18. Class: Design and Declaration

- Declare a new class in a program creates a new type – class design is type design.
- Aim for a class interface that is complete and minimal – two completely different directions.

- **Complete**: powerful and convenient to use.
- **Minimal**: fairly small number of member functions, each of which performs a distinct task.

19. Functions: Design and Declaration

- Differentiate among ***member*** functions, ***global*** functions, and ***friend*** functions (access to non-public members).
- ***class Rational {***

    Rational operator*(const Rational& rhs);

  };

  Rational a, r;

  r = r * a   // fine

  r = a * 2;   // fine because

    Rational temp(2);   // turn 2 into a Rational

    r = a * temp;

Functions: Design and Declaration (2)

- ***r = 2 * a;*** // error! because

  r = 2.operator*(a); // no member function

  r = operator*(2, a); // no global function
- The solution is to make it global (allowing compiler to perform *implicit type conversions* on all arguments):

  Rational operator*(const Rational& lhs, const Rational& rhs);
- Should it be made a friend function in this case? ..... No. Whenever you can avoid friend functions, you should.

Functions: Design and Declaration (3)

- **operator>>** and **operator<<** should not be member function (must be made friends). If they were, you have to put the **String** object on the left when you called the functions.

```
class String {
    istream& operator>>(istream& input);
    ostream& operator<<(ostream& output);
}
String s;
s >> cin;        // legal, but contrary to convention
s << cout;       // No…
```

Functions: Design and Declaration (4)

// Virtual functions must be members.

- ***if (f needs to be virtual)***
  make f a member function of C;

// *operator*>> and *operator*<< are never members.

- ***else if (f is operator>> or operator<<) {***
  make f a global function (or within a namespace);
  if (f needs access to non-public members of C)
  make f a friend of C;
  }

Functions: Design and Declaration (5)

// Only non-members can have type conversion on
// their left-most argument, like
//   result = 2 * onehalf;

- ***else if (f needs type conversion on its***
  left-most argument)
  make f a global (or a friend) function;

// Everything else should be a member function.

- ***else***
  make f a member function of C;

21. Use "const" Whenever Possible

```
char *p                    = “Hello”;     // non-const pointer,
                                          // non-const data

const char *p              = “Hello”;     // non-const pointer,
                                          // const data

char * const p             = “Hello”;     // const pointer,
                                          // non-const data

const char * const p = “Hello”;           // const pointer,
                                          // const data
```

Use "const" Whenever Possible (2)

- A function returning a const value often reduces the incidence of client errors.

```
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
Rational a, b, c;
(a * b) = c; // assign to the product of a * b
             // flat-out illegal!
```

Use "const" Whenever Possible (3)

- Member functions differing only in their "const"-ness can be overloaded.

- operator[] for non-const objects: (e.g. LHS)

  char& operator[](int position);

- operator[] for const objects: (e.g. RHS)

  const char& operator[](int position) const;

Use "const" Whenever Possible (4)

String str = "World"; // non-const String object

const String

constStr = "Hello“; // const String object

char c2 = str[0]; // fine – reading a non-const String.

char c1 = constStr[0]; // fine – reading a const String.

str[0] = 'x'; // fine – writing a non-const String.

constStr[0] = 'x';

// error! – writing a const String.

Use "const" Whenever Possible (5)

- The constructor makes data point to a copy of what value points to.

```
String(const char *value=0);
operator char *() const { return dataPtr; };

const String s = "Hello";        // declare constant object
char *nasty = s;                 // call op char *() const
*nasty = 'M';                    // modifies s.data[0]
cout << s;                       // writes "Mello"
```

Use "const" Whenever Possible (6)

- Surely there is something wrong when you create a **constant** object with a particular value and you invoke only "**const**" member functions on it, yet you are still able to change its value!

22. Pass & Return Objects

- Pass and return objects by reference instead of by value.
- ***class Person {***

  String name, address; };

  class Student : public Person {

  String schoolName, schoolAddress; };

Student returnStudent (Student p) { return p; }

Pass & Return Objects (2)

- ***Student s;***
  returnStudent (s);
- Call copy constructor to initialize *p* with *s*.
- Call copy constructor to initialize the object returned by the function with *p*.
- Destructor is called for *p*.
- Destructor is called for the object returned by ***returnStudent***.
- There are many other constructor and destructor calls.....

Pass & Return Objects (3)

- Each ***Student*** construction entails two more ***String*** constructions.
- Each ***Student*** construction entails one more ***Person*** construction.
- Each ***Person*** construction entails two more ***String*** constructions.
- Each constructor call is matched by a destructor call…..

Pass & Return Objects (4)

- Therefore,

```
Student& returnStudent (Student& p)
        { return p; }
```

- More **efficient**: no constructors or destructors are called, because no new objects are being created.

Pass & Return Objects (5)

- Passing parameters by reference has another advantage: it avoids what is called the "slicing problem". When a derived class object is turned into a base class object, all of the specialized features that made it behave like a derived class object are "sliced" off, and you are left with a simple base class object.

