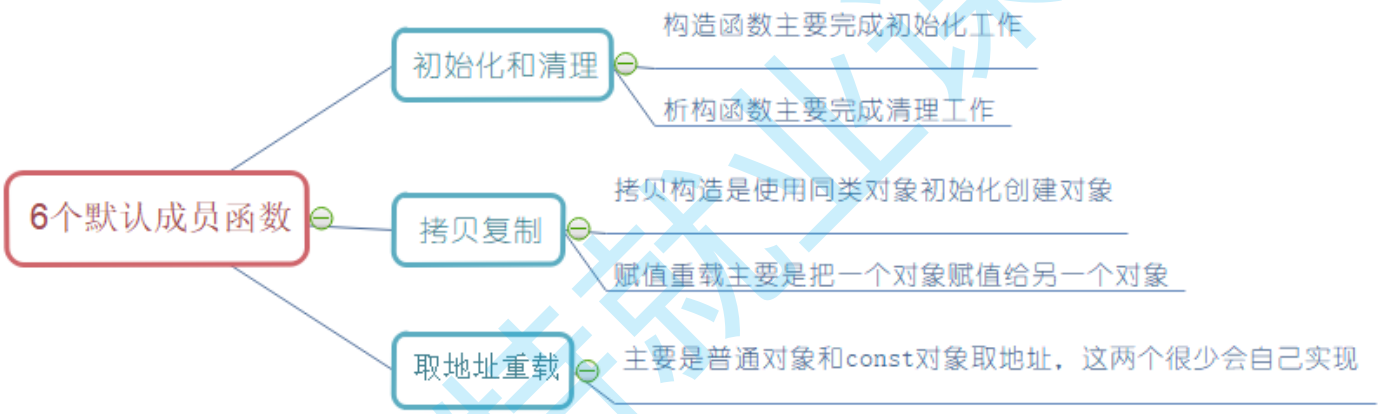
3.类和对象(中)

1. 类的默认成员函数

默认成员函数就是用户没有显式实现，编译器会自动生成的成员函数称为默认成员函数。一个类，我们不写的情况下编译器会默认生成以下6个默认成员函数，需要注意的是这6个中最重要的是前4个，最后两个取地址重载不重要，我们稍微了解一下即可。其次就是C++11以后还会增加两个默认成员函数，移动构造和移动赋值，这个我们后面再讲解。默认成员函数很重要，也比较复杂，我们要从两个方面去学习：

- 第一：我们不写时，编译器默认生成的函数行为是什么，是否满足我们的需求。
- 第二：编译器默认生成的函数不满足我们的需求，我们需要自己实现，那么如何自己实现？



2. 构造函数

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(我们常使用的局部对象是栈帧创建时，空间就开好了)，而是对象实例化时初始化对象。构造函数的本质是要替代我们以前Stack和Date类中写的Init函数的功能，构造函数自动调用的特点就完美的替代的了Init。

构造函数的特点：

1. 函数名与类名相同。
2. 无返回值。 (返回值啥都不需要给，也不需要写void，不要纠结，C++规定如此)
3. 对象实例化时系统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。

6. 无参构造函数、全缺省构造函数、我们不写构造时编译器默认生成的构造函数，都叫做默认构造函数。但是这三个函数有且只有一个存在，不能同时存在。无参构造函数和全缺省构造函数虽然构成函数重载，但是调用时会存在歧义。要注意很多同学会认为默认构造函数是编译器默认生成那个叫默认构造，实际上无参构造函数、全缺省构造函数也是默认构造，总结一下就是不传实参就可以调用的构造就叫默认构造。
7. 我们不写，编译器默认生成的构造，对内置类型成员变量的初始化没有要求，也就是说是是否初始化是不确定的，看编译器。对于自定义类型成员变量，要求调用这个成员变量的默认构造函数初始化。如果这个成员变量，没有默认构造函数，那么就会报错，我们要初始化这个成员变量，需要用初始化列表才能解决，初始化列表，我们下个章节再细细讲解。

说明：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原生数据类型，如：int/char/double/指针等，自定义类型就是我们使用class/struct等关键字自己定义的类型。

```cpp
#include<iostream>
using namespace std;
class Date
{
public:
    // 1.无参构造函数
    Date()
    {
        _year = 1;
        _month = 1;
        _day = 1;
    }

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

    // 3.全缺省构造函数
    /*Date(int year = 1, int month = 1, int day = 1)
    {
        _year = year;
        _month = month;
        _day = day;
    }*/

    void Print()
    {
        cout << _year << "/" << _month << "/" << _day << endl;
```

