面向对象分析与设计
Object-Oriented Analysis and Design

北京理工大学软件学院
马　锐
Email: mary@bit.edu.cn

1 A Tour of C++

1.1 What is the C++

- The C++ is a general-purpose programming language with a bias towards systems programming that:
  - is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming

1.2 Better C(1)

```
//example 1.1
#include <iostream>              //I/O stream base
using namespace std;             //namespace
int min(int a,int b);            //function declaration
int min(int a,int b,int c);      //function overloading
int main()
{
    const int a=10;                  //constant
    const int &b=a;                  //reference
    cout<<min(a,b,4)<<endl;          //output
    cout<<min(-2,a)<<endl;
}
```


1.2 Better C(2)

```
//example 1.1
int min(int a,int b)             //function overloading
{
    return a<b?a:b;
}
int min(int a,int b,int c)
{
    int t=min(a,b);
    return min(t,c);
}
```

Output:

4
-2

1.2 Better C(3)

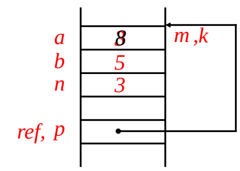
1.2.1 References(1)

- A reference is an alternative name for an object.
- Declaration (Definition)
  A. Type &<reference name>(<variable>);
  B. Type &<reference name>=<variable>;
- Usage
  - Must initialize the reference;
  - Using reference and variable are the same.
  - The value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote.

7

1.2.1 References(2)

```
int a=3,b=5;
int &m=a;    //ok
int &m;      //error
int &m=b;    //error
int &k=m;
int n=m;
……
int *p=&m;
int * &ref=p;
m=m+5;
```

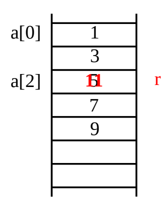


8

1.2.1 References(3)

```
//example 1.2
#include <iostream>
using namesapce std;
int& f(int index,int a[]) {
    int &r=a[index];
    return r;
}
int main() {
    int a[]={1,3,5,7,9};
    f(2,a)=11;
    for(int i=0;i<5;i++)
        cout<<a[i]<<"\t";
}
```

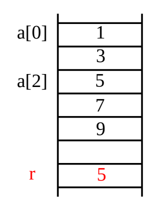
Output:
1 3 11 7 9



9

1.2.1 References(4)

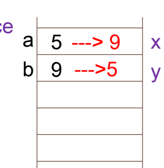
```
//example 1.2
#include <iostream>
using namesapce std;
int& f(int index,int a[]) {
    int  r=a[index];
    return r;
}
int main() {
    int a[]={1,3,5,7,9};
    f(2,a)=11;
    for(i=0;i<5;i++)
        cout<<a[i]<<"\t";
}
```



10

1.2.1 References(5)

```
//example1.3: passing by reference
#include <iostream>
using namespace std;
void swap3(int &x,int &y) {
    int temp;
    temp=x;
    x=y;
    y=temp;
    cout<<"x="<<x<<", "<<"y="<<y<<endl;
}
int main() {
    int a(5),b(9);
    swap3(a,b);
    cout<<"a="<<a<<", "<<"b="<<b<<endl;
}
```



int &x=a;
int &y=b;

11

1.2.2 Constant(1)

- Declaration
  const T <identifier> = <values>;
- Usage
  - Must be initialized;
  - Ensure its value will not change within its scope;
- Eg:
  - const int model=90; //model is a const
  - const int v[ ]={1,2,3,4}; //v[i] is a const
  - model=200; //error
  - v[2]++; //error

12

1.2.2 Constant(2)

- const int x; //error: no initilizer
- double x=1.2;
- const double &v=x; //constant reference
- v=12.3; //error
- Differences between symbolic constants & #define
  - const double PI=3.14;
    - PI has a double type and its value is 3.14.
  - #define PI 3.14
    - PI has no type and value. It is only a replacement.

13

1.2.3 Free Store(1)

- Free Store Operators
  - new/new[ ]: creates dynamic objects.
  - delete/delete[ ]: destroy dynamic object.
- new/new[] operator

  new <type>(<initializer>); //individual objects

  new <type>[<size>]; //arrays
  - new/new[] returns a pointer that points to type.
  - new[ ] does not initialize the memory returned.
  - when new/new[] can find no store to allocate, by default, the allocator throws a bad_alloc exception.

14

1.2.3 Free Store(2)

- delete/delete[] operator

  delete <pointer name>; //individual objects

  delete[ ] <pointer name>; //arrays
  - An object created by new must be destroyed by delete.
  - The delete/delete[] operator may be applied only to a pointer returned by new/new[] or to zero.
  - A pointer can be destroyed by delete/delete[] only once.
  - Notice delete[]: Its ignored the dimension and size. There is only one [] pairs.

15

1.2.3 Free Store(3)

```
//example 1.4
#include <iostream>
using namespace std;
int main(){
   int *p;
   p=new int(5);          //individual object
   cout<<*p<<endl;
   delete p;              //destroy individual object
   p=new int[5];          //array
   for(int i=0;i<5;i++)
       *(p+i)=i;
   for(i=0;i<5;i++)
       cout<<*(p+i)<<"\t";
   delete[] p;            //destroy array
}
```

Output:				
5				
0	1	2	3	4

16

1.2.4 Namespace(1)

- A namespace is a mechanism for expressing logical grouping and it is a scope.
- Declaration and Definition

```
namespace namespace-name {
    //declaration and definition
}
namespace Parser {   //declaration&definition
   double prim(bool) {/*…*/}
   double term(bool) {/*…*/}
   double expr(bool) {/*…*/}
}
```

17

1.2.4 Namespace(2)

```
namespace Parser {          //declaration
   double prim(bool);
   double term(bool);
   double expr(bool);
}
//definition
double Parser::prim(bool get) { /*…*/}
double Parser::term(bool get) { /*…*/}
double Parser::expr(bool get) { /*…*/}
```

18

1.2.5 Function Overloading(1)

- When some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name.
- Using the same name for operations on different types is called overloading.

```
void print(double);
void print(long);
void f() {
    print(1L);          //print(long)
    print(1.0);         //print(double)
    print(1);           //error:ambiguous
}
```

19

1.2.5 Function Overloading(2)

- The compiler must figure out which of the functions is to be invoked.
- Invoke the functions that is the best match on the arguments:
  - the number of the arguments;
  - the type of the arguments;
- If no function is the best match, the compiler will give a compile-time error.
- The overloading resolution is independent of the order of declaration of the functions considered.

