类与对象(中)

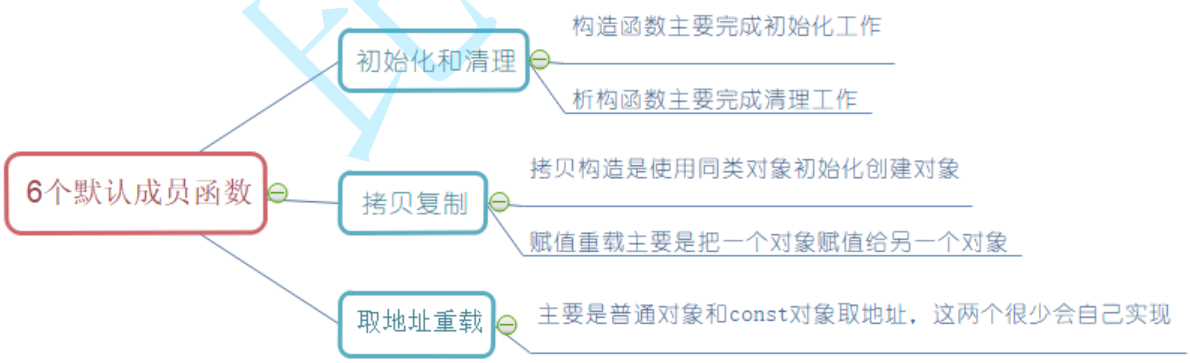
[本节目标]

- **1. 类的6个默认成员函数**
- **2. 构造函数**
- **3. 析构函数**
- **4. 拷贝构造函数**
- **5. 赋值操作符重载**
- **6. 默认拷贝构造与赋值运算符重载的问题**
- **7. const成员函数**
- **8. 取地址及const取地址操作符重载**

1.类的6个默认成员函数

> 如果一个类中什么成员都没有，简称为空类。空类中什么都没有吗？并不是的，任何一个类在我们不写的情况下，都会自动生成下面6个默认成员函数。

```
class Date {};
```



2. 构造函数

2.1 概念

> 对于以下的日期类:

```
class Date
```

```
{
public:
    void SetDate(int year, int month, int day)
    {
        _year = year;
        _month = month;
        _day = day;
    }

    void Display()
    {
        cout <<_year<< "-" <<_month << "-"<< _day <<endl;
    }

private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1,d2;
    d1.SetDate(2018,5,1);
    d1.Display();

    Date d2;
    d2.SetDate(2018,7,1);
    d2.Display();
    return 0;
}
```

对于Date类，可以通过SetDate公有的方法给对象设置内容，但是如果每次创建对象都调用该方法设置信息，未免有点麻烦，那能否在对象创建时，就将信息设置进去呢?

构造函数是一个**特殊的成员函数，名字与类名相同,创建类类型对象时由编译器自动调用**，保证每个数据成员都有 一个合适的初始值，并且**在对象的生命周期内只调用一次。**

2.2 特性

构造函数是特殊的成员函数，需要注意的是，构造函数的虽然名称叫构造，但是需要注意的是构造函数的主要任务**并不是开空间创建对象，而是初始化对象。**

其特征如下:

1. 函数名与类名相同。
2. 无返回值。
3. 对象实例化时编译器**自动调用**对应的构造函数。
4. 构造函数可以重载。

```
class Date
{
```

```
public :
    // 1.无参构造函数
    Date ()
    {}

    // 2.带参构造函数
    Date (int year, int month , int day )
    {
        _year = year ;
        _month = month ;
        _day = day ;
    }
private :
    int _year ;
    int _month ;
    int _day ;
};

void TestDate()
{
    Date d1; // 调用无参构造函数
    Date d2 (2015, 1, 1); // 调用带参的构造函数

    // 注意：如果通过无参构造函数创建对象时，对象后面不用跟括号，否则就成了函数声明
    // 以下代码的函数：声明了d3函数，该函数无参，返回一个日期类型的对象
    Date d3();
}
```

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无参的默认构造函数，一旦用户显式定义编译器将不再生成。

```
class Date
{
public:
    /*
    // 如果用户显式定义了构造函数，编译器将不再生成
    Date (int year, int month, int day)
    {
        _year = year;
        _month = month;
        _day = day;
    }
    */
private:
    int _year;
    int _month;
    int _day;
};

void Test()
{

    // 没有定义构造函数，对象也可以创建成功，因此此处调用的是编译器生成的默认构造函数
```

```
    Date d;
}
```