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

int main()
{
    // 如果留下三个构造中的第二个带参构造，第一个和第三个注释掉
    // 编译报错：error C2512: “Date”: 没有合适的默认构造函数可用
    Date d1;              // 调用默认构造函数
    Date d2(2025, 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，否则编译器无法
    // 区分这里是函数声明还是实例化对象
    // 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();

    d1.Print();
    d2.Print();

    return 0;
}
```

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

typedef int STDataType;
class Stack
{
public:
    Stack(int n = 4)
    {
        _a = (STDataType*)malloc(sizeof(STDataType) * n);
        if (nullptr == _a)
        {
            perror("malloc申请空间失败");
            return;
        }

        _capacity = n;
        _top = 0;
    }
    // ...
```

```
21 private:
22     STDataType* _a;
23     size_t _capacity;
24     size_t _top;
25 };
26
27 // 两个Stack实现队列
28 class MyQueue
29 {
30 public:
31     //编译器默认生成MyQueue的构造函数调用了Stack的构造，完成了两个成员的初始化
32 private:
33     Stack pushst;
34     Stack popst;
35 };
36
37 int main()
38 {
39     MyQueue mq;
40
41     return 0;
42 }
```

3. 析构函数

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，比如局部对象是存在栈帧的，函数结束栈帧销毁，他就释放了，不需要我们管，C++规定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释放工作。析构函数的功能类比我们之前Stack实现的Destroy功能，而像Date没有Destroy，其实就是没有资源需要释放，所以严格说Date是不需要析构函数的。

析构函数的特点：

1. 析构函数名是在类名前加上字符 ~。
2. 无参数无返回值。 (这里跟构造类似，也不需要加void)
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。
4. 对象生命周期结束时，系统会自动调用析构函数。
5. 跟构造函数类似，我们不写编译器自动生成的析构函数对内置类型成员不做处理，自定类型成员会调用他的析构函数。
6. 还需要注意的是我们显示写析构函数，对于自定义类型成员也会调用他的析构，也就是说自定义类型成员无论什么情况都会自动调用析构函数。

7. 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，如Date；如果默认生成的析构就可以用，也就不需要显示写析构，如MyQueue；但是有资源申请时，一定要自己写析构，否则会造成资源泄漏，如Stack。
8. 一个局部域的多个对象，C++规定后定义的先析构。

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

typedef int STDataType;
class Stack
{
public:
    Stack(int n = 4)
    {
        _a = (STDataType*)malloc(sizeof(STDataType) * n);
        if (nullptr == _a)
        {
            perror("malloc申请空间失败");
            return;
        }

        _capacity = n;
        _top = 0;
    }

    ~Stack()
    {
        cout << "~Stack()" << endl;

        free(_a);
        _a = nullptr;
        _top = _capacity = 0;
    }

private:
    STDataType* _a;
    size_t _capacity;
    size_t _top;
};

// 两个Stack实现队列
class MyQueue
{
public:
    //编译器默认生成MyQueue的析构函数调用了Stack的析构，释放的Stack内部的资源
```

```
41
42      // 显示写析构，也会自动调用Stack的析构
43      /*~MyQueue()
44      {}*/
45  private:
46      Stack pushst;
47      Stack popst;
48  };
49
50  int main()
51  {
52      Stack st;
53
54      MyQueue mq;
55
56      return 0;
57  }
```

对比一下用C++和C实现的Stack解决之前括号匹配问题isValid，我们发现有了构造函数和析构函数确实方便了很多，不会再忘记调用Init和Destory函数了，也方便了不少。

```
1  #include<iostream>
2  using namespace std;
3
4  // 用最新加了构造和析构的C++版本Stack实现
5  bool isValid(const char* s) {
6          Stack st;
7
8          while (*s)
9          {
10                 if (*s == '[' || *s == '(' || *s == '{')
11                 {
12                         st.Push(*s);
13                 }
14                 else
15                 {
16                         // 右括号比左括号多，数量匹配问题
17                         if (st.Empty())
18                         {
19                                 return false;
20                         }
21
22                         // 栈里面取左括号
23                         char top = st.Top();
24                         st.Pop();
```

