第11章-模板

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

1. 泛型编程

概念：通用的数据类型和算法，将算法从数据结构中抽象出来，程序写得尽可能通用，用不变的代码完成一个可变的算法。屏蔽掉数据和操作数据的细节，让算法更为通用，让编程者更多地关注算法的结构，而不是在算法中处理不同的数据类型，总之是不考虑具体数据类型的一种编程方式。
在C++ 中，模板为泛型程序设计奠定了关键的基础。使用模板需要用到两个关键字`template`，`typename`，写法：`template<typename Type>` `template` 告诉编译器，将要定义一个模板，`<>`中的是模板参数列表，类似于函数的参数列表，关键字`typename`看作是变量的类型名，该变量接受类型作为其值，把`Type`看作是该变量的名称，是一个通用的类型。

2. 函数模板

1. 常规使用

建立一个通用函数模板，它所用到的数据的类型（包括返回值类型、形参类型、函数体中局部变量类型）用一个虚拟的类型（模板类型）来代替，而实际调用时编译器根据传入的实参来逆推出真正的类型，生成对应的具体的函数。这样一个通用的模板函数不仅将数据的值作为变化的量，类型也被参数化。
两个关键字：`template`：定义模板的关键字，`typename`：定义模板类型的关键字。

```
template<typename T>
T fun(T a, T b);

int a = 10, b = 20;
fun(a, b);   //其中T 自动推导为 int 类型
```

其中`T`是模板参数，虚拟的类型，用于替换。在实际调用时，实参为int类型，那么会生成一个参数和返回值都为int类型的函数，由一个通用的模板函数生成具体类型的函数。**编译器由函数模板自动生成模板函数的过程叫模板的实例化。**

2. 显示指定及默认值

如果函数的参数中并未使用模板类型，那么编译器无法自动推导，这时就需要手动显式的指定模板类型。

```
//在调用函数时，显式的指定
add<long>(a,b);    //此时 模板类型T 为`long` 类型。
```

除此之外还可以指定模板类型的默认类型，（类似于函数参数指定默认值）。

```
template<typename T =long>
void fun();

fun();         //fun<long>    T 采用默认值
fun<int>();    //fun<int>     T 采用显式指定的类型
```

模板参数类型选择的优先级： **手动显式指定 > 编译器根据实参自动推导 ->模板类型默认值。**

```
template<typename T = int>
void fun(T t);

double d = 12.3;
fun<char>(d);    //T==char

fun(d);          //T=double
```

3. 多模板参数

类似于函数参数，模板类型可以指定多个，用逗号分割，每个模板类型都需要关键字typename修饰。
`template<typename T ,typename K>`
模板类型T替换一种类型，K则可以替换为另一种类型。多模板参数同样可以根据实参进行自动推导。不同于函数参数的默认值，模板参数默认值指定的**顺序可以是任意的没有强制的顺序要求**，但在调用函数时显式指定模板类型时必须从左向右依次指定，不能有间断。

```
template<typename T ,typename K  = int,typename M> //合法
void fun();

fun<int,double,char>();    //从左向右传递
```

经验：一般编译器能够自动推导出来的模板参数放于最后，剩余的模板参数如果有默认值的放于中间，无默认值则放于前面。

```
template<typename T,typename K=long,typename M>
void fun(M &m);

```

```
fun<double>(10);        //fun<double,long,int>
fun<double,char>(10);   //fun<double,char,int>
```

4. 模板函数的声明和定义

如果函数的声明和定义分开，那么在声明和定义处都需要加上模板，如果模板存在默认类型，那么只在函数声明时指定即可。

```
//函数声明
template <typename T=char>
void fun();

//函数定义
template <typename T>  //默认类型去掉
void fun() {...}
```

由于模板函数的定义并不是真正的函数，他们不能单独编译，所以不能将模板函数单独放到源文件中，模板必须与特定的模板实例化请求一起使用，所以最好的办法是模板函数的声明和定义放在一起。

2. 类模板

与函数模板差不多，类模板也需要在类定义的上面加上`template`及`typename`，但在定义对象时，必须使用`<>`显式的指定模板类型。

```
template<typename T>  //定义一个类模板
class CTest{ };       //模板类

CTest<int> tst;       //定义对象
```

模板类型可以替换类内的任意地方定义的类型，包括成员属性类型，成员函数。
类中成员属性若为模板类型，那么我们可以定义带参数的构造，让调用者去指定初始化值。

类模板可以有多个模板类型，且可以指定默认的模板参数，规则是从**右往左依次指定不能间断**（从后往前），在定义对象时从左向右指定，如果不指定模板参数，将使用默认的。

```
template<typename T,typename K=long>                  //合法
template<typename T,typename K=long,typename M=int>   //合法
template<typename T=long,typename K>                  //非法
template<typename T,typename K=long,typename M>       //非法
```

