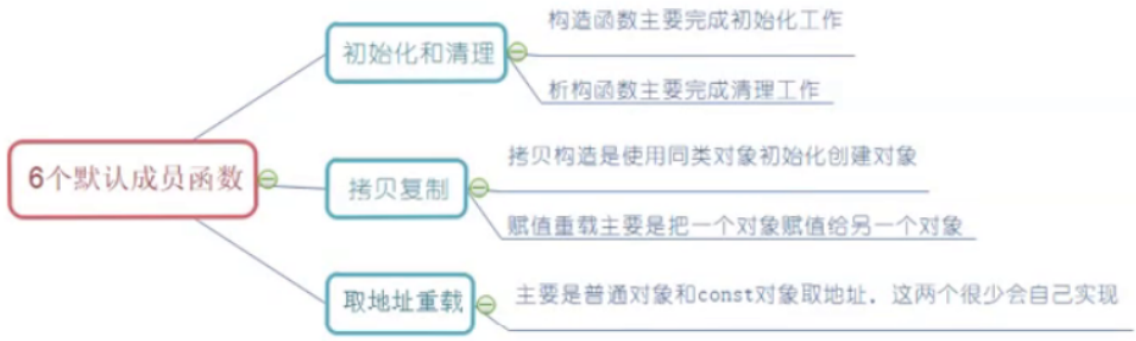
<C++> 类和对象 - 构造函数

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。



2. 构造函数

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**。

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。**

构造函数的目的就是为了创建对象的时候同时初始化。

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。也就是说在创建对象的时候，就对成员变量进行初始化。

下面是一个 Stack 栈的类：

C/C++

```
class Stack
{
public:
    //无参数构造函数 - 函数名与对象名相同
    Stack()
    {
        _a = nullptr;
        _size = _capacity = 0;
    }
    //带参数构造函数
    Stack(int n)
    {
        _a = (int*)malloc(sizeof(int) * n);
        if (nullptr == _a)
        {
            perror("malloc fail");
            exit(-1);
        }
        _capacity = n;
        _size = 0;
    }
    void Push(int x) {}
    void Destroy() {}
private:
    int* _a;
    int _size;
    int _capacity;
};
```

```cpp
int main()
{
    Stack st();

    st.Push(1);
    st.Push(2);
    st.Push(3);
    st.Push(4);
    st.Destroy();
}
```

```cpp
int main()
{
    //Stack st();  err出现歧义，编译器无法知道是在创建对象，还是调用函数
    Stack st; //无参数构造函数
    Stack st1(5); //带参数构造函数

    st.Push(1);
    st.Push(2);
    st1.Push(3);
    st1.Push(4);
    st.Destroy();
    st1.Destroy();
}
```

> 4.构造函数可以重载。(一个类可以有多个构造函数，也就是多种初始化方式)

C/C++

```cpp
//Date类举例
class Date
{
public:
    //无参构造函数
    Date()
    {
        _year = 1;
        _month = 1;
        _day = 1;
    }
    //带参构造函数
    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;
};
```

使用全缺省构造函数

在写构造函数的时候，推荐使用全缺省或者半缺省

C/C++

```
class Date
{
public:
    //全缺省
    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;
};
```

```cpp
class Date
{
public:
    //全缺省
    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;
};
```

```cpp
int main()
{
    Date d1;
    Date d2(2023, 2, 6);
    Date d3(2023);
    Date d4(2023, 2);
    d1.Print();
    d2.Print();
    d3.Print();
    d4.Print();
    return 0;
}
```

```
Microsoft Visua
1/1/1
2023/2/6
2023/1/1
2023/2/1

E:\VS\Project
按任意键关闭此
```

