C++ review

1. Class（类）
2. Template （模板）

1

Class（类）

```
class rectangle
{
 private:
   int w;
   int h;

 public:
  rectangle(int a,int b){ w=a; h=b;}
  void setWidth(int width) { w=width;}
  void setHeight(int Height) { h=Height;}
  int getWidth() { return w;}
  int getHeight()  {return h;}
  double  Area()  { return  w*h; }
  double  Perimeter()  { return  2*(w+h); }
};
```

```
class square
{
 private:
   int w;

 public:
   square(int a){ w=a; }
   void setWidth(int width) { w=width;}
   int getWidth( ) { return w;}
   double Area ()  { return  w*w; }
   double Perimeter ()  { return  4*w; }
};
```

2

Class（类）续

```
class triangle   {
private:
  int w1;
  int w2;
  int h;
public:
  triangle(int a,int b,int c){ w1=a; w2=b; h=c;}
  void setWidth1(int width1) { w1=width1;}
  void setWidth2(int width2) { w2=width2;}
  void setHeight(int Height) { h=Height;}
  int getWidth1() { return w1;}
  int getWidth2() { return w2;}
  int getHeight()  {return h;}
  double Perimeter ()  {
    return  sqrt((double)(h*h+w1*w1))+sqrt((double)(w2*w2+h*h)) + w1+ w2 ;   }
  double  area()  {return  (w2+w1)*h/2; }
};
```

3

example

```
#include <iostream>
…
#include  <shape.hpp>
void main()
{
   int a=3;
   int  b=4;
   double  c,  d;
   rectangle  r1(a, b);
   triangle   t1(5, 5, 8);
   square  s1(6);
   c = r1.Area();
   d = t1.Perimeter ;
   printf(" %f,%f \n", c,d);
   cout<<r1.Area()<<t1.Area()<<s1.Area()<<endl;
   cout<<r1.Perimeter()<<t1.Perimeter()<<s1.Perimeter()<<endl;
}
```

```
void main()
{
   rectangle  r1( 3.3, 4.8 );
   cout<<r1.Area()<<endl
   cout<<r1.Perimeter() <<endl;
}
```

4

2 模板---引子

```
int max ( int a , int b )
{
        return ( a > b ) ? a :  b ;
}

float max1 ( float a , float b )
{
        return ( a > b ) ? a :  b ;
}

… …
```

```
void main()
{
      int   x=6,y=9,z;
      float  x1=3.2, y1=10.7, z1;

      z = max(x,y);
      z1=max1(x1,y1);
}
```

5

模板—引子

1.假如设计一个求两参数最大值的函数，在实践中我们可能需要定义五个函数来保证主调函数中实参类型的任意性：

```
int max_i ( int a , int b ) { return ( a > b ) ? a , b ; }
long max_l ( long a , long b ) { return ( a > b ) ? a , b ;}
float max_f ( float a , float  b ) { return ( a >b)? a , b ; }
double max_d ( double a , double  b ) { return ( a >b)? a , b ; }
char max_c ( char a , char b ) { return ( a > b ) ? a , b ;}
```

2.这些函数几乎相同，唯一的区别就是形参及返回值的类型不同

3.需要事先知道有哪些类型会使用这些函数，对于未知类型这些函数不起作用（不匹配）

模板 template

6

Example

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

void main()
{
      int    x=6,y=9,z;
      float  x1=3.2, y1=10.7, z1;
      char   c1='d',  c2='s', c3;

      z = max(x,y);
      z1=max(x1,y1);
      c3=max(c1,c2);

}
```

```
float max ( float a , float b )
{
        return ( a > b ) ? a :  b ;
}
```

模板 template

7

模板的概念

1. 所谓模板是一种使用无类型参数来产生一系列函数或类的机制。
2. 若一个程序的功能不是对某种特定的数据类型进行处理，则可以将所处理数据的数据类型说明为参数，以便在各种数据类型的情况下使用，这就是模板的由来。
3. 模板是以一种完全通用的方法来设计函数或类，不必预先说明将被使用的参数/对象的数据类型。
4. 通过模板可以产生函数或类的集合，使它们操作不同的数据类型，从而避免需要为每一种数据类型编写一个单独的函数或类。

8

模板分类

- 函数模板(function template)
  - 是独立于类型的函数
  - 调用时可产生针对特定类型的特定版本函数
- 类模板(class template)
  - 跟类相关的模板
  - 可产生针对特定类型的特定版本类

9

9

2.1 函数模板的形式

Ex: class T1, class T2

