

深入理解C++类和对象（中）

如果一个类里面什么也不写，那么它里面真的什么都没有吗？

答案是否定的，如果一个类是空类，那么编译器会为我们提供6个默认成员函数，分别为构造函数，析构函数拷贝构造函数，赋值运算符重载函数，&操作符重载，const修饰的取地址操作符重载，那么接下来本篇文章将带你来理解这6个默认成员函数

1.构造函数

构造函数是一个特殊的成员函数，类名与函数名相同，主要完成对对象的初始化

特点：

- 1.函数名与类名相同
- 2.没有返回值，也不写void
- 3.对象实例化时编译器会自动调用对应的构造函数
- 4.构造函数可以重载
- 5.如果类中没有显式定义构造函数，则C++编译器会自动生成一个默认的构造函数，一旦用户显式定义了编译器就不会自动生成

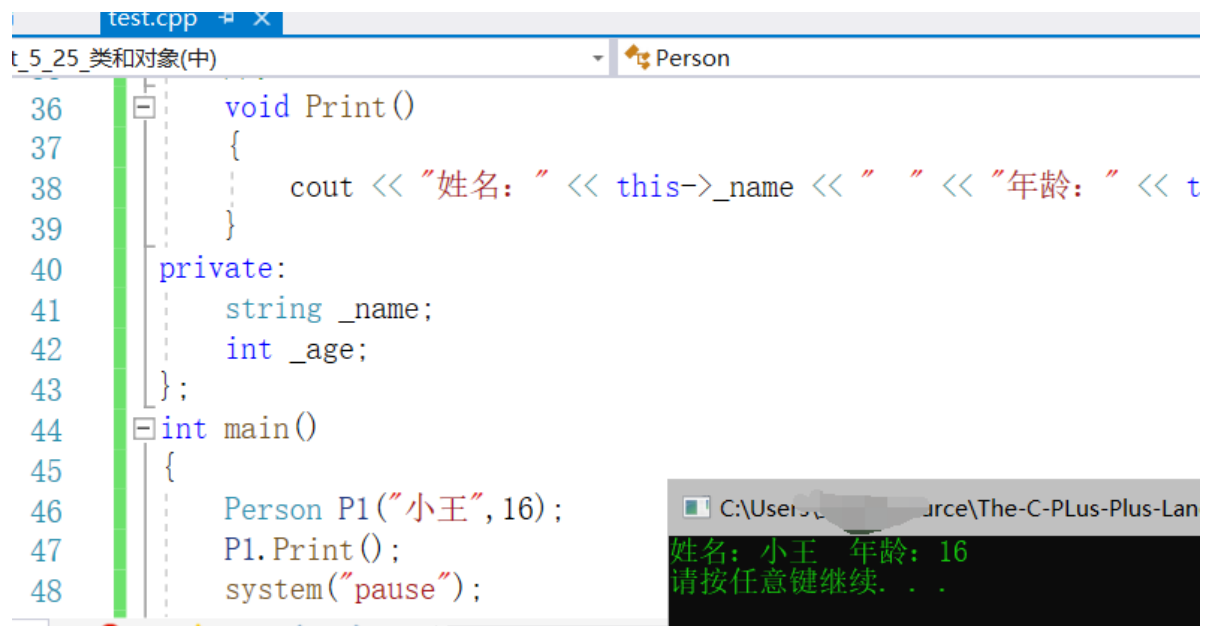
```
#include<iostream>
using namespace std;
class Person
{
public:
    //1. 编译器提供的默认构造函数，什么都不做
    Person()
    {

    }

    //2. 带参数的默认构造函数
    Person(string name,int age )
    {
        this->_name = name;
        this->_age = age;
    }
    void Print()
    {
        cout << "姓名: " << this->_name << " " << "年龄: " << this->_age << endl;
    }
private:
    string _name;
    int _age;
};
int main()
{
    Person P1("小王",16);
    P1.Print();
    Person P2;
    P2.Print();
    system("pause");
}
```

```
    return 0;
}
```

