第二章 C++对C的拓展

2.1 C++命名空间基本常识

1、创建名字是程序设计过程中一项最基本的活动，当一个项目很大时，它会不可避免地包含大量名字。c++允许我们对名字的产生和名字的可见性进行控制。

我们之前在学习c语言可以通过static关键字来使得名字只得在本编译单元内可见，在c++中我们将通过一种通过命名空间来控制对名字的访问。

2、在C++中，名称（name）可以是符号常量、变量、宏、函数、结构、枚举、类和对象等等。

3、std是c++标准命名空间，c++标准程序库中的所有标识符都被定义在std中，比如标准库中的类iostream、vector等都定义在该命名空间中，使用时要加上using声明(using namespace std) 或using指示(如std::cout)

2.2 命名空间的使用

2.2.1 命名空间的定义

```
namespace 名称

{

// 定义变量、函数、类型、对象...

}
```

2.2.2 命名空间成员的访问

- 使用作用域操作符::
- 空间名::成员

```
#include <iostream>

using namespace std;

namespace A{
        int a = 10;
}

namespace B{
        int a = 20;
}
```

```
int main(){
        cout << "A::a : " << A::a << endl;
        cout << "B::a : " << B::a << endl;
        return 0;
}
```

2.2.3 命名空间只能全局范围内定义（以下错误写法 ）

```
void test(){
        namespace A{
                int a = 10;
        }
        namespace B{
                int a = 20;
        }
        cout << "A::a : " << A::a << endl;
        cout << "B::a : " << B::a << endl;
}
```

2.2.4 命名空间的嵌套定义

```
namespace A{
        int a = 10;
        namespace B{
                int a = 20;
        }
}
void test(){
        cout << "A::a : " << A::a << endl;
        cout << "A::B::a : " << A::B::a << endl;
}
```

2.2.5 可以将命名空间的声明和实现分开

- 在test.h中声明命名空间

```
#ifndef TEST_TEST_H
#define TEST_TEST_H

namespace MySpace{
    void func1();
    void func2(int);
}

#endif //TEST_TEST_H
```

- 在test.cpp中实现命名空间

```
#include <iostream>
#include "test.h"

using namespace std;

void MySpace::func1(){
    cout << "MySpace::func1" << endl;
}
void MySpace::func2(int x){
    cout << "MySpace::func2 : " << x << endl;
}
```

- 在main.cpp中使用命名空间

```
#include <iostream>
#include "test.h"

using namespace std;

int main() {
    std::cout << "Hello, World!" << std::endl;

    MySpace::func1();
    MySpace::func2(100);

    return 0;
}
```

2.2.6 命名空间别名

```
namespace veryLongName{
        int a = 10;
        void func()
       { cout << "hello namespace" << endl; }
}

void test(){
        namespace shortName = veryLongName;
        cout << "veryLongName::a : " << shortName::a << endl;
        veryLongName::func();
        shortName::func();
}
```

2.3 引用

2.3.1 引用的概念

1、引用可以看作一个**已定义变量的别名**

2、引用的语法：**Type& name = var;**

> **注意：**
>
> 1、&在此不是求地址运算，而是起标识作用
>
> 2、类型标识符是指目标变量的类型
>
> 3、必须在声明引用变量时进行初始化
>
> 4、引用初始化之后不能改变
>
> 5、不能有NULL引用。必须确保引用是和一块合法的存储单元关联

2.3.2 引用的使用

```
void test01(){

        int a = 10;
        //给变量a取一个别名b
        int& b = a;
        cout << "a:" << a << endl;
        cout << "b:" << b << endl;
        cout << "------------" << endl;
        //操作b就相当于操作a本身
        b = 100;
        cout << "a:" << a << endl;
        cout << "b:" << b << endl;
        cout << "------------" << endl;
        //一个变量可以有n个别名
        int& c = a;
        c = 200;
        cout << "a:" << a << endl;
        cout << "b:" << b << endl;
        cout << "c:" << c << endl;
        cout << "------------" << endl;
        //a,b,c的地址都是相同的
        cout << "a:" << &a << endl;
        cout << "b:" << &b << endl;
        cout << "c:" << &c << endl;
}
//2. 使用引用注意事项
void test02(){
        //1) 引用必须初始化
        //int& ref; //报错:必须初始化引用
        //2) 引用一旦初始化，不能改变引用
        int a = 10;
        int b = 20;
        int& ref = a;
        ref = b; //不能改变引用
```

```
        cout << a << endl; //思考：为啥a的值改变了呢？   ref = b到底是什么意思呢？
}
```