全缺省构造函数不能和无参构造函数同时存在 - 出现歧义，编译器不知道调用哪个

> 5.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。

为什么只能有一个默认构造函数？ 因为调用的时候会发生歧义

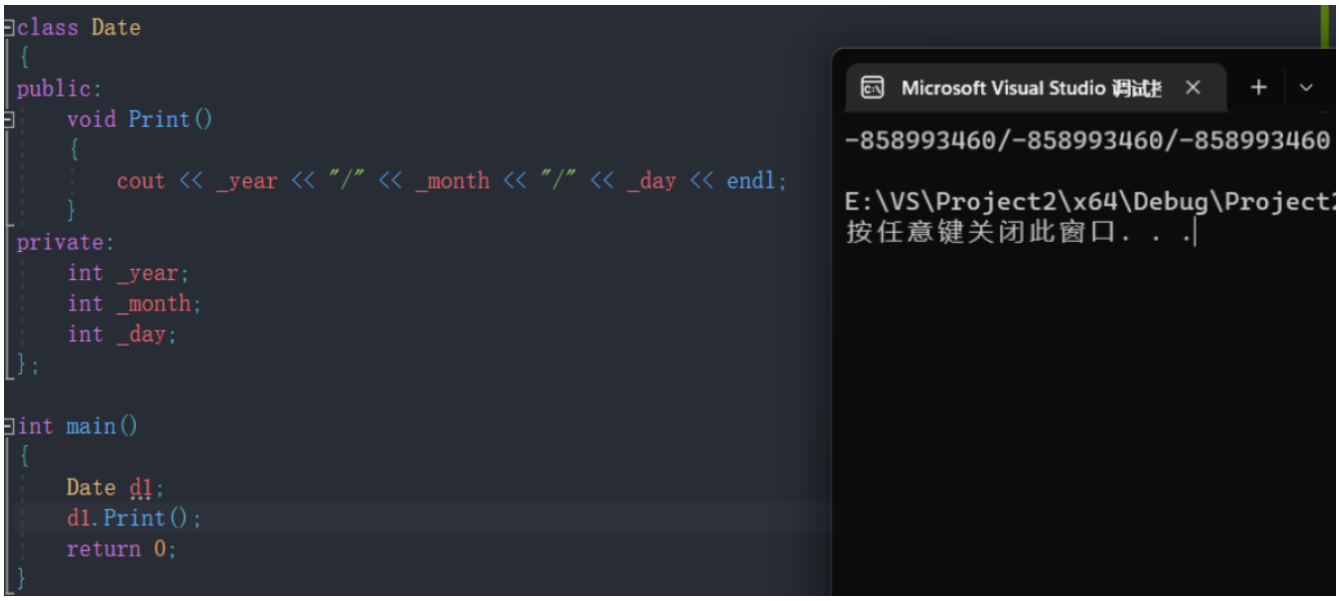
不传参数就可以调用构造函数，一般建议每个类都提供一个默认构造函数

```cpp
////无参构造函数
Date()
{
    _year = 1;
    _month = 1;
    _day = 1;
}

//全缺省 - 推荐写构造函数使用全缺省或者半缺省
//全缺省不能和无参构造函数同时存在 - 出现歧义，编译器调用的时候不知道调用哪个
Date(int year = 1, int month = 1, int day = 1)
{
    _year = year;
    _month = month;
    _day = day;
}
```

```cpp
int main()
{
    Date d1;
    Date d2(2023, 2, 6);
    Date d3(2023);
    Date d4(2023, 2);
    d1.Print();
    d2.Print();
    d3.Print();
    d4.Print();
```

```
解决方案   错误 1   警告 0   展示 1 个消息中的 0 个   生成 + IntelliSe
代码   说明
E0339   类 "Date" 包含多个默认构造函数
```

> 6.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。



为什么自动生成了默认构造函数，打印出来的变量还是随机值呢？

C++ 规定默认生成的构造函数：

> 1. 内置类型成员不做处理
> 2. 自定义类型的成员，会去调用自定义类型的类的构造函数。
> 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。

默认构造函数的场景：

比如用栈实现队列：

C/C++

```cpp
//Stack类
class Stack
{
public:
    //无参构造函数 - 函数名与对象名相同
    Stack()
    {
        cout << "Stack()" << endl;
        _a = nullptr;
        _size = _capacity = 0;
    }
    //带参数构造函数
    Stack(int n)
    {
        cout << "Stack()" << endl;
        _a = (int*)malloc(sizeof(int) * n);
        if (nullptr == _a)
        {
            perror("malloc fail");
            exit(-1);
        }
        _capacity = n;
        _size = 0;
    }
    void Push(int x) {}
    void Destroy() {}

    ~Stack()  //析构函数
    {
        cout << "~Stack()" << endl;
        free(_a);
        _a = nullptr;
        _size = _capacity = 0;
    }
private:
    int* _a;
    int _size;
    int _capacity;
};
```

