MEMORY LEAKS:

Some strategies can be taken into consideration in order to achieve clean code, free of memory leaks. The best and most common practice is to take care to delete all allocated memory with Destructors and use Valgrind to detect memory leaks and fix them. You should try to be careful to always delete[] memory

allocated for arrays and to check for proper deallocation on locally used pointers.

— A good approach to protect from memory leaks is to use a garbage (collection) collector and replace new with its allocator. There are a lot of libraries in the new standard wich can provide good garbage collectors, or you can implement you own.

— The c++ standard provides an efficient way of fighting memory leaks, which can applied by both experienced and intermediate level programmers: SMART POINTERS

One smart pointer provided by the STL is called auto_ptr (unique_ptr) and it is declared in the <memory> header

Ex:

```
void f()
{
    Dialog *dl = new Dialog;
    auto_ptr<> pdl = pdl(dl);
}
```

At the end of the function the auto_ptr object will go out of scope and will delete AUTOMATICALLY the memory allocated in the first statement. This above technique is used when there are some EXCEPTIONS thrown between allocating and deallocating memory and you want to make sure there aren't going to be any memory leaks.

For instance, it is common to write code such as:

```
void myFunction()
{
    myClass = new myClass();
    // Body of the function
    delete myClass;
}
```

And this may work. But what if somewhere in the function body an exception gets thrown? Suddenly, the delete code never gets called! What is more, you may never have intended to throw

an exception into the function but one of the functions you call may do so, or you may later need to modify you code to do so. In both case, this is a memory leak waiting to happen.
On the other hand, by letting the auto_ptr class manage your memory for you, as soon as an exception gets thrown and the auto_ptr you declared has gone out of scope, the memory allocated will automatically be freed.

Deciphering "N Bytes in M blocks are definitely lost" messages:

Memcheck's "definitely lost" messages are used to report all the unfreed memory blocks where it is clear that all pointers to this memory block have been lost.

Unfreed memory blocks in this category are usually caused by overwriting a pointer variable without freeing the memory pointed at or by holding the pointer in a local variable (i.e., on the stack) and then not freeing it before returning from the function.

Example:

```
38: buf = (char *) malloc (100);
    /* ... */
46: buf = (char *) malloc (200); /* lose pointer to original buffer */
```

we see that a 100 byte block allocated on line 38 was "definitely lost" because the pointer to it was overwritten on line 46.

Memcheck stores information about the allocation point for any unfreed blocks in a "LOSS RECORD".

Deciphering "N Bytes in M blocks are possibly Lost" messages:

Memcheck's "possibly lost" messages are used to report all the unfreed memory blocks where it is probably true that all pointers to this memory block have been lost BUT if the application was doing something clever with this unfreed block's pointer (like holding only a pointer to the ~~pointer~~ the middle of a block instead of to the beginning), it is

possible that a pointer to this unfreed memory could be reconstructed by the application (and thus it is not truly lost). if your application is doing something clever with this unfreed block's pointer, treat this message like a "blocks are still reachable" message. otherwise, treat it like a "Blocks are definitely lost" message.

Deciphering "N bytes in M Blocks are still Reachable messages:

These messages are used to report unfreed memory blocks that could (in theory) be freed at exit by the application because Memcheck has detected that the application still holds pointers to these block at exit.

majasa maja sandberg

Koronen

chjh red_17

magnlun red21

mnot red19 → mikaela nötebere

joelsu 10orange

Hamed neuzati

Marcus Dicander

fchahine

costa 21

costa 28

costa 2/6

DD2431 maskininlärning

cs.ecs.baylor.edu/~donahoo

informit.com/guides

sgi.com

Fredosaurus.com/notes_cpp

copy constructor ghostplus

diegm.uniud.it/schaerf/didattica.php

* dsi.unifi.it/~costa/lucidi_4_02/

GCC 4.5.4

— Avancerad Algorithm (avalg12) DD2440

— DD1352 Algorithm, datastruktur och komplexitet A (adk12) 9 credits

— SF1631 Diskret matematik 12 hp för D3

— DD2395 Datasäkerhet 6 hp

27 hp

www.ud.edu/chas/spring2003/cmsc311/notes

Charles Lin

— standalone, biase

— Design patterns : Erich Gamma, Helm, Johnson, Vlissides

— Esempio di scope

— writing test drivers / cmsc 214 spring 2002

stand out !?

+46 70 314 1624

cs.umd.edu/class

cs.umd.edu/class/fall2002/cmsc214/Tutorial/index.html.

" "/fall2012/cmsc359c/ slides.

johnnyb@ecert.se

~~text_a_is_second_to_last~~

vector → johnny

math.fsu.edu/finmath/computing resources/math.