2.3.3 引用的本质

1、引用的本质在c++内部实现是一个常指针.

```
Type& ref = val; // Type* const ref = &val;
```

2、c++编译器在编译过程中使用常指针作为引用的内部实现，因此**引用所占用的空间大小与指针相同**，只是这个过程是编译器内部实现，用户不可见。

```
        int& aRef = a; //自动转换为int* const aRef = &a;这也能说明引用为什么必须初始化
        aRef = 20; //内部发现aRef是引用，自动帮我们转换为: *aRef = 20;
        cout << "a:" << a << endl;
```

验证引用占用内存空间大小：

```
struct Teacher{
        int &a;
        int &b;
};
int main()
{
        cout << sizeof(Teacher)) << endl;

        return 0;
}
```

2.3.4 引用作为函数的参数

> 普通引用在声明时必须用其它的变量进行初始化，引用作为函数参数声明时不进行初始化

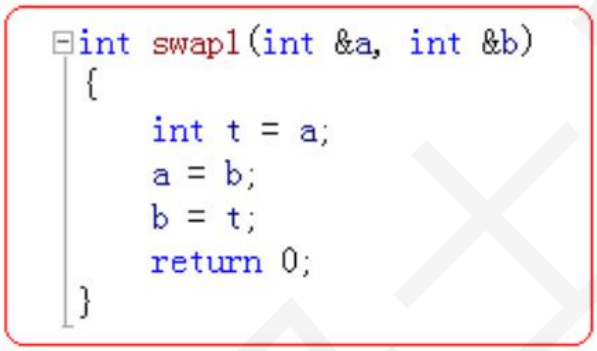
```
#include <iostream>

using namespace std;

typedef struct Teacher
{
        char name[64];
        int age ;
}Teacher;

void printfT(Teacher *pT)
{
        cout<<pT->age<<endl;
}

//pT是t1的别名 ,相当于修改了t1
```

```
void printfT2(Teacher &pT)
{
        //cout<<pT.age<<endl;
        pT.age = 33;
}

int main()
{
    Teacher t1;
    t1.age = 35;

    printfT(&t1);

    printfT2(t1); //pT是t1的别名

    return 0;
}
```

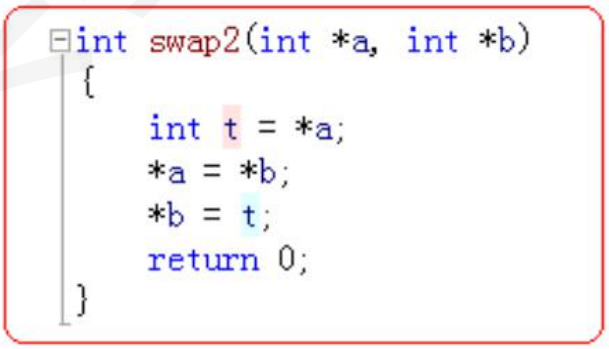
2.3.5 引用的意义

1）引用作为其它变量的别名而存在，因此在一些场合可以代替指针

2）引用相对于指针来说具有更好的可读性和实用性



```
int swap1(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
    return 0;
}
```



```
int swap2(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
    return 0;
}
```

2.3.6 引用的使用场景及实际应用

如果引用的对象是普通的数据类型的其实与指针差不多，引用在函数形参为对象的引用以及函数返回值为对象的引用上使用非常广泛，所以后面学习完类和对象后我们再详细讲解。

2.4 内联函数

2.4.1 内联函数的引入

1、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，这些都由预处理来完成。

2、下面我们来看看宏替代函数可能出现的问题

```
#define ADD(x,y) x+y
int main()
{
    int ret1 = ADD(10, 20) * 10; //希望的结果是300
    cout << "ret1:" << ret1 << endl; //210
    return 0;
}
```

为什么最终的执行结果是210而不是300呢？原因就是宏定义在预处理的时候是文本替换，所以ret1 = ADD(10, 20) * 10;被展开后就变成了：ret1 = 10+20*10

3、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).

