Unit 4

Coding Standards and C++ Language Issues

Topics

In this unit we will consider a set of coding conventions that has been put forth by AT&T [see Appendix A]. The implementers of Brickles have tried to follow these conventions and standards. [Variance from the conventions will be noted.] Our purposes for examining the standards and conventions are two-fold:

- To help you understand the code in Brickles.
- To expose some of the fine points of C++ and some problems that [new] C++ programmers typically encounter.
- To introduce you to a reasonable set of standards.

Topics

- Coding conventions for C++[1]

  Why review them?

  - We have attempted to follow these standards and conventions in developing Brickles. Your familiarity with them will help you understand the code.
  - Many address areas of C++ that must be considered in implementing a design. In understanding the rationale behind some of the standards, you can become aware of some problem areas/pitfalls in using C++.

- Please try to follow them in the development you do during this course.

1. *OO Programming Rules and Guidelines: Version 3* by George Logothetis and Nancy Wilkinson (December 1, 1994)

Coding Conventions

Conventions are in two forms:

- *Rules* that are defined objectively and that can be enforced. On a typical project, a rule must always be obeyed.
- Guidelines that are defined subjectively and that might vary from project to project. On a typical project, there is a set of guidelines that are enforced as rules—for example, naming conventions for classes and member functions—and a set that fall into a “recommended” category.

Coding Conventions

- As defined by Logothetis and Wilkinson, the scope and use of each category of convention is:
  - *Rules* are conventions which have a clear, non-subjective definition of compliance. Because of this, the programmer can ensure compliance on his or her own , with the help of tools and visual inspection.
  - *Guidelines* are conventions which have a more subjective defnition of compliance. Because of this, peer review may be needed. Tools may be of help, but human judgement and experience is needed to ensure compliance.

Naming Conventions

Naming conventions are one of the most controversial issues when establishing conventions for any organization. Names are subjective. Some people like underscores in variable and function names; others don't. We have followed these conventions—except that we have begun all class data members (instance variables) with underscores. [We borrowed this convention from the gang of four.

Furthermore, we try to be consistent in the way we abbreviate words—for example, we would always use "nbr" as an abbreviation for "number" and avoid using "no" as an alternative abbreviation.

Naming Conventions

- These conventions apply to names.

1. (G) Readable and comprehensive names should be preferred over names which are easy to type.
2. (G) Names should be as specific to the context as possible, e.g. Coordinate rather than Number.
3. (G) Names with global scope should be distinctive and unique.
4. (R) Class names should begin with an uppercase letter.

   Example: String, Message, NetworkElement
5. (R) Acronyms that are part of names should be all uppercase.
6. (R) Variable names should begin with a lowercase letter.

   Example: myString, msg2, flowError
7. (R) Function names should begin with a lowercase letter (except constructors and destructors).

   Example: length(), execute()
8. (R) In any name which contains more than one word, the first word should follow the convention for the type of the name, and subsequent words should immediately follow, with the first letter of each word capitalized.

   Example: logEvent(), PrintManager

   We will follow these naming conventions, except that all protected and private data members should begin with an underscore (_).

File Organization

This convention is widely practiced in C development environments.

Following Microsoft Visual C++ conventions, our code files have an extension of “.cpp”. We use only one header file and code file per class, but some projects use one header file and a separate code file for each member function or class data member.

Because Windows only supports file names of at most eight characters plus an extension, we have been forced to shorten some class names in order to follow this convention—for example, class PlayingField can be found in files PLYNGFLD.H and PLYNGFLD.CPP.

File Organization

9. (R) Class definitions should be placed in class header files and class member implementations in one or more source files.

10. (G) Each header file should define one class or a small, tightly connected group of classes.

11. (G) The name of the header file should be the name of the most visible or important class being defined, suffixed by ".h".

    Example: String.h, Alarm.h

12. (G) If the class name is too long, a unique and meaningful abbreviation should be chosen.

    Example: NetElem.h

13. (G) Source file names should indicate the class and the most general or visible capability being defined. They should be suffixed by ".c" or ".C" (choose one and use it consistently).

    Example: StrAssign.c, AlarmInit.c

    We will follow these conventions. Note that file names in Windows must be no more than twelve characters. Header files will have a ".h" extension and source files will have a ".cpp" extension. We use exactly one header file and one source file per class.

Class Definitions

A common error in C++ syntax is forgetting to put "public" after the colon in declaring a derived class. The default inheritance is private, which means that all public and protected members of the base class become private in the derived class—usually not the effect desired.

Inline functions can be used to improve program performance. This rule is not prohibiting their use, it just requires that the member function be defined in the header file *after* the close of the class specification, not within it. This keeps the specification less cluttered, and promotes our goal of keeping specification and implementation separate. [While the convention does not address non-inline functions, those should not appear in the specification either.]

The rule about declaration order simply makes class specifications easier to read.

Class Definitions

14. (R) The protection of every base class should be specifically stated. When one class is derived from another, the default protection is private. For example:

```
class D : B { };  // D is privately derived from B
```

Q. What are protected and private inheritance useful for?

A.

15. (R) No inline function should be defined within a class definition.

16. (R) Class organization: the definition of the class should contain the following in this order:

    1. declaration of any friend functions and classes
    2. the "public:" specifier
    3. declarations of any public member functions
    4. declarations of any public data members
    5. the "protected:" specifier
    6. declarations of any protected member functions
    7. declarations of any protected data members
    8. the "private: "specifier
    9. declarations of any private member functions
    10. declarations of any private data

Q. What are public data members useful for?

A.