C

```c
typedef struct {
    ST pushst;
    ST popst;
} MyQueue;

bool myQueueEmpty(MyQueue* obj);

MyQueue* myQueueCreate() {
    MyQueue* pq = (MyQueue*)malloc(sizeof(MyQueue));
    StackInit(&pq->pushst);
    StackInit(&pq->popst);

    return pq;
}
```

```c
void myQueueFree(MyQueue* obj) {
    assert(obj);

    StackDestroy(&obj->pushst);
    StackDestroy(&obj->popst);

    free(obj);
}
```

CPP

```cpp
class MyQueue {
public:
    // 默认生成构造函数，对自定义类型成员，会调用他的默认构造函数
    // 默认生成析构函数，对自定义类型成员，会调用他的析构函数

    void push(int x) {

    }
    //....

    Stack _pushST;
    Stack _popST;
};
```

```cpp
int main()
{
    Date d1;
    d1.Print();

    MyQueue q;

    return 0;
}
```

可以看到 MyQueue 中没有构造函数，它会自动调用 Stack 类中的构造函数。

2.1 构造函数体赋值

在创建对象时，编译器通过调用构造函数，给对象中各个成员变量一个合适的初始值。

C/C++

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

虽然上述构造函数调用之后，对象中已经有了一个初始值，但是不能将其称为对对象中成员变量的初始化，构造函数体中的语句只能将其称为赋初值，而不能称作初始化。因为初始化只能初始化一次，而构造函数体内可以多次赋值。

2.2 初始化列表

初始化列表：以一个**冒号开始**，接着是一个以**逗号分隔**的数据成员列表，每个"成员变量"后面跟 一个放在括号中的初始值或表达式。

C/C++

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

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

> 【注意】
>
> 1. 每个成员变量在初始化列表中只能出现一次(初始化只能初始化一次)
> 2. 类中包含以下成员，必须放在初始化列表位置进行初始化：
>
> ① 引用成员变量
>
> ② const 成员变量
>
> ③ 自定义类型成员(且该类没有默认构造函数时)

C/C++

```cpp
class B
{
public:
    B(int)
        :_b(0)
    {
        cout << "B()" << endl;
    }
private:
    int _b;
};


class A
{
public:
    //1、哪个对象调用构造函数，初始化列表是它所有成员变量定义的位置
    //2、不管是否显示在初始化列表写，那么编译器每个变量都会初始化列表定义初始化
    //3、三个成员变量需要写初始化，一个是const变量，一个引用变量，一个自定义类型
    A()   //这是一个初始化列表
        :_x(1)
        ,_ref(_a1)
        , _a2(1)
        ,_bb(0)
    {
        _a1++;
        _a2--;
    }
private:
    int _a1 = 1;  // 声明
    int _a2 = 2;   //缺省值是在没有构造函数的时候，缺省值才有效
    //const变量必须在定义的位置初始化，因为后面const变量就无法更改了
    //const int _x;   //加了这个const  编译器会报错，无法引用A的默认构造函数
    //const int _x = 1;  //C++11 可以这么给
    const int _x;
    int& _ref;
    B _bb;
};

```

```
40 int main()
41 {
42     A aa;    // 对象整体的定义，每个成员什么时候定义呢？
43     //必须给每个成员变量找一个定义的位置，不然像const这样的成员，没办法初始化
44
45     return 0;
46 }
```

> 3.尽量使用初始化列表初始化，因为不管你是否使用初始化列表，对于自定义类型成员变量， 一定会先使用初始化列表初始化。

C/C++

```
class Time
{
public:
    Time(int hour = 0)
        : _hour(hour)
    {
        cout << "Time()" << endl;
    }

private:
    int _hour;
};

class Date
{
public:
    Date(int day)
    {
    }

private:
    int _day;
    Time _t;     //调用Time的构造函数 ，打印Time();
};

int main()
{
    Date d(1);
}
```