20

1.2.5 Function Overloading(3)

- A series of criteria of the best match in order:
  - Exact match: match using no or only trivial conversions;
  - Match using promotions: integral promotions and float to double;
  - Match using standard conversions;
  - Match using user-defined conversions;
  - Mach using the ellipsis … in a function declaration.
- If two matches are found at the highest level where a match is found, the call is rejected as ambiguous.

21

1.2.5 Function Overloading(4)

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);
void h(char c, int i, short s, float f){
    print(c);        //exact match: print(char)
    print(i);        //exact match: print(int)
    print(s);        //integral promotion: print(int)
    print(f);        //float to double: print(double)
    print('c');      //exact match: print(char)
    print(49);       //exact match: print(int)
    print(0);        //exact match: print(int)
    print("a");      //exact match: print(const char*)
}
```

22

1.2.5 Function Overloading(5)

- Notion
  - Return types are not considered in overload resolution.

```
int add(int ,int);
void add(int,int);        //error: redefinition
```

  - The name of arguments are not considered in overload resolution.

```
int add(int x,int y);
void add(int a,int b);    //error: redefinition
```

23

1.2.5 Function Overloading(6)

- The body of functions are similar.

```
int add(int x,int y) { return x+y; }
double add(double x,double y)
    { return x-y; }
```

- Functions declared in different non-namespaces scopes do not overload.

```
void f(int);
void g() {
    void f(double);
    f(1);                 //call f(double)
}
```

24

1.2.6 Default Arguments(1)

- Assignment
  - Default arguments can be assigned in the function declaration or function definition. But it must be assigned only once that is the first time the function appears.

    int add(int x,int y=0);

    int add(int x,int y=0) {//the body of function}
- Default arguments may be provided for trailing arguments only (assignment must from right to left).

  int add(int x=1,int y=5,int z); //error

  //add(2,4);⇔add(1,2,4)? No, z is no value.

1.2.6 Default Arguments(2)

int add(int x=1,int y,int z=6); //error

//add(2); ⇔ add(1,2,6)?In fact, y is no value.

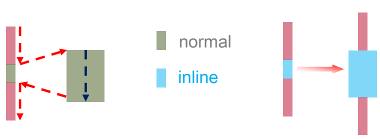
- How to Use Default Arguments?
  - If the number of actual arguments exact match formal arguments, we don't use default arguments;
  - Otherwise (less than), the compiler will complement actual arguments from left to right using default arguments.

    int add(int x,int y=5,int z=6);

    add(15); ⇔ add(15,5,6);

1.2.7 Inline Functions(1)

- Objective:
  - When a function is called, it can improve its efficiency, especially those functions which has several lines code but can be called frequently.
- Solved Methods:
  - Replacement, that is, Increasing the number of code, saving time and improving efficiency.



1.2.7 Inline Functions(2)

- Inline Function Definition

inline <type> <function name>(<formal arguments>)
{ //… }

```
inline int add(int x,int y,int z) {
  return x+y+z;
}
int main(){
  int a(1),b(2),c(3),sum(0);
  sum=add(a,b,c);  //replaced by sum=1+2+3;
}
```


1.2.7 Inline Functions(3)

```
#include <iostream>
using namespace std;
#define f(x) x*x

int main(){
    int x(2);
    cout<<f(x)<<endl;
    cout<<f(x+1)<<endl;
}
```

Output:
4 *f(x) is replaced 2*2*
5 *f(x+1) is repalced 2+1*2+1*

```
#include <iostream>
using namespace std;
inline int f(int x){
    return x*x;
}
int main(){
    int x(2);
    cout<<f(x)<<endl;
    cout<<f(x+1)<<endl;
}
```

Output:
4 *f(x) is replaced 2*2*
9 *f(x+1) is repalced 3*3*

1.3 Data Abstraction(1)

```
//example 1.5
#include <iostream>
using namespace std;
const int Max=100;
class IntSet {                    //class declaration
public:                           //public members
   IntSet();                      //Constructor
   IntSet(int a[ ], int size);
   ~IntSet() { }                  //Destructor
   void Print();                  //Member Functions
   IntSet Intersection(const IntSet& set);
private:                          //private members
```

1.3 Data Abstraction(2)

```
//example 1.5
    int element[Max];          //data members
    int end;
};
IntSet::IntSet() {             //constructor definition
    for(int i=0;i<Max;i++)
        element[i]=-1;
    end=-1;
}
IntSet::IntSet(int a[ ],int size) {
    if(size>=Max)     end=Max-1;
    else              end=size-1;
```

31

1.3 Data Abstraction(3)

```
//example 1.5
    for(int i=0;i<=end;i++)
        element[i]=a[i];
}
//Member Functions definition
IntSet IntSet::Intersection(const IntSet& set) {
    int a[Max],size=0;
    for(int i=0;i<=end;i++)
        for(int j=0;j<=set.end;j++)
            if(element[i]==set.element[j]) {
                a[size++]=element[i];
                break;
```

32

1.3 Data Abstraction(4)

```
//example 1.5
    }
    return IntSet(a,size);
}
void IntSet::Print() {
    for(int i=0;i<=end;i++)
        if((i+1)%20==0)    cout<<element[i]<<endl;
        else               cout<<element[i]<<'\t';
    cout<<endl;
}
int main() {
    int a[ ]={1,2,3,45,8,10};
```

33

1.3 Data Abstraction(5)

```
//example 1.5
    int b[ ]={2,8,9,11,30,56,67};
    IntSet set1(a,6),set2(b,7),set3;
    set3=set1.Intersection(set2);          //objects
    cout<<"set1: ";
    set1.Print();
    cout<<"set2: ";
    set2.Print();
    cout<<"set3: ";
    set3.Print();
}
```

Output:

```
set1: 1  2  3  45  8   10
set2: 2  8  9  11  30  56  67
set3: 2   8
```

34

1.3 Data Abstraction

35

1.3.1 Classes(1)

- A class is a user-defined data type.
- The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types.