```
template < 模板形参表>
返回值类型  函数名（模板函数形参表）
{
    //函数定义体
}
```

可用模板形参表中的形参定义变量

Example:

```
template < class T >
   T max(T a , T b){
         T  c;
         c = ( a > b ) ? a :  b;
         return c;
   }
```

10

10

函数模板工作方式

- 函数模板只是说明，不能直接执行，需要实例化为模板函数(即调用）后才能执行
- 在说明了一个函数模板后，当编译系统发现有一个对应的函数调用时，将根据实参中的类型来确认是否匹配函数模板中对应的形参，然后生成一个重载函数。该重载函数的定义体与函数模板的函数定义体相同，称之为模板函数

11

11

Example

函数模板

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

void main()
{
      int    x=6,y=9,z;
      float  x1=3.2, y1=10.7, z1;
      char   c1='d', c2='s', c3;

      z = max(x,y);
      z1=max(x1,y1);
      c3=max(c1,c2);

}
```

重载的模板函数

```
int max ( int a , int b )
{
      return ( a > b ) ? a :  b ;
}

float max ( float a , float b )
{
      return ( a > b ) ? a:  b ;
}

char max ( char a , char b )
{
      return ( a > b ) ? a :  b ;
}
```

12

函数模板优缺点

- 函数模板方法克服了C语言解决上述问题时用大量不同函数名表示相似功能的坏习惯
- 克服了宏定义不能进行参数类型检查的弊端
- 克服了C++函数重载用相同函数名字重写几个函数的繁琐
- 缺点，调试比较困难
  - 一般先写一个特殊版本的函数
  - 运行正确后，改成模板函数

13

13

2.2 类模板

```
class rectangle
{
 private:
   int w;
   int h;

 public:
   rectangle(int a,int b){ w=a; h=b;}
   void setWidth(int width) { w=width;}
   void setHeight(int Height) { h=Height;}
   int getWidth() { return w;}
   int getHeight()  {return h;}
   double  Area()  { return  w*h; }
   double  Perimeter()  { return  2*(w+h); }
};
```

```
void main()
{
  rectangle  r1( 3.3, 4.8 );
  cout<<r1.Area()<<endl
  cout<<r1.Perimeter() <<endl;
}
```

```
template < class T> class rectangle
{
 private:
   T  w;
   T  h;

 public:
   rectangle(T a, T b){ w=a; h=b;}
   void setWidth(T width) { w=width;}
   void setHeight(T Height) { h=Height;}
   T getWidth() { return w;}
   T getHeight()  {return h;}
   double  Area()  { return  w*h; }
   double  Perimeter()  { return  2*(w+h); }
};
```

```
void main()
{
  rectangle<float>  r1( 3.3, 4.8 );
  cout<<r1.Area()<<endl
  cout<<r1.Perimeter() <<endl;
}
```

14

类模板

- 定义语法：

Ex: class T1, class T2

template <模板参数表> class 类模板名

类体

类体中可以出现的数据类型有

a. 基本数据类型，如 int, char, float, double

b. 用户自定义类型，如 结构体名, 类名

c. 模板参数表中说明的类型参数

```
template < class T> class rectangle
{
 private:
   T  w;
   T  h;

 public:
   rectangle(T a, T b){ w=a; h=b;}
   void setWidth(T width) { w=width;}
   void setHeight(T Height) { h=Height;}
   T getWidth() { return w;}
   T getHeight()  {return h;}
   double  Area()  { return  w*h; }
   double  Perimeter()  { return  2*(w+h); }
};
```

15

15

类模板

- 2. 类模板实例化

(1) 从类模板生成具体类的过程

(2) 时机: 定义对象时; 指针或者引用解引用(*p)时;

(3) 语法:

类模板名<实参表>

```
void main()
{
  rectangle<float>  r1( 3.3, 4.8 );
  rectangle<int>  r2( 4, 6 );
  cout<<r1.Area()<<r2.Area() <<endl
  cout<<r1.Perimeter() <<endl;
  cout<<r2.Perimeter() <<endl;
}
```

