CISC 3142
Programming Paradigms in C++

Ch17-20 – Selected Topics

Abstraction Mechanisms: Elements of OO Programming

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

Constructor and Destructor

```
struct Tracer {
    string mess;
    Tracer(const string& s) :mess{s}
        { clog << mess; }
    ~Tracer() {clog << "~" << mess; }
};
void f(const vector<int>& v) {
    Tracer tr {"in f()\n"};
    for (auto x : v) {
        Tracer tr {string{"v loop "}+to_string(x)+'\n'};
        // …
    }
}
```

- Output for `f({2, 3, 5});`

```
in_f()
v loop 2
~v loop 2
v loop 3
~v loop 3
v loop 5
~v loop 5
~in_f()
```

- A destructor doesn't take an argument
- A class can have only one destructor
- Destructors typically clean up and release resources

Constructor/Destructor Execution Orders

- A constructor builds a class object “from the bottom up”
  1. First, the constructor invokes its base class constructors
  2. Then, it invokes the member constructors, and
  3. Finally, it executes its own body
- A destructor “tears down” an object in the reverse order
  1. First, the destructor executes its own body
  2. Then, it invokes its member destructors, and
  3. Finally, it invokes its base class destructors
- If a class doesn’t provide ANY constructor, the compiler will generate a default constructor, so `T t;` will not cause compile error
  - a *default constructor* is one that requires no arguments

virtual Destructors

- A destructor can be declared to be virtual, and usually should be for a class with a virtual function

```
class Shape {
public:
    // ...
    virtual void draw() = 0;
    virtual ˜Shape();
};
class Circle : public Shape {
public:
    // ...
    void draw();
    ˜Circle(); // overrides ˜Shape()
    // ...
};
```

- The reason we need a virtual destructor is that an object manipulated through the base class' interface is often deleted through that interface

```
void user(Shape∗ p) {
    p−>draw(); // invoke the appropriate draw()
    // ...
    delete p; // invoke the appropriate dtor
};
```

- Without the virtual destructor, ~Circle() won't get called and this leads to memory leak

Class Object Initialization

- Without constructors

```
struct Work {
    string author; // class member is init'ed
    string name;
    int year; // built-in type depends on scope
};
Work alpha; // is {"", "", 0}
void f() {
    Work beta; // is {"", "", unknown}
    // ...
}
```

  - **A class with a private non-static data member must have a constructor to initialize it**

- Difference with vs without Constructors
  - use {} for memberwise initialization (w/o ctor), and call constructor (with ctor) – **this is called *aggregate initialization* but only to all-members-public classes.**
  - Use () to call constructor explicitly (**now can also be used for aggregate initialization – since C++20?**)

```
struct S1 {
    int a, b; // no constructor
};
struct S2 {
    int a, b;
    S2(int aa = 0, int bb = 0) : a(aa), b(bb) {} // constructor
};
S1 x11(1,2);  // error : no constructor (OK under C++20)
S1 x12 {1,2}; // OK: memberwise initialization
S1 x13(1);    // error : no constructor (OK under C++20)
S1 x14 {1};   // OK: x14.b becomes 0
S2 x21(1,2); // OK: use constructor
S2 x22 {1,2}; // OK: use constructor
S2 x23(1);    // both OK: use constructor and one default
S2 x24 {1};   // argument
```

Member and Base Initialization

- Member initialization

```
class Club {
   string name;
   vector<string> members;
   vector<string> officers;
   Date founded;
   // ...
   Club(const string& n, Date fd);
};
```

- Constructor

```
Club::Club(const string& n, Date fd)
      : name{n}, members{}, officers{}, founded{fd} {
   // ...
}
```

- Benefit of initializer list for data members
  - Don’t need to do: `this->var = var;` // `var(var)` is allowed
  - Only way for initializing non-static `const`, and reference members
  - For member objects without default constructor, i.e. initializer list happens ahead of the default constructor

- Performance boost of initializer list against assignment:
  - Copy constructor is used, only one copy is made
  - Whereas with an assignment: default ctor is called first. Then the object created via default ctor is replaced by the passed object argument – wasted work for default ctor

- Base initialization

```
class B1 { B1(); }; // has default constructor
class B2 { B2(int); } // no default constructor
struct D1 : B1, B2 {
   D1(int i) : B1{}, B2{i} {}
};
struct D2 : B1, B2 {
   D2(int i) : B2{i} {} // B1{} is used implicitly
};
struct D1 : B1, B2 {
   D1(int i) { } // error : B2 requires an int initializer
}
```

