CISC 3142
Programming Paradigms in C++

Ch3 – A Tour of C++:
The Abstraction Mechanisms

(*Stroustrup – The C++ Programming Language, 4th Ed*)

Classes

- The central language feature of C++
- A user-defined type to represent any useful concept, idea, entity, etc.
- Libraries are commonly offered in classes
- Three kinds of classes:
  - Concrete classes
  - Abstract classes
  - Classes in class hierarchies

Concrete Types

- Behave “just like built-in types”
- Its representation is part of its definition
  - Could have pointers to more data stored elsewhere (heap)
  - Could be private, accessible only through member functions
- The keys
  - It allows objects being placed on stack, or in other objects, and
  - Being referred to directly (not through pointers/references)
  - Being initialized completely (via constructors)
  - Being copied
- Or, the way the rest of us are used to understand:
  - Can be readily instantiated, i.e. no abstract functions

An Example – complex (simplified version)

```
class complex {
    double re, im;                                  // representation: two doubles
public:
    complex(double r, double i) :re{r}, im{i} {}   // construct complex from two scalars
    complex(double r) :re{r}, im{0} {}             // construct complex from one scalar
    complex() :re{0}, im{0} {}                     // default complex: {0,0}, so "complex c;" won't trigger a compiler error
    double real() const { return re; }             // accessors/mutators, notice the const specifiers in accessors
    void real(double d) { re=d; }
    double imag() const { return im; }
    void imag(double d) { im=d; }
    complex& operator+=(complex z) { re+=z.re , im+=z.im; return *this; } // add to re and im and return the result
    complex& operator-=(complex z) { re-=z.re , im-=z.im; return *this; } // these are inlined
    complex& operator*=(complex);                  // defined out-of-class somewhere – not-inlined
    complex& operator/=(complex);                  // defined out-of-class somewhere
};
```

A Container - Vector

- The previously defined Vector has a fatal error – it never deallocates elements obtained via new – needed a mechanism to do that – *destructor*

```
class Vector {
private:
    double∗ elem;                                    // elem points to an array of sz doubles
    int sz;
public:
    Vector(int s) :elem{new double[s]}, sz{s} {      // constructor: acquire resources
        for (int i=0; i!=s; ++i) elem[i]=0;          // initialize elements
    }
    ~Vector() { delete[] elem; }                     // destructor: release resources
    double& operator[] (int i);
    int size() const;
};
```

- A destructor allows proper memory releases when objects of Vector go out of scope
- The ctor/dtor combo is the basis for **RAII** (Resource Acquisition is Initialization), which eliminates “naked new/delete operations” – avoiding allocation/release in user code

Initializing Containers

- Two ways:
  - Vector(std::initializer_list<double>); // Initialize with a list
  - void push_back(double); // Add element at end one at a time
- Implementation example for initializer-list:

```
Vector::Vector(std::initializer_list<double> lst)    // initialize with a list
    :elem{new double[lst.size()]}, sz{lst.size()} {
    copy(lst.begin(), lst.end(), elem);               // copy from lst into elem
}
```

  Note: std::initializer_list is a standard-library type: the compiler will create such object for the program when we use a {}-list
- The above allows us to use
  - Vector v1 = {1, 2, 3, 4, 5}; // v1 has 5 elements

Abstract Types

- A type that insulates a user from implementation details, i.e. it only exposes the interface
- We must allocate objects on the heap and access them via pointers/references
- Consider a more abstract version of our Vector (an *abstract class*)

```
class Container {
public:
    virtual double& operator[](int) = 0;    // pure virtual function, prevent instantiations
    virtual int size() const = 0;           // const member function
    virtual ~Container() {}                 // destructor
};
```

Note: virtual means: “may be redefined later in a class derived from this one”

- A class with a pure virtual function is called an *abstract class*

Usage of Abstract Types

- How do you use abstract types when you can't instantiate them?

```
void use(Container& c) {  // referred to by pointer/reference, as a polymorphic type
    const int sz = c.size();  // use() doesn't need to know the implementation of size() and []
    for (int i=0; i!=sz; ++i) // so abstract types allow runtime polymorphism
        cout << c[i] << '\n';
}
```

- Abstract classes generally don't have a constructor – no data to initialize
- They do have a destructor which are generally virtual – so implementation will take care of releasing resources

