类与对象(中)

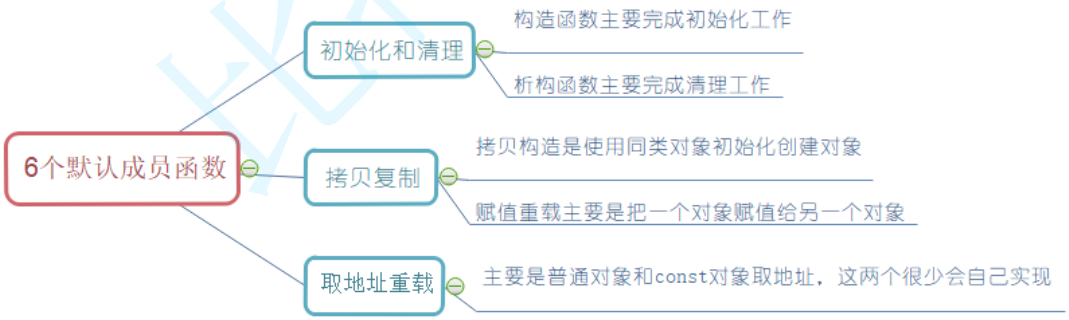
【本节目标】

- 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. 赋值运算符重载
- 6. const成员函数
- 7. 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。
>
> 空类中真的什么都没有吗？并不是，任何类在什么都不写时，编译器会自动生成以下6个默认成员函数。
>
> 默认成员函数：用户没有显式实现，编译器会生成的成员函数称为默认成员函数。

```
class Date {};
```



2. 构造函数

2.1 概念

> 对于以下Date类：

```
class Date
{
public:
    void Init(int year, int month, int day)
    {
```

```
        _year = year;
        _month = month;
        _day = day;
    }

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

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

int main()
{
    Date d1;
    d1.Init(2022, 7, 5);
    d1.Print();

    Date d2;
    d2.Init(2022, 7, 6);
    d2.Print();
    return 0;
}
```

对于Date类，可以通过 Init 公有方法给对象设置日期，但如果每次创建对象时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**，以保证每个数据成员都有 一个合适的初始值，并且**在对象整个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，构造函数虽然名称叫构造，但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
    // warning C4930: “Date d3(void)”: 未调用原型函数(是否是有意用变量定义的?)
    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;
  }
  */

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

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

int main()
{
  // 将Date类中构造函数屏蔽后，代码可以通过编译，因为编译器生成了一个无参的默认构造函数
  // 将Date类中构造函数放开，代码编译失败，因为一旦显式定义任何构造函数，编译器将不再生成
    // 无参构造函数，放开后报错：error C2512: “Date”: 没有合适的默认构造函数可用
  Date d1;
  return 0;
}
```

6. 关于编译器生成的默认成员函数，很多童鞋会有疑惑：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。内置类型就是语言提供的数据类型，如：int/char...，自定义类型就是我们使用class/struct/union等自己定义的类型，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;
}
```

   注意：C++11 中针对内置类型成员不初始化的缺陷，又打了补丁，即：**内置类型成员变量在类中声明时可以给默认值。**

```
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 = 1970;
    int _month = 1;
    int _day = 1;

    // 自定义类型
    Time _t;
};

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

7. 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;
```

```
}
```

3.析构函数

3.1 概念

> 通过前面构造函数的学习，我们知道一个对象是怎么来的，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对象本身的销毁，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，完成对象中资源的清理工作。**

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. 无参数无返回值类型。
> 3. 一个类只能有一个析构函数。若未显式定义，系统会自动生成默认的析构函数。注意：析构函数不能重载
> 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 Stack
{
public:
    Stack(size_t capacity = 3)
    {
        _array = (DataType*)malloc(sizeof(DataType) * capacity);
        if (NULL == _array)
        {
            perror("malloc申请空间失败!!!");
            return;
        }

        _capacity = capacity;
        _size = 0;
    }

    void Push(DataType data)
    {
        // CheckCapacity();
        _array[_size] = data;
        _size++;
    }

    // 其他方法...

