1 内联函数

> c++从c中继承的一个重要特征就是效率。假如c++的效率明显低于c的效率，那么就会有很大的一批程序员不去使用c++了。
> 在c中我们经常把一些短并且执行频繁的计算写成宏，而不是函数，这样做的理由是为了执行效率，宏可以避免函数调用的开销，这些都有预处理来完成。

但是在c++出现之后，使用预处理宏会出现两个问题：

> 1 c中也会出现，宏看起来像一个函数调用，但是会有隐藏一些难以发现的错误。和实际的函数调用不一致
>
> 2 面向对象是c++特有的，预处理器不允许访问类的成员，也就是说预处理器宏不能用作类的成员函数

为了保持预处理宏的效率又增加安全性，而且还能像一般成员函数那样可以在类里访问自如，

> c++引入了内联函数(inline function).

内联函数为了继承宏函数的效率，没有函数调用时开销，然后又可以像普通函数那样，可以进行参数，返回值类型的安全检查，又可以作为成员函数。

1.1 内联函数的作用

作用：不是在调用时发生控制转移，而是在编译时将函数体嵌入在每一个调用处，

适用于功能简单，规模较小又使用频繁的函数。

> 递归函数无法内联处理，内联函数不能有循环体，switch语句，不能进行异常接口声明。

主要体现在于inline关键字

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的效率会很少。

另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

1.2 内联函数和宏定义的区别

内联函数和宏的区别在于

> 1. 宏是由预处理器对宏进行替代，而内联函数是通过编译器控制来实现的。
> 2. 内联函数是真正的函数，只是在需要用到的时候，内联函数像宏一样的展开，所以取消了函数的参数压栈，减少了调用的开销。你可以象调用函数一样来调用内联函数，而不必担心会产生于处理宏的一些问题。
> 3. 内联函数与带参数的宏定义进行下比较，它们的代码效率是一样，但是内联欢函数要优于宏定义，因为内联函数遵循的类型和作用域规则，它与一般函数更相近，
> 4. 在一些编译器中，一旦关联上内联扩展，将与一般函数一样进行调用，比较方便。

另外，宏定义在使用时只是简单的文本替换，并没有做严格的参数检查，也就不能享受C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。

1.3 使用注意事项

> 1.内联函数不能包括复杂的控制语句，如循环语句和switch语句；
>
> 2.内联函数不能包括复杂的控制语句，如循环语句和switch语句；
>
> 3.只将规模很小（一般5个语句一下）而使用频繁的函数声明为内联函数。在函数规模很小的情况下，函数调用的时间开销可能相当于甚至超过执行函数本身的时间，把它定义为内联函数，可大大减少程序运行时间。

总结

```
1.不能存在任何形式的循环语句
2.不能存在过多的条件判断语句
3.函数体不能过于庞大
4.不能对函数进行取址操作
```

1.4 内联函数的优缺点优点：

1.inline 定义的类的内联函数，函数的代码被放入符号表中，在使用时直接进行替换，（像宏一样展开），没有了调用的开销，效率也很高。

2.很明显，类的内联函数也是一个真正的函数，编译器在调用一个内联函数时，会首先检查它的参数的类型，保证调用正确。然后进行一系列的相关检查，就像对待任何一个真正的函数一样。这样就消除了它的隐患和局限性。（宏替换不会检查参数类型，安全隐患较大）

3.inline函数可以作为一个类的成员函数，与类的普通成员函数作用相同，可以访问一个类的私有成员和保护成员。内联函数可以用于替代一般的宏定义，最重要的应用在于类的存取函数的定义上面。

1.5 内联函数缺点：

1.内联函数具有一定的局限性，内联函数的函数体一般来说不能太大，

如果内联函数的函数体过大，一般的编译器会放弃内联方式，而采用普通的方式调用函数。(换句话说就是，你使用内联函数，只不过是向编译器提出一个申请，编译器可以拒绝你的申请）这样，内联函数就和普通函数执行效率一样了。

2.inline说明对编译器来说只是一种建议，编译器可以选择忽略这个建议。比如，你将一个长达1000多行的函数指定为inline，编译器就会忽略这个inline，将这个函数还原成普通函数，因此并不是说把一个函数定义为inline函数就一定会被编译器识别为内联函数，具体取决于编译器的实现和函数体的大小。