> 4. 成员变量在类中声明次序就是其在初始化列表中的初始化顺序，与其在初始化列表中的先后次序无关

C/C++

```
class A
{
public:
    A(int a)
        : _a1(a), _a2(_a1)
    {
    }

    void Print()
    {
        cout << _a1 << " " << _a2 << endl;
    }

private:
    int _a2; //_a2先初始化，_a1在初始化    此时调用初始化列表，而_a1此时是随机值，所以_a2(_a1)后_a2为随机值
    int _a1;
};

int main()
{
    A aa(1);
    aa.Print();
}

/*      D
A.输出1 1
B.程序崩溃
C.编译不通过
D.输出1 随机值
*/
```

2.3 explicit关键字

构造函数不仅可以构造与初始化对象，**对于单个参数或者除第一个参数无默认值其余均有默认值的构造函数，还具有类型转换的作用。**

C/C++

```cpp
class Date
{
public:
    //1.单参构造函数，没有使用explicit修饰，具有类型转换作用
    Date(int year) : _year(year) {}

// 2，虽然有多个参数，但是创建对象时后两个参数可以不传递，没有使用explicit修饰，具有类型转换作用 - int转换为Date类
    Date(int year, int month = 1, int day = 1)
        : _year(year), _month(month), _day(day) {}
private:
    int _year;
    int _month;
    int _day;
};

int main()
{
    Date d1(2022);

    d1 = 2023;
    //用一个整型变量给日期类型对象赋值
    //实际编译器背后会用2023构造一个无名对象，最后用无名对象给d1对象进行赋值
    return 0;
}
```

C/C++

```cpp
class Date
{
public:
    // explicit修饰构造函数，禁止类型转换
    explicit Date(int year, int month = 1, int day = 1)
        : _year(year), _month(month), _day(day) {}

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

int main()
{
    Date d1(2022);

    d1 = 2023;  //err 编译器报错，没有与这些操作数匹配的 "=" 运算符，操作数类型为:  Date = int
    return 0;
}
```

C/C++

```cpp
class Date
{
public:
    Date(int year) : _year(year), _month(1), _day(1) {}
    friend ostream& operator<<(ostream& out, const Date& d);
    // operator是重载运算符，后面了解
    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;
};

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

int main()
{
    Date d1(2022); // 构造函数

    d1 = 2023; // 隐式类型转换  构造+拷贝+优化->构造  有的编译器可能不会优化
    // 构造，调用Date(year)，在拷贝调用的是operator
    // Date& ref = 10;  //err ref是类类型 不能引用10
    const Date& ref = 10; // 可以,隐式类型转换，10构造一个对象，临时对象具有常属性，所以const可以引用
    cout << ref << endl;   //10 1 1
```

```
39
40      return 0;
41  }
```

> 用 explicit 修饰构造函数，将会禁止构造函数的隐式转换。

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，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**。

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。
> 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，对自定义类型成员调用它的析构函数。(跟构造函数类似)
> 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
> 7. 析构函数的执行顺序是先创建的对象后析构，后创建的对象先析构

下面是几个可能需要使用析构函数的场景：

> 1. 动态分配内存：当一个对象在堆上分配内存时，需要手动释放内存以避免内存泄漏。在对象的析构函数中释放分配的内存是一种常见的方法。

C/C++

```
class MyClass {
public:
    MyClass() {
        data = new int[100];
    }
    ~MyClass() {
        delete[] data;
    }
private:
    int* data;
};
```

> 2.异常处理：如果在构造函数中发生了异常，那么对象可能没有完全初始化，因此析构函数应该只释放已经成功初始化的资源。

C/C++

```
class MyClass {
public:
    MyClass() {
        // 可能会抛出异常
        data = new int[100];
        // 如果抛出异常，data指针将保持为nullptr
    }
    ~MyClass() {
        if (data != nullptr) {
            delete[] data;
        }
    }
private:
    int* data = nullptr;
};
```

