课程目标

- 函数模板
- 模板实现机制
- 模板局限性
- 类模板
- 类模板的应用

课程实验

- 设计一个数组模板类(MyArray),完成对不同类型元素的管理
- Pair类模板例子

课堂引入

模板类和模板函数使得程序员能够以更高的抽象层次思考和编写代码。模板允许编写与具体数据类型无关的通用代码，同时可以在编译时进行类型检查，减少了运行时的类型错误，这些使得这些代码可以被不同类型的数据使用，提高了代码的重用性、灵活性和可维护性。模板是C++中强大的特性之一，能够有效地支持泛型编程。

模板类和函数使得通用算法的实现变得简单，可以适用于多种数据类型，从而减少了在不同类型上编写相似算法的需要。

模板提供了一种静态多态性的实现方式，即通过模板参数的多样性实现代码的多态性，而无需虚函数和运行时多态性的开销。

授课进程

一 模板

1.1 什么是模板

C++ 有各种特性来提高代码的可重用性，有助于减少开发的代码量和工作量。

C++ 提高代码的可重用性主要有两方面：

> - 继承
> - 模板

使用模板的特性设计，实际上也就是泛型程序设计。

模板是C++支持**参数化**多态的工具。

使用模板可以使用户为类或者函数声明一种一般模式，使得类中的某些数据成员或者成员函数的参数、返回值取得任意类型。

使用模板的目的就是为了实现泛型，能够让程序员编写与类型无关的代码，可以减轻编程的工作量，增强函数的重用性。

> 1. 模板是一种对**类型**进行参数化的工具；

> 2. C++提供两种模板机制：函数模板(function template)和类模板(class template)；
> 3. 函数模板针对仅**参数类型**不同的**函数**；
> 4. 类模板针对仅数据成员和成员函数类型不同的类。

> 注意：模板的声明或定义只能在全局，命名空间或类范围内进行。即不能在局部范围，函数内进行，比如不能在main函数中声明或定义一个模板。

1.2 函数模板

所谓函数模板，**实际上是建立一个通用函数，其函数类型和形参类型不具体制定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。**凡是函数体相同的函数都可以用这个模板代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现不同函数的功能。

1.2.1 函数模板通式

```
// 两种写法
template <class 形参名，class 形参名，.....> 返回类型 函数名(参数列表) { 函数体 }

template <typename 形参名，typename 形参名，.....> 返回类型 函数名(参数列表) { 函数体 }
```

其中**template**和**class**是关键字，class可以用typename 关键字代替，在这里**typename 和class没区别**，**< >**括号中的参数叫**模板形参**，模板形参和函数形参很相像，**模板形参不能为空。一但声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数，**即可以在该函数中使用内置类型的地方都可以使用模板形参名。模板形参需要调用该模板函数时提供的模板实参来初始化模板形参，一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。

比如**swap**的模板函数形式为:

```
template <class T> void swap(T& a, T& b) {}
```

当调用这样的模板函数时类型 T 就会被被调用时的类型所代替，比如**swap(a, b)**其中**a**和**b**是**int** 型，这时模板函数swap中的形参**T**就会被**int** 所代替，模板函数就变为**swap(int &a, int &b)**。而当**swap(c, d)**其中**c**和**d**是**double**类型时，模板函数会被替换为**swap(double &a, double &b)**，这样就实现了函数的实现与类型无关的代码。

注意：对于函数模板而言不存在 h(int, int) 这样的调用

不能在函数调用的参数中指定模板形参的类型，对函数模板的调用应使用实参推演来进行，即只能进行 h(2, 3) 这样的调用，或者

```
int a, b;
h(a, b)
```

1. 变量交换函数模板

假设我们设计一个交换两个整型变量的值的函数，代码如下：

```
// 交换两个整型变量的值的 swap 函数：
void swap(int & x, int & y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

如果是浮点类型的变量的值交换，则替换 int 类型为 double 即可，代码如下：

