第六章 C++对C的拓展2

6.1 const详解

6.1.1 const 修饰普通变量

被修饰的对象是只读的

1. const int a; //a的值是只读的 int const a;
2. const int * p; 该语句表示指向整形常量 的指针，它指向的值不能修改。
3. int const * p; 该语句与b的含义相同，表示指向整形常量 的指针，它指向的值不能修改。
4. int * const p; 该语句表示指向整形的常量指针，它不能再指向别的变量，但指向（变量）的值可以修改。
5. const int *const p; 该语句表示指向整形常量 的常量指针 。它既不能再指向别的常量，指向的值也不能修改。
6. int const *const p; 表示指向整形常量 的常量指针 。它既不能再指向别的常量，指向的值也不能修改。

6.1.2 const修饰成员变量

const修饰类的成员变量，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。

```
class A
{
    const int nValue; //成员常量不能被修改
    A(int x): nValue(x) { } ; //只能在初始化列表中赋值
}
```

6.1.3 const 修饰类的成员函数

使用const修饰类的成员函数，该函数为类的**常成员函数，它不改变对象的成员变量。**

```
class B
{
        public:
                int x;
        public:
                void f() const
                {this->x++;} //increment of member 'B::x' in read-only object
};
```

注意：普通的全局函数不能用const修饰

```
void f() const //non-member function 'void f()' cannot have cv-qualifier
{}
```

6.1.4 const 修饰对象

1、const 修饰的对象为常量对象，其中的任何成员都不能被修改。

2、const 修饰的对象只能访问类中的const函数。

3、const 修饰的对象可以访问public成员变量，但是不能够修改。

```
class B
{
        public:
                int x;
        public:
                void f() const
                {}
                void f1(){};
};
int main(int argc, char** argv) {
        B b;
        const B p = b;
//      p.x = 100; // assignment of member 'B::x' in read-only object
//      p.f1();//passing 'const B' as 'this' argument of 'void B::f1()' discards qualifiers
        p.f();
        return 0;
}
```

6.1.5 const 修饰引用

```
void function(const TYPE& Var); //引用参数在函数内不可以改变
void function(const int& t)
{
    t++; //[Error] increment of read-only reference 't'
}

//引用指向某个常量时，需要用const修饰
//invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
int &a = 19;
const int &b = 19;
```

6.1.6 const 修饰函数返回值

const修饰函数返回值其实用的并不很多，它的含义和const修饰普通变量以及指针的含义基本相同。

- 函数返回值为普通变量和对象时使用const修饰没有意义，接收函数的返回值可以是const也可以不是，因为不论怎样都不可能通过函数返回值的接收者改变函数的返回值
- 函数返回值是被const修饰的指针，注意返回的地址不要为函数栈上的地址，接收函数的返回值必须用const修饰
- 函数返回值是被const修饰的引用时，接收函数的返回值可以是const也可以不是，如果接收函数的返回值是引用时必须用const修饰

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

//const 修饰成员变量
class A
{
public:
    const int x; //成员常量 只能在初始化成员列表中被赋值

    A():x(100)
    {}
    /*  A a;
     *  a.x = 1000; //错！！
     * */
};

//const 修饰成员函数
class B
{
public:
    int x;
    B():x(100){}

    int getX() const
    {
        return x;
    }
};

#if 0
//const 不能修饰全局函数
void func() const //error: non-member function 'void func()' cannot have cv-qualifier
{}
#endif

class C
{
public:
    int x;
    C():x(100)
    {}
```

```
    int getX() const
    {return x;}

    void setX(int x)
    {this->x  = x;}
};
```

6.2 extern "C"

6.2.1 extern "C"的含义

1、extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉**编译器，其声明的函数和变量可以在本模块或其它模块中使用。**

2、通常，在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字extern声明。例如，如果模块B欲引用该模块A中定义的全局变量和函数时只需包含模块A的头文件即可。这样，模块B中调用模块A中的函数时，在编译阶段，模块B虽然找不到该函数，但是并不会报错；它会在**链接阶段** 中从模块A编译生成的目标代码中找到此函数。

3、extern "C" 包含双重含义，从字面上即可得到：首先，被它修饰的目标是“extern”的；其次，被它修饰的目标是“C”的。

6.2.2 extern "C"的作用

1、我们来回顾一下c语言编译器gcc和c++编译器g++在编译一个函数的时候的处理方法

我们新建三个文件，分别命名为：example.h example.c main.cpp

“example.c”的内容如下：

```
#include "example.h"
int add( int x, int y )
{
        return x + y;
}
```

"example.h"的内容如下：

```
#ifndef C_EXAMPLE_H
#define C_EXAMPLE_H
extern int add(int x,int y);
#endif
```

“main.cpp"的内容如下：

```cpp
#include <iostream>

using namespace std;