An Implementation of Container

```
class Vector_container : public Container {  // Vector_container implements Container
    Vector v;                                 // uses previously defined concrete class
public:
    Vector_container(int s) : v(s) { }        // Vector of s elements
    ˜Vector_container() {}               // implicitly calls member’s destructor ~Vector()
    double& operator[](int i) { return v[i]; } // overrides members in the base class
    int size() const { return v.size(); }
};
```

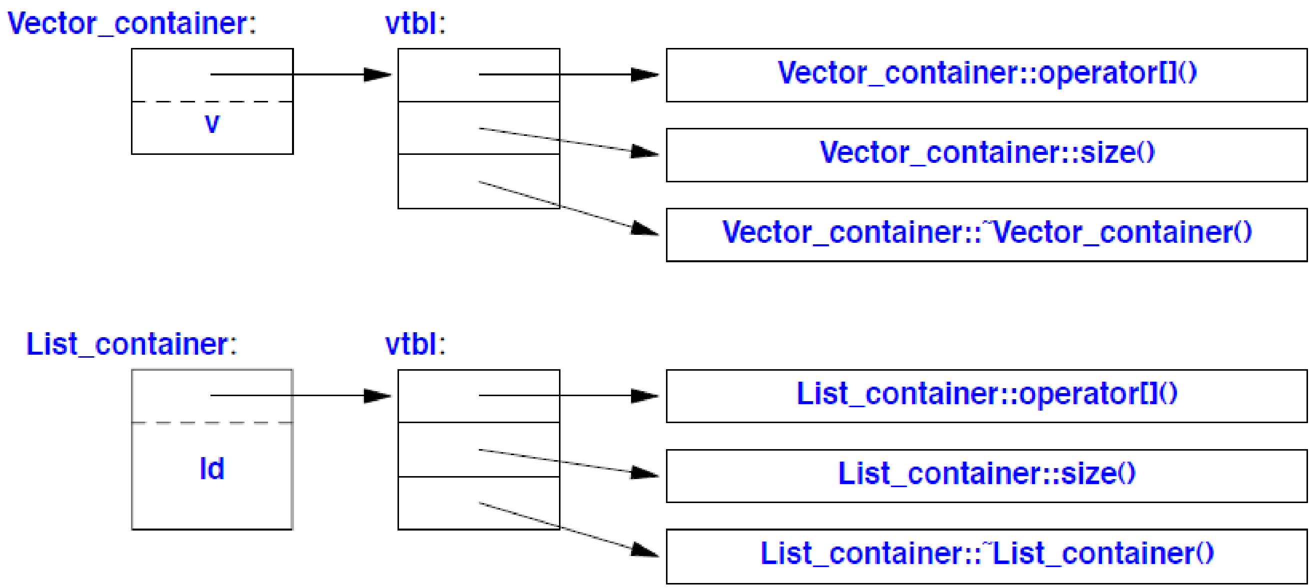
note: the : public specifier means “is derived from”

- If someone were to implement another Container with a list (instead of Vector v) – List_container, the function use() can use that object without changing any code, and need no recompiling

Virtual Functions

- For use() to resolve c.size(), it must know whether to use Vector_container's or List_container's implementation
- Compiler resolves this by making objects that include a table with all virtual functions accessed via an index – *virtual function table* or vtbl

- So to call c.size() when c could be referring to a Vector_container or List_container, the object's vtbl[1] slot is called
- That is, the slot contains the address for the correct version of size() – depending on what object c is referring to at the runtime



Class Hierarchies

- A Smiley is a kind of Circle which is a kind of Shape

```
class Shape {
public:
    virtual Point center() const = 0; // assume Point defined
    virtual void move(Point to) = 0;
    virtual void draw() const = 0;
    virtual void rotate(int angle) = 0;
    virtual ˜Shape() {}
};
```

// Based on Shape, we can write general functions like this:

```
void rotate_all(vector<Shape∗>& v, int angle) {
    for (auto p : v)
        p−>rotate(angle);
}
```

```
// partial implementation in Circle
class Circle : public Shape {
public:
    Circle(Point p, int rr); // constructor
    Point center() const { return x; }
    void move(Point to) { x=to; }
    void draw() const; // defined elsewhere
    void rotate(int) {}     // simple implementation
private:
    Point x; // center
    int r;     // radius
};
```