Class Header Files

There are two ways to get visibility to a class X in C++:

- include the specification in a header file—`#include "X.h"`
- provide a forward declaration: `class X;`

The latter form is preferable because it requires less preprocessing. It can be used if X is used in declarations that are pointers (X *) or references (X &). If you declare variables, arguments, or return types of type X, then you will have to include the specification.

Class specifications can be quite lengthy, so making declarations follow a standard order helps make specifications easier to read.

Class Header Files

17. (R) Every header file should be guarded against multiple inclusion with preprocessor directives.

    Example:

```
#ifndef CLASS_H
#define CLASS_H

// header file contents ...

#endif
```

18. (R) Header files should #include header files only for classes that are directly needed by the class declaration. Otherwise, forward declarations should be used.

    Example:

```
class Switch;
```

19. (R) Each class header file should contain the following:

    1. header file guard (as described above) at the beginning of the file
    2. a preamble C++ comment containing useful information such as file name, content, and copyright. This comment is for users of the header file.
    3. any #includes
    4. any forward declarations
    5. any extern and constant declarations
    6. any typedef definitions
    7. each class, struct, and union declaration followed by its inline function definitions

Class Source Files

Ordinarily, one would never expect to see an extern declaration in a program, except within a header file. The use of extern otherwise could result in an inconsistent view of some global function or object.

As in the case of header files, a standard order helps make a source file more readable.

Class Source Files

20. (R) Source files should #include header files instead of using extern declarations.

21. (R) Each class source file should contain the following:

    1. a preamble C++ comment containing useful information, such as file name, content, and copyright. This comment is for maintainers of the source file.
    2. #includes: this will include at least the class header file for the class being defined
    3. any global data initialization
    4. any static data initialization
    5. any static function declarations
    6. any friend functions and binary operator definitions
    7. member function definitions
    8. any static function definitions

Reference to a Class Member

C++ introduces a new scope: class scope. In a member function, a data member (instance variable) or a member function (method) can be referenced directly:

```
Class X {
public:
   X();
   int getCount() const;
private:
   int _count;
};

int X::getCount() { return _count; }
```

Thus, _count can be used directly in method getCount(). It is also permissible to use the expression “this->_count”, which is what this rule prefers.

In our code, we have followed this convention for member functions, but seldom for instance variables. We prefer a different approach: when referring to a variable outside of the object itself, we use the scope operator—for example, ::globalCounter.

Reference to a Class Member

22. (R) Any reference to a class member within the body of a member function of that class should be prefixed by this->.

*We will follow this rule, except that data members will be referred to directly. All data members are defined in either the protected or private areas of a class, and have names that start with an underscore—for example, "*_currentPuck*".*

Q. What is the value of c._foo in the following program?

```
int foo();

class C {
     public:
        C(int i = 0) : _x(i) { };
        virtual int foo() const { return _x; };[2]
        virtual int bar() const { return foo(); };
     private:
        int _x;
};

int foo() { return -1; };

void main() {
     C c(3);
     int i;

     i = c.bar();
};
```

A.

2. PLEASE NOTE!! In order to shorten the lengths of our examples, we include the code for member functions in the class specifications as we have done here. This is poor programming practice and violates earlier conventions!

Declarations

Compilers are inconsistent in their support for this language feature. Avoiding it will avoid problems with portability of code across compilers, and can avoid problems across subsequent releases of the same compiler.

Declarations

25. (R) No declaration should appear in the initializer section of a for statement.

This rule prevents developers from using a confusing feature of C++. Things were confusing enough when the scope of such a declaration was the block surrounding the for statement. But the ANSI C++ committee has recently voted to change the scope of variables in the initializer section. It is now defined to be the header and the controlled statement.

It is hard to predict when this change will be supported by compilers. During the (long) transition period, the safest thing to do is to follow this rule and write code which is legal under the old and new rules, e.g.

```
int j;
for (j = 0; j < 10; j++) // ...
```

Functions

Objects are created and destroyed automatically in many contexts within a C++ program. One of those context is in a function return value. If a function returns an object, then the value returned is copied into the scope of the caller (using the copy constructor for the class of the returned object). This is almost never the effect that you want. It can be circumvented by returning a reference to an object instead of an actual object.

However, if the reference is to a data member, then care must be taken to prevent that data member from being modified through the reference—hence the guideline about "const."

Unfortunately, the way the program is written, _value can be changed from outside the counter. For example, in the body of main, we can put:

```
int &i = counter.getValue();
i = 3;      // changes _value in counter!
```

The fix is to use "const" which would cause the compiler to reject the assignment to i:

```
class Counter {
   public:
      Counter() : _value(0) { };
      virtual const int &getValue() const { return _value; };
      virtual void step() { ++_value; };
   protected:
      int _value;
};
```

Functions

31. (G) A function parameter should be a reference to an object instead of an object whenever possible.

    Sometimes a function actually needs a copy of an object. If not, a reference to a const object should be used to avoid copying potentially large objects.

32. (G) Avoid returning non-const references or pointers to member data.