```
// 交换两个double型变量的值的 swap 函数:
void swap(double & x, double & y){
    double tmp = x;
    x = y;
    y = tmp;
}
```

那如果是其他变量类型的值交换，那不是每次都要重新写一次 swap函数？是不是很繁琐？且代码后面会越来越冗余。

能否只写一个 swap 函数，就能交换各种类型的变量？

答案是肯定有的，就是用「函数模板」来解决，「函数模板」的形式：

```
template <class 类型参数1, class 类型参数2, ...>
返回值类型 模板名（形参表）{
    函数体
};
```

具体 swap「函数模板」代码如下：

template就是模板定义的关键词，T代表的是任意变量的类型。

```
template <class T> void swap(T &x, T &y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

那么定义好「函数模板」后，在编译的时候，编译器会根据传入 swap函数的参数变量类型，自动生成对应参数变量类型的 swap函数：

```
int main(int argc, char *argv[]) {
    int n = 1,m = 2;
    swap(n, m);       // 编译器自动生成 void swap(int &, int &)函数
    double f = 1.2, g = 2.3;
    swap(f, g);       // 编译器自动生成 void swap(double &, double &)函数
    return 0;
}
```

上面的实例化函数模板的例子，是让编译器自己来判断传入的变量类型，那么我们也可以自己指定函数模板的变量类型，具体代码如下：

```
int main() {
    int n = 1, m = 2;
    swap<int>(n, m);      // 指定模板函数的变量类型为int
    double f = 1.2, g = 2.3;
    swap<double>(f, g);  // 指定模板函数的变量类型为double
    return 0;
}
```

2. 查询数组最大值函数模板

下面的maxElement函数定义成了函数模板，这样不管是 int、double、char 等类型的数组，都可以使用该函数来查数组最大的值

代码如下：

```
// 求数组最大元素的maxElement函数模板
template <typename T> T maxElement(T a[], int size) {     // size是数组元素个数
    T tmpMax = a[0];    // 设置数组首元素为最大元值
    for(int i = 1; i < size; ++i) {
        if(tmpMax < a[i]) {
            tmpMax = a[i];
        }
    }
    return tmpMax;
}
```

3. 多个类型参数模板函数

函数模板中,可以不止一个类型的参数:

T1 是传入的第一种任意变量类型,T2 是传入的第二种任意变量类型。

```
template<class T1, class T2> T2 MyFun(T1 arg1, T2 arg2) {
    cout << arg1 << " " << arg2 << endl;
    return arg2;
}
```

4. 函数模板的重载

函数模板可以重载，只要它们的形参表或类型参数表不同即可。

```
// 模板函数 1
template<class T1, class T2> void print(T1 arg1, T2 arg2) {
    cout << arg1 << " " << arg2 << endl;
}

// 模板函数 2
template<class T> void print(T arg1, T arg2) {
    cout << arg1 << " " << arg2 << endl;
}
```

```
// 模板函数 3 (和第一个一样,不是重载，报错)
template<class T, class T2> void print(T arg1, T2 arg2) {
    cout << arg1 << " "<< arg2 << endl;
}
```

上面都是 print(参数1, 参数2)模板函数的重载，因为「形参表」或「类型参数表」名字不同。

5. 函数模板和函数的次序

在有多个函数和函数模板名字相同的情况下，编译器如下规则处理一条函数调用语句：

> - 先找参数完全匹配的普通函数（非由模板实例化而得的函数）；
> - 再找参数完全匹配的模板函数；
> - 再找实参数经过自动类型转换后能够匹配的普通函数；
> - 上面的都找不到，则报错。

代码例子如下：

```
// 模板函数 - 1个参数类型
template <class T> T Max(T a, T b) {
    cout << "TemplateMax" <<endl;
    return 0;
}

// 模板函数 - 2个参数类型
template <class T, class T2> T Max(T a, T2 b) {
    cout << "TemplateMax2" << endl;
    return 0;
}

// 普通函数
double Max(double a, double b) {
    cout << "MyMax" << endl;
    return 0;
}

int main(int argc, char *argv[]) {
    int i=4, j=5;
    Max(1.2, 3.4);      // 输出MyMax - 匹配普通函数
    Max(i, j);          // 输出TemplateMax  - 匹配参数一样的模板函数
    Max(1.2, 3);        // 输出TemplateMax2 - 匹配参数类型不同的模板函数

    return 0;
}
```