```
25
26                          // 顺序不匹配
27                          if ((*s == ']' && top != '[')
28                                  || (*s == '}' && top != '{')
29                                  || (*s == ')' && top != '('))
30                          {
31                                  return false;
32                          }
33                  }
34
35                  ++s;
36          }
37
38          // 栈为空，返回真，说明数量都匹配  左括号多，右括号少匹配问题
39          return st.Empty();
40  }
41
42  // 用之前C版本Stack实现
43  bool isValid(const char* s) {
44          ST st;
45          STInit(&st);
46          while (*s)
47          {
48                  // 左括号入栈
49                  if (*s == '(' || *s == '[' || *s == '{')
50                  {
51                          STPush(&st, *s);
52                  }
53                  else // 右括号取栈顶左括号尝试匹配
54                  {
55                          if (STEmpty(&st))
56                          {
57                                  STDestroy(&st);
58                                  return false;
59                          }
60
61                          char top = STTop(&st);
62                          STPop(&st);
63
64                          // 不匹配
65                          if ((top == '(' && *s != ')')
66                                  || (top == '{' && *s != '}')
67                                  || (top == '[' && *s != ']'))
68                          {
69                                  STDestroy(&st);
70                                  return false;
71                          }
```

```
72                  }
73
74                  ++s;
75          }
76
77          // 栈不为空，说明左括号比右括号多，数量不匹配
78          bool ret = STEmpty(&st);
79          STDestroy(&st);
80
81          return ret;
82  }
83
84  int main()
85  {
86      cout << isValid("[()][]") << endl;
87      cout << isValid("[(])[]") << endl;
88
89
90      return 0;
91  }
```

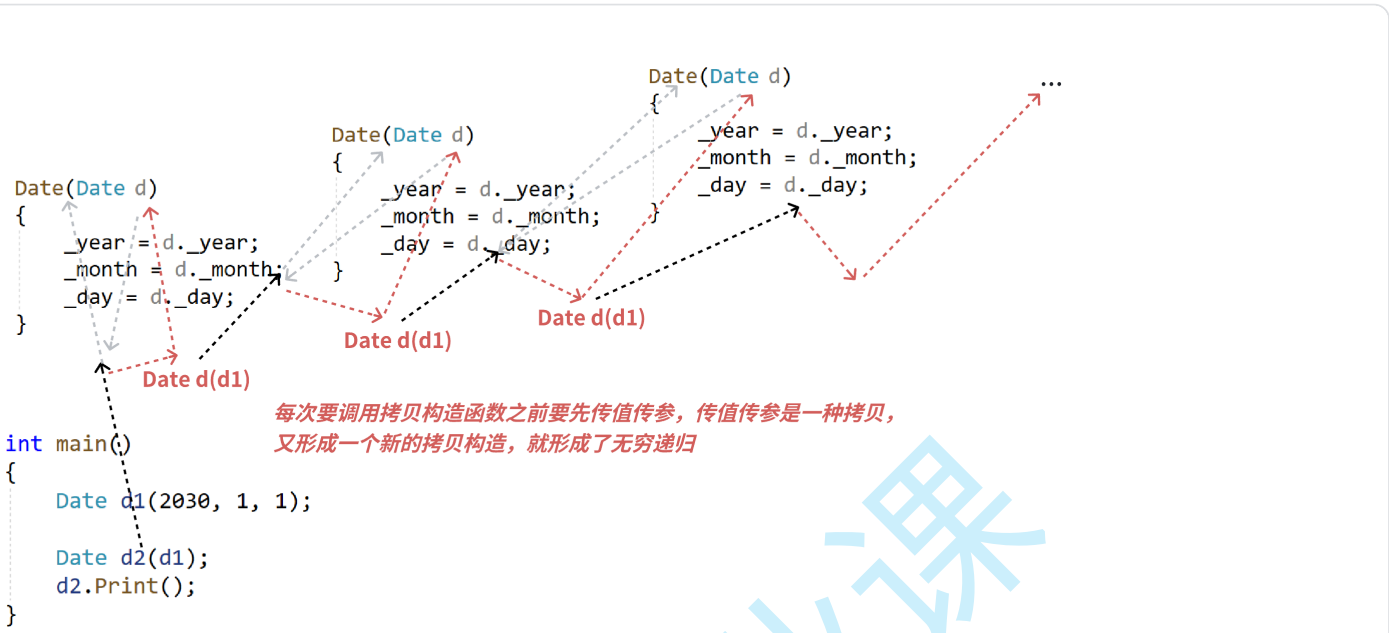
4. 拷贝构造函数

如果一个构造函数的第一个参数是自身类类型的引用，且任何额外的参数都有默认值，则此构造函数也叫做拷贝构造函数，也就是说拷贝构造是一个特殊的构造函数。

拷贝构造的特点：

1. 拷贝构造函数是构造函数的一个重载。
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，因为语法逻辑上会引发无穷递归调用。
3. C++规定自定义类型对象进行拷贝行为必须调用拷贝构造，所以这里自定义类型传值传参和传值返回都会调用拷贝构造完成。
4. 若未显式定义拷贝构造，编译器会生成自动生成拷贝构造函数。自动生成的拷贝构造对内置类型成员变量会完成值拷贝/浅拷贝(一个字节一个字节的拷贝)，对自定义类型成员变量会调用他的拷贝构造。
5. 像Date这样的类成员变量全是内置类型且没有指向什么资源，编译器自动生成的拷贝构造就可以完成需要的拷贝，所以不需要我们显示实现拷贝构造。像Stack这样的类，虽然也都是内置类型，但是_a指向了资源，编译器自动生成的拷贝构造完成的值拷贝/浅拷贝不符合我们的需求，所以需要我们自己实现深拷贝(对指向的资源也进行拷贝)。像MyQueue这样的类型内部主要是自定义类型Stack成员，编译器自动生成的拷贝构造会调用Stack的拷贝构造，也不需要我们显示实现MyQueue的拷贝构造。这里还有一个小技巧，如果一个类显示实现了析构并释放资源，那么他就需要显示写拷贝构造，否则就不需要。

6. 传值返回会产生一个临时对象调用拷贝构造，传值引用返回，返回的是返回对象的别名(引用)，没有产生拷贝。但是如果返回对象是一个当前函数局部域的局部对象，函数结束就销毁了，那么使用引用返回是有问题的，这时的引用相当于一个野引用，类似一个野指针一样。传引用返回可以减少拷贝，但是一定要确保返回对象，在当前函数结束后还在，才能用引用返回。




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

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

	// 编译报错：error C2652: “Date”: 非法的复制构造函数：第一个参数不应是“Date”
	//Date(Date d)
	Date(const Date& d)
	{
		_year = d._year;
		_month = d._month;
		_day = d._day;
	}

	Date(Date* d)
	{
```

```
25                  _year = d->_year;
26                  _month = d->_month;
27                  _day = d->_day;
28          }
29
30          void Print()
31          {
32                  cout << _year << "-" << _month << "-" << _day << endl;