2 C++函数默认参数

> 函数可以为形参分配默认参数，这样当在函数调用中遗漏了实际参数时，默认参数将传递给形参。

2.1 函数重载写法

函数默认参数通常设置在函数原型中，示例如下：

```
void showArea(int a, int b, int c = 10, int d = 20 )
```

因为在函数原型中不需要形参名称，所以示例原型也可以这样声明：

```
void showArea(int a, int b, int= 0)
```

在这两种情况下，默认参数必须是常数值或常量，在它们前面有一个赋值运算符（=）。 请注意，在这两个示例原型中

test函数具有两个 int参数。第一个被赋给了默认参数 20.0，第二个被赋予默认参数 10.0。以下是函数的定义：

```
void showArea(double length, double width)
{
    double area = length * width;
    cout <<"The area is " << area << endl;
}
```

length 的默认实参为 20.0，width 的默认实参为 10.0。因为这两个形参都有默认实参，所以可以在函数调用中省略它们，如下所示：

```
showArea();
```

在该函数调用中，这两个默认实参将被传递给形参。形参 length 将接收值 20.0，width 将接收值 10.0。该函数的输出将是：

> The area is 200

默认实参仅在调用函数时省略实际参数的情况下使用。在下面的调用语句中，指定了第一个实参，而第二个实参则被忽略：

```
showArea(12.0);
```

值 12.0 将被传递给 length，而 width 则将被传递默认值 10.0。函数的输出将是：

> The area is 120

当然，所有的默认实参都可以被覆盖。在以下函数调用中，为两个形参都提供了实参：

```
showArea(12.0,5.5);
```

此函数调用的输出将为：

> The area is 66

注意，函数的默认实参应在函数名称最早出现时分配，这通常是函数原型。但是，如果一个函数没有原型，则可以在函数头中指定默认实参。例如，test函数可以定义如下：

```
void showArea(double length = 20.0, double width = 10.0)
{
    double area = length * width;
    cout << "The area is " << area << endl;
}
```

2.2 函数默认参数实例

下面的程序说明了默认函数实参的用法。它有一个在屏幕上显示星号的函数，该函数接收 2 个实参，指定要显示多少行星号，以及在每一行上打印多少个星号。提供的默认实参是显示 1 行 10 个星号：

```
//This program demonstrates the use of default function arguments.
#include <iostream>
```

```
using namespace std;
//Function prototype with default arguments
void displayStars(int starsPerRow = 10,int numRows = 1);
int main()
{
    displayStars(); // starsPerRow & numRows use defaults (10 & 1)
    cout << endl;
    displayStars (5); // starsPerRow 5. numRows uses default value 1
    cout << endl;
    displayStars (7, 3); // starsPerRow 7. numRows 3. No defaults used.
    return 0;
}
void displayStars(int starsPerRow, int numRows)
{
    for (int row = 1; row <= numRows; row++)
    {
        for (int star = 1; star <= starsPerRow; star++)
            cout << '*';
        cout << endl;
    }
}
```

程序输出结果：

```
**********
*****
*******
*******
*******
```

2.3默认参数注意事项

虽然 C++ 的默认实参非常方便，但它们在使用中并不完全是灵活的。当一个实参在一个函数调用中被遗漏时，它之后的所有实参也必须被省略。例如，上面程序的 displayStars 函数中，不可能只省略 starsPerRow 实参而不省略 numRows 实参，换句话说，以下函数调用是非法的：

```
displayStars ( , 3); // 非法函数调用
```

2.4 重载函数与默认参数

但是，使函数的某些形参有默认实参，而某些形参则没有，这是可能的。例如，在下面函数中，只有最后一个形参具有默认实参：

```
//函数原型
void calcPay(int empNum, double payRate, double hours = 40.0);
//函数calcPay的定义
void calcPay(int empNum, double payRate, double hours)
{
    double wages;
    wages = payRate * hours;
    cout << "Gross pay for employee number ";
    cout << empNum << " is " << wages << endl;
}
```