Delegating Constructor

- Without it

```
class X {
  int a;
  validate(int x) {
    if (0<x && x<=max) a=x;
    else throw Bad_X(x);
  }
public:
  X(int x) { validate(x); }
  X() { validate(42); }
  X(string s) { int x = to<int>(s); validate(x); }
  // §25.2.5.1
  // ...
};
```

- See the multiple invocations of validate(), which can be re-written:

```
class X {
    int a;
public:
    X(int x) {
        if (0<x && x<=max) a=x;
        else throw Bad_X(x);
    }
    X() :X{42} { }
    X(string s) :X{to<int>(s)} { }
    // ...
};
```

- A constructor that calls another constructor as part of the construction is named a *delegating constructor*, or *forwarding constructor*

Copy

- Copy for a class X is defined by two operations
  - Copy constructor: X(const X&)
  - Copy assignment: X& operator=(const X&)

```
template<class T>
class Matrix {
    array<int,2> dim; // two dimensions dim[0]: row, dim[1]: col
    T∗ elem; //pointer to dim[0]*dim[1] elements of type T
public:
    Matrix(int d1, int d2) :dim{d1,d2}, elem{new T[d1∗d2]} {} // simplified (no error handling)
    int size() const { return dim[0]∗dim[1]; }
    Matrix(const Matrix&);                   // copy constructor
    Matrix& operator=(const Matrix&);        // copy assignment
    Matrix(Matrix&&);                        // move constructor
    Matrix& operator=(Matrix&&);             // move assignment
    ˜Matrix() { delete[] elem; }
    // ...
};
```

Implementation of Copy Constructor/Assignment

```
template<class T>
Matrix:: Matrix(const Matrix& m) // copy constructor
	: dim{m.dim}, elem{new T[m.size()]} { // deep copy here
	uninitialized_copy(m.elem, m.elem+m.size(), elem); // copy elements
}

template<class T>
Matrix& Matrix::operator=(const Matrix& m) { // copy assignment
	if (dim[0]!=m.dim[0] | | dim[1]!=m.dim[1]) // could have checked for self-assignment
		throw runtime_error("bad size in Matrix =");
	copy(m.elem, m.elem+m.size(), elem); // copy elements -> ok even self-assign
}
```

- A copy constructor initializes uninitialized memory
- A copy assignment reassign an object that has already been constructed

Explicit defaults and deleted functions

- Compiler generated default ctor/dtor will be suppressed when a programmer creates own versions
- You can use default to bring them back

```
class gslice {
    valarray<size_t> size;
    valarray<size_t> stride;
    valarray<size_t> d1;
public:
    gslice() = default;
    ~gslice() = default;
    gslice(const gslice&) = default;
    gslice(gslice&&) = default;
    gslice& operator=(const gslice&) = default;
    gslice& operator=(gslice&&) = default;
    // ...
}; // equivalent to removing everything after public
```

- We can also state that a function does not exist so that it is an error to use it
- It's common to use it to prevent the copy of classes

```
class Base {
    // ...
    Base& operator=(const Base&) = delete;
                                   // disallow copying
    Base(const Base&) = delete;
    Base& operator=(Base&&) = delete;
                                   // disallow moving
    Base(Base&&) = delete;
};
Base x1;
Base x2 {x1}; // error : no copy constructor
```

Operator Overloading

- All of the following operators can be overloaded:

```
+   -   *   /   %   ^   &
|   ~   !   =   <   >   +=
-=  *=  /=  %=  ^=  &=  |=
<<  >>  >>=  <<=  ==  !=  <=
>=  &&  ||  ++  --  ->* ,
->  []  ()  new  new[]  delete  delete[]
```

- The following operators cannot be redefined

::	scope resolution (§6.3.4, §16.2.12)
.	member selection (§8.2)
.*	member selection through pointer to member (§20.6)
sizeof	size of object (§6.2.8)
alignof	alignment of object (§6.2.9) (char: 1, int: 4, etc)
typeid	type_info of an object (§22.5)
?:	conditional evaluation (§9.4.1)