匹配模板函数时，当模板函数只有一个参数类型时，传入了不同的参数类型，不会进行类型自动转换

```
// 模板函数 - 1个参数类型
template<class T> T myFunction(T arg1, T arg2){
    cout << arg1 << " " << arg2 << "\n";
    return arg1;
}

// OK : 替换 T 为 int 类型
myFunction(5, 7);
// OK : 替换 T 为 double 类型
myFunction(5.8, 8.4);
// error : 没有匹配到myFunction(int, double)函数
myFunction(5, 8.4);
```

1.3 模板函数和普通函数的区别

- 函数模板不允许自动类型转换，必须严格匹配类型
- 普通函数能够自动进行类型转化

```
// 函数模板
template<class T> T MyPlus(T a, T b) {
    T ret = a + b;
    return ret;
}

// 普通函数
int MyPlus(int a, char b) {
    int ret = a + b;
    return ret;
}

void test02() {
    int a = 10;
    char b = 'a';

    // 调用函数模板，严格匹配类型
    MyPlus(a, a);   // 模板函数
    MyPlus(b, b);   // 模板函数
    // 调用普通函数
    MyPlus(a, b);
    // 调用普通函数  普通函数可以隐式类型转换
    MyPlus(b, a);
}
```

1.4 函数模板和普通函数的调用规则

- c++编译器优先考虑普通函数
- 可以通过空模板实参列表的语法限定编译器只能通过模板匹配
- 函数模板可以像普通函数那样可以被重载
- 如果函数模板可以产生一个更好的匹配，那么选择模板

```cpp
// 函数模板
template<class T> T MyPlus(T a, T b) {
    T ret = a + b;
    return ret;
}

// 普通函数
int MyPlus(int a, int b) {
    int ret = a + b;
    return ret;
}

void test_03() {
    int a = 10;
    int b = 20;
    char c = 'a';
    char d = 'b';
    // 如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
    cout << MyPlus(a, b) << endl;
    // 如果我必须要调用函数模板,那么怎么办?
    cout << MyPlus<>(a, b) << endl;
    // 此时普通函数也可以匹配，因为普通函数可以自动类型转换
    // 但是此时函数模板能够有更好的匹配
    // 如果函数模板可以产生一个更好的匹配，那么选择模板
    cout << MyPlus(c, d) << endl;
}

// 函数模板重载
template<class T> T MyPlus(T a, T b, T c){
    T ret = a + b + c;
    return ret;
}

void test_04() {
    int a = 10;
    int b = 20;
    int c = 30;
    cout << MyPlus(a, b, c) << endl;
    // 如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
}
```

1.5 模板的局限性

```cpp
template<class T> void f(T a, T b) { … }
```

如果代码实现时定义了赋值操作 a = b，但是T为数组，这种假设就不成立了。同样，如果里面的语句为判断语句if(a>b),但T如果是结构体，该假设也不成立，另外如果是传入的数组，数组名为地址，因此它比较的是地址，而这也不是我们所希望的操作。

> 总之，编写的模板函数很可能无法处理某些类型，另一方面，有时候通用化是有意义的，但C++语法不允许这样做。为了解决这种问题，可以提供模板的重载，为这些特定的类型提供具体化的模板。

```
class Person {
public:
    Person(string name, int age) {
        this->m_strName = name;
        this->m_nAge = age;
    }
    string m_strName;
    int m_nAge;
};

// 普通交换函数
template <class T> void mySwap(T &a, T &b) {
    T temp = a;
    a = b;
    b = temp;
}

// 第三代具体化，显示具体化的原型和意思以template<>开头，并通过名称来指出类型
// 具体化优先于常规模板
template<> void mySwap<Person>(Person &p1, Person &p2) {
    string nameTemp = p1.mName;
    p1.m_strName = p2.m_strName;
    p2.m_strName = nameTemp;

    int ageTemp = p1.m_nAge;
    p1.m_nAge = p2.m_nAge;
    p2.m_nAge = ageTemp;
}

void test() {
    Person P1("Tom", 10);
    Person P2("Jerry", 20);

    cout << "P1 Name = " << P1.m_strName << " P1 Age = " << P1.m_nAge << endl;
    cout << "P2 Name = " << P2.m_strName << " P2 Age = " << P2.m_nAge << endl;
    mySwap(P1, P2);
    cout << "P1 Name = " << P1.m_strName << " P1 Age = " << P1.m_nAge << endl;
    cout << "P2 Name = " << P2.m_strName << " P2 Age = " << P2.m_nAge << endl;
}
```