    ~Stack()
    {
        if (_array)
        {
            free(_array);
            _array = NULL;
            _capacity = 0;
            _size = 0;
```

```
        }
    }

private:
    DataType* _array;
    int _capacity;
    int _size;
};

void TestStack()
{
    Stack s;
    s.Push(1);
    s.Push(2);
}
```

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，对自定类型成员调用它的析构函数。

```
class Time
{
public:
    ~Time()
    {
        cout << "~Time()" << endl;
    }
private:
    int _hour;
    int _minute;
    int _second;
};

class Date
{
private:
    // 基本类型(内置类型)
    int _year = 1970;
    int _month = 1;
    int _day = 1;

    // 自定义类型
    Time _t;
};

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

// 程序运行结束后输出：~Time()
// 在main方法中根本没有直接创建Time类的对象，为什么最后会调用Time类的析构函数？
// 因为：main方法中创建了Date对象d，而d中包含4个成员变量，其中_year，_month，
_day三个是
// 内置类型成员，销毁时不需要资源清理，最后系统直接将其内存回收即可；而_t是Time类对
象，所以在
```

```
// d销毁时，要将其内部包含的Time类的_t对象销毁，所以要调用Time类的析构函数。但是：main函数
// 中不能直接调用Time类的析构函数，实际要释放的是Date类对象，所以编译器会调用Date类的析构函
// 数，而Date没有显式提供，则编译器会给Date类生成一个默认的析构函数，目的是在其内部调用Time
// 类的析构函数，即当Date对象销毁时，要保证其内部每个自定义对象都可以正确销毁

// main函数中并没有直接调用Time类析构函数，而是显式调用编译器为Date类生成的默认析构函数
// 注意：创建哪个类的对象则调用该类的析构函数，销毁那个类的对象则调用该类的析构函数
```

[两个栈实现一个队列]

同学们可以使用C++的方式，自己封装栈，实现上述oj题，深刻体会编译器生成析构函数的作用。

6. 如果类中没有申请资源时，析构函数可以不写，直接使用编译器生成的默认析构函数，比如Date类；有资源申请时，一定要写，否则会造成资源泄漏，比如Stack类。

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，可否创建一个与已存在对象一某一样的新对象呢?

拷贝构造函数：**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。**

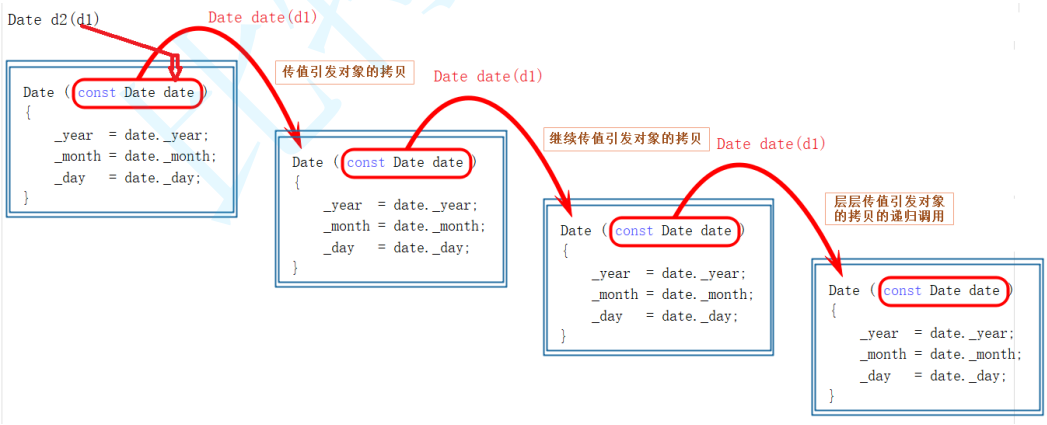
4.2 特征

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是类类型对象的引用**，使用**传值方式编译器直接报错**，因为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)   // 正确写法
    Date(const Date& d)   // 错误写法：编译报错，会引发无穷递归
    {
        _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. **若未显式定义，编译器会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，这种拷贝叫做浅拷贝，或者值拷贝。