4.拷贝构造函数

概念：

> 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。

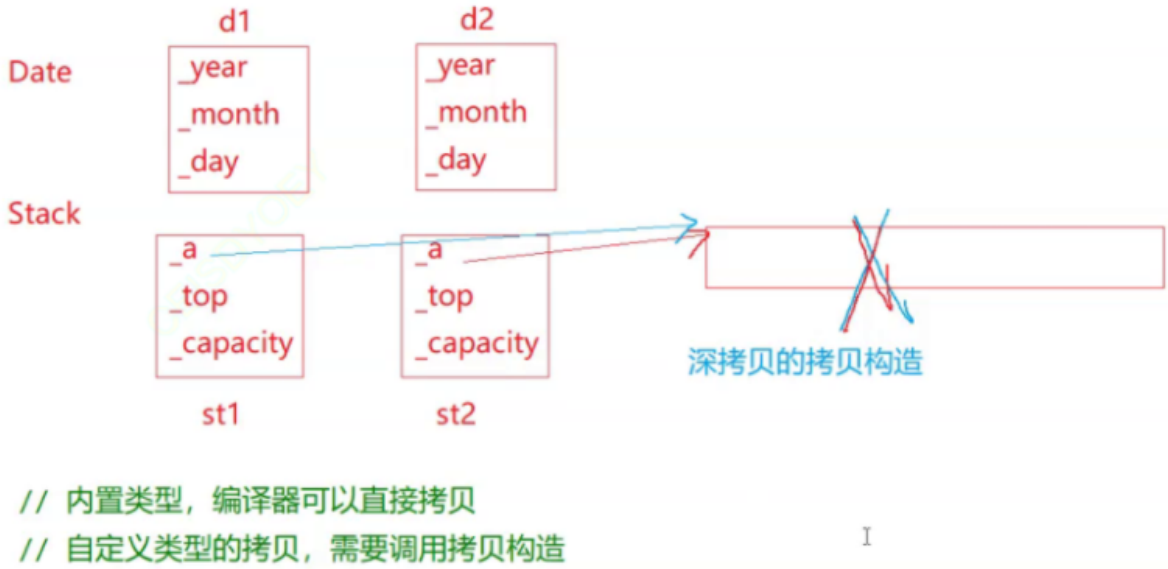
拷贝构造函数也是特殊的成员函数，其特征如下：

> 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。

为什么需要拷贝构造呢？

因为编译器只能对内置类型进行直接拷贝，而自定义类型的拷贝，需要使用拷贝构造

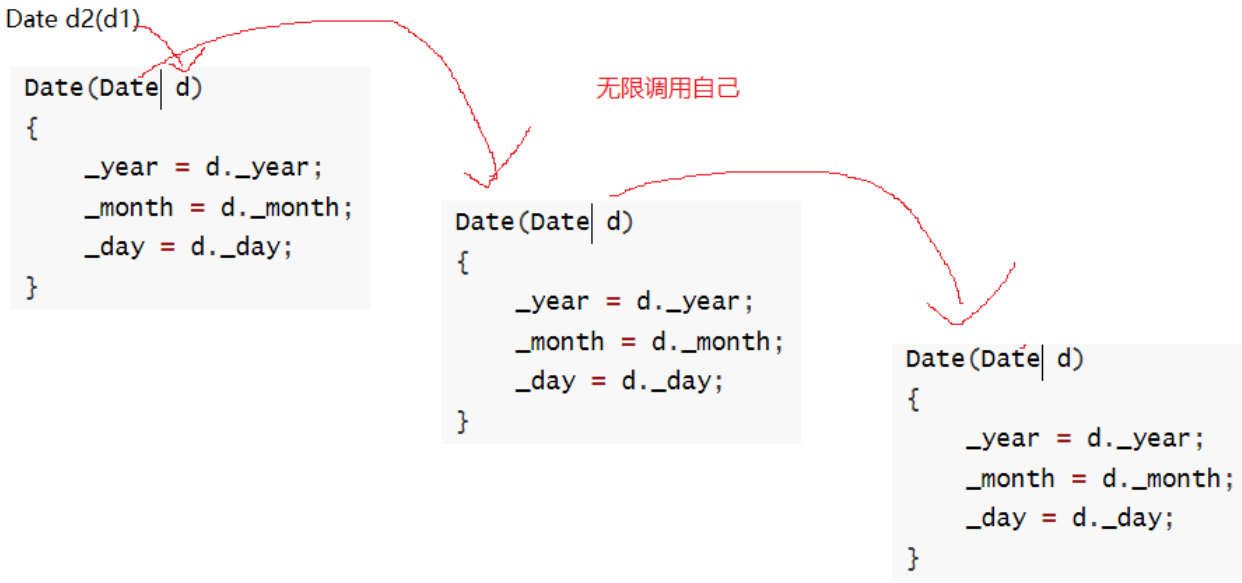


Date 类中的成员变量都是内置类型，编译器可以直接拷贝。而 Stack 栈中的 a 是指针，如果直接拷贝，st1 和 st2 指向了同一块内存地址，当 st2 进行析构处理的时候，也就把 st1 的 a 也清理了，当 st2push 一个 1，st1 也会跟着 push 一个 1，所以对于自定义不能直接拷贝。

为什么拷贝构造使用传值调用会发生无穷递归？

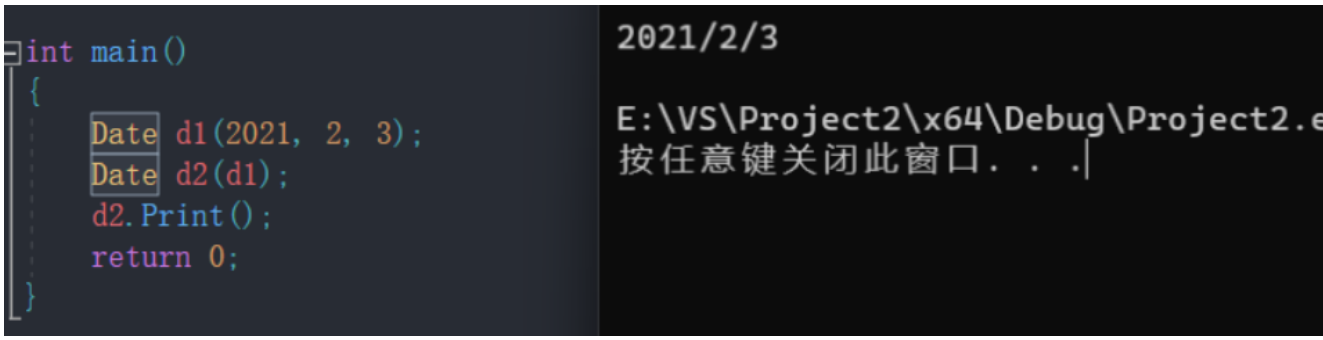
C/C++