    Q. What can happen to a counter in the following program?

```
class Counter {
     public:
        Counter() : _value(0) { };
        virtual int &getValue() { return _value; };
        virtual void step() { ++_value; };
     protected:
        int _value;
};

void main() {
     Counter counter;

     counter.step();
};
```

    A.

    What if we declare: virtual const int &getValue()? ________________
    What if we declare: virtual int &getValue() const? ________________

Canonical Class

If not specified explicitly in a class specification, the compiler will generate a default copy constructor and an assignment operator. In some cases, the generated versions are not adequate. Ordinarily, this happens when an object contains pointers to other data, and that pointer relationship is supposed to be “one-to-one”. The default copy constructors and assignment operators will not preseve the relationship:

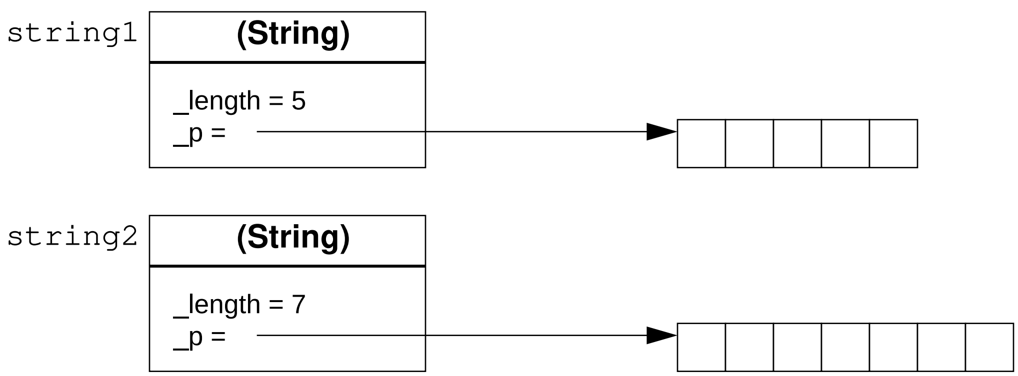
The string example shown is a classic illustration of this problem. If a default assignment operator is used for class String, then memory leakage occurs, and other undesirable effects as well—changing the characters associated with string2 will also update the value of string1!

Canonical Class

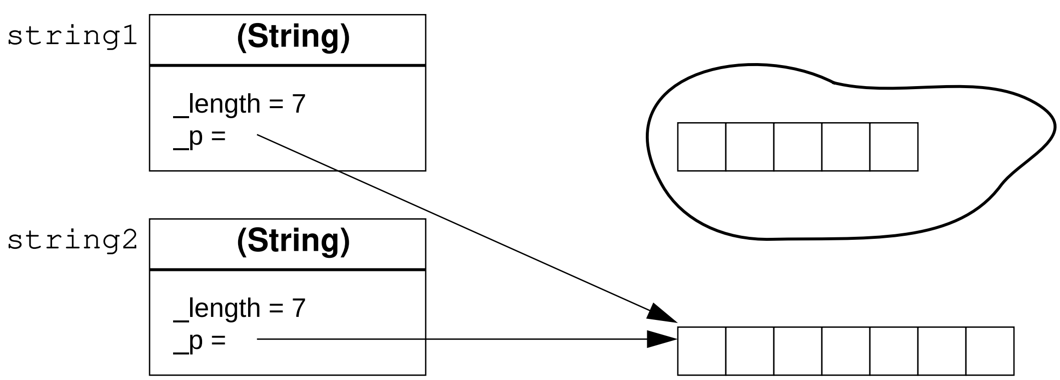
Each time a class is created, the compiler-generated copy constructor and assignment operator should be scrutinized to determine if they are adequate for the class. If found inadequate, versions of these functions should be provided by the programmer. The following addresses when and how these functions are created.

34. (R) If objects of the class allocate subsidiary data structures on the heap or consume any other shared resource, a copy constructor and assignment operator should be explicitly provided..

Here is a classic example:




```
string1 = string2;
```



Canonical Class

This is the code that corrects the problem just illustrated by overriding the copy constructor and the assignments operator

```
String::String(const String &s)

String & String::operator=(const String &string) {
```

Canonical Class

```
#include <string.h>
class String {
public:
   String(int length)
   : _length(length), _p(new char[length]) { };
   String(const String &);
   virtual ~String() { delete []_p; };
   virtual int getLength() const { return _length; };
   String &operator=(const String &);
private:
   int _length;
   char *_p;
};

class VString : public String {
public:
   VString(int maxLength)
   : String(maxLength), _curLength(maxLength) { };
   VString(const VString &);
   void setLength(int l) { _curLength = l; };
private:
   int _curLength;
};

String::String(const String &s)
   : _length(s._length), _p(new char[s._length]) {
   strcpy(_p, s._p);
};

String & String::operator=(const String &string) {
   if ( &string != this ) {   // Beware (s = s)
      delete []_p;
      _length = string._length;
      _p = new char[string._length];
      strcpy(_p, string._p);
   };
   return *this;
};

VString & VString::operator=(const VString &vstring) {
   if ( &vstring != this ) {   // Beware (s = s)
      this->String::operator=(vstring);
      _curLength = vstring._curLength;
   };
   return *this;
};

void main() {
   String string1(10);
   String string2(20);

   string1 = string2;
};
```

Assignment Operator

Good programming practice dictates that no variables in an object should be left uninitialized (at least in most cases).

