

实验二 C++对C的扩展 A

一、 问题描述

1. 实验目的：

掌握“C++对C扩展”中涉及的若干基本概念和特性，并能够应用于程序编写

掌握验证性实验的基本方法和过程(认知、实验、总结)

2. 实验内容：

分别编写一段测试代码来回答任务书中的相关问题（每一个问题，用一个工程文件，同时需要记录相应的调试过程），具体问题请参考“实验任务说明02.doc”；

调试的过程；（动态调试的相关截图，比如 设置断点、查看当前变量值等）；

编译出来的可执行程序单独放在一个目录下（bin/exe/debug目录下，同时附上输入数据说明和输出结果）

二、 具体实验

1. 程序阅读题

1.1、求结果，并上机验证；简要分析原因（关于域运算符）

```
#include <iostream>
using namespace std;
int value = 0;
void printvalue()
{
    cout << "Value=" << value;
}
int main()
{
    int value = 0;
    value = 1;
    cout << "Value=" << value << endl;
    ::value = 2;
    printvalue();
    return 0;
}
```

(1) 结果

Value=1, Value=2

(2) 上机验证



Microsoft Visual Studio 调试控制台

```
Value=1
Value=2
```

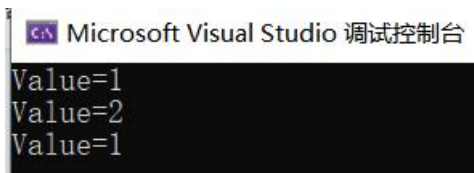
(3) 分析原因

全局变量可以被同名的局部变量所屏蔽。所以第一次输出的value为局部变量的值1。但是，可以使用域解析运算符 (::) 实现对该全局变量的访问，此时全局变量value的值变为2。再调用printvalue即可打印出全局变量Value=2。

(4) 验证拓展

::value=2是将全局变量value的值改为2，那么此时局部变量value的值应该还为1。此时对局部变量进行输出应该得到value=1的结果。上机实验后确实如此。

```
#include <iostream>
using namespace std;
int value = 0;
void printvalue()
{
    cout << "Value=" << value << endl;
}
int main()
{
    int value = 0;
    value = 1; // 更改局部变量
    cout << "Value=" << value << endl; // 打印局部变量
    ::value = 2; // 更改全局变量
    printvalue(); // 打印全局变量
    cout << "Value=" << value << endl; // 打印局部变量
    return 0;
}
```



Microsoft Visual Studio 调试控制台

```
Value=1
Value=2
Value=1
```

1.2、分析程序运行结果（关于引用）

```
#include <string>
using namespace std;
int main()
{
    int a = 10;
    int b = 20;
    int& rn = a; //r是a的引用
    int equal;
    rn = b; //a的值被改为b的值
    cout << "a = " << a << endl; //20
    cout << "b=" << b << endl; //20
    rn = 100; //a的值改为100
    cout << "a = " << a << endl; //100
    cout << "b = " << b << endl; //20
    equal = (&a == &rn) ? 1 : 0; //a的地址和rn地址相同
    cout << "equal = " << equal << endl; //1
    return 0;
}
```

(1) 结果

a=20

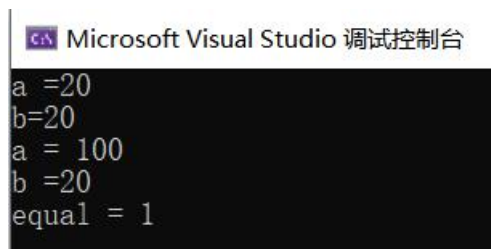
b=20

a = 100

b =20

equal = 1

(2) 上机验证



Microsoft Visual Studio 调试控制台

```
a =20
b=20
a = 100
b =20
equal = 1
```

(3) 分析原因

对一个变量的“引用”的所有操作，实际上是对其所绑定的原变量的操作。程序中rn是a的引用，rn和a的地址相同，当rn的值改变时a的值也将改变，所以虽然只是改变rn的值，最后输出a的值也发生了变化。

1.3、分析程序结果，思考运行过程及内存关系

```
1 //函数返回值为普通类型和返回值为引用之间的差别
2 #include <iostream>
3 using namespace std;
4 float temp;
5 float fn1(float r) {
6     temp = r * r * 3.14;
7     return temp;
8 }
9 float& fn2(float r) {
10     temp = r * r * 3.14;
11     return temp;
12 }
13 void main() {
14     float a = fn1(5.0); //a地址与temp不同
15     //float& b = (fn1(5.0));非常量引用的初始值必须为左值
16     float c = fn2(5.0); //c地址与temp不同
17     float& d = fn2(5.0); //d地址与temp地址相同
18     cout << a << endl;
19     //cout << b << endl;
20     cout << c << endl;
21     cout << d << endl;
22 }
```