16

Example

```
template < class T>
class rectangle
{
 private:
   T  w;
   T  h;

 public:
  rectangle(T a, T b){ w=a; h=b;}
  void setWidth(T width) { w=width;}
  void setHeight(T Height) { h=Height;}
  T getWidth() { return w;}
  T getHeight()  {return h;}
  double  Area()  { return  w*h; }
  double  Perimeter()  { return  2*(w+h); }
};
```

```
void main()
{
  rectangle<float>  r1( 3.3, 4.8 );
  cout<<r1.Area()<<endl
  cout<<r1.Perimeter() <<endl;

  rectangle<int>  r2( 3, 4 );
  cout<<r2.Area()<<endl
  cout<<r2.Perimeter() <<endl;

}
```

17

Review after Class

- C++的函数特征
- C++的数据声明
- C++的作用域
- C++的类
- C++的对象
- C++的输入/输出
- C++的函数
- C++的参数传递
- C++的函数名重载
- C++的操作符重载
- C++的动态存储分配(new)
- 虚(virtual)函数
- 模板(template)

18

18

课后练习

1） 改写下列程序，要求将求最小值的函数设计成函数模板，以保证主程序的正确调用。

```
int min(int a[], int n)
{
    int i;
    int minv=a[0];
    for( i = 1;i < n ; i++)
    {
        if(minv>a[i])
            minv=a[i];
    }
    return minv;
}
```

```
#include <iostream>
void main()
{ int a[]={1,3,0,2,7,6,4,5,2};
  double b[]={1.2,-3.4,6.8,9,8};
  cout<<"a数组的最小值为："
      <<min(a,9)<< endl;
  cout<<"b数组的最小值为："
      <<min(b,4)<<endl;  }
```

19

课后练习

2） 增改类square的定义及main程序，实现边长分别为int，float，及double类型的正方形s1,s2,s3的面积&周长计算。

```
class square
{
 private:
  int w;

 public:
    square(int a){ w=a; }
    void setWidth(int width) { w=width;}
    int getWidth( ) { return w;}
    double Area ()  { return  w*w; }
    double Perimeter ()  { return  4*w; }
};
```

```
void main()
{
  float a=3.2;
  double   b=5.789;
  square s1(6);
  square s2(a);
  square s3(b);
  cout<<s1.Area() << endl;
  cout<<s1.Perimeter()<<endl;
  cout<<s2.Area() << endl;
  cout<<s2.Perimeter()<<endl;
  cout<<s3.Area() << endl;
  cout<<s3.Perimeter()<<endl;
}
```

20

Chapter 4

Lists, Stacks and Queue

07:57 2

2

topic

4.1 List（线性表）
4.2 Stack（栈）
4.3 Queue（队列）

07:57 3

3

4.1 List

4.1.1 List ADT
4.1.2 Array-based list(顺序表)
4.1.3 Linked list (链表)
4.1.4 Freelists
4.1.5 Double linked list(双链表)

07:57 4

4

4.1.1 list(线性表)的定义与特点

- 定义
  - n（≥0）个元素的有限序列，记作（$a_0, a_1, a_2, \ldots, a_{n-1}$）
  - a_i 是表中第i个元素，n 是表长度，n=0时称为空表
- 线性表的特点
  - 除第一个元素外，其他每一个元素有一个且仅有一个直接前驱。
  - 除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。

07:57 5

5

4.1.1 list(线性表) 的ADT

- Data： n（≥0）个元素
- Relation： 序列，线性 1-1
- Operation：
  - 插入，删除， 查找等

07:57 6

6

4.1.1 list(线性表) 的ADT

- Data： n（≥0）个元素
- Relation： 序列，1-1
- Operation：
  - 在当前位置 插入insert ，删除remove
  - 改变或获取当前位置---prev， next， MovetoPos, currPos， movetoStart， movetoEnd
  - 获得当前位置的元素值--getValue
  - 获得线性表长度---length

07:57 8

8

Several Concepts(逻辑上) for List Implementation(实现) in this book

Our list implementation will support the concept of a current position.

We will do this by defining the list in terms of left and right partitions.

- Either or both partitions may be empty.

Partitions are separated by a fence(分隔线).