```
int x;
int x(5);

Date d;
Date d(2010,2,14);
```

36

1.3.1 Classes(2)

Keyword　Name

Member functions

Access Specifiers

Data Member

```
class Date
{
public:
    void SetDate(int y,int m,int d);
    int  IsLeapYear();
    void Print();
private:
    int year,month,day;
};
// member function's definition
//…
```

37

1.3.1 Classes(3)

- Access Specifier
  - public
    - The "public" part constitutes the public interface to objects of the class.
  - private
    - The "private" part can be used only by member functions and nonmember functions are barred from using private members.
  - protected

38

1.3.1 Classes(4)

```
class Date{
public:
    void SetDate(int y,int m,int d);           //declaration
    int  IsLeapYear();                         //declaration
    void Print()                               //definition
        {cout<<year<<"."<<month<<"."<<day<<endl;}
private:
    int year,month,day;
};
void Date::SetDate(int y,int m,int n){         //definition
    year=y;
    month=m;
    day=d;
}
```

Scope Resolution Operator "::"

39

1.3.2 Object(1)

- An object is an entity of the creating system.
- An object has three elements:
  - name　　　Xmas
  - state
    - attribute　　　year,month,day
    - value　　　2009,12,25
  - action
    - methods or functions　　　SetDate,Print, IsLeapYear
- Eg. Date Xmas(2009,12,25);

40

1.3.2 Object(2)

- Definition
  - <class-name> <objects-name>;
  - <class-name> <objects-name>(<initilizer>);
- Examples

```
Date firstDay;
Date Xmas(2009,12,25);
Date *pDate;
Date &refDate=Xmas;
Date arrDate[ ]={firstDay, Xmas};
```

41

1.3.3 Constructors & Destructor(1)

```
//date.h
class Date {
public:
    Date(int y,int m,int d)                    //constructor
        :year(y),month(m),day(d)
        { cout<<"Constructor called. "<<endl; }
    ~Date()                                    //destructor
        { cout<<"Destructor called. "<<endl;   }
    void Print() {
        cout<<year<<'.'<<month<<'.'<<day<<'.'<<endl;}
private:
    int year,month,day;
};
```

42

1.3.3 Constructors & Destructor(2)

A constructor is recognized by having the same name as the class itself.

A destructor is recognized by having the same name as the class itself with the complement symbol (~).

A constructor can be overloaded.

A destructor has not formal arguments and cannot be overloaded.

A constructor has not a return value.

A destructor has not a return value.

1.3.2 Constructors & Destructor(3)

- Constructors
  - A constructor initializes objects and constructs values of a given type.
  - A constructor can be invoked automatically when an object is created.
- Destructors
  - A destructor cleans up and release resources.
  - A destructor can be invoked automatically when an object is destroyed.
- The destructors are called in the reverse order of construction.

44

1.3.3 Constructors & Destructor(4)

- the execution of a constructor
  - initializer list
    - All items in the initializer list are called in the order in which items are declared in the class.
    - The initializer list(item) is still executed (initialized) although it has been left out.
  - the body of the constructor
    - All items in the body of the constructor are called in the order of definition.

45

1.3.3 Constructors & Destructor(5)

- There is an efficiency advantage to using the initializer syntax. (Initialization & Assignment)

```
class Date {
public:
    // year is initialized with a copy of y
    Date(int y,int m,int d):month(m),year(y)  {day=d;}
    // day is first initialized to a value, then a copy of
    // d is assigned
private:
    int year,month,day;
};
```

46

1.3.4 The this Pointer(1)

- In a non-static member function, the keyword this is a pointer to the object for which the function was invoked.
- *this refers to the object for which a member function is invoked.
- In a non-static member function, this is the first implicit formal argument.
- Most used of this are implicit.

47

1.3.4 The this Pointer(2)

```
Date& Date::AddOneMonth() {
    if(month==12) { year++; month=1; }
    else month++;
    return *this;
}
Date& Date::AddOneMonth() {
    if(this->month==12) {
        this->year++; this->month=1; }
    else this->month++;
    return *this;
}
```

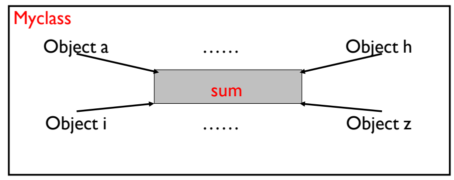
48

1.3.5 Static Members(1)

- The Objective
  - It can get the convenience of data shares without the encumbrance of a publicly accessible global variable.
- The features
  - static member is part of a class, yet is not part of an object of that class.
  - There is exactly one copy of a static member instead of one copy per object, as for ordinary non-static member.
- Declaration: static <members>;

1.3.5 Static Members(2)

```
class MyClass { //…
private:
   static int sum;         //static data member
};
```



1.3.5 Static Members(3)

- The usage of static members
  - within the class body, a static member can be referred to like any other member.
  - outside the class body

    <class-name>::<static-member-name>
- Static members has been created before any objects of its class has been created.
- Static members is independent of any objects.

1.3.5 Static Members(4)

- static data members must be initialized.

<type> <class-name>::<member-name>=<value>

  - static data members must be defined outside the class body.
  - Its initialization is independent of its access specifier.
  - There is no static keyword when static data member is initialized.
  - static data member outside the class body must be qualified by the name of its class.

1.3.5 Static Members(5)

```
//example1.6
#include <iostream>
using namespace std;
class Myclass {
public:
   Myclass(int a,int b,int c) : x(a),y(b),z(c)
      { sum+=x+y+z; }
   void GetNumber();
   void GetSum()  {  cout<<"Sum="<<sum<<endl; }
private:
   int x,y,z;
   static int sum;           //static data member
};
```


1.3.5 Static Members(6)

```
//example1.6
int Myclass::Sum=0;              //initialization
inline void Myclass::GetNumber() {
   cout<<"Number="<<x<<","<<y<<","<<z<<endl;
}
int main() {
   Myclass M(3,7,10),
           N(14,9,11);
   M.GetNumber();
   N.GetNumber();
   M.GetSum();
   N.GetSum();
}
```

```
Output:

Number=3,7,10
Number=14,9,11
Sum=54
Sum=54
```

1.3.5 Static Members(7)

- A function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a static member function.
- There is no static keyword when static member functions are defined outside the class body.
- A static member function can be accessed static members belonging to its class directly and cannot be accessed non-static members directly.
- A static member function can be accessed non-static members through a object.

55

1.3.6 Const Members(1)

- Const Objects Definition

  const <class-name> <object-name>(initializer);
  <class-name> const <object-name>(initializer);

```
class Point {
public:
   Point(int i=0,int j=0) : x(i),y(j) { }
private:
   int x,y;
};
const Point origin;              //constant object
Point const vertex(3,4);         //constant object
```

56

1.3.6 Const Members(2)

- Const Member Functions Declaration

  <type> <function-name>(<arguments>) const;
  - The const after the argument list in the function declarations.