(1) 结果

输出a,c,d为78.5。

(2) 上机验证

```
float& b = fn1(5.0);
float c = fn1(5.0);
float& d = fn1(5.0);
cout << a << endl;
cout << b << endl;
cout << c << endl;
cout << d << endl;
```

D:\XPfile\学习资料\年级分类\大二下\课程资料\c++\myexp\第一周2\

78.5
78.5
78.5

temp	78.500000
&temp	0x0070a140 (1.3.exe!float temp) (78.5000000)
a	78.500000
&a	0x00affadc (78.5000000)
b	78.500000
&b	0x00affac4 (78.5000000)
c	78.500000
&c	0x00affab8 (78.5000000)
d	78.500000
&d	0x0070a140 (1.3.exe!float temp) (78.5000000)

添加要监视的项

(3) 思考

函数返回值为普通类型时，要生成一个值的副本，而引用返回值时，不生成值的副本。这里fn1函数返回值为普通类型，将temp的值78.5赋给了a。但是对引用b初始化时VS2019报错“非常量引用的初始值必须为左值”，这里如果要引用初始化，可以改为常量引用。

而fn2的返回值类型为引用。用它给int型变量c赋值时，c被赋值为78.5，但其地址是另外申请的，temp和c的地址不同。而d由于是引用，它是temp的别名，所以d与temp共用一块内存。

(4) 拓展实验

VS中对b报错“非常量引用的初始值必须为左值”，改为const int&b即常量引用后代码运行成功，这里b并不是对temp的引用，而是绑定常量的引用，故b与temp地址也不同。

```
1 //函数返回值为普通类型和返回值为引用之间的差别
2 #include <iostream>
3 using namespace std;
4 float temp;
5 float fn1(float r) {
6     temp = r * r * 3.14;
7     return temp;
8 }
9 float& fn2(float r) {
10     temp = r * r * 3.14;
11     return temp;
12 }
13 void main() {
14     float a = fn1(5.0); //a地址与temp不同
15     //float& b = fn1(5.0);非常量引用的初始值必须为左值
16     const float& b = fn1(5.0); //常量引用
17     float c = fn2(5.0); //c地址与temp不同
18     float& d = fn2(5.0); //d地址与temp地址相同
19     cout << a << endl;
20     cout << b << endl; 已用时间 <= 1ms
21     cout << c << endl;
22     cout << d << endl;
23 }
```

Microsoft Visual Studio 调试控制台

```
78.5
78.5
78.5
78.5
```

1.4、运行程序，分析程序结果（考察：字面常量）

```

#include <iostream>
using namespace std;
int fun(char* s);
int main() {
    cout << fun("Hello");
    return 0;
}
int fun(char* s) {
    *s = 'h';
    return *s;
}

```

(1) 结果

运行失败

(2) 上机验证

```

("Hello");
) {

```

▢ (const char [6])"Hello"

[联机搜索](#)

"const char *" 类型的实参与 "char *" 类型的形参不兼容

[联机搜索](#)

(3) 分析

字面常量是通过直接写出常量值使用的常量。“Hello”是字符串常量，const char*类型的实参与char*类型的形参不兼容。又由于它是字符串常量，值无法更改，最后程序运行失败。

(4) 实验拓展

基于以上分析，可以修改代码，用字符串数组表示“Hello”，可以顺利运行程序，并将字符串改为hello。

```

#include <iostream>
using namespace std;
char* fun(char* s);
char a[] = "Hello";
int main() {
    cout << fun(a) << endl; //用字符串数组代替字面常量储存hello
    cout << a; //经过传址调用, a中储存的字符串变为hello
    return 0;
}
char* fun(char* s) {
    *s = 'h';
    return s;
}

```

Microsoft Visual Studio 调试控制台

```

hello
hello

```

1.5、运行程序，分析程序结果（考察：const在参数传递中更彰显其价值）

```

#include <iostream>
using namespace std;
int f(const int* p)
{
    int y;
    y = (*p) * 2;
    (*p)++;
    return y;
}
int main() {
    int x;
    x = 10;
    cout << x + f(&x) << endl;
    return 0;
}