Assignment of an object to itself can be disastrous. Consider, for example, the String class defined earlier. Assignment of string to itself would result in loss of the character array.

Since assignment is a concept that is well-established in C and in C++, its semantics should be preserved, including allowing of expressions of the form a = b = c.

Assignment Operator

35. (R) In the assignment operator all members, and base classes, should be assigned values.

36. (R) The assignment operator should have a check for assignment to self.

37. (R) The return type of the assignment operator for class T should be const T& and it should return *this.

    This allows expressions of the form a = b = c to be used and to behave as the built-in assignment operator behaves.

Constructors

Initialization is not assignment!

When an object is created, space is allocated for all its data members, including those that are instances of other classes. If an initialization list contains an initialization for a contained object, then a copy constructor is used to create that object. Otherwise, a default constructor is used, and then the assignment operator is applied.

Care must be taken in C++ to avoid such situations because it slows down execution of programs.

Constructors

38. (R) Constructors should use initialization lists to initialize class members rather than assign to them in the body of the constructor.

Initialization lists are generally the most efficient way to initialize object member data; in fact, they are the only way to initialize reference and const members. Initialization entails just one constructor call; but an assignment inside the body may result in a constructor call followed by a call to an assignment operator.

Example:

```
class Circle{
public:
     Circle(int radius, const Point &p);
private:
     int _radius;
     Point _centerPoint;
};

// Constructor version 1:
Circle::Circle(int radius, const Point &p)
     : _radius(radius), _centerPoint(p) {
}

// Constructor version 2:
Circle::Circle(int radius, const Point &p) {
     _radius = radius;
     _centerPoint = p;
};
```

The second version involves more computation, but produces the same effect.

Constructors

Data members are initialized in the order they are declared in the class specification, not in the order they are listed in an initialization list. By following this rule, you can avoid confusion and perhaps uncover an anomaly.

The example is illustrating an interesting characteristic of constructors: no dynamic binding occurs in the construction of an object.

The implementation for these two counter classes makes use of an instance variable _stepValue that holds the value to increment the counter by: 1 in the case of a Counter, 2 in the case of a ByTwosCounter. The step value is initialized during object construction by the setStepValue member function (invoked during the execution of Counter::Counter()). The method is overridden in the subclass to set the step value to 2 instead of 1, the value used in he method as defined in class Counter.

Answer: i is 1 and so is j. The initialization of the step value to 2 for ByTwosCounter instances fails since no dynamic binding occurs in the execution of the base class's constructor, so this->setStepValue() invokes the member Counter::setStepValue.

Constructors

39. (R) The members in an initialization list should be listed in the same order as they are declared inside the class.

    Members are initialized in declaration order whether you list them that way or not.

    What are the values of i and j at the end of the following program?

```
class Counter {
public:
     Counter() : _value(0) { this->setStepValue(); };
     virtual int getValue() const { return _value; };
     virtual void step() { _value += _stepValue; };
protected:
     virtual void setStepValue() { _stepValue = 1; };
     int _stepValue;
     int _value;
};

class ByTwosCounter : public Counter {
public:
protected:
     virtual void setStepValue() { _stepValue = 2; };
};

void main() {
     Counter counter;
     ByTwosCounter byTwosCounter;
     int i;
     int j;

     counter.step();
     i = counter.getValue();
     byTwosCounter.step();
     j = byTwosCounter.getValue();
};
```

Const

The “const” feature of C++ is one of the most powerful vehicles available for improving the reliability (and in some sense the efficiency) of code. [At the same time, it can be one of the most frustrating to use!]

The use of “const” in a declaration denotes that the declared object will not change its value once it has been initialized upon entry to the scope in which the object is declared. Thus, it has more semantics than just permitting the declaration of constant values such as strings, numbers, and so on.

A member functions can be declared to return a const value, but it can also be declared to be const—that is, it is a member function that will not [logically] change the state of the object for which it is invoked. This “const member function” is denoted by putting the keyword `const` at the end of the declaration. If a member function *m* for some class `X` is declared `const`, then `this` is implicitly declared as `const X * const this` within the body of *m*. [This means that `this` is a constant pointer to a constant object of type `X`.]

The example given shows how a “logical” const can be casted away.

Const

40. (R) Functions that do not alter the values of any data member should be declared const.

    This ensures that the function can be used on the widest range of objects. Not declaring a function const needlessly restricts the set of objects upon which it can be called.

41. (G) Functions that do not alter the logical state of an object should be declared const, even if that means casting away const inside the function.

    Here is an example of "logical" vs. "physical" state from Increment 1:

```
class ArcadeGamePiece;
class Timer {
public:
     Timer() : _enrollmentCount(0) { };
     void enroll(ArcadeGamePiece *) const;
     void withdraw(ArcadeGamePiece *) const;
private:
     ArcadeGamePiece *_enrollees[10];
     int _enrollmentCount;
};

void Timer::enroll(ArcadeGamePiece *piecePtr) const {
     ++(((Timer *)(this))->_enrollmentCount);
     ((Timer *)(this))->_enrollees[_enrollmentCount]
        = piecePtr;
};

void Timer::withdraw(ArcadeGamePiece *piecePtr) const  {
     // ...
};
```

42. (G) Reference or pointer parameters that are not modified by the function should be declared to be references or pointers to const.

    Not declaring such parameters as pointers or reference to const needlessly restricts the set of objects which can be passed to the function.