如果模板类中的成员函数在类中声明，类外实现时，定义的函数也要加上模板，如下：

```
template<typename T>
class CTest{
    void show(T);
};

template<typename T>  //通用的模板函数
void CTest<T>::show(T){ ... }
```

如果类模板指定了默认的类型，为了避免歧义，默认的模板类型应当去掉。

如果在模板类中声明、类外定义的成员函数存在函数模板，那么在定义的时，两个模板都需要指定。

```
template<typename T = int>
class CTest{
    template<typename K>
    void fun();
};

//顺序为：先类模板，后函数模板
template<typename T>
template<typename K>
void fun(){ ... }
```

模板类中嵌套的情况：

```
template<typename T>
class A {
public:
    T m_t;
    A() {
        m_t = 10;
    }
    A(T t) {
        m_t = t;
    }
};

class B {
public:
    A<long> m_a;
    B(){  //匹配无参数的构造
    }
```

```
    B(long a):m_a(a) {  //匹配带参数的构造
    }
};

template<typename T>
class C {
public:
    A<T> m_a;
    C(T a) :m_a(a) {  //匹配带参数的构造
    }
};

template<typename K>
class D {
public:
    K m_k;
    D(const K k) :m_k(k) {  //匹配拷贝构造
    }
};

int main(){
    B b;
    cout << b.m_a.m_t << endl;   //10

    B b2(30);
    cout << b2.m_a.m_t << endl;  //30

    C<char> c('a');
    cout << c.m_a.m_t << endl;  //a

    A<double> aa(12.3);

    D<A<double>> d(aa);
    cout << d.m_k.m_t << endl;  //12.3
}
```

3. 优化链表

让链表可以装任意类型的数据结构。

```
template<typename VT>
struct Node {
    VT m_value;
```

```
    Node* pNext;
    //结构体构造函数
    Node(const VT& v):m_value(v), pNext(nullptr){}
    //结构体析构
    ~Node() {}
};

template<typename T>
class CIterator {
public:
    Node<T>* m_pTemp;
public:
    CIterator() : m_pTemp(nullptr) {}
    CIterator(Node<T>* pNode) :m_pTemp(pNode) {}
    ~CIterator() {}
public:
    Node<T>* operator=(Node<T>* pNode) {
        m_pTemp = pNode;
        return m_pTemp;
    }
    bool operator==(Node<T>* pNode) {
        return m_pTemp == pNode;
    }
    bool operator!=(Node<T>* pNode) {
        return m_pTemp != pNode;
    }
    T& operator*() {
        return m_pTemp->m_value;
    }
    Node<T>* operator++() {
        m_pTemp = m_pTemp->pNext;
        return m_pTemp;
    }
    Node<T>* operator++(int) {
        Node<T>* pTemp = m_pTemp;
        m_pTemp = m_pTemp->pNext;
        return pTemp;   //返回的是加之前的
    }
    operator bool() {
        return m_pTemp;
    }
};

template<typename T>
```

```
48  class CMyList {
49  public:
50      Node<T>* m_pHead;  //头指针
51      Node<T>* m_pEnd;  //头指针
52      int   m_nLength; //链表的长度
53  public:
54      CMyList():m_pHead(nullptr), m_pEnd(nullptr), m_nLength(0){}
55      ~CMyList() {
56          Node<T>* pTemp = nullptr;
57          while (m_pHead) {
58              pTemp = m_pHead;            //标记
59              m_pHead = m_pHead->pNext;  //移动
60              delete pTemp;               //回收标记的
61          }
62          m_pHead = nullptr;
63          m_pEnd = nullptr;
64          m_nLength = 0;
65      }
66  public:
67      void ShowList() {
68          CIterator<T> ite(m_pHead);  //带参数的构造,
69          while (ite) {
70              cout << *ite << "    ";
71              ite++;
72          }
73          cout << endl;
74      }
75      void PushBack(const T& v) {
76          Node<T>* pNode = new Node<T>(v);
77
78          if (m_pHead)   //非空链表
79              m_pEnd->pNext = pNode;  //先连接
80          else //空链表
81              m_pHead = pNode;
82
83          m_pEnd = pNode;
84          ++m_nLength;   //长度增长
85      }
86      void PopFront() {
87          if (m_pHead) { //非空链表
88              Node<T>* pTemp = m_pHead;  //标记
89              if (m_pHead == m_pEnd) {//1个节点
90                  m_pHead = nullptr;
91                  m_pEnd = nullptr;
```

```
            }
            else   //多个节点
                m_pHead = m_pHead->pNext;

            delete pTemp;   //回收标记的
            pTemp = nullptr;

            --m_nLength;  //长度减少
        }
    }
    int GetLength() {   //获取链表长度
        return m_nLength;
    }
};

class CStudent {
public:
    string m_name;
    int    m_age;
    bool   m_sex;
    CStudent():m_name(""), m_age(0), m_sex(true){}
    CStudent(const string & name,const int& age,const int& sex)
        :m_name(name), m_age(age), m_sex(sex){}

    ~CStudent() {
        m_name = "";
        m_age = 0;
        m_sex = true;
    }
};

//这里需要重载输出操作符，保证可以输出自定义类型
ostream& operator<<(ostream& os, CStudent& s) {
    os << s.m_name << "  " << s.m_age << "   " << s.m_sex ;
    return os;
}

int main() {
    CMyList<char> lst;
    lst.PushBack('a');
    lst.PushBack('b');
    lst.PushBack('c');
    lst.PushBack('d');