```
class Date  {
public:
    int GetDay() const { return day; }
    int GetMonth() const { return month; }
    int GetYear() const;
private:
    int year,month,day;
};
```

57

1.3.6 Const Members(3)

- Const Member Functions Definition
  - When a const member function is defined outside its class, the const suffix is required.

```
inline int Date::GetYear() const {     //correct
    { return year;  }
inline int Date::GetYear()           //error
    { return year;  }
```

- A const member function indicates that it does not modify the state of an object.

```
inline int Date::GetYear() const
    { return year++; }                 //error
```

58

1.3.6 Const Members(4)

- Const data members must be explicitly initialized in a member initializer list in the definition of the constructor of the class.
- Because consts and references data members must be initialized, a class containing const or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor.

59

1.3.7 Friends(1)

- The keyword friend is a function specifier that gives a non-member function access to the hidden members of the class and provides a method to improve efficiency.
- A friend (function) must be declared inside the class declaration to which it is a friend.
- A friend cannot be inherited, commutated or transferred.
- It can be divided three kinds:
  - friend functions
  - friend member functions
  - friend class

60

1.3.7 Friends(2)

```
//example1.7
#include <iostream>
#include <string>
using namespace std;
class Girl;                     //forward declaration
class Boy {
public:
    Boy(char *str) {
        name=new char[strlen(str)+1];
        strcpy(name,str);
    }
    ~Boy() { delete[ ] name;}
    void Show(const Girl&);     //member function
```

61

1.3.7 Friends(3)

```
//example1.7
private:
    char *name;
};
class Girl {
public:
    Girl(char *str) {
        name=new char[strlen(str)+1];
        strcpy(name,str);
    }
    ~Girl() { delete[ ] name;}
    //friend member function
    friend void Boy::Show(const Girl&);
```

62

1.3.7 Friends(4)

```
//example1.7
private:
    char *name;
};
void Boy::Show(const Girl & g) {
    cout<<"Boy's name is "<<name<<endl
        <<"Girl's name is "<<g.name<<endl;
}
int main() {
    Boy b("Bob");
    Girl g("Kitty");
    b.Disp(g);
}
```

Output:

Boy's name is Bob
Girl's name is Kitty

63

1.3.8 Member Objects(1)

```
//part.h
class Part {
public:
    Part();
    Part(int i);
    ~Part();
    void Print() const   {  cout<<val<<endl;  }
private:
    int val;
};
```

64

1.3.8 Member Objects(2)

```
//part.cpp
#include <iostream>
#include "part.h"
using namespace std;
Part::Part():val(0)
    {  cout<<"Default Cons. of Part. "<<endl;  }
Part::Part(int i):val(i)
    {  cout<<"Constructor of Part "<<val<<endl;  }
Part::~Part()
    {  cout<<"Destructor of Part "<<val<<endl;  }
```

65

1.3.8 Member Objects(3)

```
//whole.h
#include "part.h"
class Whole {
public:
    Whole();
    Whole(int i,int j,int k);
    ~Whole() {cout<<"Destructor of Whole. "<<endl;}
    void Print() const;
private:
    Part one,two;          //member objects
    int date;
};
```

66

1.3.8 Member Objects(4)

```
//whole.cpp
#include <iostream>
#include "whole.h"
using namespace std;
Whole::Whole():date(0)
   { cout<<"Default cons. of Whole. "<<endl; }
Whole::Whole(int i,int j,int k):two(i),one(j),date(k)
   { cout<<"Constructor of Whole. "<<endl; }
void Whole::Print() const {
   one.Print();          two.Print();
   cout<<date<<endl;
}
```

67

1.3.8 Member Objects(5)

```
//example1.8
#include "whole.h"
int main()
{
   Whole anObject(5,6,10);
   anObject.Print();
}
```

Output:

Constructor of Part 6.
Constructor of Part 5.
Constructor of Whole.
6
5
10
Destructor of Whole.
Destructor of Part 5.
Destructor of Part 6.

68

1.4 Object-oriented Programming(1)

```
//example 1.9
#include <iostream>
using namespace std;
class Animal
{
public:
   //virtual functions
   virtual char *getType() const { return "Animal"; }
   virtual char *getVoice() const { return "Voice"; }
};
class Dog : public Animal        //inheritance
{
```

69

1.4 Object-oriented Programming(2)

```
//example 1.9
public:
   char *getType() const { return "Dog"; }
   char *getVoice() const { return "Woof"; }
};
class Cat : public Animal        //inheritance
{
public:
   char *getType() const { return "Cat"; }
   char *getVoice() const { return "Miaow"; }
};
```

70

1.4 Object-oriented Programming(3)

```
//example 1.9
void type(Animal &a)
{
   cout<<a.getType();
}
void speak(Animal a)
{
   cout<<a.getVoice();
}
int main()
{
   Dog d;
```

71

1.4 Object-oriented Programming(4)

```
//example 1.9
   type(d);
   cout<<" speak ";
   speak(d);
   cout<<endl;
   Cat c;
   type(c);
   cout<<" speak ";
   speak(c);
   cout<<endl;
}
```

Output:

Dog speak Voice
Cat speak Voice

72

1.4 Object-oriented Programming

73

1.4.1 Inheritance

74

1.4.1.1 Base Classes & Derived Classes(1)