<20, 23 | 12, 15>

07:57 7

7

List ADT class

```
template <class Elem> class List {
Private:
  void operator =(const list&)  {}
  List (const List&) {}
public:
  list() {}
  virual ~list()  {}
  virtual void clear() = 0;
  virtual void insert(const Elem&) = 0;
  virtual void append(const Elem&) = 0;
  virtual Elem remove() = 0;
```

07:57 9

9

List ADT class(cont)

```
  virtual Elem getValue() const = 0;
  virtual void moveToStart() = 0;
  virtual void moveToEnd() = 0;
  virtual void prev() = 0;
  virtual void next() = 0;
  virtual void moveToPos(int pos) = 0;
  virtual int currPos() const = 0;
  virtual int length() const = 0;
};
```

07:57 10

10

List 声明及其公用函数调用样式

List <int> List1; List1: <1,5,9,18 | 32, 100，15>

List1.insert(12)； 将实参插入当前元素之前
List1.prev( ); 将当前位置向前移一个
List1.remove( )； 删除当前位置处的元素
List1.append(20)；将实参插入线性表的末尾
List1.moveToPos(6); 将当前位置 移到实参指定的位置
x =List1.getvalue(); 将当前位置元素的值赋给x
List1.moveToStart(); 将当前位置移到表头
List1.next(); 将当前位置向后移一个
List1.moveToEnd(); 将当前位置移到表尾
i= List1.currPos(); 获取当前位置 元素在表中的索引
l= List1.length(); 获取线性表的长度

07:57 11

11

List Examples

List <int> MyList;
MyList: <12 | 32, 15>

```
MyList.insert(99);
```

Result: <12 | 99, 32, 15>

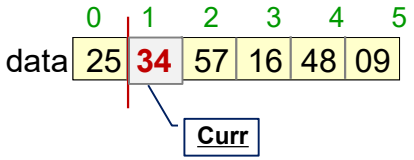
```
MyList.moveToStart();   Result:

MyList.getValue(it);    it:
```

07:57 12

12

4.1.2 Array-based/Sequential List (顺序表)

- 定义 将线性表中的元素相继存放在一个连续的存储空间中。
- 可用一维数组描述此存储结构
- 特点 在物理空间顺序存储

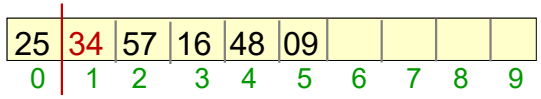


07:57 13

13

4.1.2 array-based/Sequential List (顺序表)

- 一般用1个数组，3个整型变量可描述顺序表
  - ✓ 1个数组listArray存放各个元素值
  - ✓ 1个整型变量maxSize 存放数组的大小
  - ✓ 1个整型变量listSize存放顺序表的实际长度，listSize <= maxsize
  - ✓ 1个整数curr存放当前元素的下标



maxSize=10 listSize=6, curr=1

14

14

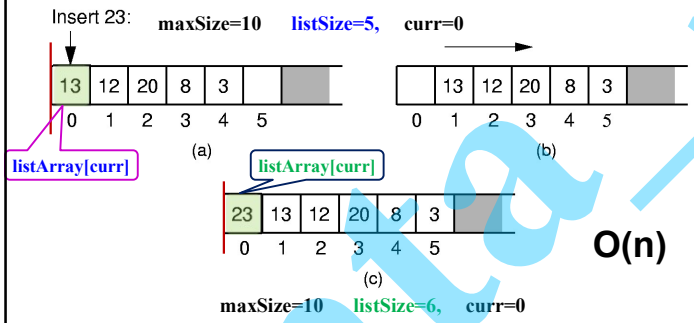
4.1.2 array-based/Sequential List (顺序表)

- 顺序表各种操作的实现及其时间复杂度：
  - 插入insert，删除remove
    - O(n) listArray，listSize的值会改变
  - 改变当前位置---prev， next， MovetoPos, movetoStart， movetoEnd
    - O(1) 根据要求改变变量curr的值
  - 获得当前位置元素值或其索引—getValue，currPos
    - O(1) ， 返回 listArray[curr], curr的值
  - 获得线性表长度---length
    - O(1) ， return 变量listSize的值