```
1 //传值调用
2 Date(Date d)
3     {
4         _year = d._year;
5         _month = d._month;
6         _day = d._day;
7     }
```



C/C++

```cpp
//正确的写法
class Date
{
public:
    Date(int year = 2023, 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;
    }

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

int main()
{
    Date d1(2021, 2, 3);
    Date d2(d1);
    d2.Print();
    return 0;
}
```



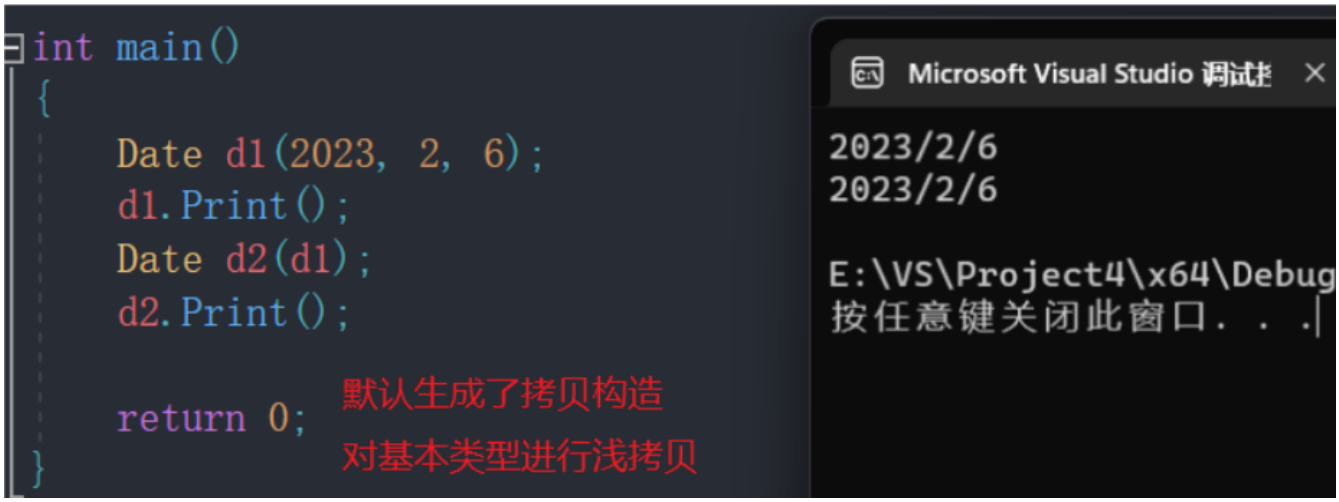
> 3.若未显式定义，编译器会生成默认的拷贝构造函数。 默认的拷贝构造函数对象按内存存储按字节序完成拷贝，这种拷贝叫做浅拷贝，或者值拷贝。