33          }
34  private:
35          int _year;
36          int _month;
37          int _day;
38  };
39
40  void Func1(Date d)
41  {
42          cout << &d << endl;
43          d.Print();
44  }
45
46  // Date Func2()
47  Date& Func2()
48  {
49      Date tmp(2024, 7, 5);
50      tmp.Print();
51
52      return tmp;
53  }
54
55  int main()
56  {
57          Date d1(2024, 7, 5);
58          // C++规定自定义类型对象进行拷贝行为必须调用拷贝构造，所以这里传值传参要调用拷贝
    构造
59          // 所以这里的d1传值传参给d要调用拷贝构造完成拷贝，传引用传参可以较少这里的拷贝
60          Func1(d1);
61
62          cout << &d1 << endl;
63
64          // 这里可以完成拷贝，但是不是拷贝构造，只是一个普通的构造
65          Date d2(&d1);
66          d1.Print();
67          d2.Print();
68
69          //这样写才是拷贝构造，通过同类型的对象初始化构造，而不是指针
70          Date d3(d1);
```

```
71          d2.Print();
72
73          // 也可以这样写，这里也是拷贝构造
74          Date d4 = d1;
75          d2.Print();
76
77          // Func2返回了一个局部对象tmp的引用作为返回值
78          // Func2函数结束，tmp对象就销毁了，相当于了一个野引用
79          Date ret = Func2();
80          ret.Print();
81
82          return 0;
83  }
```

```
1   #include<iostream>
2   using namespace std;
3
4   typedef int STDataType;
5   class Stack
6   {
7   public:
8       Stack(int n = 4)
9       {
10          _a = (STDataType*)malloc(sizeof(STDataType) * n);
11          if (nullptr == _a)
12          {
13              perror("malloc申请空间失败");
14              return;
15          }
16
17          _capacity = n;
18          _top = 0;
19      }
20
21      Stack(const Stack& st)
22          {
23          // 需要对_a指向资源创建同样大的资源再拷贝值
24                  _a = (STDataType*)malloc(sizeof(STDataType) * st._capacity);
25                  if (nullptr == _a)
26                  {
27                          perror("malloc申请空间失败!!!");
28                          return;
29                  }
30
31                  memcpy(_a, st._a, sizeof(STDataType) * st._top);
```

```
32
33          _top = st._top;
34                  _capacity = st._capacity;
35          }
36
37      void Push(STDataType x)
38      {
39          if (_top == _capacity)
40          {
41              int newcapacity = _capacity * 2;
42              STDataType* tmp = (STDataType*)realloc(_a, newcapacity *
    sizeof(STDataType));
43              if (tmp == NULL)
44              {
45                  perror("realloc fail");
46                  return;
47              }
48
49              _a = tmp;
50              _capacity = newcapacity;
51          }
52
53          _a[_top++] = x;
54      }
55
56
57      ~Stack()
58      {
59          cout << "~Stack()" << endl;
60
61          free(_a);
62          _a = nullptr;
63          _top = _capacity = 0;
64      }
65
66  private:
67      STDataType* _a;
68      size_t _capacity;
69      size_t _top;
70  };
71
72  // 两个Stack实现队列
73  class MyQueue
74  {
75  public:
76  private:
77      Stack pushst;
```

```
78      Stack popst;
79  };
80
81  int main()
82  {
83      Stack st1;
84      st1.Push(1);
85      st1.Push(2);
86      // Stack不显示实现拷贝构造，用自动生成的拷贝构造完成浅拷贝
87      // 会导致st1和st2里面的_a指针指向同一块资源，析构时会析构两次，程序崩溃
88      Stack st2 = st1;
89
90      MyQueue mq1;
91      // MyQueue自动生成的拷贝构造，会自动调用Stack拷贝构造完成pushst/popst
92      // 的拷贝，只要Stack拷贝构造自己实现了深拷贝，他就没问题
93      MyQueue mq2 = mq1;
94
95      return 0;
96  }
```