Encapsulation and Information Hiding

An important goal in object-oriented design and programming is the separation of specification—*what* a class does—from implementation—*how* a class does it. This means, among other things, that all data members should be kept hidden—that is, restricted to non-public areas of classes. Guideline 43 goes so far as to say that even *protected* data members should be avoided. The rationale is based on the fact that inheritance violates encapsulation. A data member in the protected area of a class is available for use in derived class member functions, so if for some reason a change is made in the class , then the corresponding code for a derived class might be required. The use of private data circumvents this violation of encapsulation by inheritance by forcing derived class's methods to use member functions provided in the public and protected areas of any base class. [This is a guideline since efficiency concerns might preclude the use of this convention.]

Rule 45 is more restrictive. If putting data in a public area is not a good idea, then neither is it a good idea to provide pointers or references to non-public data. [Recall our example of the counter in Guideline 32.]

Encapsulation and Information Hiding

43. (G) Public and protected data members should be avoided when possible.

44. (R) Inline accessor and mutator functions should be provided to access and modify state components.

    State information should be only accessed and modified via functions. These functions should be defined as inline for efficiency reasons.

45. (R) An accessor function should not let you modify the object; thus it should return either an object, a reference to a const object or a pointer to a const object.

    Returning a non-const reference or pointer to a data member would subvert the encapsulation and information hiding of the class.

Inlines

The ability to inline member functions in C++ can promote the efficiency of program execution, but if care is not used, then code size can increase dramatically. The guidelines are intended to avoid bloat.

Note that the use of inlin can defeat the ability to easily patch software, if that is necessary in the application environment.

Inlines

46. (G) Inlining should be reserved for accessor, mutator, and very short functions.

    A very short function is generally 1-2 lines. Counting lines however can be complicated if one of the lines is a call to another inline function. What gets generated by the compiler for an inline function may be more than meets the eye. Other than simple forwarding functions, e.g.

    ```
    int length(const String &s) { return s.length(); }
         // length()s are inline
    ```

    Users should call inlines within inlines only if they understand what the compiler is likely to generate.

47. (R) Constructors, destructors, and virtual functions should not be inline.

    The address of these functions may need to be preserved (even implicitly) which will result in multiple copies being created.

48. (R) Inline functions should do no I/O.

    Otherwise, iostream.h must be included in the header! This causes code bloat.

NULL

Old habits die hard., but “0” is the proper denotation of a pointer to “nowhere”. A project might wish to make use of a symbolic constant, but that is not required by C++. [Note that strong typing is the mechanism by which a compiler can determine whether “0” means a number or a pointer. That was not possible in C, hence the need for a symbolic constant.]

NULL

49. (R) 0 should be used instead of NULL.

The symbol NULL is an artifact of old C when the compiler had no way to distinguish 0, the number, from 0, the pointer. In C++, this is no longer the case (except when passed to a function with ellipses arguments). NULL is not guaranteed to be compatible with all pointer types in C++. In particular, in some implementations, it is defined to be (void*)0, which cannot be assigned to all pointer types.

Note that this is counter to the C convention to use NULL instead of 0.

This rule does not mean that the literal 0 must be used; only that some constant whose value is an unadorned 0 must be used and that NULL should not be. A project-wide symbolic constant could be defined.

Memory Management

Memory management is a difficult task in some C++ applications. It is important in designing a class implementation that the constructors and destructors allocate (via `new`) and free (via `delete`) memory needed for the correct operation of the class. Sometimes, as we have seen, this is also an issue in copy constructors and assignment operators.

Note that one frequent source of “memory leakage” is failure to declare the destructor for a class as “virtual”.

The example illustrates how to define new and delete for a class. This is sometimes useful in real-time systems to address performance issues associated with memory management.

Memory Management

50. (R) new and delete should be used instead of malloc() and free().

    Operators new and delete are more convenient to use than malloc() andfree() and understand C++ constructs.

51. (R) delete [] should be used whenever an array is deleted.

    The empty brackets are not strictly necessary when deleting arrays of objects of built-in type. They should always be used for consistency and to increase readability.

52. (R) delete should be defined whenever new has been defined.

    Memory allocation and deletion should share underlying assumptions. malloc() and free() use different assumptions than new and delete.

    Here's an example of how to do it. Why define new and delete for a class?

```
#include <stddef.h>

class Counter {
     public:
        Counter() : _value(0), _nextFreeCounter(0) { };
        virtual int &getValue() { return _value; };
        virtual void step() { ++_value; };
        static void *operator new(size_t);
        static void operator delete(void *);
     protected:
        int _value;
     private:
        static Counter *freeList;
        Counter *_nextFreeCounter;
};

Counter * Counter::freeList = 0;
void *Counter::operator new(size_t s) {…};
void Counter::operator delete(void *p) {…};
```

Preprocessor

Many of the things we relied on the preprocessor for in C are now better accomplished using features of C++ as the examples given here illustrate.

While they are not readily available for our use in the version of Visual C++ we are using, templates are very useful in defining “macros”.

Note that the inline version of max eliminates the old problem of the macro of having the larger of the arguments evaluated twice.

Preprocessor

53. (R) const and enum should replace use of #define for manifest constants.