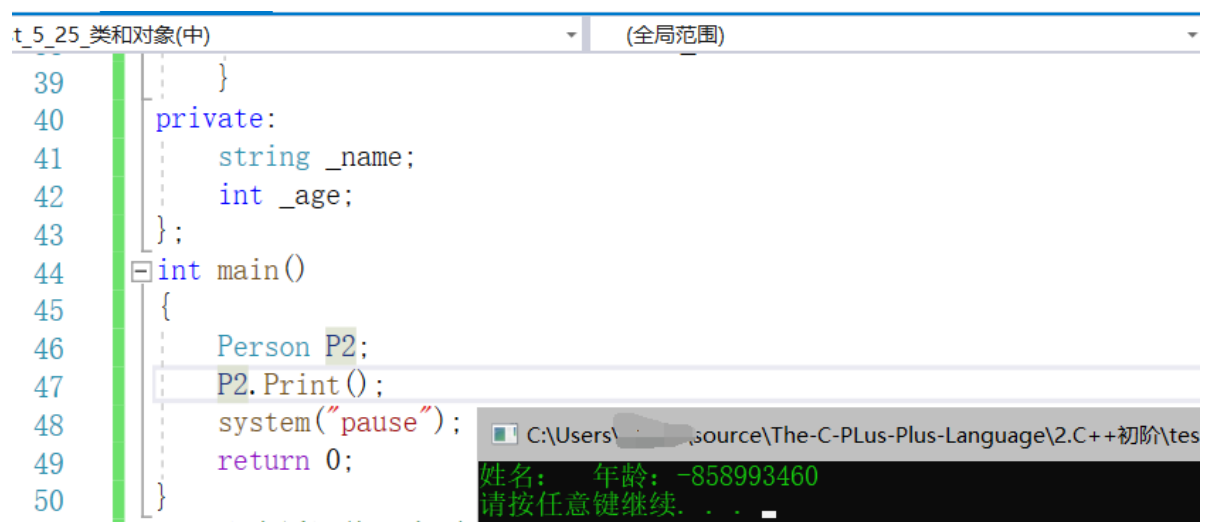
如上所示，P1调用的即为我们写的构造函数，它能够对对象进行初始化，结果如下



```
test.cpp x
t_5_25_类和对象(中) Person
36 void Print()
37 {
38     cout << "姓名: " << this->_name << " " << "年龄: " << t
39 }
40 private:
41     string _name;
42     int _age;
43 };
44 int main()
45 {
46     Person P1("小王", 16);
47     P1.Print();
48     system("pause");
}
```

C:\Users\...source\The-C-Plus-Plus-Lan
姓名: 小王 年龄: 16
请按任意键继续. . .

当我们调用默认的构造函数时，结果如下：



```
t_5_25_类和对象(中) (全局范围)
39 }
40 private:
41     string _name;
42     int _age;
43 };
44 int main()
45 {
46     Person P2;
47     P2.Print();
48     system("pause");
49     return 0;
50 }
```

C:\Users\...source\The-C-Plus-Plus-Language\2.C++初阶\tes
姓名: 年龄: -858993460
请按任意键继续. . .

编译器提供的默认构造函数里面什么都没有，注意调用默认构造函数时写成：P2,不要写成P2(),编译器会将P2()当成函数的声明

当然关于构造函数的书写，我们更加推荐下面这种写法：写成缺省函数，这样在我们为其赋值时就会方便很多

```
//3.全缺省的默认构造函数
Person(string name = "小王", int age = 16)
{
    this->_name = name;
    this->_age = age;
}
```

Ps:无参的构造函数和全缺省的构造函数都称为默认构造函数，标签默认构造函数只能有一个。注意：无参构造函数，全缺省构造函数，我们没写编译器默认生成的构造函数，都可以认为是默认成员函数

2.析构函数

在对象销毁时自动调用析构函数，完成对类的一些资源的清理工作

特点：

- 1.析构函数名是在类名前面加上字符~
- 2.无参数，没有返回值，不写void
- 3.一个类有且只有一个析构函数，系统会自动生成默认的析构函数
- 4.对象生命周期结束时，C++编译系统自动调用析构函数
- 5.如果有成员变量是在堆区创建出来的，我们要在析构函数中手动释放

例如：

```
//默认构造函数
```

```
~Person()  
{  
  
}
```

```
#include<iostream>  
using namespace std;  
class Person  
{  
public:  
    Person(string name = "小王", int age = 16)  
    {  
        this->_name = name;  
        this->_age = age;  
    }  
    void Print()  
    {  
        cout << "姓名: " << this->_name << " " << "年龄: " << this->_age << endl;  
    }  
    ~Person()  
    {  
        cout << "person析构函数的调用" << endl;  
    }  
private:  
    string _name;  
    int _age;  
};  
int main()  
{  
    Person P2("小王",16);  
    P2.Print();  
    system("pause");  
    return 0;  
}
```

效果如下：

```
25 类和对象(中) Person
24 void Print()
25 {
26     cout << "姓名: " << this->_name << " " << "年龄: " << this->_age << endl;
27 }
28 ~Person()
29 {
30     cout << "person析构函数的调用" << endl;
31 }
32 private:
33     string _name;
34     int _age;
35 };
36 int main()
37 {
38     Person P2("小王", 16);
39     P2.Print();
40     system("pause");
41     return 0;
42 }
```

Microsoft Visual Studio 调试控制台

```
姓名: 小王 年龄: 16
请按任意键继续. . .
person析构函数的调用
```

在这种在栈上定义的变量析构函数可能没什么用，但是要是该对象中有变量在堆区被malloc/new出来，很多人会忘记最后在函数结束时free/delete，这就造成了内存泄漏问题，但是有了析构函数，我们就可以将free或者delete函数写到析构函数中去，然后在程序结束的时候编译器会自动free/delete掉这块空间，就会很方便

```
#include<iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        arr = (int*)malloc(sizeof(int) * 1);
    }
    ~Person()
    {
        free(arr);
        cout << "Person的析构函数的调用" << endl;
    }
    int* arr;
};
int main()
{
    Person p;
    return 0;
}
```

```

16  class Person
17  {
18  public:
19      Person()
20      {
21          arr = (int*)malloc(sizeof(int) * 1);
22      }
23      ~Person()
24      {
25      }
26          free(arr);
27          cout << "Person的析构函数的调用" << endl;
28      }
29      int* arr;
30  };
31  int main()
32  {
33      Person p;
34      return 0;