4、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。

2.4.2 内联函数的基本概念

1、在c++中，预定义宏的概念是用内联函数来实现的，而**内联函数本身也是一个真正的函数** 。内联函数具有普通函数的所有行为。唯一不同之处在于内联函数会在适当的地方像预定义宏一样展开，所以不需要函数调用的开销。因此应该不使用宏，使用内联函数

2、在普通函数(非成员函数)函数前面加上inline关键字使之成为内联函数。但是必须注意必须函数体和声明结合在一起，否则编译器将它作为普通函数来对待。

```
inline void func(int a);
```

以上写法没有任何效果，仅仅是声明函数，应该如下方式来做:

```
inline int func(int a){return ++a;}
```

注意: 编译器将会检查函数参数列表使用是否正确，并返回值(进行必要的转换)。这些事预处理器无法完成的。

2.4.3 内联函数注意事项

1、C++中推荐使用内联函数替代宏代码片段

2、内联函数在最终生成的代码中是没有定义的，C++编译器直接将函数体插入在函数调用的地方，内联函数没有普通函数调用时的额外开销(压栈，跳转，返回)

3、因为内联函数在最终生成的代码中是没有定义的，所以内联函数的作用域可以理解为只在定义的文件内。假如在a.cpp中定义了inline int func(int a){return ++a;}，如果在b.c中需要调用func函数则在b.cpp中需要重新定义内联函数inline int func(int a){return ++a;}

4、inline只是对编译器的一个内联请求，c++内联编译会有一些限制，以下情况编译器可能考虑不会将函数进行内联编译：

> 存在任何形式的循环语句
>
> 存在过多的条件判断语句
>
> 函数体过于庞大
>
> 对函数进行取址操作

因此， **内联仅仅只是给编译器一个建议，编译器不一定会接受这种建议，如果你没有将函数声明为内联函数，那么编译器也可能将此函数做内联编译。一个好的编译器将会内联小的、简单的函数。**

2.5 函数的默认参数

1、c++在声明函数原型的时可为一个或者多个参数指定默认(缺省)的参数值，当函数调用的时候如果没有指定这个值，编译器会自动用默认值代替。

```
void TestFunc01(int a = 10, int b = 20){
        cout << "a + b  = " << a + b << endl;
}

int main(){
        //1.如果没有传参数，那么使用默认参数
        TestFunc01();
        //2. 如果传一个参数，那么第二个参数使用默认参数
        TestFunc01(100);
        //3. 如果传入两个参数，那么两个参数都使用我们传入的参数
        TestFunc01(100, 200);

        return EXIT_SUCCESS;
}
```

2、注意事项

> 函数的默认参数从左向右，如果一个参数设置了默认参数，那么这个参数之后的参数都必须设置默认参数。
>
> 如果函数声明和函数定义分开写，函数声明和函数定义不能同时设置默认参数

```
//注意点:
//1. 形参b设置默认参数值，那么后面位置的形参c也需要设置默认参数
void TestFunc02(int a,int b = 10,int c = 10){}
//2. 如果函数声明和函数定义分开，函数声明设置了默认参数，函数定义不能再设置默认参数
void TestFunc03(int a = 0,int b = 0);
void TestFunc03(int a, int b){}
```

2.6 函数重载

2.6.1 函数重载概念

函数重载(Function Overload)

> 用同一个函数名定义的不同函数
>
> 当函数名和不同的参数搭配时函数的含义不同

2.6.2 实现函数重载的条件

- 同一个作用域
- 参数个数不同
- 参数类型不同
- 参数顺序不同

```
void func(){}
void func(int x){}
void func(char x){}
void func(int x, int y){}
void func(int x, char y){}
void func(char y, int x){}
```

2.6.3 函数重载的实现原理

1、编译器在将我们的程序编译完成后会将变量和函数变成一个一个的符号，存放这些符号的表格我们称之为符号表

2、对程序进行编译查看函数对应的符号

- [ ] 编译命令：g++ -c main.cpp
- [ ] 执行命令：nm main.o 查看符号表

```
T _Z4funcc
T _Z4funcci
T _Z4funci
T _Z4funcic
T _Z4funcii
T _Z4funcv
```

我们发现g++编译器在将函数转换成符号时，根据函数名、形参类型进行转化的。

3、我有一个大胆的假设，在一个main.c 的文件中实现函数的重载，如果使用gcc 和 g++编译器编译，哪个会编译成功呢?