5. 赋值运算符重载

5.1 运算符重载

- 当运算符被用于类类型的对象时，C++语言允许我们通过运算符重载的形式指定新的含义。C++规定类类型对象使用运算符时，必须转换成调用对应运算符重载，若没有对应的运算符重载，则会编译报错。
- 运算符重载是具有特名字的函数，他的名字是由operator和后面要定义的运算符共同构成。和其他函数一样，它也具有其返回类型和参数列表以及函数体。
- 重载运算符函数的参数个数和该运算符作用的运算对象数量一样多。一元运算符有一个参数，二元运算符有两个参数，二元运算符的左侧运算对象传给第一个参数，右侧运算对象传给第二个参数。
- 如果一个重载运算符函数是成员函数，则它的第一个运算对象默认传给隐式的this指针，因此运算符重载作为成员函数时，参数比运算对象少一个。
- 运算符重载以后，其优先级和结合性与对应的内置类型运算符保持一致。
- 不能通过连接语法中没有的符号来创建新的操作符：比如operator@。
- `.*  ::  sizeof  ?:  .` 注意以上5个运算符不能重载。(选择题里面常考，大家要记一下)
- 重载操作符至少有一个类类型参数，不能通过运算符重载改变内置类型对象的含义，如：`int operator+(int x, int y)`
- 一个类需要重载哪些运算符，是看哪些运算符重载后有意义，比如Date类重载operator-就有意义，但是重载operator+就没有意义。

- 重载++运算符时，有前置++和后置++，运算符重载函数名都是operator++，无法很好的区分。C++规定，后置++重载时，增加一个int形参，跟前置++构成函数重载，方便区分。
- 重载<<和>>时，需要重载为全局函数，因为重载为成员函数，this指针默认抢占了第一个形参位置，第一个形参位置是左侧运算对象，调用时就变成了 对象<<cout，不符合使用习惯和可读性。重载为全局函数把ostream/istream放到第一个形参位置就可以了，第二个形参位置当类类型对象。

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

// 编译报错：“operator +”必须至少有一个类类型的形参
int operator+(int x, int y)
{
    return x - y;
}

class A
{
public:
    void func()
    {
        cout << "A::func()" << endl;
    }
};

typedef void(A::*PF)(); //成员函数指针类型

int main()
{
    // C++规定成员函数要加&才能取到函数指针
    PF pf = &A::func;