A Smiley implementation

```
class Smiley : public Circle {
// use the circle as the base for a face
public:
     Smiley(Point p, int r) : Circle{p,r}, mouth{nullptr} { }
    ˜Smiley() {  // destructor implementation
            delete mouth;
            for (auto p : eyes) delete p;
     }
     void draw() const;
     void add_eye(Shape∗ s) { eyes.push_back(s); }
     void set_mouth(Shape∗ s);
    // ...
private:
    vector<Shape∗> eyes; //usually two eyes
    Shape∗ mouth;
};
```

- The implementation for draw() is outside of the class:

```
void Smiley::draw()
{
      Circle::draw();
      for (auto p : eyes)
            p−>draw();
      mouth−>draw();
}
```

Class hierarchy offers two kinds of benefits

- Interface inheritance
  - The base class acts as an interface only – abstract classes (e.g. Container, Shape)
  - An object of a derived class can be used wherever a pointer/reference to base class is required
- Implementation inheritance
  - The base class provides functions/data that simplify the implementation of derived classes (e.g. Circle, whose constructor and draw() are used by Smiley)
  - Such base classes often have data members and constructors

Copy and Move

- Memberwise copy is ok for simple concrete types, but not for sophisticated ones, such as Vector:

```
void bad_copy(Vector v1)
{
    Vector v2 = v1;        // copy v1's representation into v2
    v1[0] = 2;             // v2[0] is now also 2!
    v2[1] = 3;             // v1[1] is now also 3!
}
```

- Solutions (define the following):
  - Copy constructor – for use in `Thing t2(t1);` or `Thing t2 = t1;`
  - Copy assignment – for use in `Thing t2;` then `t2 = t1;`

Copy Constructor/Assignment for Vector

- Defined in class Vector:

```
Vector(const Vector& a);              // copy constructor
Vector& operator=(const Vector& a);   // copy assignment
```

- Implementation:

```
// copy constructor
Vector::Vector(const Vector& a)
   :elem{new double[a.sz]}, sz{a.sz}
{
      for (int i=0; i!=sz; ++i)
         elem[i] = a.elem[i];
}
```

```
// copy assignment
Vector& Vector::operator=(const Vector& a)  {
    double∗ p = new double[a.sz]; // copy first
    for (int i=0; i!=a.sz; ++i)
          p[i] = a.elem[i];
    delete[] elem; // delete old elements
    elem = p; // assignment later
    sz = a.sz;
    return ∗this;
}
```

- **Copy ctor/assignment + dtor: the Big Three**

Move operations

- Copying can be expensive for large containers, and redundant for temporary objects (those will be discarded anyway)
- Since C++11, there is a new "move" semantics

```
class Vector {
  // move constructor and assignment:
  Vector(Vector&& a);
  Vector& operator=(Vector&& a);
};
```

A move constructor

```
Vector::Vector(Vector&& a)
:elem{a.elem},       // "grab the elements" from a, which
 sz{a.sz} {          // is a shallow copy, but it's ok.
   a.elem = nullptr; // now a has no elements
   a.sz = 0;  }
```

- Here && means "*rvalue* reference"
  - *lvalues* can always appear on the left-hand side of an assignment (with a name or address)
  - *rvalues* generally can't (like literals, or temporary primitive types) – if they are temporary objects returned by a function, they could appear on lhs but it's really meaningless
  - *rvalue* reference allows such object to be coupled with an address and become modifiable – its content will be stolen (this is good!) and itself can be destroyed.
  - std::move() can force its argument to become a move-from object
  - The **Big Three** plus the move ctor/assignment are termed the **Big Five** (or Rule of Five)

Resource Management

- Try to avoid using new and delete directly, or even pointers for managing resource
- Instead, use resource handles – Vector is a good example where the resource management is repackaged in the form of constructor/destructor.
- This helps eliminate resource leaks and achieve *strong resource safety*

Templates – Parameterized Types

- A *template* is a class or function that we parameterize with a set of types or values
- Parameterized Types (to make Vector a container of any possible type)

```
template<typename T>
class Vector {
private:
    T∗ elem;                  // elem points to an array of sz elements of type T
    int sz;
public:
    Vector(int s);            // constructor: establish invariant, acquire resources
    ˜Vector() { delete[] elem; } // destructor: release resources
    // ... copy and move operations ...
    T& operator[](int i);
    const T& operator[](int i) const; // the const qualifier at the end can't be missed
    int size() const { return sz; }
};
```