调用此函数时，必须始终为前 2 个形参（empNum 和 payRate）指定实参，因为它们没有默认实参，以下是有效调用的示例：

```
calcPay (769, 15.75);  // 使用 hours 的默认实参
calcPay (142, 12.00, 20); //指定 hours 数字
```

当函数使用了带默认实参和不带默认实参这两种混合的形参时，带默认实参的形参必须最后声明，因此以下原型是非法的：

```
//非法原型
void calcPay(int empNum, double hours = 40.0, double payRate);
//非法原型
void calcPay(double hours = 40.0, int empNum, double payRate);
```

以下是关于默认实参的重点总结：

> 1. 默认实参的值必须是常数值或命名常量。
> 2. 当在函数调用中遗漏了一个实参时（因为它有默认值），它后面的所有实参也必须 被省略。
> 3. 当函数使用了带默认实参和不带默认实参这两种混合的形参时，带默认实参的形参 必须最后声明。

2.5 占位参数的意义

在C++中可以为函数提供占位参数

1）占位参数只有参数类型声明，而没有参数名声明

2）一般情况下，在函数体内部无法使用占位参数

```
int func(int a, int b, int)
{
    return a+b;
}
int main(int argc, char *argv[])
{
    printf("fun(1,2,3) = %d\n", func(1,2,3));
    printf("Press enter to contiue ...");
    getchar();
return 0;
}

123456789101112
```

C++支持这样的函数占位参数有什么意义？？？

可以将占位参数与默认参数结合起来使用

意义：

1）为以后程序的扩展留下线索

2）兼容C语言程序中可能出现的不规范写法

```
int func(int a, int b, int = 0)
{
    return a+b;
}
int main(int argc, char *argv[])
{
    printf("func(1,2) = %d\n", func(1,2));

    printf("Press enter to continue ...");
    getchar();
return 0;
}
```

总结

> C++中在声明函数的时候指定参数的默认值
>
> C++可以声明占位符参数，占位符参数一般用于程序扩展和对C代码的兼容

3 函数重载

- **函数重载**

  函数重载是指在同一作用域内，可以有一组具有相同函数名，不同参数列表的函数，这组函数被称为重载函数。重载函数通常用来命名一组功能相似的函数，这样做减少了函数名的数量，避免了名字空间的污染，对于程序的可读性有很大的好处。

  1. 试想如果没有函数重载机制，如在C中，你必须要这样去做：为这个print函数取不同的名字，如print_int、print_string。这里还只是两个的情况，如果是很多个的话，就需要为实现同一个功能的函数取很多个名字，如加入打印long型、char*、各种类型的数组等等。这样做很不友好！
  2. 类的构造函数跟类名相同，也就是说：构造函数都同名。如果没有函数重载机制，要想实例化不同的对象，那是相当的麻烦！
  3. 操作符重载，本质上就是函数重载，它大大丰富了已有操作符的含义，方便使用，如+可用于连接字符串等！

  两个重载函数必须在下列一个或两个方面有所区别：

  > 1. 函数有不同参数。
  > 2. 函数有不同参数类型。

  返回值不作为判断两个函数区别的标志。

  C++运算符重载的相关规定如下：

  > 1. 不能改变运算符的优先级；
  > 2. 不能改变运算符的结合型；
  > 3. 默认参数不能和重载的运算符一起使用；
  > 4. 不能改变运算符的操作数的个数；
  > 5. 不能创建新的运算符，只有已有运算符可以被重载；
  > 6. 运算符作用于C++内部提供的数据类型时，原来含义保持不变。

  总结示例：

  （1）普通函数(非类成员函数)形参完全相同，返回值不同，如：

```
1 void print();
2 int print(); //不算重载，直接报错
```

（2）普通函数形参为非引用类型，非指针类型，形参一个带const，一个不带const

```
1 void print(int x);
2 void print(const int x); //不算重载，直接报错重定义
```

（3）普通函数形参为引用类型或指针类型，一个形参带const，一个不带const

```
1 void print(int *x);
2 void print(const int *x); //算重载，执行正确，实参为const int *时候调用这个，为int* 的时候调用上面一个
3
4 void print(int &x);
5 void print(const int &x); //算重载，执行正确，实参为const int &时候调用这个，为int& 的时候调用上面一个
```