15

15

Array-Based List Insert



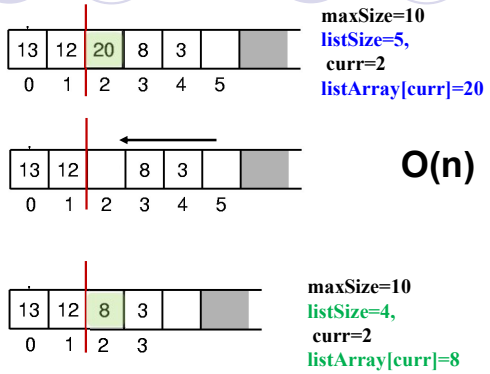
O(n)

顺序表有表满情况：maxSize==listSize，不能再插入

16

16

Array-Based List Remove



O(n)

顺序表有表空情况：listSize==0，不能再删除

17

17

Array-Based List Class (1)

```
template <class Elem> // Array-based list
class AList  {
private:
  int maxSize;
  int listSize;
  int curr;
  Elem* listArray;
public:
  AList(int size) {
    maxSize = size;
    listSize = curr = 0;
    listArray = new Elem[maxSize];
  }
  ~AList() { delete [] listArray; }
```

构造函数

18

07:57

18

Array-Based List Class (2)

```
void clear() {
  delete [] listArray;
  listSize = curr = 0;
  listArray = new Elem[maxSize];
}
void moveToStart() {curr = 0; }
void moveToEnd() {curr = listSize; }
void prev()   {if(curr != 0) curr --;}
void next()   { if (curr < listSize)
                  curr ++; }
void moveToPos(int pos) {
  Assert ((pos >= 0) && (pos <= listSize),
  "Pos out of range");
  curr = pos;
}
```

O(1)

19

07:57

19

Array-Based List Class (3)

```
Elem  getValue() const {

  Assert ((curr>= 0)&&curr<listSize), "No
  current element");

  return listArray[curr];
}
int currPos() const  { return curr; }

int Length() const  { return listSize; }

};
```

O(1)

20

07:57

20

Insert

```
// Insert at front of Current position item
template <class Elem>
void AList<Elem>::insert(const Elem& item)
{
  Assert( listSize < maxSize, "List capacity
  exceeded");
  for(int i=listSize; i> curr; i--)
    // Shift Elems up to make room
    listArray[i] = listArray[i-1];
  listArray[curr] = item;
  listSize++; // Increment list size
}
```

O(n)

21

07:57

21

Append

```
// Append Elem to end of the list
template <class Elem>
void AList<Elem>::append(const Elem& item)
{
  Assert( listSize < maxSize, "List capacity
  exceeded");
  listArray[listSize++] = item;
}
```

22

07:57

22

22

Remove

```
// Remove and return first Elem in right
// partition
template <class Elem>
Elem AList<Elem>::remove( ) {
  Assert( (curr>=0)&& (curr<listSize, "No
  element");
  Elem it = listArray[curr]; // Copy Elem
  for(int i=curr; i<listSize-1; i++)
    // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;    // Decrement size
  return it;
}
```

O(n)

23

07:57

23

23

Array based List 应用举例

```
#include "AList.h"
……
void main() {
int a,b;
    AList <int> myList(100);   //using the array-based list
    cout << myList.currPos() <<endl ;
    myList.insert(12);
    myList.insert(20);
    myList.insert(31);
    a = myList.getValue( );  cout << a <<endl;
    myList.next();  cout << myList.currPos() <<endl ;
    b =myList.getValue( ); cout << b <<endl ;
}
```

24

07:57

24

24

AList.h (课件p18-23)

```
template <class Elem>  // Array-based list
class AList {
private:
  int maxSize;      // Maximum size of list
  int listSize;     // Actual elem count
  int curr;        // Position of fence
  Elem* listArray; // Array holding list
public:
    AList(int size) {
    maxSize = size;
    listSize = curr = 0;
    listArray = new Elem[maxSize];
  }
  ……
};
……
```

25

07:57

25

25

本章作业一

4.2

4.4

课堂小测验

1. AList <**int**> MyList; 假定当前MyList为:
<12 | 32, 26，78，30, 15>

写出执行下列要求的调用语句（按顺序）
1）在元素78和30之间插入元素20；
2）删除线性表中第一个元素12。

2. 试为Alist 编写一个查找函数

```
int find(const Elem& item )
```