6. 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，并且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。

```
// 默认构造函数
class Date
{
public:
    Date()
    {
        _year = 1900 ;
        _month = 1 ;
        _day = 1;
    }

    Date (int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }

private :
    int _year ;
    int _month ;
    int _day ;
};

// 以下测试函数能通过编译吗?
void Test()
{
    Date d1;
}
```

7. 关于编译器生成的默认成员函数，很多童鞋会有疑惑：在我们不实现构造函数的情况下，编译器会生成默认的构造函数。但是看起来默认构造函数又没什么用？d对象调用了编译器生成的默认构造函数，但是d对象*year*/month/_day，依旧是随机值。也就说在这里**编译器生成的默认构造函数并没有什么卵用？？**

   解答：C++把类型分成内置类型(基本类型)和自定义类型。内置类型就是语法已经定义好的类型：如int/char...，自定义类型就是我们使用class/struct/union自己定义的类型，看看下面的程序，就会发现编译器生成默认的构造函数会对自定类型成员_t调用的它的默认成员函数

```
class Time
{
public:
    Time()
    {

        cout << "Time()" << endl;
```

```
        _hour = 0;
        _minute = 0;
        _second = 0;
    }
private:
    int _hour;
    int _minute;
    int _second;
};

class Date
{
private:
    // 基本类型(内置类型)
    int _year;
    int _month;
    int _day;

    // 自定义类型
    Time _t;
};

int main()
{
    Date d;
    return 0;
```

8. 成员变量的命名风格

```
// 我们看看这个函数，是不是很僵硬?
class Date
{
public:
    Date(int year)
    {
        // 这里的year到底是成员变量，还是函数形参?
        year = year;
    }
private:
    int year;
};

// 所以我们一般都建议这样
class Date
{
public:
    Date(int year)
    {
        _year = year;
    }
private:

    int _year;
```

```
};

// 或者这样。
class Date
{
public:
    Date(int year)
    {
        m_year = year;
    }
private:
    int m_year;
};

// 其他方式也可以的，主要看公司要求。一般都是加个前缀或者后缀标识区分就行。
```

3.析构函数

3.1 概念

> 前面通过构造函数的学习，我们知道一个对象时怎么来的，那一个对象又是怎么没呢的?
>
> 析构函数：与构造函数功能相反，析构函数不是完成对象的销毁，局部对象销毁工作是由编译器完成的。而**对象在销毁时会自动调用析构函数，完成类的一些资源清理工作。**

3.2 特性

> **析构函数**是特殊的成员函数。
>
> 其**特征**如下：
>
> 1. 析构函数名是在类名前加上字符 ~。
> 2. 无参数无返回值。
> 3. 一个类有且只有一个析构函数。若未显式定义，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++编译系统系统自动调用析构函数。

```
typedef int DataType;
class SeqList
{
public :
    SeqList (int capacity = 10)
    {
        _pData = (DataType*)malloc(capacity * sizeof(DataType));
        assert(_pData);

        _size = 0;
        _capacity = capacity;
    }
```

```
    ~SeqList()
    {
        if (_pData)
        {
            free(_pData );   // 释放堆上的空间
            _pData = NULL;   // 将指针置为空
            _capacity = 0;
            _size = 0;
        }
    }

private :
    int* _pData ;
    size_t _size;
    size_t _capacity;
};
```

5. 关于编译器自动生成的析构函数，是否会完成一些事情呢？下面的程序我们会看到，编译器生成的默认析构函数，对会自定类型成员调用它的析构函数。

```
class String
{
public:
    String(const char* str = "jack")
    {
        _str = (char*)malloc(strlen(str) + 1);
        strcpy(_str, str);
    }

    ~String()
    {
        cout << "~String()" << endl;
        free(_str);
    }
private:
    char* _str;
};

class Person
{
private:
    String _name;
    int    _age;
};

int main()
{
    Person p;
    return 0;
}
```