（4）类的成员函数，形参完全相同，一个函数为const成员函数，一个函数为普通成员函数

```
1 void print();
2 void print() const; //算重载。const对象或const引用const指针调用时调用这个函数，普通对象或普通引用调用时调用上面一个。
```

函数指针

函数指针（或称为回调函数）是一个很有用也很重要的概念。当发生某种事件时，其它函数将会调用指定的函数指针指向的函数来处理特定的事件。

注意：定义的一个函数指针是一个变量。

定义一个函数指针：

两种格式：

1. 返回类型 (*函数指针名称)(参数类型,参数类型,参数类型，...);
2. 返回类型 （类名称::*函数成员名称）（参数类型，参数类型，参数类型，....）

示例：

```
1 int (*pFunction)(float,char,char)=NULL;//C语言的函数指针
2 int (MyClass::*pMemberFunction)(float,char,char)=NULL;//C++的函数指针，非静态函数成员
3 int (MyClass::*pConstMemberFunction)(float,char,char) const=NULL;//C++的函数指针，静态函数成员
```

C函数指针赋值和调用：

赋值：

```
1 int func1(float f,int a,int b){return f*a/b;}
2 int func2(float f,int a,int b){return f*a*b;}
3 //然后我们给函数指针pFunction赋值
4 pFunction=func1;
5 pFunction=&func2;
```

上面这段代码说明了两个问题：

1. 一个函数指针可以多次赋值。
2. 取地址符号是可选的，却是推荐使用的。

调用：

```
1 pFunction(10.0,'a','b');
2 (*pFunction)(10.0,'a','b');
```

C++类里的函数指针赋值和调用：

定义类：

```
 1 MyClass
 2 {
 3 public:
 4     int func1(float f,char a,char b)
 5     {
 6         return f*a*b;
 7     }
 8     int func2(float f,char a,char b) const
 9     {
10         return f*a/b;
11     }
12 }
```

赋值：

```
1 MyClass mc;
2 pMemberFunction= &mc.func1;//必须要加取地址符号
3 pConstMemberFunction = &mc.func2;
```

调用：

```
1 (mc.*pMemberFunction)(10.0,'a','b');
2 (mc.*pConstMemberFunction)(10.0,'a','b');
```

函数指针作为参数：

```
 1 #include<stdio.h>
 2
 3 float add(float a,float b){return a+b;}
 4
 5 float minus(float a,float b){return a-b;}
 6
 7 float multiply(float a,float b){return a*b;}
 8
 9 float divide(float a,float b){return a/b;}
10
11 int pass_func_pointer(float (*pFunction)(float a,float b))
```

```
12 {
13       float result=pFunction(10.0,12.0);
14       printf("result=%f\n",result);
15 }
16
17 int main()
18 {
19       pass_func_pointer(add);
20       pass_func_pointer(minus);
21       pass_func_pointer(multiply);
22       pass_func_pointer(divide);
23       return 0;
24 }
```

使用函数指针作为返回值：

对于以下形式：

```
1 float (* func(char op) ) (float ,float)
```

其具体含义就是，声明了这样一个函数：

1. 其名称为func，其参数的个数为1个；
2. 其各个参数的类型为：op—char；
3. 其返回变量（函数指针）类型为：float(*)(float,float)

4 宏定义防止多次导入

> 想必很多人都看过“头文件中的 #ifndef/#define/#endif 防止该头文件被重复引用”。
>
> 但是是否能理解“被重复引用”是什么意思？是不能在不同的两个文件中使用include来包含这个头文件吗？
>
> 如果头文件被重复引用了，会产生什么后果？是不是所有的头文件中都要加入 #ifndef/#define/#endif 这些代码？

4.1重复引用定义

定义： **被重复引用 是指一个头文件在同一个cpp文件中被include了多次**

这种错误常常是由于include 嵌套 造成的。比如：存在a.h文件代码中包含

> a.h

```
#include "c.h"
```

而此时b.cpp代码中导入了

> b.cpp

```
#include "a.h"
#include "c.h"
```

此时就会造成 c.h 重复引用

4.2头文件被重复引用引起的后果：

有些头文件重复引用只是增加了编译工作的工作量，不会引起太大的问题，

仅仅是编译效率低一些，但是对于大工程而言编译效率低下那将是一件多么痛苦的事情。

有些头文件重复包含，会引起错误，比如在头文件中定义了全局变量(虽然这种方式不被推荐，但确实是C规范允许的)这种会引起重复定义

是不是所有的头文件中都要加入

```
#ifndef

#define

#endif
```

这些代码?