1.6 类模板

1.6.1 类模板基本概念

类模板和函数模板的定义和使用类似。如果有两个或多个类，其功能是相同的，仅仅是数据类型不同。为了方便地定义出一批相似的类，可以定义「类模板」，然后由类模板生成不同的类。

类模板的定义形式如下：

```
template <class 类型参数1, class 类型参数2, ...> // 类型参数表
class 类模板名 {
    成员函数和成员变量
};
```

用类模板定义对象的写法：

> 类模板名<真实类型参数表> 对象名(构造函数实参表);

01 Pair类模板例子

接下来，用 Pair 类用类模板的方式的实现，Pair 是一对的意思，也就是实现一个键值对（key-value）的关系的类。

```
// 类模板
template <class T1, class T2> class Pair {
public:
    Pair(T1 k, T2 v) : m_key(k), m_value(v) {};
    friend bool operator<(const Pair<T1, T2> &p) const;
private:
    T1 m_key;
    T2 m_value;
};

// 类模板里成员函数的写法
template <class T1, class T2> bool Pair<T1, T2>::operator<(const Pair<T1, T2>
&p) const {
    return m_value < p.m_value;
}

int main() {
    Pair<string, int> Astudent("Jay", 20);
    Pair<string, int> Bstudent("Tom", 21);
    cout << (Astudent < Bstudent) << endl;
    return 0;
}
```

> 输出结果：1

需要注意的是，同一个类模板的两个模板类是不兼容的：

```
Pair<string, int> *p;
Pair<string, double> a;
p = &a; // 错误！！
```

1.6.2 函数模板作为类模板成员

当函数模板作为类模板的成员函数时，是可以单独写成函数模板的形式，成员函数模板在使用的时候，编译器才会把函数模板根据传入的函数参数进行实例化，例子如下：

```
// 类模板
template<class T> class A {
public:
    template<class T2> void Func(T2 t) { cout << t << endl; }     // 成员函数模板内
联
    template<class T3> void Func2(T3 t);                          // 类外定义
};
```

```
template<class T> template<class T3> void A<T>::func2(T3 t) { }

int main(){
    A<int> a;
    a.Func('K');        // 成员函数模板 Func被实例化
    a.Func("hello");    // 成员函数模板 Func再次被实例化

    return 0;
}
```

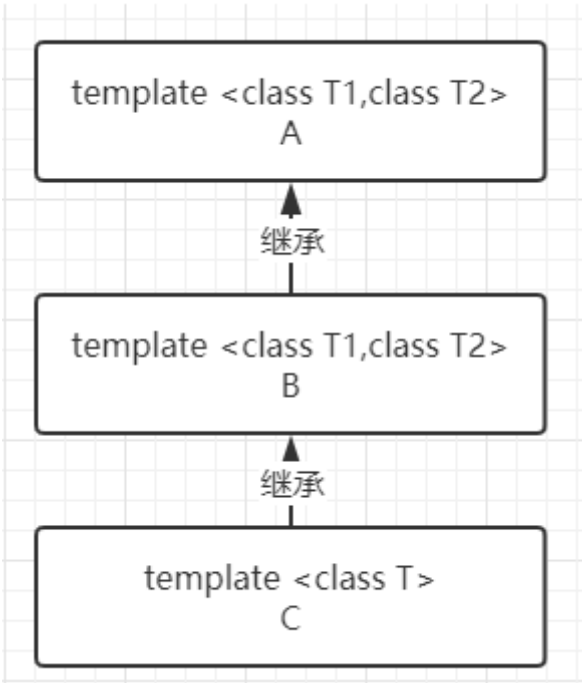
1.6.3 类模板与非类型参数

类模板的"<类型参数表>"中可以出现非类型参数：

```
template<class T, int size> class CArray {
public:
    void print() {
        for(int i = 0; i < size; ++i) {
            cout << array[i] << endl;
        }
    }
private:
    T array[size];
};

CArray<double, 40> a2;
CArray<int, 50> a3;             // a2和a3属于不同的类
```