```

Microsoft Visual Studio 调试控制台

Person的析构函数的调用

C:\Users\...\source\The-C-Plus-Plus-Language\18572)已退出，代码为 0。

要在调试停止时自动关闭控制台，请启用“工具”->“

如上所示我们在堆区开辟成员变量，到结束时析构函数自动释放

3.拷贝构造函数

完成对一个对象的拷贝，例如：

```

//系统提供的拷贝构造函数
#include<iostream>
using namespace std;
class Person
{
public:
    Person(int a)
    {
        this->_a = a;
    }
    void Printf()
    {
        cout << "_a=" << this->_a << endl;
    }
private:
    int _a;
};
int main()
{
    Person P1(10);
    P1.Printf();
    Person P2(P1);
    P2.Printf();
    return 0;
}

```

test_5_25_类和对象(中) (全局范围)

```
1 //系统提供的拷贝构造函数
2 #include<iostream>
3 using namespace std;
4 class Person
5 {
6 public:
7     Person(int a)
8     {
9         this->_a = a;
10    }
11    void Printf()
12    {
13        cout << "_a=" << this->_a << endl;
14    }
15 private:
16     int _a;
17 };
18 int main()
19 {
20     Person P1(10);
21     P1.Printf();
22     Person P2(P1);
23     P2.Printf();
24     return 0;
25 }
```

Microsoft Visual Studio 调试控制台

```
_a=10
_a=10
```

01 % 未找到相关问题

如上所示，我们定义了一个person类，创建了P1，P2两个对象，并将P1->a初始化为10，然后我们写了P2(P1)，然后打印P2->a发现也变成了10，这其实是调用了拷贝构造函数，下面我们先来了解一下什么是拷贝构造函数：

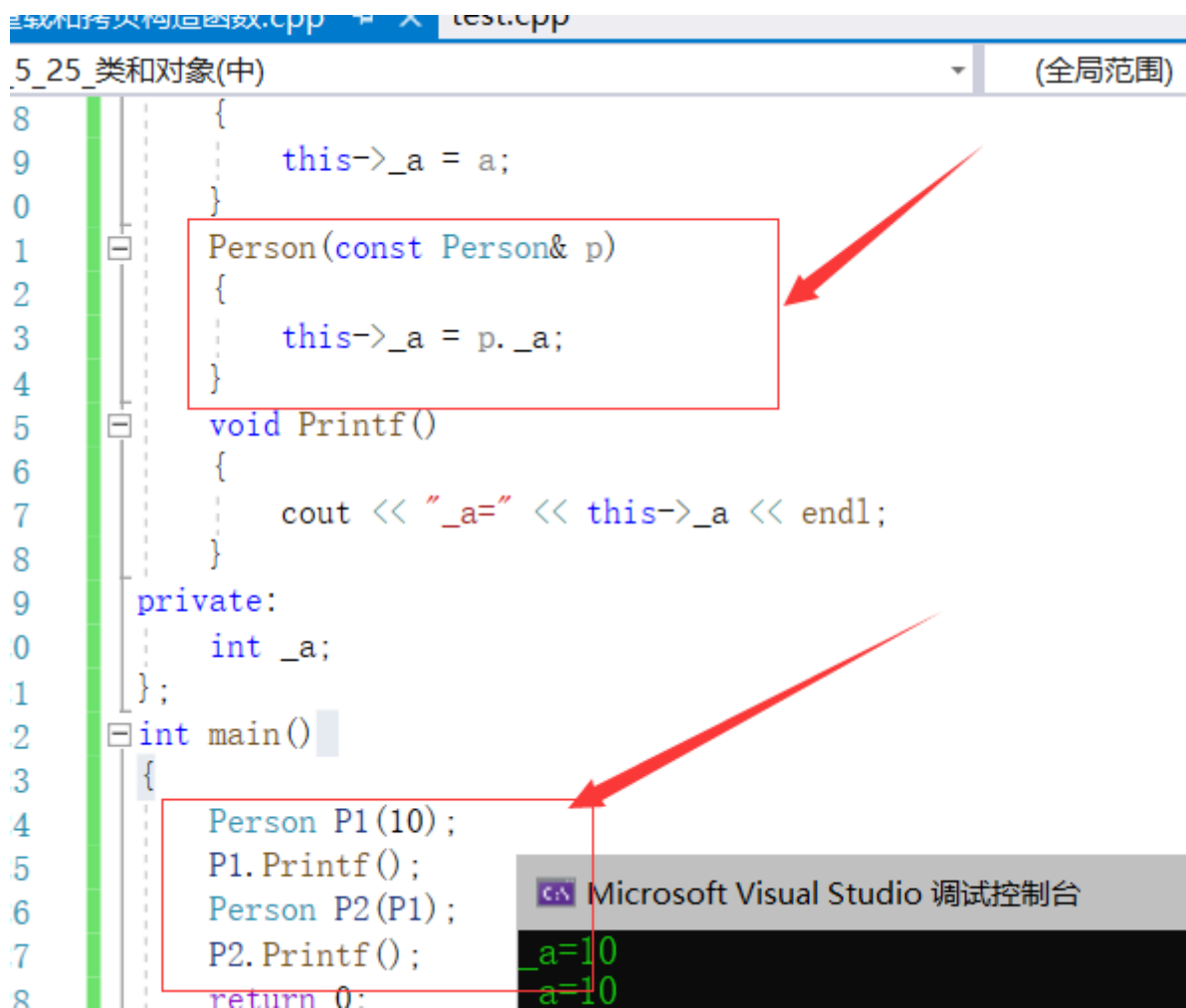
- 1.拷贝构造函数是构造函数的一个重载形式
- 2.拷贝构造函数的参数有且只有一个必须使用引用传参，使用传值方式会引发无穷递归调用
- 3.若显示定义，系统生成默认的拷贝构造函数，默认的拷贝构造函数对象按内存存储字节序完成拷贝，这种拷贝我们称为浅拷贝，或者值拷贝
- 4.编译器提供的构造函数是按照字节序进行拷贝的，因此又叫做值拷贝或者浅拷贝