答案：不是一定要加，但是不管怎样，

```
ifnde xxx

define xxx

#endif
```

或者其他方式避免头文件重复包含，只有好处没有坏处。个人觉得培养一个好的编程习惯是学习编程的一个重要分支。

下面给一个#ifndef/#define/#endif的格式：

> #ifndef A_H意思是"if not define a.h" 如果不存在a.h**
>
> **接着的语句应该#define A_H 就引入a.h**
>
> **最后一句应该写#endif 否则不需要引入**

5 this定义

> this 是 C++ 中的一个关键字，也是一个 const 指针，它指向当前对象，通过它可以访问当前对象的所有成员。

所谓当前对象，是指正在使用的对象。例如对于`stu.show();`，stu 就是当前对象，this 就指向 stu。

5.1this使用

下面是使用 this 的一个完整示例：

```
#include <iostream>
using namespace std;
class Student{
public:
    void setname(char *name);
    void setage(int age);
    void setscore(float score);
    void show();
private:
    char *name;
    int age;
    float score;
};
void Student::setname(char *name){
    this->name = name;
}
void Student::setage(int age){
    this->age = age;
}
void Student::setscore(float score){
    this->score = score;
}
void Student::show(){
    cout<<this->name<<"的年龄是"<<this->age<<"，成绩是"<<this->score<<endl;
}
int main(){
    Student *pstu = new Student;
    pstu -> setname("李华");
    pstu -> setage(16);
    pstu -> setscore(96.5);
    pstu -> show();
    return 0;
}
```

运行结果：
李华的年龄是16，成绩是96.5

this 只能用在类的内部，通过 this 可以访问类的所有成员，包括 private、protected、public 属性的。

本例中成员函数的参数和成员变量重名，只能通过 this 区分。以成员函数 `setname(char *name)` 为例，它的形参是 `name`，和成员变量 `name` 重名，如果写作 `name = name;` 这样的语句，就是给形参 `name` 赋值，而不是给成员变量 `name` 赋值。而写作 `this -> name = name;` 后，`=` 左边的 `name` 就是成员变量，右边的 `name` 就是形参，一目了然。

注意，this 是一个指针，要用 `->` 来访问成员变量或成员函数。

this 虽然用在类的内部，但是只有在对象被创建以后才会给 this 赋值，并且这个赋值的过程是编译器自动完成的，不需要用户干预，用户也不能显式地给 this 赋值。本例中，this 的值和 pstu 的值是相同的。

我们不妨来证明一下，给 Student 类添加一个成员函数 `printThis()`，专门用来输出 this 的值，如下所示：

```
void Student::printThis(){
    cout<<this<<endl;
}
```

然后在 main() 函数中创建对象并调用 printThis()：

```
Student *pstu1 = new Student;
pstu1 -> printThis();
cout<<pstu1<<endl;
Student *pstu2 = new Student;
pstu2 -> printThis();
cout<<pstu2<<endl;
```

运行结果

> 0x7b17d8
> 0x7b17d8
> 0x7b17f0
> 0x7b17f0

可以发现，this 确实指向了当前对象，而且对于不同的对象，this 的值也不一样。

几点注意：

> - this 是 const 指针，它的值是不能被修改的，一切企图修改该指针的操作，如赋值、递增、递减等都是不允许的。
> - this 只能在成员函数内部使用，用在其他地方没有意义，也是非法的。
> - 只有当对象被创建后 this 才有意义，因此不能在 static 成员函数中使用（后续会讲到 static 成员）。

5.2 this 到底是什么

this 实际上是成员函数的一个形参，在调用成员函数时将对象的地址作为实参传递给 this。不过 this 这个形参是隐式的，它并不出现在代码中，而是在编译阶段由编译器默默地将它添加到参数列表中。

this 作为隐式形参，本质上是成员函数的局部变量，所以只能用在成员函数的内部，并且只有在通过对象调用成员函数时才给 this 赋值。