```
class Time
{
public:
    Time()
    {
        _hour = 1;
        _minute = 1;
        _second = 1;
```

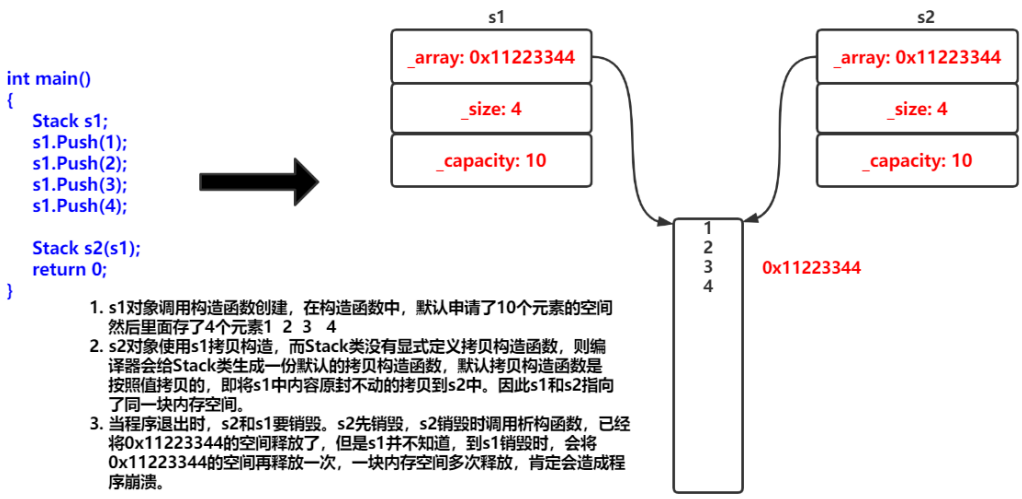
```
    }

    Time(const Time& t)
    {
        _hour = t._hour;
        _minute = t._minute;
        _second = t._second;
        cout << "Time::Time(const Time&)" << endl;
    }
private:
    int _hour;
    int _minute;
    int _second;
};

class Date
{
private:
    // 基本类型(内置类型)
    int _year = 1970;
    int _month = 1;
    int _day = 1;

    // 自定义类型
    Time _t;
};

int main()
{
    Date d1;

    // 用已经存在的d1拷贝构造d2，此处会调用Date类的拷贝构造函数
    // 但Date类并没有显式定义拷贝构造函数，则编译器会给Date类生成一个默认的拷贝构造函数
    Date d2(d1);
    return 0;
}
```

注意：在编译器生成的默认拷贝构造函数中，内置类型是按照字节方式直接拷贝的，而自定义类型是调用其拷贝构造函数完成拷贝的。

4. **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**，还需要自己显式实现吗？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。
typedef int DataType;
class Stack
{
public:
    Stack(size_t capacity = 10)
    {
        _array = (DataType*)malloc(capacity * sizeof(DataType));
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            return;
        }
```

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

    void Push(const DataType& data)
    {
        // CheckCapacity();
        _array[_size] = data;
        _size++;
    }

    ~Stack()
    {
        if (_array)
        {
            free(_array);
            _array = nullptr;
            _capacity = 0;
            _size = 0;
        }
    }

private:
    DataType *_array;
    size_t _size;
    size_t _capacity;
};
int main()
{
    Stack s1;
    s1.Push(1);
    s1.Push(2);
    s1.Push(3);
    s1.Push(4);