当我们写下Person P2(P1)时，编译器调用了P2.Person(P1)，以上面例子为例，编译器提供的默认拷贝构造函数如下所示：

```
Person(const Person& p)
{
    this->_a = p._a;
}
```

运行结果如下所示：

```
5_25_类和对象(中) (全局范围)
8 {
9     this->_a = a;
10 }
11 Person(const Person& p)
12 {
13     this->_a = p._a;
14 }
15 void Printf()
16 {
17     cout << "_a=" << this->_a << endl;
18 }
19 private:
20     int _a;
21 };
22 int main()
23 {
24     Person P1(10);
25     P1.Printf();
26     Person P2(P1);
27     P2.Printf();
28     return 0;
}
```



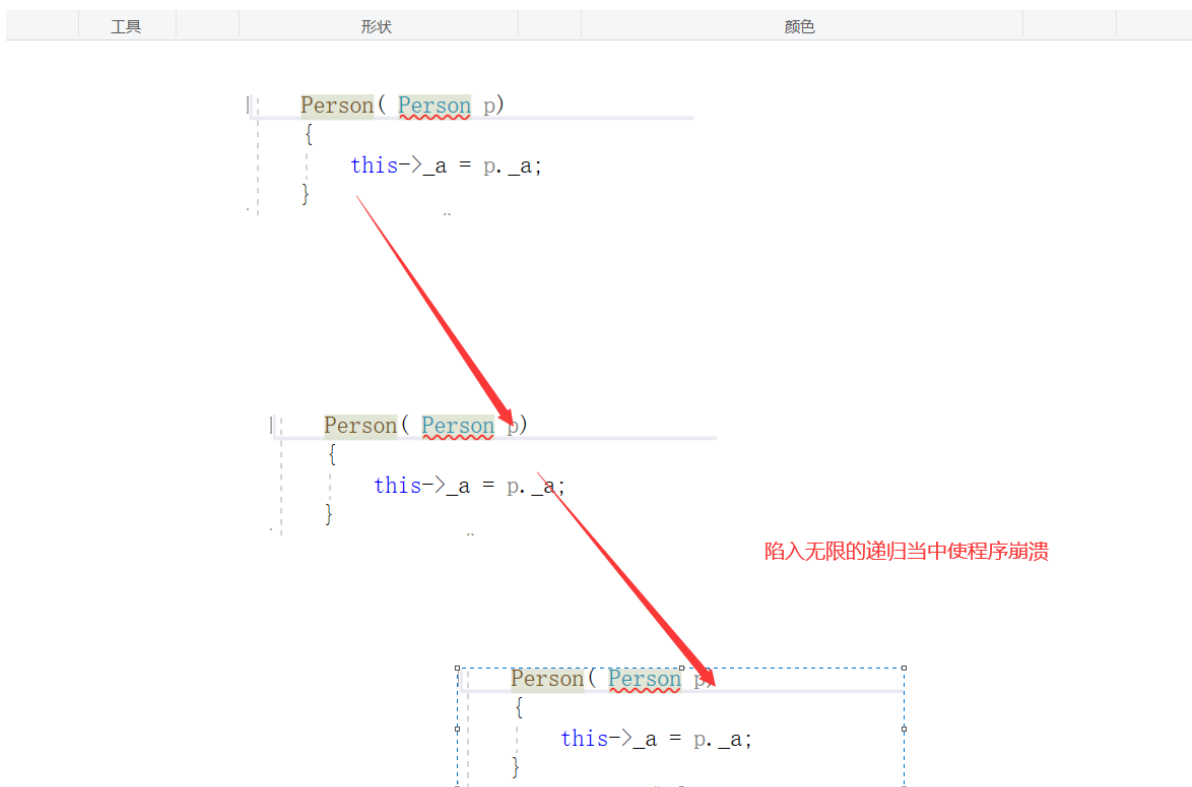
看上面的例子，这是最简单的拷贝构造函数，大家仔细看上面的函数原型，要是我把里面的 `Person(const Person&p)` 换成 `Person(const Person p)` 可以吗，答案不行，要是换了，那么要调用拷贝构造函数，形参在传递的时候又要调用拷贝构造函数，之后一直调用拷贝构造函数，陷入无限的递归当中，最终使程序崩溃，而当使用引用时就不会存在这样的问题，看下面的实例：