```
class Employee {          //…
    string name;
    Date hiring_date;
};
class Manager{            //…
    Employee emp;      //manager's employee record
    short level;
};
class Manager : public Employee {
    short level;
    //…
};
```

75

1.4.1.1 Base Classes & Derived Classes(2)

- Concepts
  - base class
  - derived class
  - inheritance
- the relation between base classes and derived classes
  - Base classes are abstraction of derived classes, and derived classes are concretion of base classes.
  - A derived class is composition of base classes.

76

1.4.1.1 Base Classes & Derived Classes(3)

- A derived class can itself be a base class. (hierarchy)
- Derived classes has an object of the derived class represented as an object of the base class.
- Expression
  - DAG: directed acyclic graph

base class	Employee
↑	↑
derived class	Manager

77

1.4.1.2 Single Inheritance

- A derived class has just one immediately base class.
- Declaration:

class <derived-class-name>
  :<access specifier> <base-class-name>
{
  //derived class's new members
};

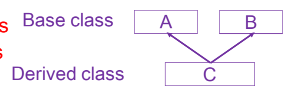
Base class	A
	↑
Derived class	B

78

1.4.1.3 Multiple Inheritance

- A derived class has more than one immediately base class.
- Declaration:

class <derived-class-name>
    :<access specifier> <base-class-name1> ,
    :<access specifier> <base-class-name2>
    //……
{
    //derived class's
    //new members
};



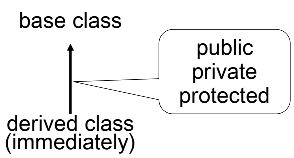
79

1.4.1.4 Access Specifiers(1)

- Access Specifiers
    - public
    - private
    - protected
- The access specifier for a base class controls the access to members of the base class and the conversion of pointers and references from the derived class type to the base class type.

80

1.4.1.4 Access Specifiers(2)



81

1.4.1.4 Access Specifiers(3)

Access ability to members belonging to base class

Inheritance / Member	public	private	protected
private	×	×	×
public	√	√(private)	√(protected)
protected	√	√(private)	√

82

1.4.1.4 Access Specifiers(4)

```
class Location {
public:
    void InitL(int i,int j);
    void Move(int xOff,int yOff);
    int GetX() { return x; }
    int GetY() { return y; }
private:
    int x,y;
};
void Location::InitL(int i,int j){
    x=i;
```

83

1.4.1.4 Access Specifiers(5)

```
    y=j;
}
void Location::Move(int xOff,int yOff){
    x+=xOff;
    y+=yOff;
}
class Rectangle : public Location {
public:
    void InitR(int i,int j,int k,int l);
    int GetH() { return h; }
```

84

1.4.1.4 Access Specifiers(6)

```
    int GetW() { return w; }
private:
    int h,w;
};
void Rectangle::InitR(int i,int j,int k,int l) {
    InitL(i,j);
    h=k;
    w=l;
}
```

85

1.4.1.4 Access Specifiers(7)

```
#include <iostream>
using namespace std;
int main()
{
    Rectangle rect;
    rect.InitR(2,3,10,20);
    rect.Move(3,2);
    cout<<rect.GetX()<<", "<<rect.GetY()<<", "
        <<rect.GetH()<<", <<rect.GetW()<<endl;
}
```

Output:

5,5,10,20

86

1.4.1.4 Access Specifiers(8)

```
//derived class
class V:public Rectangle {
public:
    void Function();
};
void V::Function() {
    Move(3,2);            //correct
}
```

87

1.4.1.4 Access Specifiers(9)

```
class Rectangle : private Location {
public:
    void InitR(int i,int j,int k,int l);
    int GetH() { return h; }
    int GetW() { return w; }
private:
    int h,w;
};
void Rectangle::InitR(int i,int j,int k,int l) {
    InitL(i,j);              //correct
    h=k;
    k=l;
}
```

88

1.4.1.4 Access Specifiers(10)

```
#include <iostream>
using namespace std;
int main()
{
    Rectangle rect;
    rect.InitR(2,3,10,20);
    rect.Move(3,2);
    cout<<rect.GetX()<<", "<<rect.GetY()<<", "
        <<rect.GetH()<<", <<rect.GetW()<<endl;
}
```

89

1.4.1.4 Access Specifiers(11)

```
#include <iostream>
using namespace std;
int main()
{
    Rectangle rect;
    rect.InitR(2,3,10,20);
    rect.Move(3,2);
    cout<<rect.GetX()<<", "<<rect.GetY()<<", "
        <<rect.GetH()<<", <<rect.GetW()<<endl;
}
```

90

1.4.1.4 Access Specifiers(12)

```
class Rectangle : private Location {
public:
    void InitR(int i,int j,int k,int l);
    void Move(int xOff,int yOff)
        { Location::Move(xOff,yOff); }
    void GetX() { Location::GetX(); }
    void GetY() { Location::GetY(); }
    int GetH() { return h; }
    int GetW() { return w; }
private:
    int h,w;
};
```

91

1.4.1.4 Access Specifiers(13)

```
class A {
protected:
    int x;
};
int main() {
    A a;
    a.x=5;
}
```

```
class B:public A {
public:
    void Function();
};
void B::Function()
{
    x=5;
}
```

wrong, protected member cannot be accessed by object a for class A.

right, protected member can be accessed by derived B's member function.

92

1.4.1.5 Constructors&Destructors(1)

- Arguments for base classes are specified in a member initializer list in the definition of the constructor of the derived class.
- The multiple base's constructor are called in the order in which the base class are declared in the class rather than the order in which the base class appear in the initializer list.
- If a base class has constructors, then a constructor must be invoked. Default constructors can be invoked implicitly.

93

1.4.1.5 Constructors&Destructors(2)

```
//example1.10
#include <iostream>
using namespace std;
class M {
public:
    M():m1(0),m2(0) { }
    M(int i,int j);
    ~M();
    void Print() const {  cout<<m1<<", "<<m2<<", "; }
private:
    int m1,m2;
};
```

94

1.4.1.5 Constructors&Destructors(3)

```
//example1.10
M::M(int i,int j):m1(i),m2(j) {
    cout<<"M's constructor called. “
          <<m1<<", "<<m2<<endl;
}
M::~M() {
    cout<<"M's destructor called. "
          <<m1<<", "<<m2<<endl;
}
class N:public M {
public:
    N():n(0) { }
```

95

1.4.1.5 Constructors&Destructors(4)

```
//example1.10
    N(int i,int j,int k);
    ~N();
    void Print() const;
private:
    int n;
};
N::N(int i,int j,int k):M(i,j),n(k) {
    cout<<"N's costructor called. "<<n<<endl;
}
N::~N() {
    cout<<"N's destructor called. "<<n<<endl;
}
```

96

1.4.1.5 Constructors&Destructors(5)

```
//example1.10
void N::Print() const {
    M::Print();
    cout<<n<<endl;
}
int main()
{
    N n1(5,6,7),
       n2(-2,-3,-4);
    n1.Print();
    n2.Print();
}
```

```
Output:

M's constructor called.5,6
N's constructor called.7
M's constructor called.-2,-3
N's constructor called.-4
5,6,7
-2,-3,-4
N's destructor called.-4
M's destructor called.-2,-3
N's destructor called.7
M's destructor called.5,6
```


1.4.1.6 Ambiguity Resolution(1)

- Two base classes may have member functions with the same name (multiple inheritance).