```

```
136         lst.ShowList();
137         cout << lst.GetLength() << endl;
138 
139         lst.PopFront();
140         lst.PopFront();
141         lst.ShowList();
142         cout << lst.GetLength() << endl;
143         //-----------------------------------------
144         CMyList<CStudent> lst2;
145         {
146             CStudent st1("小明", 1, 1);
147             lst2.PushBack(st1);
148             CStudent st2("小张", 2, 0);
149             lst2.PushBack(st2);
150             CStudent st3("小李", 3, 1);
151             lst2.PushBack(st3);
152         }
153         cout << lst2.GetLength() << endl;
154         lst2.ShowList();
155 
156         lst2.PopFront();
157         cout << lst2.GetLength() << endl;
158         lst2.ShowList();
159 
160         return 0;
161     }
```

```
a    b    c    d
4
c    d
2
3
小明  1   1    小张  2   0    小李  3   1
2
小张  2   0    小李  3   1
```

4. 模板类封装数组

```
1   template<typename T>
2   class CDynamicArray {
3   private:
```

```
4       T*  m_pArr;     //指针指向堆区开辟的真实数组
5       size_t m_capacity;
6       size_t m_size;
7   public:
8       explicit CDynamicArray(size_t len):m_capacity(len), m_size(0), m_pA
    rr(nullptr){
9           if (len > 0) {
10              m_pArr = new T[len]();
11          }
12      }
13      explicit CDynamicArray(const CDynamicArray& arr) :m_capacity(arr.m_
    capacity), m_size(arr.m_size), m_pArr(nullptr) {
14          //判断形参中的指针是否为空
15          if (arr.m_pArr) {//判空后再决定是否申请数组
16              m_pArr = new T[m_capacity]();
17              for (size_t i = 0; i < m_size; i++) {  //依次拷贝
18                  m_pArr[i] = arr.m_pArr[i];
19              }
20          }
21      }
22      CDynamicArray& operator=(const CDynamicArray& arr){
23          if (this != &arr) {  //判断是否是自己
24              if (m_pArr) {  //回收旧空间
25                  delete[] m_pArr;
26                  m_pArr = nullptr;
27              }
28              m_capacity = arr.m_capacity;
29              m_size = arr.m_size;
30              if (arr.m_pArr) {  //申请新空间，在依次拷贝
31                  m_pArr = new T[m_capacity]();
32                  for (size_t i = 0; i < m_size; i++) {
33                      m_pArr[i] = arr.m_pArr[i];
34                  }
35              }
36          }
37          return *this;
38      }
39      void PushBack(const T& t) {
40          if (m_size<m_capacity) { //正常尾添加
41              m_pArr[m_size++] = t;
42          }
43          else { //扩容
44              size_t oldSize = m_size++;
45              //1.5倍扩容
```

```
            m_capacity = m_size < (oldSize + oldSize / 2) ? (oldSize + oldSize / 2) : m_size;

            //申请新空间并依次拷贝
            T* pTemp = new T[m_capacity]();
            for (size_t i = 0; i < oldSize; i++) {
                pTemp[i] = m_pArr[i];
            }

            pTemp[oldSize] = t;   //添加新的值

            delete[]m_pArr;   //删除旧空间
            m_pArr = pTemp;   //指针指向新空间
        }
    }
    void PopBack() {
        if (m_size > 0) {
            --m_size;
        }
    }
    T& operator[](size_t index) {
        if (index >= m_size) {
            _ASSERT_EXPR(false, "arr is out of size");
        }
        return m_pArr[index];
    }
    size_t GetLength()const {
        return m_size;
    }
    size_t GetCapacity()const {
        return m_capacity;
    }

    //支持范围for遍历
    T* begin() {
        return &m_pArr[0];
    }
    T* end() {  //注意遍历不包含end
        return &m_pArr[m_size];
    }
};
```