工具	形状	颜色

```
Person( Person p)
{
    this->_a = p._a;
}

Person( Person p)
{
    this->_a = p._a;
}

Person( Person p)
{
    this->_a = p._a;
}
```



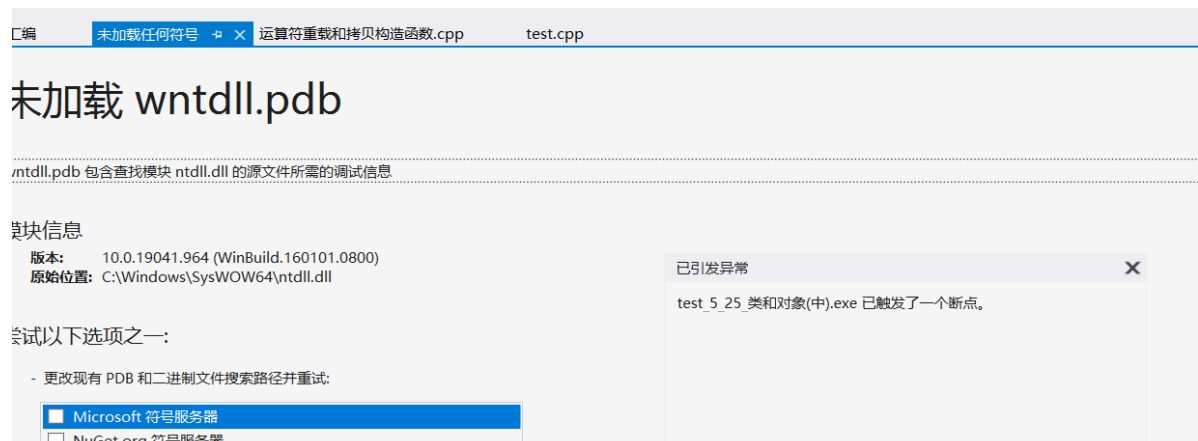
有了拷贝构造函数我们在对象的拷贝上就会省去大量的时间，但是在有些情况下我们仍然需要自己写拷贝构造函数，例如：

```
#include<iostream>
using namespace std;
class Person
{
public:
    Person(int a)
    {
        _a = (int*)malloc(sizeof(int) * a);
    }

    ~Person()
    {
        if (_a != NULL)
        {
            free(_a);
            _a = NULL;
        }
    }

private:
    int *_a;
};

int main()
{
    Person P1(10);
    Person P2(P1);
    return 0;
}
```



当我启动程序时程序直接奔溃，上面的例子我们将变量创建到堆区，在构造函数中我们初始化这块内存空间，在析构函数中释放这块空间，然后创建P1，P2，将P1拷贝给P1，这是错误的，程序在执行时，首先调用P1的构造函数，再调用P2的构造函数，在程序结束时，调用P2的析构函数，再调用P1的析构函数，但是此时P1->a和P2->a指向同一块内存空间，我们首先调用P2的析构函数释放这块内存空间，此时_a所指向的这块空间已经被释放，然后我们又调用P1的析构函数对这块已经释放过的空间再次进行释放，那么毫无疑问会出现错误，这就是C++中非常经典的深浅拷贝问题，关于这个问题下次我会专门开一个专题详细探讨这个问题，这里就不在赘述了

4.赋值运算符重载

1.赋值运算符重载：

在C++中默认提供对赋值运算符的重载

例如：

```
#include<iostream>
using namespace std;
class Person
{
public:
    Person()
    {

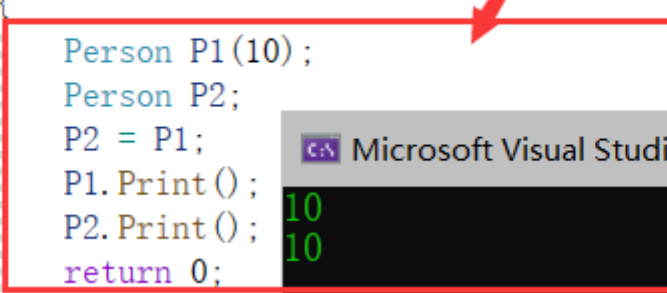
    }

    Person(int a)
    {
        this->a = a;
    }
    void Print()
    {
        cout << a << endl;
    }
    int a;
};

int main()
{
    Person P1(10);
    Person P2;
    P2 = P1;
    P1.Print();
    P2.Print();
    return 0;
}
```

```
67  
68     }  
69     Person(int a)  
70     {  
71         this->a = a;  
72     }  
73     void Print()  
74     {  
75         cout << a << endl;  
76     }  
77     int a;  
78 };  
79 int main()  
80 {  
81     Person P1(10);  
82     Person P2;  
83     P2 = P1;  
84     P1.Print();  
85     P2.Print();  
86     return 0;  
}
```