4. 拷贝构造函数

4.1 概念

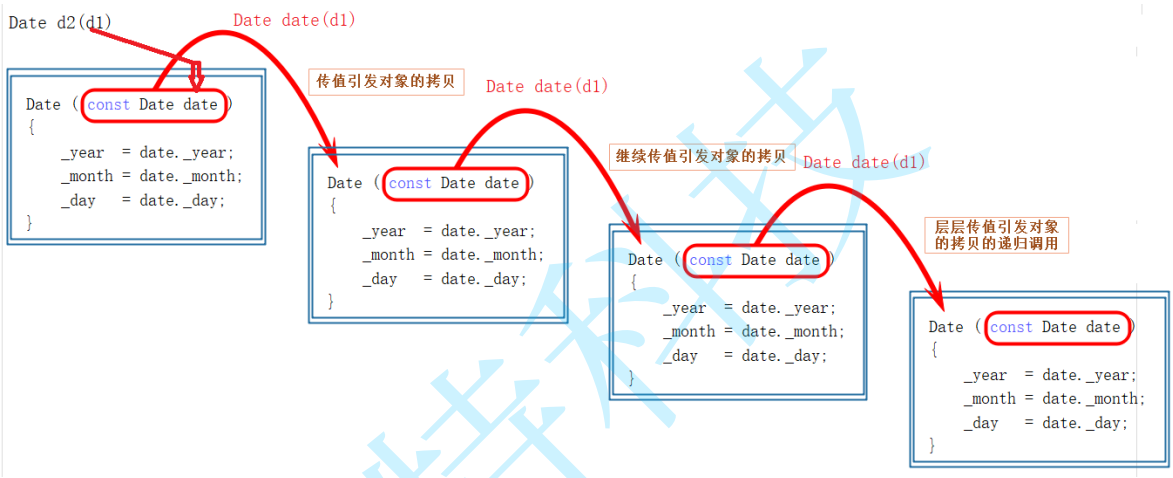
> 在现实生活中，可能存在一个与你一样的自己，我们称其为双胞胎。
>
> 
>
>
> 那在创建对象时，可否创建一个与一个对象一某一样的新对象呢?
>
> **构造函数**：**只有单个形参**，该形参是对本**类类型对象的引用(一般常用const修饰)**，在用**已存在的类类型对象创建新对象时由编译器自动调用。**

4.2 特征

> 拷贝构造函数也是特殊的成员函数，其**特征**如下：
>
> 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**参数只有一个**且**必须使用引用传参**，使用**传值方式会引发无穷递归调用。**

```
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }

    Date(const Date& d)
    {
        _year = d._year;
        _month = d._month;
```

```
        _day = d._day;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1;
    Date d2(d1);

    return 0;
}
```



3. **若未显示定义，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按字节序完成拷贝，这种拷贝我们叫做浅拷贝，或者值拷贝。

```
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1;
    // 这里d2调用的默认拷贝构造完成拷贝，d2和d1的值也是一样的。
    Date d2(d1);
```

```
    return 0;
}
```

4. 那么**编译器生成的默认拷贝构造函数已经可以完成字节序的值拷贝了**，我们还需要自己实现吗？当然像日期类这样的类是没必要的。那么下面的类呢？验证一下试试？

```
// 这里会发现下面的程序会崩溃掉？这里就需要我们以后讲的深拷贝去解决。
class String
{
public:
    String(const char* str = "jack")
    {
        _str = (char*)malloc(strlen(str) + 1);
        strcpy(_str, str);
    }

    ~String()
    {
        cout << "~String()" << endl;
        free(_str);
    }
private:
    char* _str;
};

int main()
{
    String s1("hello");
    String s2(s1);
}
```


5.赋值运算符重载

5.1 运算符重载

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，也具有其返回值类型，函数名字以及参数列表，其返回值类型与参数列表与普通的函数类似。

函数名字为：关键字**operator后面接需要重载的运算符符号。**

函数原型：**返回值类型 operator操作符(参数列表)**

注意：

- 不能通过连接其他符号来创建新的操作符：比如operator@
- 重载操作符必须有一个类类型或者枚举类型的操作数
- 用于内置类型的操作符，其含义不能改变，例如：内置的整型+，不 能改变其含义

- 作为类成员的重载函数时，其形参看起来比操作数数目少1成员函数的操作符有一个默认的形参this，限定为第一个形参
- .* 、:: 、sizeof 、?: 、. 注意以上5个运算符不能重载。这个经常在笔试选择题中出现。

```
// 全局的operator==
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }
//private:
    int _year;
    int _month;
    int _day;
};

// 这里会发现运算符重载成全局的就需要成员变量是共有的，那么问题来了，封装性如何保证?
// 这里其实可以用我们后面学习的友元解决，或者干脆重载成成员函数。
bool operator==(const Date& d1, const Date& d2)
{
    return d1._year == d2._year;
        && d1._month == d2._month
        && d1._day == d2._day;
}

void Test ()
{
    Date d1(2018, 9, 26);
    Date d2(2018, 9, 27);
    cout<<(d1 == d2)<<endl;
}
```

```
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }

    // bool operator==(Date* this, const Date& d2)
    // 这里需要注意的是，左操作数是this指向的调用函数的对象
    bool operator==(const Date& d2)
    {

        return _year == d2._year;
```

```
            && _month == d2._month
            && _day == d2._day;
    }
private:
    int _year;
    int _month;
    int _day;
};

void Test ()
{
    Date d1(2018, 9, 26);
    Date d2(2018, 9, 27);
    cout<<(d1 == d2)<<endl;
}
```