    Stack s2(s1);
    return 0;
}
```

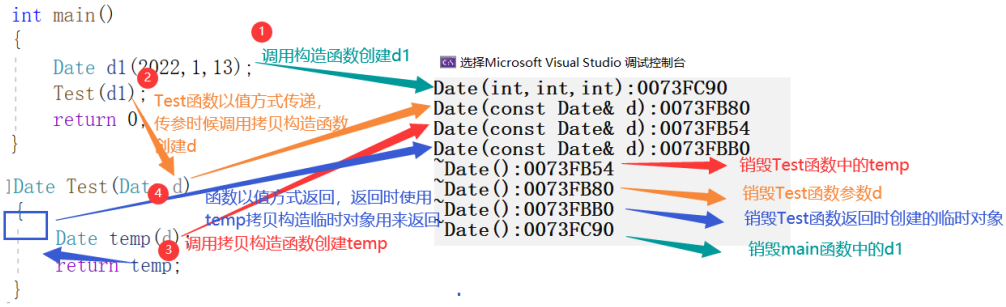


注意：类中如果没有涉及资源申请时，拷贝构造函数是否写都可以；一旦涉及到资源申请时，则拷贝构造函数是一定要写的，否则就是浅拷贝。

5. 拷贝构造函数典型调用场景：

- 使用已存在对象创建新对象
- 函数参数类型为类类型对象
- 函数返回值类型为类类型对象

```
class Date
{
public:
    Date(int year, int minute, int day)
    {
        cout << "Date(int,int,int):" << this << endl;
    }

    Date(const Date& d)
    {
        cout << "Date(const Date& d):" << this << endl;
    }

    ~Date()
    {
        cout << "~Date():" << this << endl;
    }

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

Date Test(Date d)
{
    Date temp(d);
    return temp;
}

int main()
{
    Date d1(2022,1,13);
    Test(d1);
    return 0;
}
```



为了提高程序效率，一般对象传参时，尽量使用引用类型，返回时根据实际场景，能用引用尽量使用引用。

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参数
> - 用于内置类型的运算符，其含义不能改变，例如：内置的整型+，不 能改变其含义
> - 作为类成员函数重载时，其形参看起来比操作数数目少1，因为成员函数的第一个参数为隐藏的this
> - `.*` `::` `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公有的，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;
};
```

5.2 赋值运算符重载

1. **赋值运算符重载格式**
   - **参数类型**：const T&，传递引用可以提高传参效率
   - **返回值类型**：T&，返回引用可以提高返回的效率，有返回值目的是为了支持连续赋值
   - **检测是否自己给自己赋值**
   - **返回*this** ：要复合连续赋值的含义

```
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;
        }

        return *this;
    }
private:
```

```
    int _year ;
    int _month ;
    int _day ;
};
```

2. **赋值运算符只能重载成类的成员函数不能重载成全局函数**

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

    int _year;
    int _month;
    int _day;
};

// 赋值运算符重载成全局函数，注意重载成全局函数时没有this指针了，需要给两个参数
Date& operator=(Date& left, const Date& right)
{
    if (&left != &right)
    {
        left._year = right._year;
        left._month = right._month;
        left._day = right._day;
    }

    return left;
}

// 编译失败：
// error C2801: “operator =”必须是非静态成员
```

原因：赋值运算符如果不显式实现，编译器会生成一个默认的。此时用户再在类外自己实现一个全局的赋值运算符重载，就和编译器在类中生成的默认赋值运算符重载冲突了，故赋值运算符重载只能是类的成员函数。

《C++ prime》第5版---p500页

我们可以重载赋值运算符。不论形参的类型是什么，赋值运算符都必须定义为成员函数。

3. **用户没有显式实现时，编译器会生成一个默认赋值运算符重载，以值的方式逐字节拷贝**。注意：内置类型成员变量是直接赋值的，而自定义类型成员变量需要调用对应类的赋值运算符重载完成赋值。