54. (G) Inlines and templates should replace uses of #define for macros whenever possible.

```
#define GREETING "Hello!"

const char S[] = "Hello!";


#define MAX 1000

const int MAX = 1000;


#define abs(x) ( ((i) < 0) ? -(i) : (i) )

inline float abs(float x) { return ( (x < 0) ? -x : x ); };
inline int abs(int x) { return ( (x < 0) ? -x : x ); };
// A job for a template!


#define max(a,b) ( ((a) < (b)) ? (b) : (a) )

#inline int max(int a, int b) { return ( (a < b) ? b : a ) }
```

Inheritance and Virtual Functions

When deleting an object, you usually expect the destructor for the class of that object to be executed. However, that is not always the case as the example illustrates. The problem arises in situations involving polymorphic references (or pointers). When an object denoted by a reference (or a pointer) is deleted, the destructor for the class associated with the reference (pointer) in its declaration is used—unless the destructor for that class was defined as virtual.

```
class B {
public:
    B() : _b(1) { };
    virtual ~B() { cout << "_b was " << _b; };
private:
    int _b;
};
```

Inheritance and Virtual Functions

55. (R) Every base class should have a virtual destructor.

*Q. What is printed when *bPtr is destroyed?*

```
#include <iostream.h>

class B {
public:
     B() : _b(1) { };
     ~B() { cout << "_b was " << _b; };
private:
     int _b;
};

class D : public B {
public:
     D() : _d(2) { };
     ~D() { cout << "_d was" << _d; };
private:
       int _d;
};

void main() {
     D d;
     B *bPtr = new D();

     delete bPtr;
};
```

How can it be fixed?

56. (R) A derived class should not redefine a default parameter for a virtual function.

Because default parameters are statically bound while virtual functions are dynamically bound, you could end up calling the derived class function with the base class default parameter. So the 2 default parameters better be the same. See Meyers pp. 132-134 for a complete discussion of this.

Inheritance and Virtual Functions

Overriding a member function that is not declared virtual in a base class will almost always result in unexpected behavior in a program, as the example illustrates.

Incidently, a common source of errors is *forgetting* to declare the member function as virtual in the base class.

Inheritance and Virtual Functions

57. (R) A derived class should not redefine an inherited non-virtual function.

Redefining a non-virtual function in a derived class will cause different behavior depending on how the function is called. For example, for a redefined function foo():

```
Derived der;
Derived* d = &der;
Base* b = &der;
b->foo();// calls Base::foo()
d->foo();// calls Der::foo()
```

Also, redefining an inherited non-virtual function violates the is-a relationship implied by inheritance.

Global Namespace

It is better to make use of class scope instead of global scope when declaring data and functions shared by large numbers of program components. As the guidelines point out, avoidance of global names can make components easier to reuse and integrate with other components. Consider, for example, a class C that uses a global integer value MAX and another class D that uses a global integer value MAX. Using C and D in the same program will be dificult. Scoping MAX—C::MAX and D::MAX—eliminates the conflict in global name space.

Global Namespace

Keeping the global namespace uncluttered reduces name collisions and makes it easier to reuse other people's code. It also makes it easier to figure out where a name is coming from. In general, global variables should be used only when absolutely necessary.

58. (G) Nested classes and enums should be used when possible.

    Types used only by a particular class can be declared within the scope of that class. The name is then not known outside of that one class.

59. (G) Static members should be used when possible.

    Static members often provide a good alternative to global variables and functions.

    Q. Which is better?

```
const int BRICKTHICKNESS = 10;


class Brick {
public:
     static const int THICKNESS;
private:
     // …
};

const int Brick::THICKNESS = 10;
```

Use of Friends

While the use of friends can improve program execution efficiency, it does so at the expense of violating data hiding. Workarounds are sometimes possible, including nesting class definitions or providing accessor functions that are inlined to increase execution speed.

One idiom in C++ is the provision of a stream insertion operator (<<) for any class that has an external representation. The friend feature is needed to permit the definition of the function that defines such an operator.

Use of Friends

62. (G) Friend functions should be used only for non-member binary operators and in other exceptional cases where efficiency demands it.

    Since friends violate encapsulation and information hiding, they should be generally avoided. Sometimes efficiency concerns demand that a function have direct access to the internals of a class; but this should happen very rarely. Generally, the class accessor functions can be used just as efficiently.

```
#include <iostream.h>

class X {
     friend ostream& operator<<(ostream &, X &);
public:
     X() : _i(1) { };
private:
     int _i;
};

ostream& operator<<(ostream &os, X &x) {
     return os << "X(" << x._i << ')';
};
```

63. (G) Classes should only declare other classes as friends when the two are very tightly coupled.

    A common example of tightly coupled classes is a container class and its iterator, e.g. List and ListIterator. Another example is a class which is part of the implementation of another class, e.g. StringRep and String. This case, however, can often be handled more effectively with a nested class.

Overloading

If you use overloading of member functions in a class, make sure that the meanings of those member functions correspond to some common semantics.

If you use overloaded operators in your design, make sure that their use is consistent with reasonable expectations. For example, if you define an operator == to compare two objects, make sure it returns a boolean result, and make sure that the matching operator != returns the boolean complement.

Conversions occur in C++ quite transparently, and care must be taken to avoid them.

Overloading

64. (G) Except for constructors, overloading should be reserved for functions that are really doing the same thing and that have unrelated argument types.

65. (G) Overloaded operators should be used only for classes where they make sense and in a way that mimics the built-in operators.