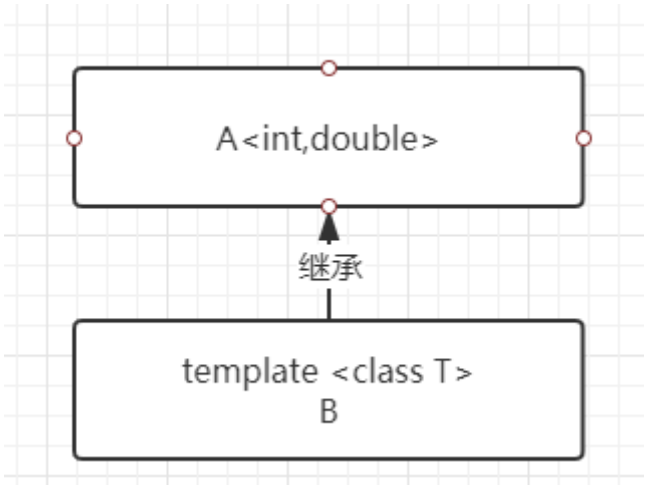
1.6.4 类模板与派生

01 类模板从类模板派生



```cpp
#include <iostream>
using namespace std;

// 基类 - 类模板
template<class T1, class T2> class A {
private:
    T1 v1;
    T2 v2;
};

// 派生类 - 类模板
template<class T1, class T2> class B : public A<T1, T2> {
private:
    T1 v3;
    T2 v4;
};

// 派生类 - 类模板
template<class T1, class T2> class C : public B<T1, T2> {
    T1 v5;
};

int main() {
    B<int, double> obj1;
    C<int, string> obj2;

    return 0;
}
```

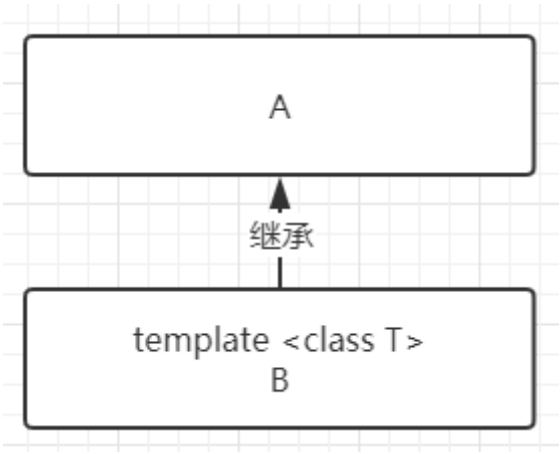
02 类模板从模板类派生




```cpp
template <class T1, class T2> class A {
private:
    T1 v1;
    T2 v2;
};

// A<int,double> 模板类
template <class T> class B : public A<int, double> {
```

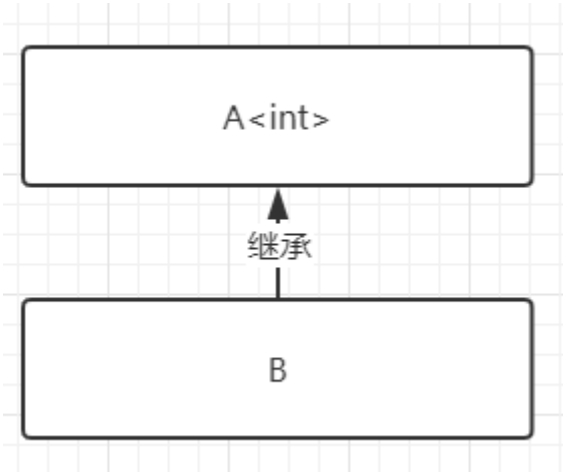
```
private:
    T v;
};

int main() {
    // 自动生成两个模板类 : A<int,double> 和 B<char>
    B<char> obj1;
    return 0;
}
```