```
class Time
{
public:
    Time()
    {
        _hour = 1;
```

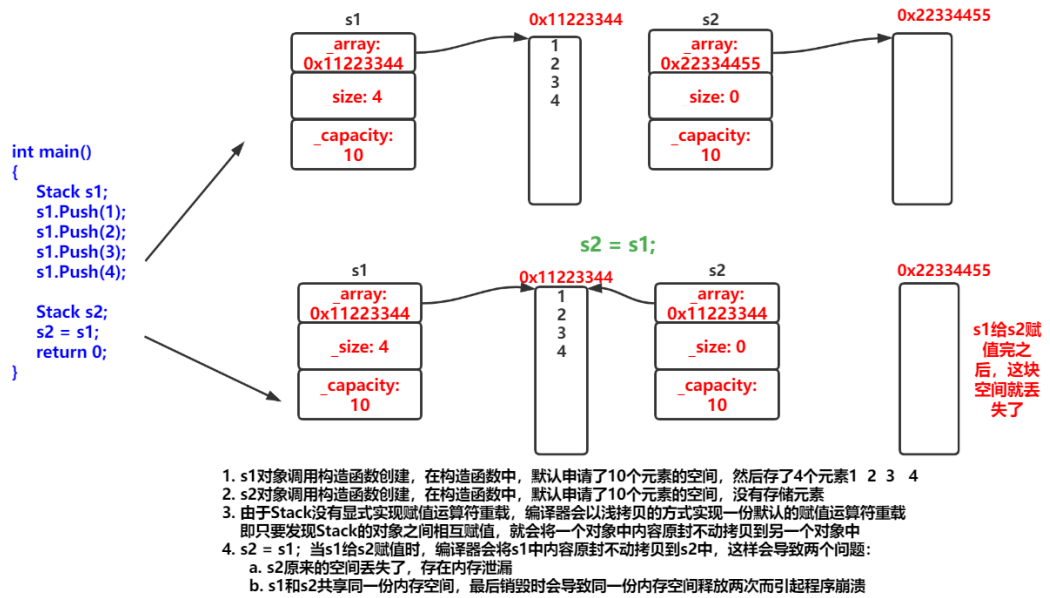
```
        _minute = 1;
        _second = 1;
    }

    Time& operator=(const Time& t)
    {
        if (this != &t)
        {
            _hour = t._hour;
            _minute = t._minute;
            _second = t._second;
        }

        return *this;
    }
private:
    int _hour;
    int _minute;
    int _second;
};

class Date
{
private:
    // 基本类型(内置类型)
    int _year = 1970;
    int _month = 1;
    int _day = 1;

    // 自定义类型
    Time _t;
};

int main()
{
    Date d1;
    Date d2;
    d1 = d2;
    return 0;
}
```

既然**编译器生成的默认赋值运算符重载函数已经可以完成字节序的值拷贝了**，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。
typedef int DataType;
class Stack
{
public:
    Stack(size_t capacity = 10)
    {
        _array = (DataType*)malloc(capacity * sizeof(DataType));
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            return;
        }
```

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

    void Push(const DataType& data)
    {
        // CheckCapacity();
        _array[_size] = data;
        _size++;
    }

    ~Stack()
    {
        if (_array)
        {
            free(_array);
            _array = nullptr;
            _capacity = 0;
            _size = 0;
        }
    }

private:
    DataType *_array;
    size_t _size;
    size_t _capacity;
};
int main()
{
    Stack s1;
    s1.Push(1);
    s1.Push(2);
    s1.Push(3);
    s1.Push(4);

    Stack s2;
    s2 = s1;
    return 0;
}
```

注意：如果类中未涉及到资源管理，赋值运算符是否实现都可以；一旦涉及到资源管理则必须要实现。



5.3 前置++和后置++重载

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

    // 前置++：返回+1之后的结果
    // 注意：this指向的对象函数结束后不会销毁，故以引用方式返回提高效率
    Date& operator++()
    {
        _day += 1;
        return *this;
    }

    // 后置++：
    // 前置++和后置++都是一元运算符，为了让前置++与后置++形成能正确重载
    // C++规定：后置++重载时多增加一个int类型的参数，但调用函数时该参数不用传递，编译器自动传递
    // 注意：后置++是先使用后+1，因此需要返回+1之前的旧值，故需在实现时需要先将this保存一份，然后给this+1
    //       而temp是临时对象，因此只能以值的方式返回，不能返回引用
    Date operator++(int)
    {
        Date temp(*this);
        _day += 1;
        return temp;
    }

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

