面向对象程序设计(OOP)

(C++)

福州大学·软件学院·计算机教研室
王灿辉(wangcanhui@fzu.edu.cn)

第4章：模板(template)

- 模板可以帮助实现编程中最难达到的一个目标：创建可复用的代码。
- 使用模板可以创建**通用的函数和类**，在通用的函数或类中，函数和类要操作的**数据的类型被指定为参数**。这样就可以将一个函数或类与多个不同的数据类型一起使用，而不必明确地为各个数据类型重新编码。

函数模板(通用函数)

```
template <class Type>
void myfunc(Type x)
{
    //body of function
}
```

第4章：模板(template)

通用函数（函数模板）

- 例如下述两个函数：

```
int abs(int x)
  {return x>0?x:-x}
double abs(double x)
  {return x>0?x:-x}
```


- 功能完全相同，只是参数类型不同。在C中要编写不同名的2个函数，C++允许重载。

第4章：模板(template)

通用函数（函数模板）

```
template <typename Type>//声明函数模板
/*可改为：template <class Type>结果完全相
  同，Type与形参一样可以为任意标识符*/
Type myabs(Type x) {
  return x>0?x:-x;
}
```

- 用模板实现通用函数的简单实例

第4章：模板(template)

通用函数（函数模板）

- **函数模板**与重载密切相关，从函数模板产生的相关函数都是同名的，编译程序用重载的解决方法调用相应的**模板函数**。
- 类型参数T可以是内部类型（int等）或自定义类型，它可以用来**指定函数的形参类型**和（或）返回值类型，以及声明函数中的局部变量。函数体定义与普通函数一样。

第4章：模板(template)

通用函数（函数模板）

- 函数模板参数表中指明的**类型参数必须用于函数参数表**，例如：

```
template <class T> T f1() {……} //错误
template <class T> void f2() { //错误
    T a;
    ……
}
```

第4章：模板(template)

通用函数（函数模板）的调用

（1）调用格式：

函数模板名（实参表）

或

函数模板名<类型实参表>(实参表)

（2）调用过程：

①函数模板实例化，生成模板函数。把模板的类型形参 T 实例化为具体的数据类型(模板类型实参)，这样得到一个具体函数，该函数称为模板函数。

②调用模板函数，完成具体功能。

第4章：模板(template)

通用函数（函数模板）

- 用函数模板实现交换两个变量值
- 输出结果：


```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

第4章：模板(template)

通用函数（函数模板）

- 函数模板可以有两个以上的参数，例如：

```
#include <iostream>
using namespace std;
template <class Type1, class Type2>
void myfunc(Type1 x, Type2 y) {
  cout << x << ' ' << y << '\n';
}
void main() {
  myfunc(10, "hi");
  myfunc(0.23, 10L);
}
```

第4章：模板(template)

显式重载通用函数

- 可以显式重载通用函数
- 输出结果：

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Inside swapargs int specialization.
Inside template swapargs.
Inside template swapargs.
Swapped i, j: 20 10
Swapped x, y: 23.3 10.1
Swapped a, b: z x
```

第4章：模板(template)

通用函数（函数模板）

- 函数模板不具有隐式类型转换的能力，在这种情况下，可以利用函数重载机制，用普通函数（**优先于模板函数**）重载一个同名的函数模板，或采用“函数模板名<类型实参表>(实参表)”调用格式显式给出类型实参。
- 显式重载、函数原型、类型转换实例

第4章：模板(template)

通用函数（函数模板）

- 你可以在函数模板以混合方式使用标准类型参数和通用类型参数。这些标准类型参数的使用与它们在其他函数中的使用是一样的。
- 混合参数实例
- 函数模板也可以定义成内联函数。

第4章：模板(template)

通用函数（函数模板）

- 允许用一个函数模板重载另一个函数模板,例如：


```
template <class T> void f(T a)
   {cout << a << endl;}
template <class T1,class T2>
void f(T1 a,T2 b)
   {cout << a << b << endl;}
```

第4章：模板(template)

通用函数（函数模板）

- C++编译器在匹配函数时遵循的规则：

（1）首先寻找一个参数完全匹配的函数；

（2）其次，寻找一个函数模板，把它实例化产生一个匹配的模板函数；

（3）最后，通过显式类型转换<...>达到参数匹配。

类模板(通用类)

```
template <class Type>
class class_name
{
    //body of class
}; //;不可少！
```

第4章：模板(template)

通用类（类模板）

- 使用类模板（通用类）使用户可以为类声明一种模式，使得类中的某些数据成员、成员函数的参数/返回值能取任意类型（系统定义的内部类型或自定义类型）。
- 类是一组对象的公共性质的抽象，而类模板(也称为参数化类)是属于更高层次的抽象。

第4章：模板(template)

通用类（类模板）

- 类模板声明格式：

```
template <class/typename Type>
class class_name {
    //body of class
};
```

其中的Type为类型的占位符(形参)，在类实例化时指定，参数可以有多个。

- 用下述格式创建该类的实例：

class_name<Type> ob;//这里的type为实参

第4章：模板(template)