```
class Task {
    //…
    debug_info* get_debug();
};
class Displayed {
    //…
    debug_info* get_debug();
    void draw();
};
class Satellite : public Task, public Displayed {
    //…
    void draw();
};
```


1.4.1.6 Ambiguity Resolution(2)

- disambiguation:
  - explicit (scope resolution operator ::)
  - defining a new function in the derived class

```
void f(Satellite* sp){
    debug_info* dip=sp->get_debug(); //error: //ambiguous
    dip=sp->Task::get_debug();        //ok
    dip=sp->Displayed::get_debug();  //ok
    sp->draw();   //ok: Satellite::draw()
}
```


1.4.1.6 Ambiguity Resolution(3)

- If a derived class has two base classes, and these two bases have a common base class, there may be ambiguous. (a class act as a base twice)

```
struct Link {   Link* next;   };
class Task : public Link {
    //…
};
class Displayed : public Link {
    //…
};
class Satellite : public Task, public Displayed {
    //…
};
```

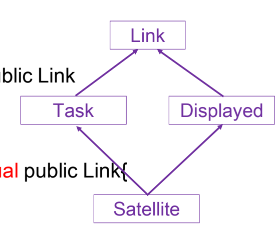

1.4.1.6 Ambiguity Resolution(4)

- Disambiguation
  - explicit
  - defining a new function in the derived class
  - virtual base class

```
void mess_with_links(Satellite* p){
    p->next=0;           //error:ambiguous(which Link?)
    p->Link::next=0;    //error:ambiguous(which Link?)
    p->Task::next=0;           //ok
    p->Displayed::next=0;     //ok
}
```


1.4.1.6 Ambiguity Resolution(5)

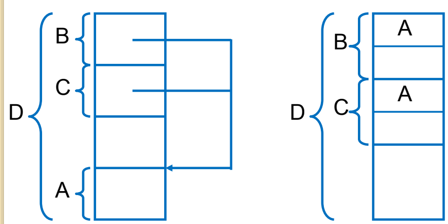
```
struct Link {
    Link* next;
};
class Task : virtual public Link
{
    //…
};
class Displayed : virtual public Link{
    //…
};
class Satellite : public Task, public Displayed {
    //…
};
```



1.4.1.6 Ambiguity Resolution(6)

The layout of virtual and non-virtual base class

virtual base class non-virtual base class



1.4.2 Subtypes of Base Classes(1)

- In general, if a derived class has a public base class , then an object of a derived class can be treated as an object of its base class, especially when it manipulated through pointers and references.
- The opposite is not true.

1.4.2 Subtypes of Base Classes(2)

- The assignment relation between derived classes and base classes:

```
derived d;
base b;

b=d;            //ok
base &ref=d;    //ok
base *pb=&d;    //ok
```


1.4.2 Subtypes of Base Classes(3)

```
//example1.11
#include <iostream>
using namespace std;
class A {
public:
    A():x(0) { }
    A(int i):x(i) { }
    void Print() const { cout<<x<<endl; }
    int GetA() {return x;}
private:
    int x;
};
```


1.4.2 Subtypes of Base Classes(4)

```
//example1.11
#include <iostream>
using namespace std;
class B : class A {
public:
    B():y(0) { }
    B(int i,int j):A(i),y(j) {}
    void Print() const { A::Print(); cout<<y<<endl; }
private:
    int y;
};
```


1.4.2 Subtypes of Base Classes(5)

```
//example1.11
void fun(A& d)  { cout<<d.GetX()*10<<endl; }
int main(){
    B bb(9,5);
    A aa(5);
    aa=bb;
    aa.Print();
    A *pa=new A(8);
    B *pb=new B(1,2);
    pa=pb;
    pa->Print();
    fun(bb);
}
```

Output:
9
1
90

An object of a derived class can use only the members of its base class.

1.4.3 Virtual Functions(1)

```
//example1.12
#include <iostream>
using namespace std;
class Point{
public:
   Point(double i,double j):x(i),y(j) { }
   virtual double Area() const { return 0; }
private:
   double x,y;
};
class Rectangle:public Point {
public:
   Rectangle(int i,int j,int k,int l);
   virtual double Area() const { return w*h; }
```

109

1.4.3 Virtual Functions(2)

```
//example1.12
private:
   double w,h;
};
Rectangle::Rectangle(int i,int j,int k,int l)
      :Point(i,j),w(k),h(l)   { }
void fun(Point& s) {
   cout<<s.Area()<<endl;
}
int main(){
   Rectangle rect(3.0,5.2,15.0,25.0);
   fun(rect);
}
```

Output:

375

Calling Rectangle::Area()

110

1.4.3 Virtual Functions(3)

```
//example1.13
#include <iostream>
using namespace std;
class Point{
public:
   Point(double i,double j):x(i),y(j) { }
   double Area() const { return 0; }
private:
   double x,y;
};
class Rectangle:public Point {
public:
   Rectangle(int i,int j,int k,int l);
   double Area() const { return w*h; }
```

111

1.4.3 Virtual Functions(4)

```
//example1.13
private:
   double w,h;
};
Rectangle::Rectangle(int i,int j,int k,int l)
      :Point(i,j),w(k),h(l)   { }
void fun(Point& s) {
   cout<<s.Area()<<endl;
}
int main(){
   Rectangle rect(3.0,5.2,15.0,25.0);
   fun(rect);
}
```

Output:

0

Calling Point::Area()

112

1.4.3 Virtual Functions(5)

- Virtual functions allow the programmer to declare functions in a base class that can be redefined in each derived class.
- The compiler and loader will guarantee the correct correspondence between objects and the functions applied to them.
- Virtual functions are non-static member functions.
- Declaration

  virtual <type> <function-name>(<arguments>);
  - Keyword virtual can be used only in declaration.

113

1.4.3 Virtual Functions(6)

- Run-time Polymorphism
  - the derived class has the public base class
  - the member functions called must be virtual
  - objects must be manipulated through pointers or references.
    - When manipulating an object directly, rather than through a pointer or reference, its exact type is known by the compiler so that run-time polymorphism is not needed.

114

1.4.3 Virtual Functions(7)