赋值有算



Microsoft Visual Studio 调试控制台

```
10  
10
```

符主要有以下几点：

上述代码中我们直接将P1赋值给P2（P2=P1），实现了自定义类型的赋值，这是由于C++中成员中默认提供承载的赋值函数，它的主要特点如下所示：

- 1.参数类型
- 2.返回值
- 3.检测是否自己给自己赋值
- 4.返回*this
- 5.一个类如果没有显示定义赋值运算符，编译器也会自动生成一个，完成对象按字节序的值拷贝

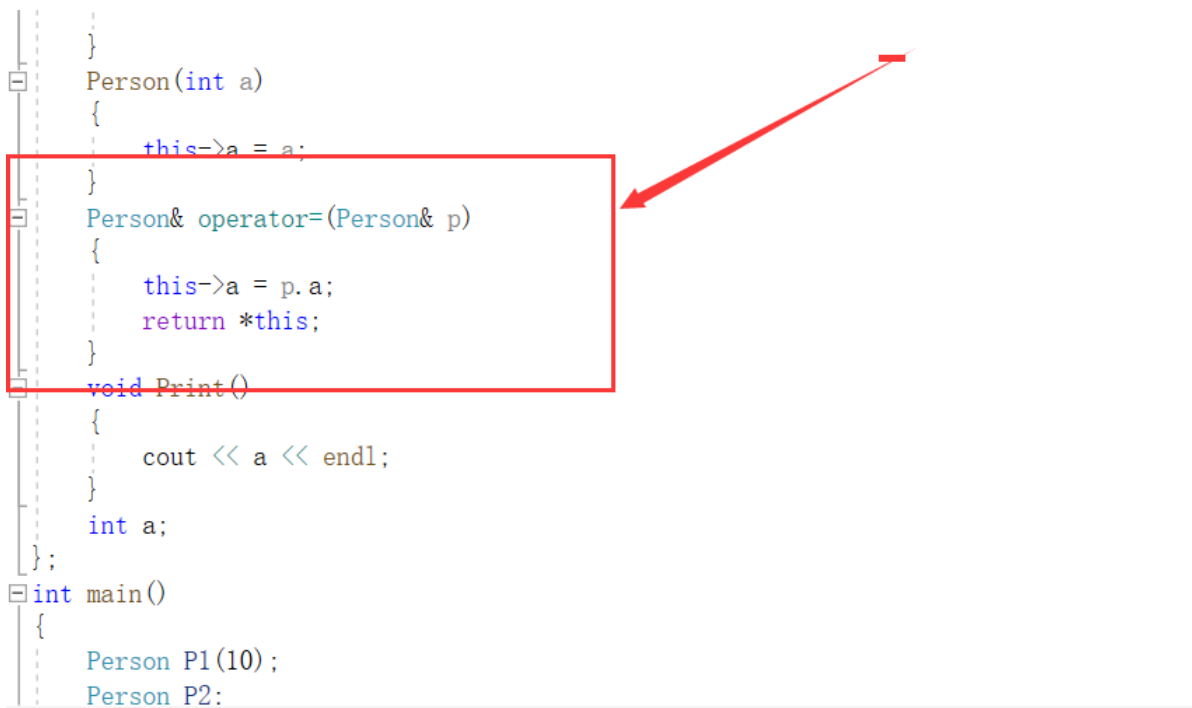
它的函数原型如下所示：

```

    }
    Person(int a)
    {
        this->a = a;
    }
    Person& operator=(Person& p)
    {
        this->a = p.a;
        return *this;
    }
    void Print()
    {
        cout << a << endl;
    }
    int a;
};

int main()
{
    Person P1(10);
    Person P2;
}

```



当时，上述只是最简单的运算符重载，我们还可以写点稍微复杂一点的

2. 其它运算符的重载

C++为了增强代码的可读性引入了运算符重载，运算符重载是具有特殊函数名的函数，也具有返回值类型，函数名及其参数列表，其返回值类型与参数列表与普通的函数类似

函数名字：关键字operator+操作符+（参数列表）

函数原型：返回值类型 operator+操作符+（参数列表）

注意：

1. 不能通过连接其它符号创建新的操作符
2. 重载操作符必须有一个类类型或者枚举类型的操作数
3. 用于内置类型的操作数，其含义不能改变，例如：+，不能改变其含义
4. 作为类成员的重载函数时，其形式看起来比操作数数目少1的成员函数的操作符有一个默认的形参this，限定为第一个参数
5. *、.、::、sizeof、?:、注意以上5个运算符不能重载

下面我们将对几种比较常见的运算符进行重载操作：

+运算符重载：实现两个自定义类型的相加

```

#include<iostream>
using namespace std;
class person
{
public:
    person()
    {

    }
    person(int a, int b)
    {
        this->m_A = a;
        this->m_B = b;
    }
}