    A obj;//定义ob类对象temp

    // 对象调用成员函数指针时，使用.*运算符
    (obj.*pf)();

    return 0;
}
```

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

```

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

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

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

// 重载为全局的面临对象访问私有成员变量的问题
// 有几种方法可以解决:
// 1、成员放公有
// 2、Date提供getxxx函数
// 3、友元函数
// 4、重载为成员函数
bool operator==(const Date& d1, const Date& d2)
{
        return d1._year == d2._year
               && d1._month == d2._month
               && d1._day == d2._day;
}

int main()
{
    Date d1(2024, 7, 5);
    Date d2(2024, 7, 6);

    // 运算符重载函数可以显示调用
    operator==(d1, d2);

    // 编译器会转换成 operator==(d1, d2);
    d1 == d2;

    return 0;
}
```

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

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

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

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

    Date& operator++()
    {
        cout << "前置++" << endl;
        //...
        return *this;
    }

    Date operator++(int)
    {
        Date tmp;
        cout << "后置++" << endl;

        //...
        return tmp;
    }
private:
    int _year;
    int _month;
    int _day;
};
```

```cpp

int main()
{
    Date d1(2024, 7, 5);
    Date d2(2024, 7, 6);

    // 运算符重载函数可以显示调用
    d1.operator==(d2);

    // 编译器会转换成 d1.operator==(d2);
    d1 == d2;

    // 编译器会转换成 d1.operator++();
    ++d1;
    // 编译器会转换成 d1.operator++(0);
    d1++;

    return 0;
}
```

5.2 赋值运算符重载

赋值运算符重载是一个默认成员函数，用于完成两个已经存在的对象直接的拷贝赋值，这里要注意跟拷贝构造区分，拷贝构造用于一个对象拷贝初始化给另一个要创建的对象。

赋值运算符重载的特点：

1. 赋值运算符重载是一个运算符重载，规定必须重载为成员函数。赋值运算重载的参数建议写成const 当前类类型引用，否则会传值传参会有拷贝
2. 有返回值，且建议写成当前类类型引用，引用返回可以提高效率，有返回值目的是为了支持连续赋值场景。
3. 没有显式实现时，编译器会自动生成一个默认赋值运算符重载，默认赋值运算符重载行为跟默认构造函数类似，对内置类型成员变量会完成值拷贝/浅拷贝(一个字节一个字节的拷贝)，对自定义类型成员变量会调用他的拷贝构造。
4. 像Date这样的类成员变量全是内置类型且没有指向什么资源，编译器自动生成的赋值运算符重载就可以完成需要的拷贝，所以不需要我们显示实现赋值运算符重载。像Stack这样的类，虽然也都是内置类型，但是_a指向了资源，编译器自动生成的赋值运算符重载完成的值拷贝/浅拷贝不符合我们的需求，所以需要我们自己实现深拷贝(对指向的资源也进行拷贝)。像MyQueue这样的类型内部主要是自定义类型Stack成员，编译器自动生成的赋值运算符重载会调用Stack的赋值运算符重载，也不需要我们显示实现MyQueue的赋值运算符重载。这里还有一个小技巧，如果一个类显示实现了析构并释放资源，那么他就需要显示写赋值运算符重载，否则就不需要。

```cpp
class Date
```

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

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

        _year = d._year;
        _month = d._month;
        _day = d._day;
    }

    // 传引用返回减少拷贝
    // d1 = d2;
    Date& operator=(const Date& d)
    {
        // 不要检查自己给自己赋值的情况
        if (this != &d)
        {
            _year = d._year;
            _month = d._month;
            _day = d._day;
        }

        // d1 = d2表达式的返回对象应该为d1，也就是*this
        return *this;
    }

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


int main()
{
```

```cpp
    Date d1(2024, 7, 5);
    Date d2(d1);

    Date d3(2024, 7, 6);
    d1 = d3;

    // 需要注意这里是拷贝构造，不是赋值重载
    // 请牢牢记住赋值重载完成两个已经存在的对象直接的拷贝赋值
    // 而拷贝构造用于一个对象拷贝初始化给另一个要创建的对象
    Date d4 = d1;

    return 0;
}
```

5.3 日期类实现

```cpp
#pragma once
#include<iostream>
using namespace std;