```
//example1.14
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout<<1; }
    virtual void Print() { cout<<'B'; }
};
class Derived : public Base
{
public:
    Derived() { cout<<2; }
```

115

1.4.3 Virtual Functions(8)

```
//example1.14
    virtual void Print() { cout<<'D'; }
}
int main()
{
    Base *ptr=new Derived;
    ptr->Print();
    delete ptr;
}
```

Output:

12D

116

1.4.4 Abstract Classes

1.4.4.1 Pure Virtual Functions

1.4.4.2 Abstract Classes

117

1.4.4.1 Pure Virtual Functions(1)

- If it is not possible to provide sensible definitions for its virtual functions in the base, the virtual functions can be pure virtual functions.
- A virtual function is "made pure" by the initializer "=0".

virtual <type> <func-name>(<arguments>)=0;

  - virtual void draw()=0;
  - Differs from virtual void draw() { }
- The pure virtual functions provide a consistent interface for its class hierarchies.

118

1.4.4.1 Pure Virtual Functions(2)

```
//example1.15
#include <iostream>
using namespace std;
class Number{
public:
    Number(int i):val(i) { }
    virtual void show()=0;
protected:
    int val;
};
class Hextype:public Number{
public:
```

119

1.4.4.1 Pure Virtual Functions(3)

```
//example1.15
    Hextype(int i):Number(i) { }
    virtual void show()
        { cout<<hex<<val<<dec<<endl;}
};
class Dectype:public Number{
public:
    Dectype(int i):Number(i) { }
    virtual void show()
        { cout<<dec<<val<<endl; }
};
void fun(Number& n){
```

120

1.4.4.1 Pure Virtual Functions(4)

```
//example1.15
    n.show();
}
int main()
{
    Dectype d(50);
    fun(d);
    Hextype h(16);
    fun(h);
}
```

Output:

50
10

121

1.4.4.2 Abstract Classes(1)

- A class with one or more pure virtual functions is an abstract class.
- A pure virtual function that is not defined in a derived class remains a pure virtual function, so the derived class is also an abstract class.
- No objects of an abstract class can be created, but objects that point or refer to an abstract class can be defined.
- An abstract class can be used only as an interface without exposing any implementation details and as a base for other classes.

122

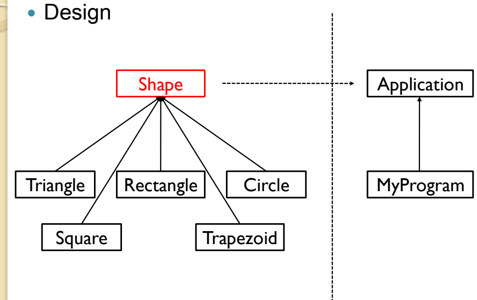
1.4.4.2 Abstract Classes(2)

- Problem
  - Five simple shapes are as following:
    - A triangle
    - A rectangle
    - A circle
    - A trapezoid
    - A square
  - Please sum the area of all above shapes.

123

1.4.4.2 Abstract Classes(3)

- Design



124

1.4.4.2 Abstract Classes(4)

```
//shape.h
#if !defined(_SHAPE_H)
#define _SHAPE_H
class Shape{
public:
    virtual double Area() const=0;
};
#endif
```

125

1.4.4.2 Abstract Classes(5)

```
//application.h
#if !defined(_PROGRAM_H)
#define _PROGRAM_H
#include "shape.h"
class Application {
public:
    double Compute(Shape *s[],int n) const;
};
#endif
```

126

1.4.4.2 Abstract Classes(6)

```
//application.cpp
#include “application.h"
double Application::Compute(Shape *s[],
                            int n) const
{
    double sum=0;
    for(int i=0;i<n;i++)
        sum+=s[i]->Area();
    return sum;
}
```

127

1.4.4.2 Abstract Classes(7)

```
//polygon.h
#include <iostream>
#include "shape.h"
using namespace std;
class Triangle:public Shape{
public:
    Triangle(double i,double j):high(i), width(j)  { }
    double Area() const { return high*width*0.5; }
private:
    double high,width;
};
```

128

1.4.4.2 Abstract Classes(8)

```
//polygon.h
class Rectangle:public Shape{
public:
    Rectangle(double i,double j):high(i),width(j)  { }
    double Area() const { return high*width; }
private:
    double high,width;
};
class Circle:public Shape{
public:
    Circle(double i):radius(i) { }
```

129

1.4.4.2 Abstract Classes(9)

```
//polygon.h
    double Area() const { return radius*radius*3.14; }
private:
    double radius;
};
class Trapezoid:public Shape{
public:
    Trapezoid(double i,double j,double k)
        :top(i),bottom(j),high(k) { }
    double Area() const {return (top+bottom)*high/2;}
private:
```

130

1.4.4.2 Abstract Classes(10)

```
//polygon.h
private:
    double top,bottom,high;
};
class Square:public Shape{
public:
    Square(double):side(i) { }
    double Area() const { return side*side; }
private:
    double side;
};
```

131

1.4.4.2 Abstract Classes(11)

```
//example1.16
#include <iostream>
#include "polygon.h"
#include “application.h“
using namespace std;
class MyProgram:public Application{
public:
    MyProgram();
    ~MyProgram();
    int Run();
private:
```

132

1.4.4.2 Abstract Classes(12)

```
//example1.16
    Shape **s;
};
MyProgram::MyProgram(){
    s=new Shape* [5];
    s[0]=new Triangle(3.0,4.0);
    s[1]=new Rectangle(6.0,8.0);
    s[2]=new Circle(6.5);
    s[3]=new Trapezoid(10.0,8.0,5.0);
    s[4]=new Square(6.7);
}
```

133

1.4.4.2 Abstract Classes(13)

```
//example1.16
MyProgram::~MyProgram(){
    for(int i=0;i<5;i++)
        delete s[i];
    delete[] s;
}
int MyProgram::Run() {
    return Compute(s,4);
}
int main()
 {cout<<"the sum of Area: "<<MyProgram().Run();}
```

Output:

the sum of Area: 276.618

134

1.5 Generic Programming(1)