5.2 赋值运算符重载

```
class Date
{
public :
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }

    Date (const Date& d)
    {
        _year = d._year;
        _month = d._month;
        _day = d._day;
    }

    Date& operator=(const Date& d)
    {
        if(this != &d)
        {
            _year = d._year;
            _month = d._month;
            _day = d._day;
        }
    }
private:
    int _year ;
    int _month ;
    int _day ;
};
```

赋值运算符主要有四点：

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. 一个类如果没有显式定义赋值运算符重载，编译器也会生成一个，完成对象按字节序的值拷贝。

```
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1;
    Date d2(2018, 10,  1);

    // 这里d1调用的编译器生成operator=完成拷贝，d2和d1的值也是一样的。
    d1 = d2;

    return 0;
}
```

那么**编译器生成的默认赋值重载函数已经可以完成字节序的值拷贝了**，我们还需要自己实现吗？当然像日期类这样的类是没必要的。那么下面的类呢？验证一下试试？

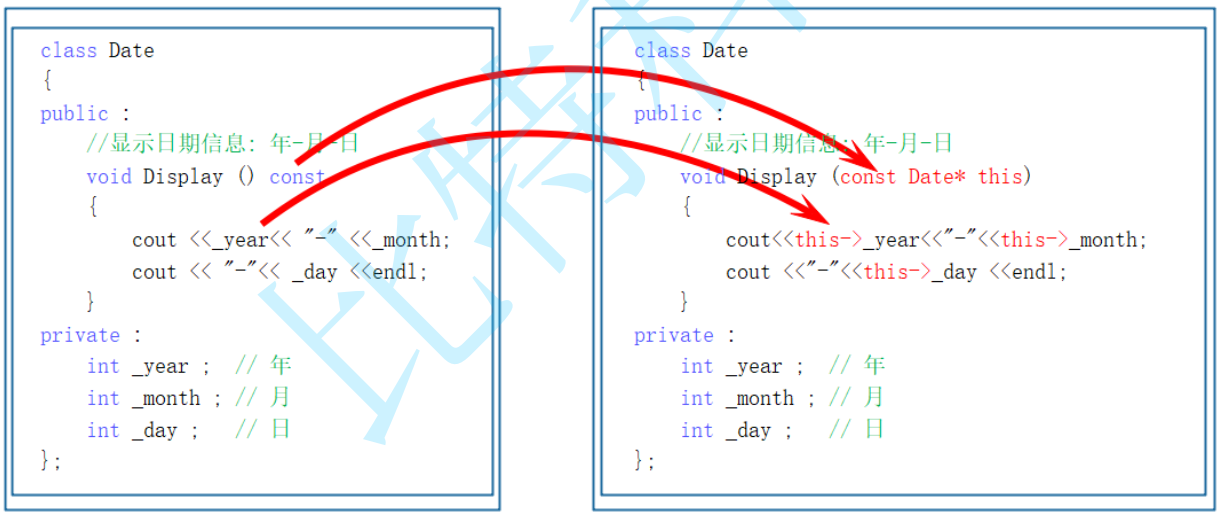
```
// 这里会发现下面的程序会崩溃掉？这里就需要我们以后讲的深拷贝去解决。
class String
{
public:
    String(const char* str = "")
    {
        _str = (char*)malloc(strlen(str) + 1);
        strcpy(_str, str);
    }

    ~String()
    {

        cout << "~String()" << endl;
```

```
        free(_str);
    }
private:
    char* _str;
};

int main()
{
    String s1("hello");
    String s2("world");

    s1 = s2;
}
```

7.const成员

7.1 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**隐含的this指针**，表明在该成员函数中**不能对类的任何成员进行修改。**