Implementation and Instantiations

- Implementation of the constructor

```
template<typename T>
Vector<T>::Vector(int s) {
    if (s<0) throw Negative_size{};
    elem = new T[s];
    sz = s;
}
```

- Implementation of the [] operator overload

```
template<typename T>
const T& Vector<T>::operator[](int i) const {
    if (i<0 || size()<=i)
    throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

- Some instantiation examples

```
Vector<char> vc(200);
Vector<string> vs(17);
Vector<list<int>> vli(45);
```

- A function using Vector of strings

```
void write(const Vector<string>& vs) {
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Templates – Function Templates

- To define a general summing function sum<T,V>

```
template<typename Container,
         typename Value>
Value sum(const Container& c, Value v)
{
    for (auto x : c)
        v+=x;
    return v;
}
```

// v needs to be passed with initial value

- Usages

```
void user(Vector<int>& vi, std::list<double>& ld,
          std::vector<complex<double>>& vc) {
    int x = sum(vi, 0); // the sum of a vector of ints (add ints)
    // Note: type parameters can be deduced for sum
    // i.e. sum<Vector, int>
    double d = sum(vi, 0.0); // the sum of a vector of ints
                             // (add doubles)
    double dd = sum(ld, 0.0); // the sum of a list of doubles
    auto z = sum(vc, complex<double>{});
            // the sum of a vector of complex<double>
            // the initial value is {0.0,0.0}
}
```

Function Objects (Functors)

- A template class used as a function - with overloaded operator()

```
template<typename T>
class Less_than {
        const T val; // value to be compared against
public:
        Less_than(const T& v) :val(v) { }
        bool operator()(const T& x) const { return x<val; } // call operator
};
```

- Instantiated versions

```
Less_than<int> lti {42}; // lti(i) will compare i to 42 using < (i.e. i<42)
Less_than<string> lts {"Backus"}; // lts(s) will compare s to "Backus" using < (i.e. s<"Backus")
```

- Main benefits
    - Carries values (e.g. the value to be compared against) – it can have states.
    - Can be easily inlined for efficiency (while function pointers can't)
    - Commonly used as arguments to general algorithms (as *policy objects*)

Aliases

- It is useful to introduce a synonym for a type or a template
- <cstddef> contains an alias size_t
  - using size_t = unsigned int; // in another implementation, could be unsigned long
  - Making use of size_t makes the code more portable
- It's common for a parameterized type to provide an alias for its template arguments (every standard-library container uses one)

```
template<typename T>
class Vector {
public:
        using value_type = T;
        // ...
};
```

Usage:

```
template<typename C>
   using Element_type = typename C::value_type;
template<typename Container>
void algo(Container& c) { // a template function
     Vector<Element_type<Container>> vec;
     // created a Vector of the same value_type of the Container c
     // or: Vector<typename Container::value_type> vec;
}
```

Chapter-end Advice

[1] Express ideas directly in code; §3.2.

[2] Define classes to represent application concepts directly in code; §3.2.

[3] Use concrete classes to represent simple concepts and performance-critical components; §3.2.1.

[4] Avoid ‘‘naked’’ **new** and **delete** operations; §3.2.1.2.

[5] Use resource handles and RAII to manage resources; §3.2.1.2.

[6] Use abstract classes as interfaces when complete separation of interface and implementation is needed; §3.2.2.

[7] Use class hierarchies to represent concepts with inherent hierarchical structure; §3.2.4.

[8] When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance; §3.2.4.

[9] Control construction, copy, move, and destruction of objects; §3.3.

[10] Return containers by value (relying on move for efficiency); §3.3.2.

[11] Provide strong resource safety; that is, never leak anything that you think of as a resource; §3.3.3.

[12] Use containers, defined as resource handle templates, to hold collections of values of the same type; §3.4.1.

[13] Use function templates to represent general algorithms; §3.4.2.

[14] Use function objects, including lambdas, to represent policies and actions; §3.4.3.

[15] Use type and template aliases to provide a uniform notation for types that may vary among similar types or among implementations; §3.4.5.