03 类模板从普通类派生




```
// 基类 - 普通类
class A {
private:
    int v1;
};

// 派生类 - 类模板
template<class T> class B : public A {          // 所有从B实例化得到的类，都以A为基类
private:
    T v;
};

int main() {
    B<char> obj1;
    return 0;
}
```

04 普通类从模板类派生




```
template <class T> class A {
    T v1;
};

class B : public A<int> {
    double v;
};

int main() {
    B obj1;
    return 0;
}
```

1.6.5 类模板与友元

01 函数、类、类的成员函数作为类模板的友元

```
// 普通函数
void Func1() {}

// 普通类
class A {};

// 普通类
class B {
public:
    void Func() {}        // 成员函数
};

// 类模板
template<class T> class Tmp {
    friend void Func1();    // 友元函数
    friend class A;         // 友元类
    friend void B::Func();  // 普通类的成员函数作为该类的友元函数
}; // 任何从 Tmp 实例化来的类 ，都有以上三个友元
```

02 函数模板作为类模板的友元

```
// 类模板
template<class T1, class T2> class Pair {
private:
    T1 key;          // 关键字
    T2 value;        // 值
public:
    Pair(T1 k, T2 v):key(k),value(v) { };
    // 友元函数模板(注意:这不是一个友元模板成员函数,这只是一个全局模板函数)
    template<class T3, class T4> friend ostream& operator<<(ostream &o, const
Pair<T3, T4> &p);
};

// 函数模板
template <class T3, class T4>ostream& operator<<(ostream& o, const Pair<T3, T4>
&p) {
    o << "(" << p.key << "---" << p.value << ")" ;
    return o;
}

int main() {
    Pair<string, int> student("Tom", 29);
    Pair<int, double> obj(12, 3.14);
    cout << student << " " << obj << endl;;
    return 0;
}
```

输出结果:

> (Tom---29) (12---3.14)

03 函数模板作为类的友元

```
// 普通类
class A {
private:
    int v;
public:
    explicit A(int n) : v(n) { }
    template<class T> friend void print(const T &p);    // 函数模板
};

// 函数模板
template<class T> void print(const T &p){
    cout << p.v;
}

int main() {
    A a(4);
    print(a);
    return 0;
}
```

> 输出结果: 4

04 类模板作为类模板的友元

```
// 类模板
template<class T> class B {
private:
    T v;
public:
    B(T n) : v(n) { }
    template<class T2> friend class A; // 友元类模板
};

// 类模板
template<class T> class A {
public:
    void Func() {
        B<int> o(10); // 实例化B模板类
        cout << o.v << endl;
    }
};

int main() {
    A<double> a;
    a.Func();
    return 0;
}
```

> 输出结果： 10

1.6.6 类模板与静态成员变量

类模板中可以定义静态成员，那么从该类模板实例化得到的所有类，都包含同样的静态成员。

```
template<class T> class A {
private:
    static int count;     // 静态成员
public:
    A() { count++; }
    A(A &) { count++ ; }
    ~A() { count-- ; };
    // 静态函数
    static void printCount() {
        cout << count << endl;
    }
};

template<> int A<int>::count = 0;         // 初始化
template<> int A<double>::count = 0;      // 初始化

int main() {
    A<int> ia;
    A<double> da;     // da和ia不是相同模板类
    ia.printCount();
```

```
    da.printCount();
    return 0;
}
```

输出:

```
1
1
```

上面的代码需要注意的点:

- 类模板里的静态成员初始化的时候，最前面要加 `template<>`。
- ia 和 da 对象是不同的模板类，因为类型参数是不一致，所以也就是不同的模板类。