#include<assert.h>

class Date
{
        // 友元函数声明
        friend ostream& operator<<(ostream& out, const Date& d);
        friend istream& operator>>(istream& in, Date& d);

public:
        Date(int year = 1900, int month = 1, int day = 1);
        void Print() const;

        // 直接定义类里面，他默认是inline
        // 频繁调用
        int GetMonthDay(int year, int month)
        {
                assert(month > 0 && month < 13);

                static int monthDayArray[13] = { -1, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

                // 365天 5h +
                if (month == 2 && (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
                {
```

```
28                          return 29;
29                  }
30                  else
31                  {
32                          return monthDayArray[month];
33                  }
34          }
35
36          bool CheckDate();
37
38          bool operator<(const Date& d) const;
39          bool operator<=(const Date& d) const;
40          bool operator>(const Date& d) const;
41          bool operator>=(const Date& d) const;
42          bool operator==(const Date& d) const;
43          bool operator!=(const Date& d) const;
44
45          // d1 += 天数
46          Date& operator+=(int day);
47          Date operator+(int day) const;
48
49          // d1 -= 天数
50          Date& operator-=(int day);
51          Date operator-(int day) const;
52
53          // d1 - d2
54          int operator-(const Date& d) const;
55
56          // ++d1 -> d1.operator++()
57          Date& operator++();
58
59          // d1++ -> d1.operator++(0)
60          // 为了区分，构成重载，给后置++，强行增加了一个int形参
61          // 这里不需要写形参名，因为接收值是多少不重要，也不需要用
62          // 这个参数仅仅是为了跟前置++构成重载区分
63          Date operator++(int);
64
65          Date& operator--();
66          Date operator--(int);
67
68          // 流插入
69          // 不建议，因为Date* this占据了一个参数位置，使用d<<cout不符合习惯
70          //void operator<<(ostream& out);
71  private:
72          int _year;
73          int _month;
74          int _day;
```

```cpp
};

// 重载
ostream& operator<<(ostream& out, const Date& d);
istream& operator>>(istream& in, Date& d);

// Date.cpp
#include"Date.h"

bool Date::CheckDate()
{
	if (_month < 1 || _month > 12
		|| _day < 1 || _day > GetMonthDay(_year, _month))
	{
		return false;
	}
	else
	{
		return true;
	}
}

Date::Date(int year, int month, int day)
{
	_year = year;
	_month = month;
	_day = day;

	if (!CheckDate())
	{
		cout << "日期非法" << endl;
	}
}

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

// d1 < d2
bool Date::operator<(const Date& d) const
{
	if (_year < d._year)
	{
		return true;
	}
	else if (_year == d._year)
```

```
122         {
123                 if (_month < d._month)
124                 {
125                         return true;
126                 }
127                 else if (_month == d._month)
128                 {
129                         return _day < d._day;
130                 }
131         }
132
133         return false;
134 }
135
136 // d1 <= d2
137 bool Date::operator<=(const Date& d) const
138 {
139         return *this < d || *this == d;
140 }
141
142 bool Date::operator>(const Date& d) const
143 {
144         return !(*this <= d);
145 }
146
147 bool Date::operator>=(const Date& d) const
148 {
149         return !(*this < d);
150 }
151
152 bool Date::operator==(const Date& d) const
153 {
154         return _year == d._year
155                 && _month == d._month
156                 && _day == d._day;
157 }
158
159 bool Date::operator!=(const Date& d) const
160 {
161         return !(*this == d);
162 }
163
164 // d1 += 50
165 // d1 += -50
166 Date& Date::operator+=(int day)
167 {
168         if (day < 0)
```

```
169         {
170                 return *this -= -day;
171         }
172
173         _day += day;
174         while (_day > GetMonthDay(_year, _month))
175         {
176                 _day -= GetMonthDay(_year, _month);
177                 ++_month;
178                 if (_month == 13)
179                 {
180                         ++_year;
181                         _month = 1;
182                 }
183         }
184
185         return *this;
186 }
187
188
189 Date Date::operator+(int day) const
190 {
191         Date tmp = *this;
192         tmp += day;
193
194         return tmp;
195 }
196
197 // d1 -= 100
198 Date& Date::operator-=(int day)
199 {
200         if (day < 0)
201         {
202                 return *this += -day;
203         }
204
205         _day -= day;
206         while (_day <= 0)
207         {
208                 --_month;
209                 if (_month == 0)
210                 {
211                         _month = 12;
212                         _year--;
213                 }
214
215                 // 借上一个月的天数
```