通用类（类模板）

- 在类内部定义成员函数与平常定义一致。
- 当在类体外定义时，如果成员函数中用到类型参数，则要按函数模板格式进行定义，并且函数名前要用类模板名加以限定。例如：

```
template <class Type>
void TwoNum<Type>::SetAB(Type aa,Type bb)
   {a=aa;b=bb;}
```

第4章：模板(template)

通用类（类模板）

- 用模板实现通用类的简单实例
- 输出结果：

  double division: 3.33333

  integer division: 3
- 多个参数的通用类实例
- 输出结果：

  10 0.23

  X This is a test

第4章：模板(template)

类模板：显式的类具体化

- 通用类的显式类具体化的实例
  (格式：template <> class MyClass<int>)
- 输出结果：

```
Inside generic MyClass
double: 10.1
Inside MyClass<int> specialization
int: 25
```

- 类模板实例(求数组元素之和及求元素位置)

第4章：模板(template)

通用类（类模板）

- 类模板可以使用默认参数：

```
template <class/typename Ttype=int>
class class_name {
    //body of class
}
```

- 用下述格式创建该类(默认参数)的实例：

```
class_name<> ob;
```

第4章：模板(template)

通用类（类模板）

- 类模板同样可以使用标准参数：

```
template <class T,int size>
class buffer {T v[size];};
```


- 然后按下述格式来实例化相应的类：

```
buffer<char,128> buf1;
buffer<Mydate,80> buf2;
```

第4章：模板(template)

通用类（类模板）与继承

- 类模板的派生与类的派生一样，有公有派生和私有派生之分，派生类不能访问基类的私有成员，派生时的语法形式也是相似的。
- 具体介绍：略（参见其他书籍）

综合实例

- 实例：用数组实现(线性)表(list)
- 用链表实现(线性)表、栈、队列等的详细介绍参见“算法与数据结构”

第4章：综合实例

```
template <class T> //建立T类的线性表
class List {
  private:
    int n;                  //表的当前长度
    int MaxSize;            //表元素的最大长度
    T *data;                //表元素数组指针
  public:
    ……
};
```

第4章：综合实例

```
List(int Max=10);
  //构造函数，默认表元素的最大长度为10
~List(){delete[] data;}
  //析构函数，释放表元素数组
bool isEmpty() const  //判表为空吗？
  {return n==0;}
int getLength() const //获得表长度
  {return n;}
```

第4章：综合实例

```
int Locate(const T &x) const;
  //返回表中元素x的位置
bool Retrieve(int k,T &x) const;
  //返回表中第k个元素x
List<T> &Insert(int k,const T &x);
  //在表的第k个位置插入元素x
List<T> &Delete(int k,T &x);
  //在表中删除第k个位置的元素x
```

第4章：综合实例

```
template <class T>
List<T>::List(int Max) {
//在类的外部定义构造函数
   n=0;
   MaxSize=Max;
   data=new T[MaxSize];//必须有析构！
}
```

第4章：综合实例

```
template <class T>
int List<T>::Locate(const T &x) const {
//返回表中元素x的位置
  for (int i=0;i<n;i++)
    if (data[i]==x) return ++i;
  return 0;
}
```

第4章：综合实例

```
template <class T>
bool List<T>::Retrieve(int k,T &x) const {
//返回表中第k个元素x
  if (k<1||k>n) return false;
  x=data[k-1];
  return true;
}
```

第4章：综合实例

```
template <class T> //在表的第k个位置插入元素x
List<T> &List<T>::Insert(int k,const T &x) {
  if (k<0||k>n) throw OutOfBounds();
  if (n==MaxSize) throw NoMem();
  for (int i=n-1;i>=k;i--) data[i+1]=data[i];
    //从第k到n-1的元素往后移1位
  data[k]=x; //插入元素x
  n++;  //表长度加1
  return *this;
}
```

第4章：综合实例

```
template <class T>//在表中删除第k个位置的元素x
List<T> &List<T>::Delete(int k,T &x) {
  if (Retrieve(k,x)) {
    for (int i=k;i<n;i++) data[i-1]=data[i];
      //从第k到n-1的元素往前移1位
    n--;  //表长度减1
    return *this;
  }
  else throw OutOfBounds();
}
```

第4章：综合实例

```
template <class T>  //用普通函数重载运算符<<
ostream &operator<<(ostream &out,const List<T> &x){
  T temp;
  for (int i=1;i<=x.getLength();i++) {
    x.Retrieve(i,temp);
    out << temp << " ";
  }
  return out;
}
```

第4章：综合实例

```
void main() {  //主函数，用于测试类的功能！
  int x;
  List<int> list(10);
  for (int i=0;i<10;i++) list.Insert(i,i+1);
  cout << "数组为：" << list << endl;
  x=list.Locate(5);
  cout << "值5的位置为：" << x << endl;
  list.Delete(3,x);
  cout<<"第3个元素，值[" <<x<< "]已被删除！
   \n";
  cout << "现在的数组为：" << list << endl;
}
```

第4章：综合实例

- 用数组实现(线性)表(list)
- 完整的程序

- 该程序存在致命的bug，必须对赋值运算符=进行重载，同时编写自己的“拷贝构造函数”（留做作业）

本章内容讲授到此结束！

福州大学·软件学院·计算机教研室
王灿辉(wangcanhui@fzu.edu.cn)