```
//example 1.17
#include <iostream>
using namespace std;
const int size=10;
template <class Type>              //class template
class Stack
{
  public:
      void Init() { pos=0; }
      void Push(Type ch);
      Type Pop();
      int GetPos() { return pos; }
```

135

1.5 Generic Programming(2)

```
//example 1.17
  private:
      Type obj[size];
      int pos;
};
template <class Type>  //function template
void Stack<Type>::Push(Type ch)
{
  if(pos==size)
  {
      cout<<"Stack is full."<<endl;
      return;
  }
```

136

1.5 Generic Programming(3)

```
//example 1.17
  obj[pos++]=ch;
}
template <class Type>  //member function template
Type Stack<Type>::Pop()
{
  if(pos==0)
  {
      cout<<"Stack is empty."<<endl;
      return -1;
  }
  return obj[--pos];
}
```

137

1.5 Generic Programming(4)

```
//example 1.17
int main()
{
  Stack<char> s1;              //template class
  s1.Init();
  s1.Push('a');
  s1.Push('b');
  s1.Push('c');
  cout<<"Pop s1: ";
  int count=s1.GetPos();
  for(int i=0;i<count;i++)
      cout<<s1.Pop()<<'\t';
  cout<<endl;
```

138

1.5 Generic Programming(5)

```
//example 1.17
  Stack<int> s2;                //template class
  s2.Init();
  s2.Push(1);
  s2.Push(2);
  s2.Push(3);
  cout<<"Pop s2: ";
  count=s2.GetPos();
  for(i=0;i<count;i++)
       cout<<s2.Pop()<<'\t';
  cout<<endl;
}
```

Output:

Pop s1: c b a
Pop s2: 3 2 1

139

1.5 Generic Programming(6)

140

1.5.1 Function Templates(1)

- Declaration

```
template <type arguments>
<type> <function-name>(<formal arguments>)
{
   //the body of function
}

template <class T>
T max(T x, T y)
{
   return (x>y)?x:y;
}
```

141

1.5.1 Function Templates(2)

- Type arguments
  - Format
    template <class T1, class T2, …>
  - T1,T2,… is type arguments. They are not any kinds of data types, but similar to any data types.
  - It must be instantiated (replaced) by a data type when function templates are invoked.
- Formal arguments
  - All type arguments must be used in the formal arguments list.

142

1.5.1 Function Templates(3)

```
//example1.18
#include <iostream>
using namespace std;
template <class T>              //function template
T max(T x,T y)
{
   return (x>y) ? x : y;
}
int main()
{
   int x1(1),y1(2);
```

143

1.5.1 Function Templates(4)

```
//example1.18
   double x2(3.4),y2(5.6);
   char x3='a',y3='b';
   //T is instantiated by int
   cout<<max(x1,y1)<<endl;
   //T is instantiated by double
   cout<<max(x2,y2)<<endl;
   //T is instantiated by char
   cout<<max(x3,y3)<<endl;
}
```

Output:

2
5.6
b

144

1.5.2 Class Templates(1)

- Declaration

  template <type arguments>
  class <class-name>
  {
  //the body of class
  }
- If member functions are defined outside the class body in a class template, it must be defined as the function template.

1.5.2 Class Templates(2)

```
//test.h
#include <iostream>
using namespace std;
template <class T>
class Test
{
  public:
    Test():b(0) {}
    Test(T x,int y);
    int Getb() { return b; }
    void Print() const { cout<<a<<','<<b<<endl; }
```


1.5.2 Class Templates(3)

```
//test.h
  private:
    T a;
    int b;
};
//the constructor must be defined as function
//member
template <class T>
Test<T>::Test(T x,int y):a(x),b(y)
{
}
```


1.5.2 Class Templates(4)

- The name of a class template followed by a type bracketed by <> is the name of a class (as defined by the template) and can be used exactly like other class names.

1.5.2 Class Templates(5)

```
//example1.19
#include <iostream>
#include "test.h"
using namespace std;
int main(){
  Test<int> obj1(10,1);
  Test<char> obj2('A',2);
  obj1.Print();
  obj2.Print();
}
```

Output:

10,1
A,2

1.5.3 Standard Template Library(1)

- Components
  - Containers
  - Algorithms
  - Iterators
  - Functors
  - Adpators
  - Allocators
- Containers
  - Sequence Containers
    - vector, list, deque

1.5.3 Standard Template Library(2)

- Associative containers
  - sset, multiset, map, multimap
- Container Adapters
  - queue, stack, priority queue
- Algorithms
  - A large number of algorithms to perform activities such as searching and sorting are provided in the STL.
- Iterators
  - Types: input, output, forward, bidirectional, random access

1.5.3 Standard Template Library(3)

- Functors
  - The STL includes classes that overload the function call operator (operator()). Instances of such classes are called functors or function objects

1.5.3 Standard Template Library(4)

```
//example1.20
#include <iostream>
#include <vector>        //
using namespace std;
int main (){
    vector<int> v(100);      // 100 is vector's size
    for (int i = 0; i < 100; ++i)
        v[i] = i;
    for (vector<int>::iterator p = v.begin();
          p != v.end(); ++p)
        cout << *p << '\t';
   cout << endl;
}
```

Output:

```
0     1     …    9
   ……
90    91    …    99
```


1.5.3 Standard Template Library(5)

```
//example1.21
#include <iostream>
#include <list>                  // list container
#include <numeric>               // for accumulate
using namespace std;
void print(list<double> &lst){
    list<double>::iterator p;        // traverse iterator
    for (p = lst.begin(); p != lst.end(); ++p)
        cout << *p << '\t';
    cout << endl;
}
int main(){
```


1.5.3 Standard Template Library(6)

```
//example1.22
    double w[4] = { 0.9, 0.8, 88, -99.99 };
    list<double> z;
    for (int i = 0; i < 4; ++i)
        z.push_front(w[i]);
    print(z);
    z.sort();
    print(z);
    cout << "sum is "
          << accumulate(z.begin(), z.end(), 0.0)
          << endl;            //algorithm
}
```

Output:

```
-99.99    88     0.8    0.9
-99.99    0.8    0.9    88
sum is -10.29
```