**C++提高**

1. **函数模板**

类型参数化 编译代码可以忽略类型

为了让编译器区分是普通函数 还是 模板函数

template<class T>

template<typename T>

**函数模板与普通函数的区别？**

A.函数模板不允许类型转化，必须严格类型匹配

B.普通函数能够自动进行类型转化

**函数模板和普通函数在一起调用原则：**

1. 函数模板可以像普通函数那样被重载
2. C++编译器优先考虑普通函数
3. 如果函数模板可以产生一个更好的匹配，那么选择模板
4. 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

**编译过程**

Index.cpp **->**预编译器**->**Index.i**->**编译器**->**Index.s 汇编文件**->**汇编器**->**目标文件.obj(windows) .o(linux) **->**连接器**->**Index.exe

g++ -E Index.cpp -o Index.i 预编译器E

g++ -S Index.i -o Index.s 编译器S

g++ -C Index.s -o Index.o 汇编器C

g++ Index.s -o Index 连接器

**软件推荐 SercureCRT**

**模板函数实现原理剖析**

函数模板 -----------------> 模板函数 -----------------> 被调用

template<class T> int MyAdd(int a,int b){

int MyAdd(T a,T b){ return a+b;

return a+b; }

}

做手链模子 -----------------> 生产具体手链 -----------------> 才能带到手上

**函数模板机制结论**

编译器并不是把函数模板处理成能够处理任何类型的函数

函数模板通过具体类型产生不同的函数

编译器会对函数模板进行两次编译，在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译

**2.类模板**

类模板和函数模板的定义和使用类似。有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同。

类模板用于实现类所需数据类型参数化

类模板在表示如数组、表、图等数据结构显得特别重要，这些数据结构的表示和算法不受所包含数据类型的影响。

template<class T>

class Person{

public:

Person(T id,T age){ // 构造函数

this->mAge = age;

this->mId = id;

}

void Show(){

cout<<mId <<mAge <<endl;

}

public:

T mId ;

T mAge ;

}

void test01(){

//函数模板在调用的时候，可以自动类型推导;

//类模板必须显式指定类型

Person<int> p(10,20);

p.Show;

}

**2.1类模板派生普通类**

//为什么？

//类区定义对象，这个对象是不是需要编译分配内存

template<class T>

class SubPersion : public Persion<int> {

}

2.2**类模板派生类模板**

template<class T>

class BubPersion : public Persion<T> {

}

**2.3 .h和.cpp分离**

.h文件 //只做类的声明

#ifndef PERSON\_H //防止头文件被重复包含

#pragma once //c++中防止头文件

.cpp文件

声明具体化，具体实现

**2.4类模板类内实现**

|  |
| --- |
| template<class T1,class T2> |
| class Person { |
| public: |
| Person(T1 name, T2 age) { |
| this->mAge = age; |
| this->mName = name; |
| } |
| void Show() |
| { |
| cout << "Name:" << this->mName << " Age:" << this->mAge << endl; |
| } |
| public: |
| T1 mName; |
| T2 mAge; |
| }; |
|  |
| void test01() |
| { |
| Person<string, int> p("AAA", 20); |
| p.Show(); |
| } |
|  |
| int main() |
| { |
| test01(); |
| return 0; |

}

**2.5**