66. (R) If an operator is defined which has a complementary operator, its complement should also be defined and in terms of the original operator.

    Example:

```
int operator==(const String&, const String&);

int operator!=(const String& s1, const String& s2)
{
     return !(s1 == s2);
}
```

67. (G) Conversion functions should be used sparingly.

68. (R) All unary operators should be defined as members.

69. (R) Compound assignments (+=, -=, *=, etc.) should be defined as members.

70. (R) All other binary operators should be defined as friends.

Type Casting

Type casting has always been suspect in C programs—at least to software engineers!—and they remain so in C++. The typing system for C++ is quite strong. That degree of checking helps produce higher-quality programs. Any need to circumvent the typing system can reduce quality. Be especially wary of casts "up" an inheritance hierarchy, which are almost always innecessary, and casts "down" a hierarchy, which can be dangerous if the target object is not of the class it is being casted to.

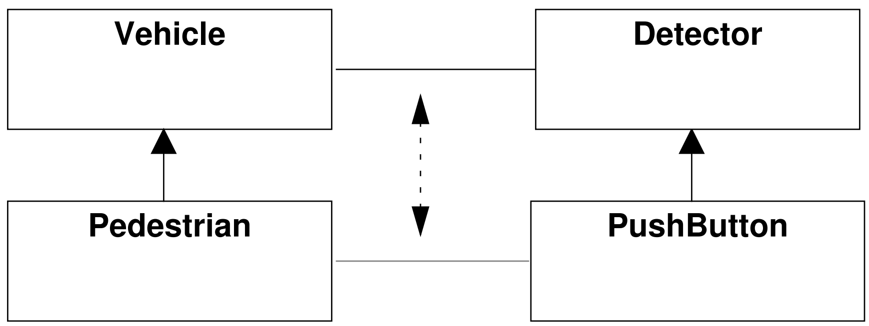
There are some situations in which type casting can be justified. We saw an example of that with respect to operations that are "logically" const. Another is in a situation that gives rise to *contravariance*. The example shows such a situation at a high level. In an object model, we establish an association between two abstract classes. That relationship, however, gets "refined" as we move down parallel hierarchies. A vehicle is associated with (detected by) a detector, but when we talk about pedestrians, we restrict the kinds of detectors we use to push buttons. We can establish an operation, such as `Detector::detect(Vehicle &)` in the protocol for Detector, but we want to refine the parameter type in the version used for push buttons—`PushButton::detect(Pedestrian &).` The problem is that in so doing, we are *overloading* detect in Pedestrian, not *overriding* it. Consequently, we are forced to preserve the parameter type established in Detector, or remove `detect` from Detector's protocol. If we choose to do the former, then the only way that the code for the method `PushButton::detect(Vehicle &v)` can call member functions specific to pedestrians is to cast `v` to a reference to a Pedestrian.

Type Casting

72. (G) Type casts should not be used unless absolutely necessary.

    Each occurrence of a cast should alert the reader to some special need to override the type system of C++. They should not be inserted routinely or for insignificant reasons.

    An example of contravariance.



An example of the use of casts can be found just about anywhere that CObjectList is used in the implementation [because "CObject" is too general an abstraction to be useful in most contexts.]

References and Pointers

References are a nice feature available in C++. In many cases, a reference parameter should be used instead of a pointer (old habits are hard to break!). The use of reference parameters can make code more easy to read.

However, there are times when a pointer is the appropriate thing to pass as an parameter. How can you tell? It is usually a judgement call. One rule of thumb: if you have a function that needs to take the address of a reference parameter, then perhaps the parameter itself is better passed as a pointer instead of a reference. This convention alerts the caller that a pointer to that argument might be stored by the “callee.”

Many people (mistakenly) believe that a reference is just a smart pointer. However, that is not quite the case: A reference can be set only once, whereas a pointer (unless it is const) can be assigned to point to various targets. Think of a reference as an alias and a pointer as an address. Store pointers to implement networks of objects. Use references when you are referring to objects along the thread of computation.

References and Pointers

Using references and pointers correctly can increase the readability of C++ code by forcing a programmer to state explicitly how a variable is used.

73. (G) Pointers should be used when you want multiple references to the same object or a dynamic data structure.

    In C, pointers were used to simulate "call by reference" as well as for multiple references to the same object. In C++, pointers should not be used to pass an object by reference.

74. (G) References should be used whenever a parameter is being passed "by reference".

    This also leads to more readable code inside the function because parameters are not dereferenced throughout.

    When should a pointer be passed?

```
class X;

class Y {
public:
     Y() : _xPtr(0) { };
     virtual void foo(X &x) { _xPtr = &x; };
     virtual void foo(X *xPtr) { _xPtr = xPtr; };
private:
     X *_xPtr;
};
```

Templates

The availability of templates in C++ is a boost to software reuse. It has given rise to a new area of software enginering: generic programming.

Unfortunately, templates are not readily available in the version of Visual C++ that we are using. [A tool exists in the environment to give the effect of templates.]

Templates are important in object-oriented programming languages, such as C++, that employ strong typing. Templates complement inheritance with respect to code sharing. Inheritance supports code sharing when the types of arguments to member functions are the same from one class to another. Templates allow the definition of a family of classes that use, in essence, the same implementation, only the types or the number of things changes. The template class Queue defines a family of queues, each having a particular type of element that can be inserted and removed from the queue. [Note that the same effect can't be achieved through inheritance since any type associated with, say, the argument to `put()` would have to be preserved in any subclass's member function `put()`.]