编译器对const成员函数的处理

我们来看看下面的代码

```
class Date
{
public :
    void Display ()
    {
        cout<<"Display ()" <<endl;
        cout<<"year:" <<_year<< endl;
        cout<<"month:" <<_month<< endl;
        cout<<"day:" <<_day<< endl<<endl ;
```

```
    }
    void Display () const
    {
        cout<<"Display () const" <<endl;
        cout<<"year:" <<_year<< endl;
        cout<<"month:" <<_month<< endl;
        cout<<"day:" <<_day<< endl<<endl;
    }
private :
    int _year ; // 年
    int _month ; // 月
    int _day ; // 日
};
void Test ()
{
    Date d1 ;
    d1.Display ();

    const Date d2;
    d2.Display ();
}
```

请思考下面的几个问题：

1. const对象可以调用非const成员函数吗?
2. 非const对象可以调用const成员函数吗?
3. const成员函数内可以调用其它的非const成员函数吗?
4. 非const成员函数内可以调用其它的const成员函数吗?

8.取地址及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， 编译器默认会生成。

```
class Date
{
public :
    Date* operator&()
    {
        return this ;
    }

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

> 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重载即可，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！**

[总结]

> 本节内容回顾：

知识块	知识点	分类	掌握程度
类的6个默认成员函数	构造函数、析构函数、拷贝构造函数、赋值操作符重载、取地址操作符重载、const修饰的取地址操作符重载	概念型	了解
构造函数	**定义**：公有函数，用来初始化类成员，仅在定义对象时自动执行一次	概念型	了解
	特征：函数名与类名相同；无返回值；对象实例化时自动调用；可重载；如果没有给出，则C++编译器自动产生一个缺省的；缺省的构造函数只能有一个(无参；不带缺省参数；带缺省参数)	概念型	掌握
析构函数	**定义**：公有函数，对象的生命周期结束时，C++编译系统会自动调用	概念型	了解
	特征：函数名是在类名前加上字符 ~ ；无参数无返回值；类有且只有一个析构函数；如果没有给出，则C++编译器自动产生一个缺省的	概念型	掌握
	不是删除对象，而是做一些对象删除前的相关清理工作	概念型	掌握
拷贝构造函数	**定义**：公有函数，创建对象时使用同类对象来进行初始化新对象	概念型	了解
	特征：是构造函数的重载；参数必须使用引用传参；如果没有给出，则C++编译器自动产生一个缺省的	概念型	掌握
运算符重载	C++支持运算符重载是为了增强程序的可读性	概念型	了解
	特征：operator + 合法的运算符 构成函数名；不改变运算符的 **优先级/结合性/操作数个数**	概念型	掌握
	不能被重载的5个运算符：.* / :: / sizeof / ?: / .	概念型	掌握
赋值操作符的重载	**定义**：是对一个已存在的对象进行拷贝赋值	概念型	了解
	特征：如果没有给出，则C++编译器自动产生一个缺省的	概念型	掌握
深拷贝与浅拷贝问题	如果一个类拥有资源(堆，或者是其它系统资源)，当这个类的对象发生复制过程的时候，这个过程就可以叫做**深拷贝**， 反之对象存在资源，但复制过程并未复制资源的情况视为**浅拷贝**。	原理型	了解
	由编译器默认生成的**拷贝构造函数**和**赋值运算符重载**，默认对类的成员采用位拷贝(只是按照基本类型进行值拷贝)	原理型	了解
const成员函数	**定义**：在成员函数后面加const，const修饰this指针所指向的对象，也就是保证**调用这个const成员函数的对象在函数内不会被改变**	概念型	了解
	const对象可以调用其它的const函数； 非const对象可以调用非const成员函数和const成员函数； const成员函数内可以调用其它的const成员函数； 非const成员函数内可以调用其它的const成员函数和非const成员函数	考点型	掌握
取地址操作符重载、const修饰的取地址操作符重载	这两个默认成员函数一般不用重新定义 ，编译器默认会生成	概念型	了解

[作业]

1. 好好总结本节内容，写一篇博客。
2. 完善Date类 。

```
class Date
{
public:
    Date(int year = 1900, int month = 1, int day = 1);
    Date(const Date& d);
    Date& operator=(const Date& d);
    Date operator+(int days);
```

```
    Date operator-(int days);
    int operator-(const Date& d);
    Date& operator++();
    Date operator++(int);
    Date& operator--();
    Date operator--(int);
    bool operator>(const Date& d)const;
    bool operator>=(const Date& d)const;
    bool operator<(const Date& d)const;
    bool operator<=(const Date& d)const;
    bool operator==(const Date& d)const;
    bool operator!=(const Date& d)const;
private:
    int _year;
    int _month;
    int _day;
};
```