Operator Overloading Examples
(for Complex numbers)

```
class complex {
   double re, im;
public:
   complex& operator+=(complex a) {
      re += a.re;
      im += a.im;
      return *this;
   }
   complex& operator+=(double a) {
      re += a;
      return *this;
   }
   // ...
};
complex operator+(complex a, complex b) {
   return a += b; // calls complex::operator+=(complex)
}
```

```
complex operator+(complex a, double b) {
   return {a.real()+b, a.imag()};
}
complex operator+(double a, complex b) {
   return {a+b.real(), b.imag()};
}
```

```
void f(complex x, complex y) {
   auto r1 = x+y; // calls operator+(complex,complex)
   auto r2 = x+2; // calls operator+(complex,double)
   auto r3 = 2+x; // calls operator+(double,complex)
   auto r4 = 2+3; // built-in integer addition
}
```

friends and Members

- An ordinary member function
  - can access the private part of the class declaration
  - is in the scope of the class
  - must be invoked on an object (has a this pointer)
- By declaring a nonmember function a friend, we can give it the first property only
- Example
  - To define an operator* (multiplication) for a Matrix by a Vector
  - Naturally, both of them hide their representations from each other
  - Our multiplication routine cannot be a member of both (need simultaneous access to both)
- One could also use friend class

```
class List {
    friend class List_iterator; // all of List_iterator's member functions are friends of List
    // ...
};
```

Friend function example

```
constexpr int rc_max {4}; // row/col size
class Matrix;
class Vector {
    float v[rc_max];
    // ...
    friend Vector operator∗(const Matrix&,
                            const Vector&);
};
class Matrix {
     Vector v[rc_max];
    // ...
    friend Vector operator∗(const Matrix&,
                            const Vector&);
};
```

- No need to say friend in definition

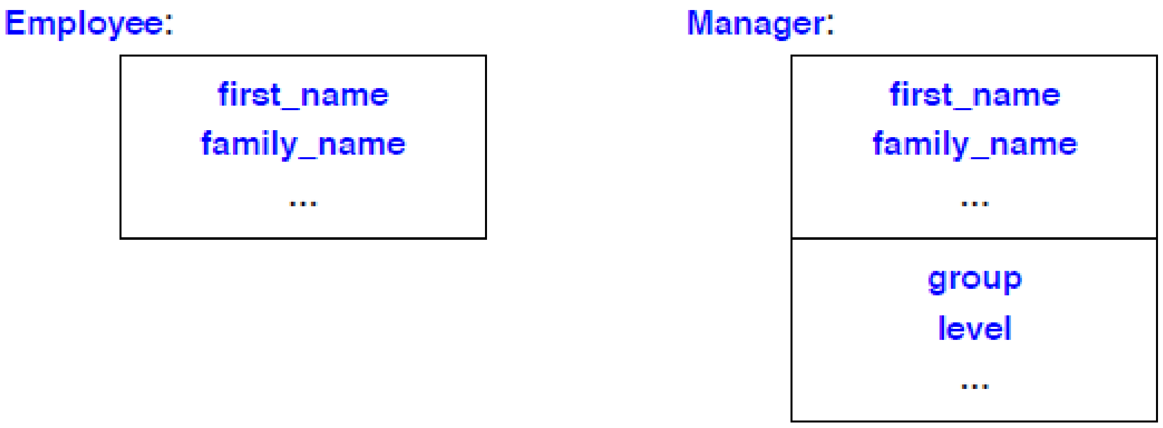
```
Vector operator∗(const Matrix& m,
                 const Vector& v) {
  Vector r;
  for (int i = 0; i!=rc_max; i++) {
    // r[i] = m[i] * v;
    r.v[i] = 0;
    for (int j = 0; j!=rc_max; j++)
         r.v[i] += m.v[i].v[j] ∗ v.v[j];
  }
  return r;
}
```

Derived Classes

- Basis for object-oriented programming
  - **Implementation inheritance**: using implemented facilities provided by a base class
  - **Interface inheritance**: allowing different derived classes to use uniform interface
- Example:

```
struct Employee { // Employee is a base class for Manager
        string first_name, family_name;
        char middle_initial;
        Date hiring_date;
        short department;
        // ...
};
struct Manager : public Employee { // Manager
        // is derived from and a subtype of Employee
        list<Employee∗> group;
        short level;
        // ...
};
```

Memory Layout:



Manager is an Employee

- Manager can be used wherever an Employee is accepted
- Manager* (or Manger&) can be used as an Employee* (or Employee&) → upcast is safe
- The opposite (downcast) is not safe (use dynamic_cast to be safe)

```
void g(Manager& mm, Employee& ee)
{
    Employee∗ pe = &mm; // OK: every Manager is an Employee
    Manager∗ pm = &ee; // potential error : not every Employee is a Manager
    pm−>level = 2; // disaster : ee doesn't have a level
    pm = static_cast<Manager∗>(pe); // brute force downcast: works here because pe points
    // to the Manager mm, but in general downcast is not safe, the following is better:
    if ( (pm = dynamic_cast<Manager*>(pe)) != nullptr ) // dynamic_cast returns nullptr if not safe
       pm−>level = 2; // safe: pm now points to the Manager mm that has a level
}
```