1.7 类模板的应用

设计一个数组模板类(MyArray),完成对不同类型元素的管理

```
template<class T> class MyArray {
public:
    explicit MyArray(int capacity) {
        this->m_Capacity = capacity;
        this->m_Size = 0;
        // 如果T是对象，那么这个对象必须提供默认的构造函数
        pAddress = new T[this->m_Capacity];
    }

    // 拷贝构造
    MyArray(const MyArray &arr) {
        this->m_Capacity = arr.m_Capacity;
        this->m_Size = arr.m_Size;
        this->pAddress = new T[this->m_Capacity];
        for (int i=0; i<this->m_Size; i++) {
            this->pAddress[i] = arr.pAddress[i];
        }
    }

    // 重载[] 操作符  arr[0]
    T& operator[](int index) {
        return this->pAddress[index];
    }

    // 尾插法
    void push_back(const T &val) {
        if (this->m_Capacity == this->m_Size) {
            return;
        }
        this->pAddress[this->m_Size] = val;
        this->m_Size++;
    }

    void pop_back() {
        if (this->m_Size == 0) {
            return;
```

```
        }
        this->m_Size--;
    }

    int getSize() {
        return this->m_Size;
    }
    // 析构
    ~MyArray() {
        if (this->pAddress != NULL) {
            delete[] this->pAddress;
            this->pAddress = nullptr;
            this->m_Capacity = 0;
            this->m_Size = 0;
        }
    }

private:
    T *pAddress;        // 指向一个堆空间，这个空间存储真正的数据
    int m_Capacity;     // 容量
    int m_Size;         // 大小
};
```

```
// 测试代码:

class Person {
public:
    Person(string name, int age) {
        this->mName = name;
        this->mAge = age;
    }
public:
    string mName;
    int mAge;
};

void printMyArrayInt(MyArray<int>& arr) {
    for (int i = 0; i < arr.getSize(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

void printMyPerson(MyArray<Person>& personArr) {
    for (int i = 0; i < personArr.getSize(); i++) {
        cout << "姓名: " << personArr[i].mName << " 年龄:  " << personArr[i].mAge
<< endl;
    }
    cout << endl;
}

int main() {
    MyArray<int> myArrayInt(10);
    for (int i = 0; i < 9; i++) {
```

```
        myArrayInt.push_back(i);
    }
    myArrayInt.push_back(100);
    printMyArrayInt(myArrayInt);

    MyArray<Person> myArrayPerson(10);
    Person p1("德玛西亚", 30);
    Person p2("提莫", 20);
    Person p3("孙悟空", 18);
    Person p4("赵信", 15);
    Person p5("赵云", 24);
    myArrayPerson.push_back(p1);
    myArrayPerson.push_back(p2);
    myArrayPerson.push_back(p3);
    myArrayPerson.push_back(p4);
    myArrayPerson.push_back(p5);
    printMyPerson(myArrayPerson);

    return 0;
}
```

课堂小结

- 函数模板及函数模板的通式
- 函数模板和普通函数的调用规则
- 模板实现机制和局限性
- 类模板
- 函数模板作为类模板的成员
- 类模板与非类型参数
- 类模板与派生
- 类模板与友元
- 类模板与静态变量
- 类模板的应用

随堂作业

1. 使用函数模板实现对char和int类型数组进行排序?

```
// 模板打印函数
template<class T> void printArray(T arr[], int len) {
    for (int i = 0; i < len;i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// 模板排序函数
template<class T> void CSort(T arr[],int len) {
    for (int i = 0; i < len;i++) {
        for (int j = len - 1; j > i;j--) {
```

```
            if (arr[j] > arr[j - 1]) {
                T temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

void test() {
    // char数组
    char tempChar[] = "aojtifysn";
    int charLen = strlen(tempChar);

    // int数组
    int tempInt[] = {7,4,2,9,8,1};
    int intLen = sizeof(tempInt) / sizeof(int);

    // 排序前 打印函数
    printArray(tempChar, charLen);
    printArray(tempInt, intLen);
    // 排序
    CSort(tempChar, charLen);
    CSort(tempInt, intLen);
    // 排序后打印
    printArray(tempChar, charLen);
    printArray(tempInt, intLen);
}
```