```

（1）运行结果

报错

（2）上机验证

```

    (*p)++;
    re
}
int main

```

表达式必须是可修改的左值

联机搜索

(3) 思考

p指向一个整型，不可通过该指针改变其指向的内容，但可改变指针本身所指向的地址。所以这里，试图让p指向的内容的值加一失败了。若仅是想让子函数使用传递给它的信息，不希望对源头（实参信息）进行修改，则可以使用常量引用。

(4) 代码修改

这里去掉const，程序可以正常运行，同时x的值被改变成了11。输出结果为31。

```
#include <iostream>
using namespace std;
int f(int* p) //去掉const, 这时可以对p指向的值进行修改
{
    int y;
    y = (*p) * 2;
    (*p)++;
    return y;
}
int main() {
    int x;
    x = 10;
    cout << x + f(&x) << endl;
    return 0;
}
```

The screenshot shows the Visual Studio IDE with the code from the previous block. The variable `x` is being monitored. The initial value of `x` is 10. After the function `f` is called, the value of `x` changes to 11. The output of the program is 31.

名称	值
x	10

添加要监视的项

13 | `cout << x + f(&x) << endl;` 已用时间 <= 1ms

14 | `return 0;`

15 | `}`

名称	值
x	11

添加要监视的项

Microsoft Visual Studio 调试控制台

31

2. 程序阅读题

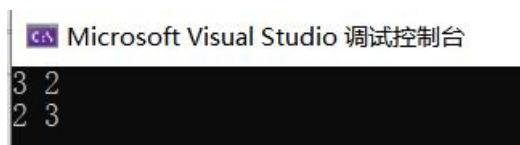
2.1 分别用指针和引用写程序，程序中包含一个子函数Myswap()，该函数能交换两个实参变量的值。

(1) 代码内容

```
#include <iostream>
using namespace std;
//指针作为函数参数
void Myswap1(int* x, int* y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
//引用作为函数参数
void Myswap2(int& x, int& y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}
//主函数
int main() {
    int a = 2, b = 3;
    Myswap1(&a, &b); //指针
    cout << a << " " << b << endl;
    Myswap2(a, b); //引用
    cout << a << " " << b << endl;
    return 0;
}
```

(3) 运行结果

都能实现实参值的交换。



```
CA Microsoft Visual Studio 调试控制台
3 2
2 3
```

(3) 思考

- ① 指针作为函数参数时，在Myswap1函数中，*x和*y表示的是x和y指向的空间（主函数中的空间）。在myswap1函数结束时，指针变量x和y自动回收（生命周期结束），但是“交换”的结果保留在main函数中。
- ② 引用作为函数参数时，通过参数传递，实现变量与引用的绑定。在被调用函数中通过引用来改变调用函数的实参的值。函数

能访问调用函数中的原始数据，而不是副本。

- ③ 两者比较：指针变量需要额外开辟内存单元，而引用不是一个独立的变量；在引用调用中，引用对应了绑定的变量，访问时无须类似“*”；形式上，引用调用中实参就是变量，而不是指针变量或是地址。

2.2 声明一个复数结构体类型struct Complex，

（一）编写函数：

- 1、set_value(): 该函数可实现对复数值的设定；
- 2、display(): 该函数可实现对复数的输出；
- 3、void addi1(): 实现两个复数的和；
- 4、struct Complex addi2(): 实现两个复数的和；
- 5、struct Complex & addi3(): 实现两个复数的和；

（1）实验代码

```
#include <iostream>
using namespace std;
//复数结构体类型
struct Complex{
    double real;
    double image;
};
//该函数可实现对复数值的设定
void set_value(Complex& ft,double real,double image) {
    ft.real = real;
    ft.image = image;
}
//该函数可实现对复数的输出
void display(const Complex& ft) {
    cout << ft.real << ft.image;
}
//实现两个复数的和
void addi1(Complex& target, const Complex& source1,const Complex& source2) {
    target.real = source1.real+ source2.real;
    target.image = source1.image+ source2.image;
}
//实现两个复数的和
struct Complex addi2(Complex target, const Complex& source1, const Complex& source2){
    target.real = source1.real + source2.real;
    target.image = source1.image + source2.image;
    return target;
}
```

```

//实现两个复数的和，结果储存在target中
struct Complex& addi3(Complex& target, const Complex& source1, const Complex& source2) {
    target.real = source1.real + source2.real;
    target.image = source1.image + source2.image;
    return target;
}

//主函数
int main() {
    set_value(num1, 3.1, 4.2); //设定复数num1的值
    set_value(num2, 2.1, 1.2); //设定复数num2的值
    set_value(test, 1, 1);
    cout << "num1=" << num1.real << "+" << num1.image << "i" << endl;
    cout << "num2=" << num2.real << "+" << num2.image << "i" << endl;

    addi1(ans1, num1, num2); //调用addi1函数
    cout << "ans1=" << ans1.real << "+" << ans1.image << "i" << endl;

    ans2=addi2(ans2, num1, num2); //调用addi2函数
    //addi2(ans2, num1, num2) = test; //验证该函数不可以作为左值。
    cout << "ans2=" << ans2.real << "+" << ans2.image << "i" << endl;

    addi3(ans3, num1, num2); //调用addi3函数
    //addi3(ans3, num1, num2) = test; //验证该函数可以作为左值。
    cout << "ans3=" << ans3.real << "+" << ans3.image << "i" << endl;
    return 0;
}