#include "example.h"

int main(int argc, char* argv[])
{
        add(2,3);
        return 0;
}
```

接下来我们对这几个文件进行编译，分别执行如下命令：

> gcc -c example.c
> g++ -c main.cpp
> g++ main.o example.o

我们会发现编译报错：

```
main.o:main.cpp:(.text+0x1f): undefined reference to `add(int, int)'
collect2.exe: error: ld returned 1 exit status
```

这是为什么呢？我们来看下example.o 和 main.o中的符号表：

```
D:\untitled>nm example.o
0000000000000000 b .bss
0000000000000000 d .data
0000000000000000 p .pdata
0000000000000000 r .rdata$zzz
0000000000000000 t .text
0000000000000000 r .xdata
0000000000000000 T add
```

上图中标识出来的”add“是编译器gcc对函数add编译完后生成的符号

```
0000000000000085 t _GLOBAL__sub_I_main
                 U _Z3addii
0000000000000049 t _Z41__static_initialization_and_destruction_0ii
                 U _ZNSt8ios_base4InitC1Ev
                 U _ZNSt8ios_base4InitD1Ev
0000000000000000 r _ZStL19piecewise_construct
0000000000000000 b _ZStL8__ioinit
                 U atexit
```

上图中标识出来的"_Z3addii"是编译g++函数add编译完后生成的符号

通过查看example.o 和 main.o中的符号表我们发现g++编译器和gcc编译器在对同一函数编译时生成的符号表不一样，所以导致使用g++编译时会找不到add函数的声明（因为在编译的链接阶段，编译器首先会在main.o中查找_Z3addii，如果没有找到会去example.o中查找，但是在example.o中函数add被生成的符号时add，所以编译器就报错了）

2、在C++编写的源文件中使用extern "C"告诉编译器该文件中哪些函数是来外来的，并且告诉编译器这些函数需要使用”C语言“的规则进行编译。

将main.cpp修改成这样既可：

```cpp
#include <iostream>

using namespace std;
extern "C"
{
    #include "example.h"
}
int main(int argc, char* argv[])
{
        add(2,3);
        return 0;
}
```

或者将example.h修改成这样：