- A derived class can access base class's public and protected members, but not its private members

Virtual Functions

- Allow base class to declare functions that can be redefined (overridden) in each derived class

```
class Employee {
public:
    Employee(const string& name, int dept);
    virtual void print() const; // ensures the correct derived version invoked, when accessed via an Employee* or Employee&
    // ...                      // print() implementation skipped here, but assume it prints name and department
private:
    string name;
    short department;
    // ...
};
```

- A virtual function *must* be defined for the class in which it is first declared unless it is declared to be a pure virtual function (no body, with = 0)
- Argument types must be the same in the derived class

```
void Manager::print() const { // This is called overriding
    Employee::print(); // can make use of functionality provided by the base, being explicit on using which version of function
    cout << "\tlevel " << level << '\n';        // using scope operator :: disables virtual mechanism
}
```

Run-time Polymorphism

- A list of Employees can be printed (**without knowing about the existence of a derived class named Manager!**)

```
void print_list(const list<Employee∗>& s) {
        for (auto x : s)
            x−>print();
}
```

- Making use of the above function (assuming Manager has name, dept, level)

```
int main() {
        Employee e {"Brown", 1234};
        Manager m {"Smith", 1234, 2};
        print_list({&m, &e});
}
```

- Output:

```
Smith 1234
        level 2
Brown 1234
```

- A type with virtual functions is called a run-time polymorphic type
  - Two key ingredients: virtual functions and objects manipulated via pointers/references
  - Such mechanism can be stopped by a trailing final at derived class’ function declaration

Abstract Classes

- While the base class Employee is useful as itself, there are classes that represent abstract concepts for which objects cannot exist

```
class Shape {
public:
    virtual void rotate(int) { throw runtime_error{"Shape::rotate"}; } // inelegant to prevent any behavior of Shape
    virtual void draw() const { throw runtime_error{"Shape::draw"}; }
    // …
};
```

- It’s actually legal to instantiate an object of Shape

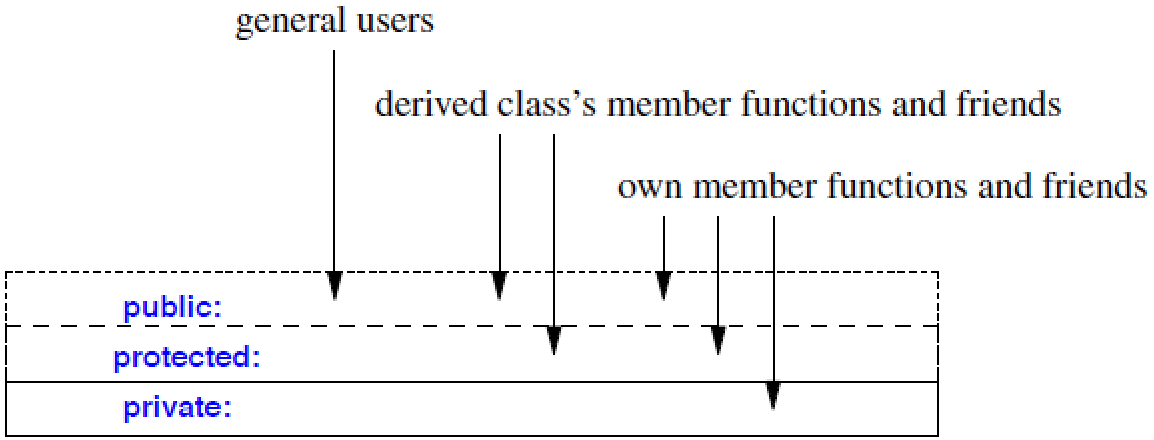
```
Shape s; // silly: ‘‘shapeless shape’’, any operation on s results in error
```

- More elegant way is to declare *pure virtual functions* with the “pseudo initializer” = 0

```
class Shape { // abstract class, now it can’t be instantiated, and only used as an Interface
public:
    virtual void rotate(int) = 0; // pure virtual function
    virtual void draw() const = 0; // pure virtual function
    // …
    virtual ~Shape(); //virtual
};
```

Access Control

- A member of a class can be private (default for class), protected, or public (default for struct)



- A base class can be declared private (default for class), protected, or public (default for struct)

  class X : public B { /* … */ };

  // X is a subtype of B, All B's `public`/`protected` members become X's `public`/`protected`, B's `private` remains `private` (this is always true)

  class Y : protected B { /* … */ };

  // B's `public`/`protected` become Y's `protected`

  class Z : private B { /* … */ };

  // B's `public`/`protected` become Z's `private`