```
int main()
{
    Date d;
    Date d1(2022, 1, 13);
    d = d1++;    // d: 2022,1,13   d1:2022,1,14
    d = ++d1;    // d: 2022,1,15   d1:2022,1,15
    return 0;
}
```

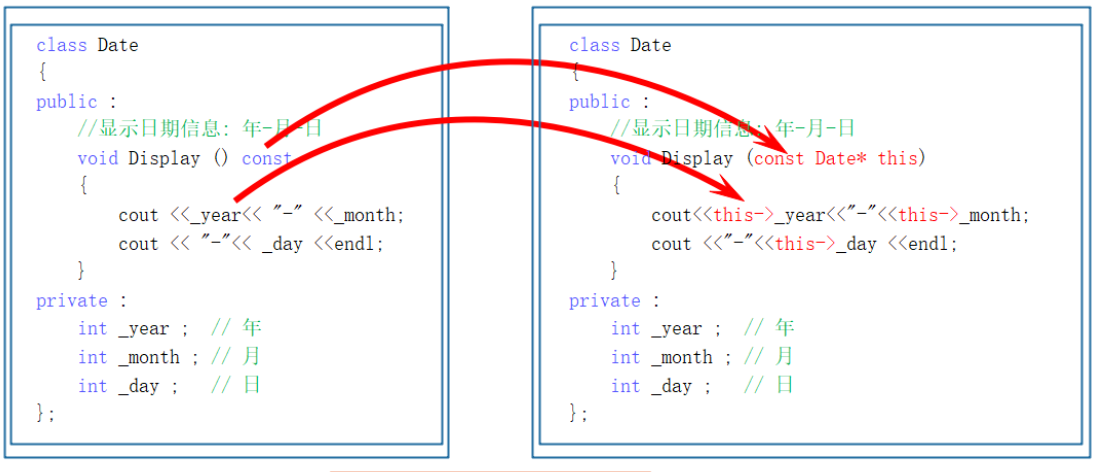
6.日期类的实现

```
class Date
{
public:
    // 获取某年某月的天数
    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);

    // 拷贝构造函数
    // d2(d1)
    Date(const Date& d);

    // 赋值运算符重载
    // d2 = d3 -> d2.operator=(&d2, d3)
    Date& operator=(const Date& d);

    // 析构函数
    ~Date();

    // 日期+=天数
    Date& operator+=(int day);

    // 日期+天数
    Date operator+(int day);

    // 日期-天数
    Date operator-(int day);

     // 日期-=天数
    Date& operator-=(int day);

    // 前置++
```

```
    Date& operator++();

    // 后置++
    Date operator++(int);

    // 后置--
    Date operator--(int);

    // 前置--
    Date& operator--();

    // >运算符重载
    bool operator>(const Date& d);

    // ==运算符重载
    bool operator==(const Date& d);

    // >=运算符重载
    bool operator >= (const Date& d);

    // <运算符重载
    bool operator < (const Date& d);

     // <=运算符重载
    bool operator <= (const Date& d);

    // !=运算符重载
    bool operator != (const Date& d);

    // 日期-日期  返回天数
    int operator-(const Date& d);
private:
    int _year;
    int _month;
    int _day;
};
```

7.const成员

> **将const修饰的“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:
    Date(int year, int month, int day)
    {
        _year = year;
        _month = month;
        _day = day;
    }

    void Print()
    {
        cout << "Print()" << endl;
        cout << "year:" << _year << endl;
        cout << "month:" << _month << endl;
        cout << "day:" << _day << endl << endl;
    }
    void Print() const
    {
        cout << "Print()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(2022,1,13);
    d1.Print();

    const Date d2(2022,1,13);
    d2.Print();
}
```

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！**