```

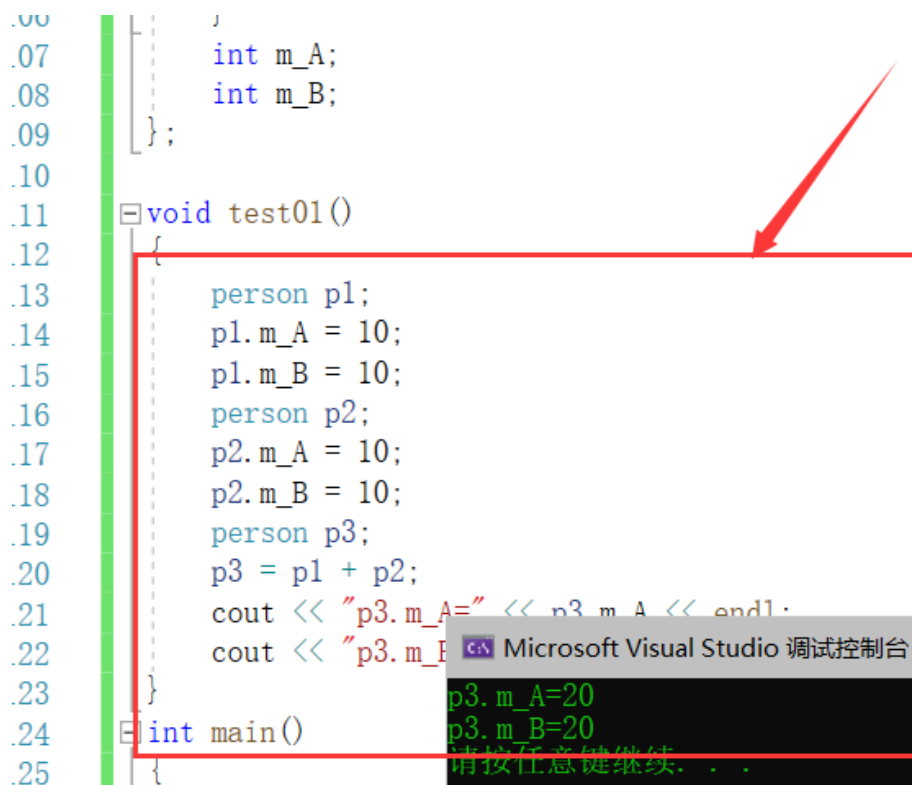
```

    }
    person operator+(person& p)
    {
        person temp;
        temp.m_A = this->m_A + p.m_A;
        temp.m_B = this->m_B + p.m_B;
        return temp;
    }
    int m_A;
    int m_B;
};

void test01()
{
    person p1;
    p1.m_A = 10;
    p1.m_B = 10;
    person p2;
    p2.m_A = 10;
    p2.m_B = 10;
    person p3;
    p3 = p1 + p2;
    cout << "p3.m_A=" << p3.m_A << endl;
    cout << "p3.m_B=" << p3.m_B << endl;
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```



```

.00
.07     int m_A;
.08     int m_B;
.09 };
.10
.11 void test01()
.12 {
.13     person p1;
.14     p1.m_A = 10;
.15     p1.m_B = 10;
.16     person p2;
.17     p2.m_A = 10;
.18     p2.m_B = 10;
.19     person p3;
.20     p3 = p1 + p2;
.21     cout << "p3.m_A=" << p3.m_A << endl;
.22     cout << "p3.m_B=" << p3.m_B << endl;
.23 }
.24 int main()
.25 {

```

Microsoft Visual Studio 调试控制台

```

p3.m_A=20
p3.m_B=20
请按任意键继续. . .

```

下面的例子我们将通过最简单的例子实现对运算符的重载：

```

class Person
{
public:
    Person()
    {
        this->m_num = 0;
    }
private:
    int m_num;
};

```

前置++运算符重载：实现对一个自定义数据类型的前置++

```

//假设类中只有一个数字m_num,实现对num的前置++
person &operator++()
{
    this->m_num++;
    return *this;
}
//重载前置++运算符，返回引用是为了一直对一个数据进行递增

```

后置++运算符重载：实现对自定义数据类型的后置++

```

//和上面例子一样，实现对类中唯一成员m_num的后置++
person operator++(int)//int为占位参数，为了与前面的前置运算++区分开
{
    person temp;
    temp.m_num =this->m_num;
    m_num++;
    return temp;
}
//重载后置++运算符，int代表占位参数，用来区分前置与后置递增
//后置递增返回值，前置递增返回引用

```

前置--运算符重载：实现对自定义数据类型前置--

```

//前置--，先-之后直接返回结果，用引用的方式返回
person& operator--()
{
    this->m_num--;
    return *this;
}

```

后置--运算符重载：实现对自定义数据类型的后置--

```

//后置--，先返回值，后减减，直接返回
//其中int为函数重载，防止命名冲突
person operator--(int)
{
    person temp = *this;//这里必须是*this，否则会报错
    this->m_num--;
    return temp;
}

```

==运算符重载

```
bool operator==(person& p)
{
    if (this->m_num==p->m_num)
    {
        return true;
    }
    return false;
}
```

!=运算符重载

```
bool operator==(person& p)
{
    if (this->m_num!=p->m_num)
    {
        return true;
    }
    return false;
}
```

5.&操作符重载及const修饰的&操作符重载

1.const修饰类的成员函数

将const修饰的类成员函数称之为const成员函数，const修饰类成员函数，实际修饰该成员函数中的this指针，表明该成员函数中不能对类的任何成员进行修改