Templates

75. (G) The necessary member functions should be defined for T in order for T to be used as a template parameter.

Member functions of a parameterized (template) class may rely on functions of its parameter, T. For example, a sort() operation of a List<T> may assume that T has defined an operator<().

Templates can be used to complement inheritance:

```
template<class T>
class Queue {
public:
     Queue() { };
     ~Queue() { };
     virtual void put(const T &) { };
     virtual T &get() { };
     virtual BOOL isEmpty() const { };
private:

};
```

Or to customize interfaces and/or implementations:

```
class ArcadeGamePiece;

template<const int MAX, const int UnitInterval>
class Timer {
public:
     Timer()
     : _unitInterval(UnitInterval), _enrollmentCount(0) { };
     ~Timer() { };
     void enroll(const ArcadeGamePiece *) const;
     void withdraw(const ArcadeGamePiece *) const;
private:
     ArcadeGamePiece *enrollees[MAX];
     const int _unitInterval;
     int _enrollmentCount;
};
```

Static Object Construction

Since the declaration of a variable whose type is a class name causes a constructor to be executed for the variable, much system initialization can occur before the main program begins execution. Care must be taken, however, if the construction of a static object relies on the existence of another static object defined in another file, because the order in which such objects in different files are actually created is not defined by the language. Therefore, avoid this situation either by avoiding static objects or by declaring them all in a single file, in which the order of construction is defined by the language.

Static Object Construction

76. (G) Static objects from other files should not be used in functions called at static constructor time.

    A well-known problem with C++ is that the order of execution of static object constructors is defined only within a single file. Therefore, references to a static object from another file with a static object constructor may fail mysteriously at the time of execution. This situation should be avoided altogether.

Initialization

Initialization is not assignment!

We have already addressed this issue wit respect to the use of initialization lists in constructors.

This guideline calls for variables—specifically, those that are instances of classes—be declared close to where they are used. This is counter-intuitive in some sense because most programming languages require that all variables used in a block be declared at the start of that block. [Interestingly, that was a reaction to languages such as Fortran that required no declarations of variables, and in which typographical errors produced bugs at run time.] Some coding conventions require all variables be declared at the start of a block, accompanied by comments as to their role in the block.

C++ permits the declaration of variables at any point in a block. There is areason for it, as the example illustrates.

Initialization

77. (G) Initialization should be preferred to assignment.

An initialization is generally more efficient than a declaration followed by an assignment. In the case of a class object, the latter will result in two function calls (constructor plus assignment operator); but initialization entails one call to a constructor.

This implies that variables should be declared very close to where they are used. Note that this is counter to the C convention to put all declarations at the top of a block.

```
{
     CRect r1(10, 10, 20,20);
     CRect r2(15, 15, 25,25);

     CRect r3;
     …
     r3 = r1.union(r3);

        versus

     CRect r3 = r1.union(r2);

        versus

     CRect r3(r1.union(r2));
```

Initialization

This is a classic example from Stroustrup that illustrates the distinction between initialization and assignment. The class Complex has three constructors, as well as a member function, operator+, that returns newly constructed instances of Complex.

Initialization

This is a classic example derived from Stroustrup:

```
class Complex { // a + ib
public:
     Complex() { };
     Complex(double a) : _a(a), _b(0.0) { };
     Complex(double a, double b) : _a(a), _b(b) { };
     Complex operator+(const Complex &c) const {
       return Complex(_a + c.a(), _b + c.b());
     };
     double a() const { return _a; };
     double b() const { return _b; };
     // ...
private:
     double _a;
     double _b;
};
Complex a(1.0);     // initialize by call to Complex(double)
Complex b = a;      // initialize by a copy of 'a'
Complex c =         // construct Complex(1, 2) using
     Complex(1, 2); // Complex(double, double)
                    // copy it into 'c'
Complex d = (a + b);// call Complex::operator+ and copy the
                    // result to 'd'
Complex e;          // initialize by a call of Complex()
Complex f = 3.0;    // construct Complex(3) using
                    // Complex(double) and copy it into 'f'
class C {
public:
     C(double x) : _c(x) { };
     C(double x, double y) { _c = Complex(x, y); };
private:
     Complex _c;
};

void main() {
     Complex d;

     d = (a + c);
};
```

Exercise

The rationale behind these coding conventions we have reviewed is frequently based on the definition of C++ itself. It is a very large and complex language, and mastering it is a bigger challenge than mastering most programming languages. [Throw on top of that the paradigm shift and the challenge becomes much greater!]

Exercise

- Are there any other questions you have about C++?
- Perhaps you’ve heard some stories.
- Have you been cautioned against using some some language feature? If so, do you know why?

Summary

Take a few minutes to review the material in this unit. Note the three most important concepts from your perspective.

-
-
-

Summary

- The adoption of coding conventions—rules and guidelines—can improve the quality of a software application
- The conventions we have reviewed have been followed, with a few noted exceptions, in our implementation of Brickles. If you notice a violation in the code you review, please bring it to the instructor's attention.
- C++ is a complex language and care must be taken to avoid unnecessary [default] construction and copying of objects.
- C++ provides many features; don't feel obliged to use them all! [There's a lot of rope there!]