```cpp
#ifndef C_EXAMPLE_H
#define C_EXAMPLE_H

#ifdef __cplusplus
extern "C" {
#endif

int add(int, int);
#ifdef __cplusplus
}
```

```
#endif
#endif
```

这样，编译器在编译” add(2,3);“这条代码的时候，会用C语言的规则进行编译，这样生成的符号也叫做"add"，因为在example.o中存在"add"函数，所以编译能够通过。

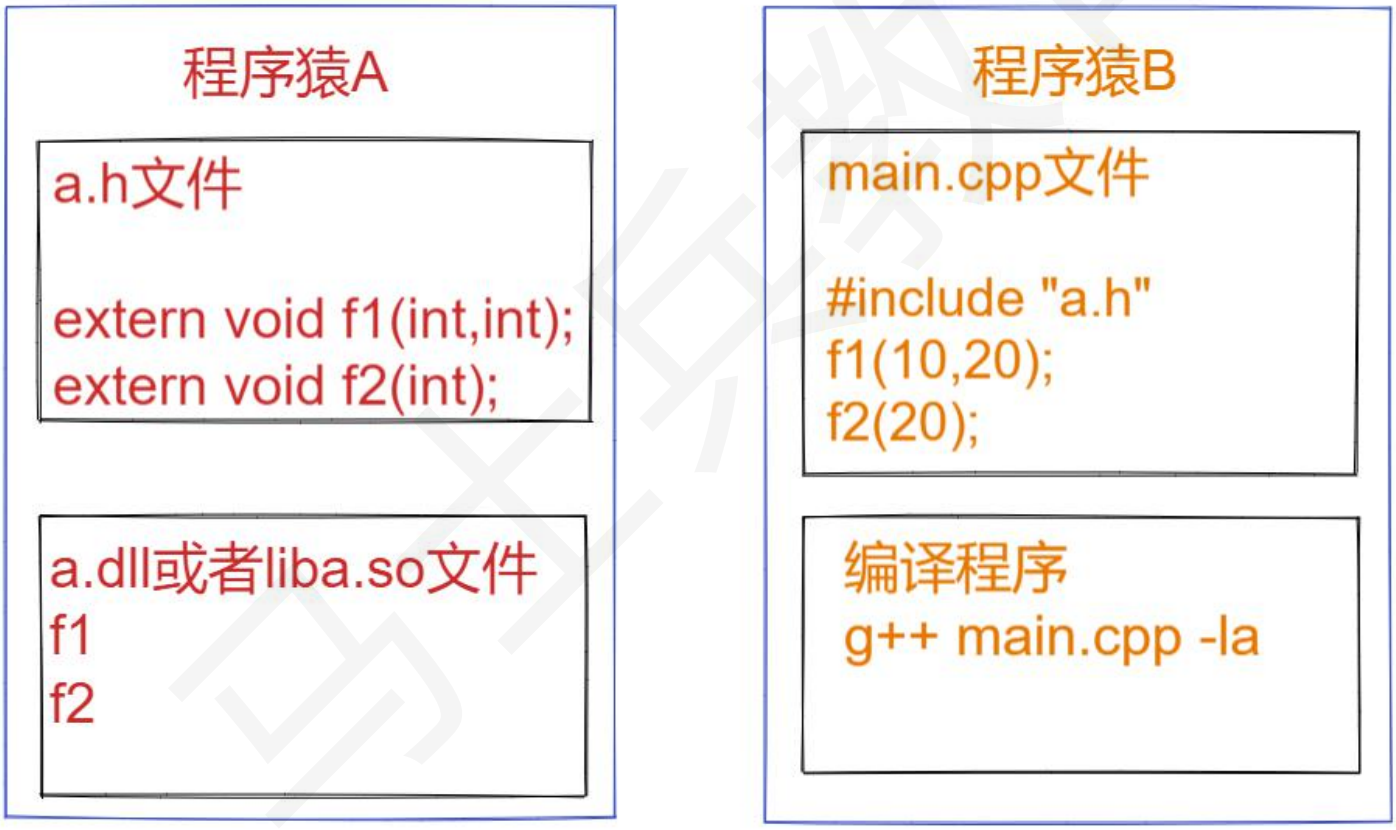
6.2.3 奇思妙想

1、既然都提供了example.c 文件了，如果我使用g++编译器编译example.c文件是不是就不会出现符号表不一致的情况了？

答：是的！！！

2、那这个extern "C"是不是就没用啦？

小伙，你还是太年轻了呀， 哈哈哈。在实际工作中，我们可能多人合作，如下图所示：



"程序猿A"将自己实现的f1和f2函数的源代码编程生成库文件(.dll是windows下动态库文件的后缀，.so文件为linux下动态库文件的后缀)

"程序员B"编写自己的程序，在程序中调用f1和f2两个函数，最后编译时链接"程序猿A"提供的a.dll或者liba.so文件，如上图所示，"程序猿B"在编译项目时是会编译失败的。怎么解决呢？请查看6.2.2章节。。。聪明的你找到答案了吗？

下面我们一起来实现一下吧。

第一步：在example.h中添加extern "C"

第二步：将example.c 使用gcc编译器编程成window下的动态库

> gcc example.c -fPIC -shared -o example.dll

第三步：使用g++编译器编译main.cpp，编译时链接example.dll库

> g++ main.cpp -L. -lexample -o main

解释：-L. 告诉编译器需要链接的库文件就在当前文件夹， “.”表示当前文件

-lexample 表示链接example.dll 动态库

6.3 nullptr

C++中为了避免“野指针”（即指针在首次使用之前没有进行初始化）的出现，我们声明一个指针后最好马上对其进行初始化操作。如果暂时不明确该指针指向哪个变量，则需要赋予NULL值。除了NULL之外，C++11新标准中又引入了nullptr来声明一个“空指针”，这样，我们就有下面三种方法来获取一个“空指针”：

```
int *p1 = NULL;
int *p2 = 0;
int *p3 = nullptr;
```

为什么C++11要引入nullptr？它与NULL相比又有什么不同呢？

C/C++中的NULL到底是什么呢？

1、NULL在C++中的定义，NULL在C++中被明确定义为整数0：

```
/* Define NULL pointer value */
#ifndef NULL
    #ifdef __cplusplus
        #define NULL    0
    #else  /* __cplusplus */
        #define NULL    ((void *)0)
    #endif  /* __cplusplus */
#endif  /* NULL */
```

2、NULL在C中的定义在C中，NULL通常被定义为如下：

```
#define NULL ((void *)0)
```

也就是说NULL实质上是一个void *指针。

那么问题又来了，我们从一开始学习C++的时候就被告诫C++是兼容C的，为什么对于NULL C++却不完全兼容C呢？C++之所以做出这样的选择，根本原因和C++的函数重载机制有关。考虑下面这段代码：

```
void Func(char *);
void Func(int);

int main()
{
    Func(NULL);
}
```

如果C++让NULL也支持void *的隐式类型转换，这样编译器就不知道应该调用哪一个函数。

为什么要引入nullptr

为了让c++传递空指针调用void Func(char *);传参时传 nullptr

```
void Func(char *);
void Func(int);

int main()
{
    Func(nullptr);
}
```

由于我们经常使用NULL表示空指针，所以从程序员的角度来看，Func（NULL）应该调用的是Func（char *）但实际上NULL的值是0，所以调用了Func（int）。nullptr关键字真是为了解决这个问题而引入的。

6.4 struct和class的区别

预知后事如何，请听下回分解