```
class Date
{
public :
    Date* operator&()
    {
        return this ;
    }
    const Date* operator&() const
    {
        return this ;
    }
private :
    int _year ; // 年
    int _month ; // 月
    int _day ; //日
};
```

2.&及const取地址操作符重载

这两个默认成员函数一般不用重新定义，编译器会默认生成

这两个运算符一般不需要重载，使用编译器生成的默认取地址的宠爱即可，只有特殊情况，才需要重载，比如想让别人获取到指定的内容

综合案例：

设计一个日期类，实现对日期的++, --, +, -, =等操作

```
class Date
{
public:
    // 获取某年某月的天数
    inline int GetMonthDay(int year, int month)
    {
        static int days[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        int day = days[month];
        if (month == 2
            &&((year % 4 == 0 && year % 100 != 0) || (year%400 == 0)))
        {
            day += 1;
        }
        return day;
    }

    // 全缺省的构造函数
    Date(int year = 1900, int month = 1, int day = 1)
    {
        if(year < 1900 || month < 1 || month > 12 || day < 1 || day > GetMonthDay(year, month) )
        {
            cout<<"非法日期"<<endl;
        }
        _year = year;
        _month = month;
        _day = day;
    }

    // 拷贝构造函数
    Date(const Date& d)
    {
        this->_year = d._year;
        _month = d._month;
        _day = d._day;
    }

    // 赋值运算符重载
    // d2 = d3 -> d2.operator=(&d3, d3)
    Date& operator=(const Date& d)
    {
        if (this != &d)
        {
            this->_year = d._year;
            this->_month = d._month;
            this->_day = d._day;
        }
        return *this;
    }
};
```

```

}

// 析构函数
~Date()
{
    // 清理工作
}

//打印函数
void Print()
{
    cout<<_year<<"-"<<_month<<"-"<<_day<<endl;
}

// 日期+=天数
// d1 += 10
// d1 += -10
Date& operator+=(int day)
{
    if (day < 0)
    {
        return *this -= -day;
    }
    _day += day;
    while (_day > GetMonthDay(_year, _month))
    {
        _day -= GetMonthDay(_year, _month);
        _month++;
        if (_month == 13)
        {
            _year++;
            _month = 1;
        }
    }
    return *this;
}

// 日期+天数
// d + 10
Date operator+(int day)
{
    Date ret(*this);
    ret += day;
    return ret;
}

// 日期-天数
Date operator-(int day)
{
    Date ret(*this);
    ret -= day;
    return ret;
}

```



```

}

// 日期-=天数
// d -= 100
// d -= -100
Date& operator--(int day)
{
    if (day < 0)
    {
        return *this += -day;
    }
    _day -= day;
    while (_day <= 0)
    {
        --_month;
        if (_month == 0)
        {
            --_year;
            _month = 12;
        }
        _day += GetMonthDay(_year, _month);
    }
    return *this;
}

```

```

// 前置++
// ++d -> d.operator++(&d)
Date& operator++()
{
    *this += 1;
    return *this;
}

```

```

// 后置++
// d++ -> d.operator++(&d, 0)
Date operator++(int)
{
    Date ret(*this);
    *this += 1;
    return ret;
}

```

```

// // 后置--
Date operator--(int)
{
    Date ret(*this);
    *this -= 1;
    return ret;
}

```

```

// 前置--
Date& operator--()
{
    *this -= 1;
    return *this;
}

// d1 > d2
// >运算符重载
bool operator>(const Date& d)
{
    if (_year > d._year)
    {
        return true;
    }
    else if (_year == d._year)
    {
        if (_month > d._month)
        {
            return true;
        }
        else if (_month == d._month)
        {
            if (_day > d._day)
            {
                return true;
            }
        }
    }
    return false;
}

// ==运算符重载
bool operator==(const Date& d)
{
    return _year == d._year
        && _month == d._month
        && _day == d._day;
}

// 下面复用上面两个的实现
// >=运算符重载

inline bool operator >= (const Date& d)
{
    return *this > d || *this == d;
}

// <运算符重载

```

```

bool operator < (const Date& d)
{
    return !(*this >= d);
}

// <=运算符重载

bool operator <= (const Date& d)
{
    return !(*this > d);
}

// !=运算符重载

bool operator != (const Date& d)
{
    return !(*this == d);
}

// d1 - d2
// 日期-日期 返回天数
int operator-(const Date& d)
{
    int flag = 1;
    Date max = *this;
    Date min = d;
    if (*this < d)
    {
        max = d;
        min = *this;
        flag = -1;
    }
    int day = 0;
    while (min < max)
    {
        ++(min);
        ++day;
    }
    return day*flag;
}
private:
    int _year;
    int _month;
    int _day;
};

```

本篇文章到此就告一段落了，感谢大家的评论，点赞，收藏，订阅，您的满意就是对我最大的支持，同时本篇文章也存在许多不足之处，还请大家多多斧正，感谢大家支持

期待与大家下次相见，To be continued....