23. Returning an Object

- Do not try to return a reference when you must return an object.

- ***Complex a(3, 2);***
  Complex b(-5, 22);
  Complex c = a + b;

- Which one of the following is better? // (1)

Returning an Object (2)

1. ***const Complex operator+(const Complex& lhs,***
   const Complex& rhs)
   { return Complex(lhs.r + rhs.r, lhs.i + rhs.i); }

   Call constructor? (and later destructor)

2. ***const Complex& operator+(const Complex& lhs,***
   const Complex& rhs)
   { Complex result(lhs.r + rhs.r, lhs.i + rhs.i);
   return result; }

   Call constructor? Return a reference to a local object?

Returning an Object (3)

3. ```
const Complex& operator+(const Complex& lhs,
                         const Complex& rhs)
{ Complex *result = new Complex(lhs.r + rhs.r,
                                lhs.i + rhs.i);
      return *result; }
```

Call constructor? A guaranteed memory leak. For example,

```
Complex w, x, y, z;
w = x + y + z;
```
```

Returning an Object (4)

```
4.   const Complex& operator+(const Complex& lhs,
                              const Complex& rhs)
     { static Complex result; // static object to which a
                              // reference will be returned
          .....
          return result; }

Complex a, b, c, d;
if ( (a + b) == (c + d) )     // will always evaluate to true
          // regardless of the values of a, b, c, and d
```

24. Function Overloading vs Parameter Defaulting

1. *void f();* // f is overload

   void f(int x)

   f(); // calls f()

   f(10); // calls f(int)

2. *void f(int x = 0);* // f has a default parameter value

   f(); // calls f(0)

   f(10); // calls f(10)

Which should be used **when**?

Function Overloading vs Parameter Defaulting (cont)

- If you can choose a reasonable default value and you want to employ only a single algorithm – use the default parameters.

- Otherwise – use function overloading.

25. Overloading

- Avoid overloading on a pointer and a numerical type.

void f(int x);

void f(char *p);

f(0); // calls *f(int)* or *f(char*)*?

26. Guard against Potential Ambiguity

- ***class B;*** // forward declaration for class B

- ***class A {***

    A (const class B&);

    // an *A* can be constructed from a *B*. ***};***

- ***class B {***

    operator A() const;

    // a *B* can be converted into an *A*. ***};***

Guard against Potential Ambiguity (2)

- ***void g(const A&);***
  B b;
  g(b);

  1. calls *A*'s constructor using *b* as an argument.
  2. converts *b* into an *A*.

Guard against Potential Ambiguity (3)

- ***void f(int);***
  void f(char);

  double d = 1.2;
  f(d); // error – ambiguous

Guard against Potential Ambiguity (4)

- ***class Base1 { public: int f(); };***
  class Base2 { public: int f(); };
  class Derived: public Base1,
  public Base2 { … };
  // ***Derived*** does not declare a function called ***f***

  Derived d;
  d.f(); // error – ambiguous

30. Reference to a Less-Accessible Member

- Never write functions to access (references of) restricted members.

- ***class Address {…};***
  class Person {
  public: Address& personAddr() { return addr; }
  private: Address addr; };

  Person s;
  Address addr = s.personAddr();
  // s.addr is no longer private.

31. Returning a Reference

- Never return a reference to a local object: local objects are destructed when they go out of scope.

- How about calling ***new*** instead of using a local object?
- Writing a function that returns a dereferenced pointer is a memory leak just waiting to happen.

32. Variable Definitions

- Postpone variables definitions as long as possible – there is a cost (constructor or destructor) associated with unused variables, avoid them whenever you can.

“enum”

- Use enums for integral class constants.
- ***class X {***

  char buffer[256];

  };

- ***const BUFSIZE = 256;***

  ....

  class X {

  char buffer[BUFSIZE];

  };

“enum” (2)

- ***class X {***
  ```
  static const BUFSIZE = 256;        // error!
  char buffer[BUFSIZE];
  };
  ```

- ***class X {***
  ```
  static const BUFSIZE;
  char buffer[BUFSIZE];              // error!
  };
  const X::BUFSIZE = 256;
  ```

“enum” (3)

- ***class X {***

```
class X {
    enum { BUFSIZE = 256; }        // fine
    char buffer[BUFSIZE];
};
```

33. Use "inlining" Judiciously

- "inline" functions ***look like*** functions, ***act like*** functions, ***better than*** macros, ***no overhead like*** calling a function.....

- There is no free lunch.

- Increase overall size of object code, reduce instruction cache hit rate.

Use “inlining” Judiciously (2)

- “inline” directive is a "hint" to the compiler not a "command“ (just like register).
- // this is file "example.h“

  inline void *f*();

  // this is file "source1.cc“

  #include "example.h“ // include definition of *f*

  // this is file "source2.cc"

  #include "example.h“ // also include definition *f*

Use “inlining” Judiciously (3)

- Assume that *f* is not being inline: linker complains multiple definitions.

- To prevent the problem, compilers treat an un-inlined inline function as "**static**", unit that includes the definition of *f* will have its own static copy of *f*.