```
216                 _day += GetMonthDay(_year, _month);
217         }
218
219         return *this;
220 }
221
222 Date Date::operator-(int day) const
223 {
224         Date tmp = *this;
225         tmp -= day;
226
227         return tmp;
228 }
229
230 //++d1
231 Date& Date::operator++()
232 {
233         *this += 1;
234
235         return *this;
236 }
237
238 // d1++
239 Date Date::operator++(int)
240 {
241         Date tmp(*this);
242         *this += 1;
243
244         return tmp;
245 }
246
247 Date& Date::operator--()
248 {
249         *this -= 1;
250         return *this;
251 }
252
253 Date Date::operator--(int)
254 {
255         Date tmp = *this;
256         *this -= 1;
257
258         return tmp;
259 }
260
261 // d1 - d2
262 int Date::operator-(const Date& d) const
```

```cpp
{
	Date max = *this;
	Date min = d;
	int flag = 1;
	if (*this < d)
	{
		max = d;
		min = *this;
		flag = -1;
	}

	int n = 0;
	while (min != max)
	{
		++min;
		++n;
	}

	return n * flag;
}

ostream& operator<<(ostream& out, const Date& d)
{
	out << d._year << "年" << d._month << "月" << d._day << "日" << endl;
	return out;
}

istream& operator>>(istream& in, Date& d)
{
	cout << "请依次输入年月日:>";
	in >> d._year >> d._month >> d._day;

	if (!d.CheckDate())
	{
		cout << "日期非法" << endl;
	}

	return in;
}

// Test.cpp
#include"Date.h"
void TestDate1()
{
	// 这里需要测试一下大的数据+和-
	Date d1(2024, 4, 14);
	Date d2 = d1 + 30000;
```

```cpp
        d1.Print();
        d2.Print();

        Date d3(2024, 4, 14);
        Date d4 = d3 - 5000;
        d3.Print();
        d4.Print();

        Date d5(2024, 4, 14);
        d5 += -5000;
        d5.Print();
}

void TestDate2()
{
        Date d1(2024, 4, 14);
        Date d2 = ++d1;
        d1.Print();
        d2.Print();

        Date d3 = d1++;
        d1.Print();
        d3.Print();

        /*d1.operator++(1);
        d1.operator++(100);
        d1.operator++(0);
        d1.Print();*/
}

void TestDate3()
{
        Date d1(2024, 4, 14);
        Date d2(2034, 4, 14);

        int n = d1 - d2;
        cout << n << endl;

        n = d2 - d1;
}

void TestDate4()
{
        Date d1(2024, 4, 14);
        Date d2 = d1 + 30000;

        // operator<<(cout, d1)
```

```
        cout << d1;
        cout << d2;

        cin >> d1 >> d2;
        cout << d1 << d2;
}

void TestDate5()
{
        const Date d1(2024, 4, 14);
        d1.Print();

        //d1 += 100;
        d1 + 100;

        Date d2(2024, 4, 25);
        d2.Print();

        d2 += 100;

        d1 < d2;
        d2 < d1;
}

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


6. 取地址运算符重载

6.1 const成员函数

- 将const修饰的成员函数称之为const成员函数，const修饰成员函数放到成员函数参数列表的后面。
- const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。const 修饰Date类的Print成员函数，Print隐含的this指针由 `Date* const this` 变为 `const Date* const this`

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

class Date
{
```

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

    // void Print(const Date* const this) const
    void Print() const
    {
        cout << _year << "-" << _month << "-" << _day << endl;
    }
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    // 这里非const对象也可以调用const成员函数是一种权限的缩小
    Date d1(2024, 7, 5);
    d1.Print();

    const Date d2(2024, 8, 5);
    d2.Print();

    return 0;
}
```


6.2 取地址运算符重载

取地址运算符重载分为普通取地址运算符重载和const取地址运算符重载，一般这两个函数编译器自动生成的就可以够我们用了，不需要去显示实现。除非一些很特殊的场景，比如我们不想让别人取到当前类对象的地址，就可以自己实现一份，胡乱返回一个地址。

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

```
 9
10          const Date* operator&()const
11          {
12                return this;
13                // return nullptr;
14          }
15  private :
16          int _year ; // 年
17          int _month ; // 月
18          int _day ; // 日
19  };
```