```

(2) 运行结果



```

Microsoft Visual Studio 调试控制台
num1=3.1+4.2i
num2=2.1+1.2i
ans1=5.2+5.4i
ans2=5.2+5.4i
ans3=5.2+5.4i

```

(二) 重点分析:

1、在set_value()的参数传递应该采用哪种参数传递的方式？并分析原因。

答：在set_value()的参数传递中使用了引用传递的方式。原因如下：

①因为给复数赋值需要在函数中修改实参的值，需要使用引用或者指针。

②引用作为参数可以提高效率；像指针那样工作，而使用方式如一般变量，具有更好的可读性和直观性，比指针更有优势。

2、在display()形参中使用const &的优点

答：由于display()函数只是用于输出复数，不需修改变量的值。使用const &可以让子函数使用传递给它的信息，同时避免了对源头（实参信息）进行修改。

3、addi1()和addi2()函数功能目标一样，返回值不一样。从返回值视角分析，addi2()有什么特性？

答：addi1()不返回值。而addi2()返回Complex结构体。可以将结构体变量传递给函数并以与普通自变量类似的方式返回，它可以作为右值赋给其他结构体。

4、addi2()和addi3()函数功能目标一样，返回值不一样。从返回值视角分析，addi3()有什么特性？

答：addi3()返回Complex结构体引用。add2()返回值时，要生成一个值的副本，而add3()返回值时，不生成值的副本。同时一个返回引用的调用函数可以做左值。

5、分析并验证，addi2()函数是否能作为左值？

答：addi2()函数返回的是结构体，不能作为左值。上机验证，ans2未被赋值成功，因为其实只是对返回的临时变量进行赋值，而没有对ans2赋值。

```
addi2(ans2, num1, num2) = test; //验证该函数不可以作为左值。
cout << "ans2=" << ans2.real << "+" << ans2.image << "i" << endl;
```



6、验证addi3()函数是否能作为左值？分析代码的执行过程。

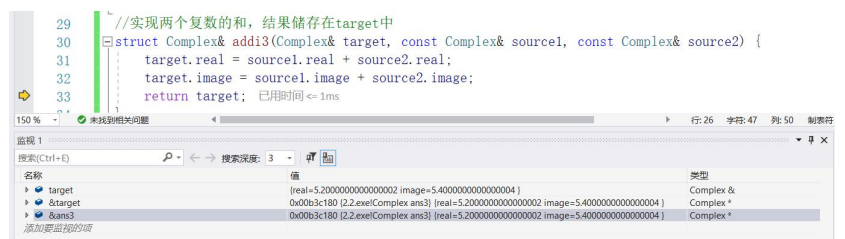
答：addi3()函数可以作为左值。addi3()返回的是引用，一个返回引用的调用函数可以做左值。

如下图所示，addi3()函数返回值target是对ans3的引用。两者地址是一致的，共用一块内存。用结构体test给函数赋值即是给ans3赋值。ans3变为了test的值，即输出1+1i。

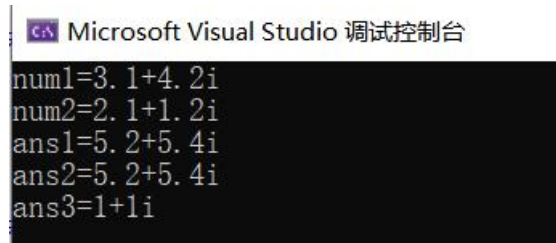
(1) 验证代码

```
46 //addi3(ans3, num1, num2); //调用addi3函数
47 addi3(ans3, num1, num2) = test; //验证该函数可以作为左值。
```

(2) 调试过程



(3) 实验结果



三、 附录

源程序文件项目清单: 1.1 1.2 1.3 1.4 1.5 2.1 2.2