默认生成拷贝构造和赋值重载：

> a、内置类型完成浅拷贝 / 值拷贝 -- 按 byte 一个一个拷贝
> b、自定义类型，去调用这个成员拷贝构造 / 赋值重载

C/C++

```
//自动生成构造拷贝函数对内置类型进行拷贝
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;
};
```

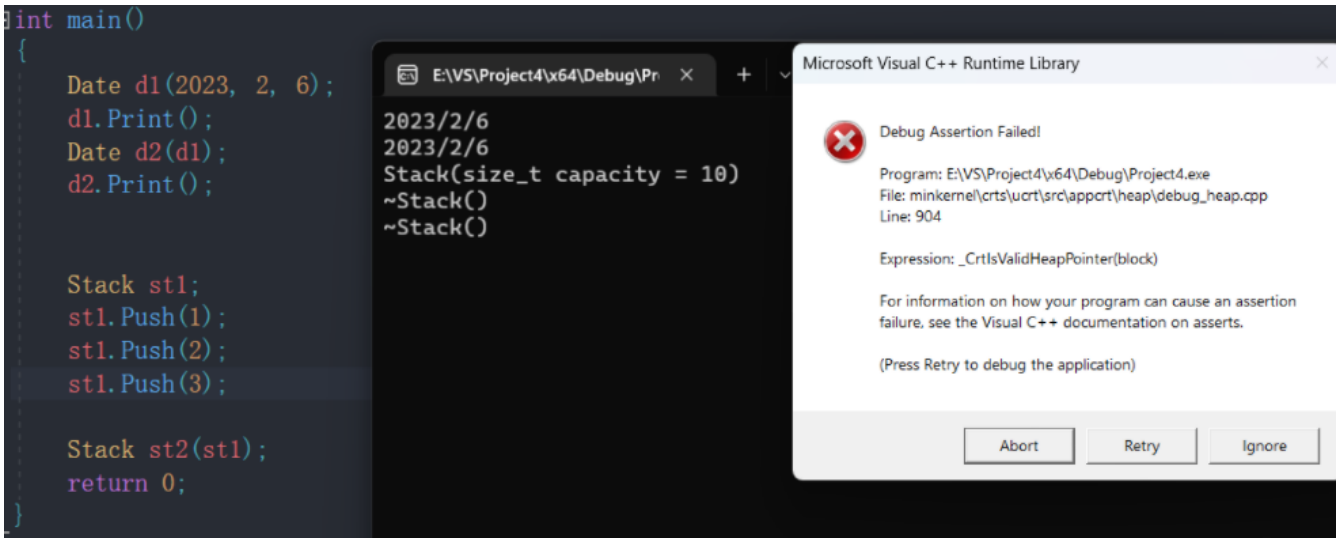


> 4.编译器生成的默认拷贝构造函数已经可以完成字节序的值拷贝了，还需要自己显式实现吗？ 当然像日期类这样的类是没必要的。那么下面的类呢？验证一下试试？

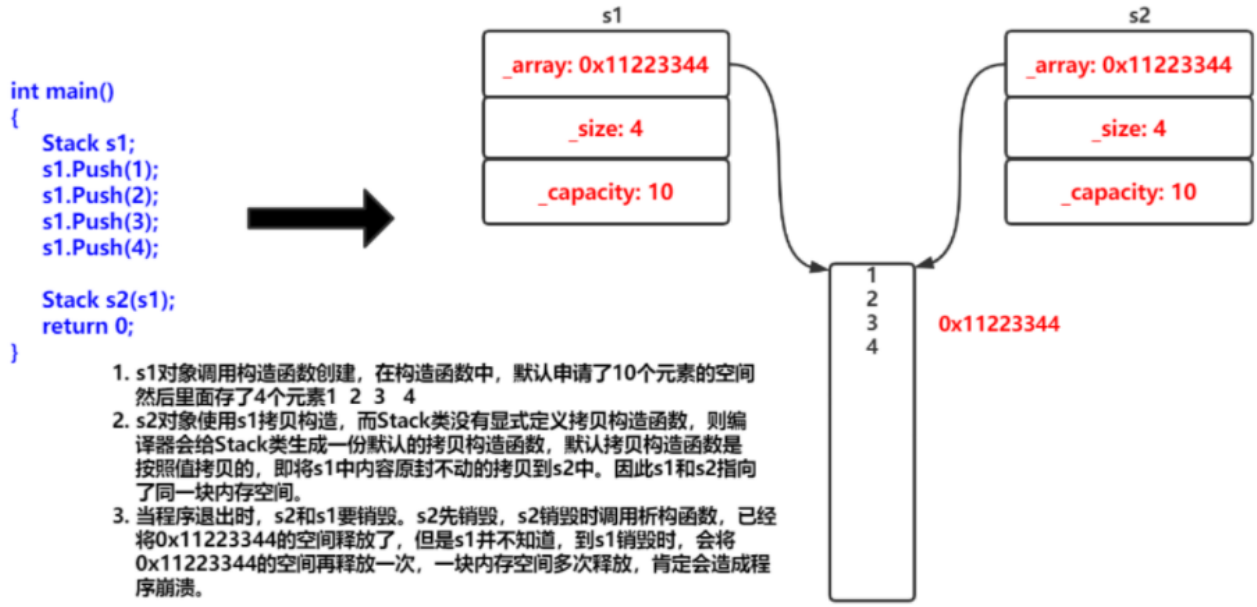
C/C++

```cpp
//自动生成构造拷贝函数对自定义类型进行拷贝
typedef int DataType;
class Stack
{
public:
    Stack(size_t capacity = 10)
    {
        cout << "Stack(size_t capacity = 10)" << endl;

        _array = (DataType*)malloc(capacity * sizeof(DataType));
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            exit(-1);
        }

        _size = 0;
        _capacity = capacity;
    }

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

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

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

```
40
41 private:
42     DataType* _array;
43     size_t    _size;
44     size_t    _capacity;
45 };
```



这里会发现上面的程序会崩溃掉？这里就需要我们以后讲的深拷贝去解决。



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：

a、使用已存在对象创建新对象

b、函数参数类型为类类型对象

c、函数返回值类型为类类型对象

C/C++

```cpp
typedef int DataType;
class Stack
{
public:
    Stack(size_t capacity = 10)
    {
        cout << "Stack(size_t capacity = 10)" << endl;

        _array = (DataType*)malloc(capacity * sizeof(DataType));
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            exit(-1);
        }

        _size = 0;
        _capacity = capacity;
    }

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

    //stack类的拷贝构造深拷贝
    Stack(const Stack& st)
    {
        cout << "Stack(const Stack& st)" << endl;
        //深拷贝开额外空间，为了避免指向同一空间
        _array = (DataType*)malloc(sizeof(DataType)*st._capacity);
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            exit(-1);
        }
        //进行字节拷贝
        memcpy(_array, st._array, sizeof(DataType)*st._size);
```

```
40          _size = st._size;
41          _capacity = st._capacity;
42      }
43
44      ~Stack()
45      {
46          cout << "~Stack()" << endl;
47
48          if (_array)
49          {
50              free(_array);
51              _array = nullptr;
52              _capacity = 0;
53              _size = 0;
54          }
55      }
56
57  private:
58      DataType *_array;
59      size_t    _size;
60      size_t    _capacity;
61  };
62
63  class MyQueue
64  {
65  public:
66      //MyQueue什么都不写，会调用默认的构造函数，也就是Stack类的构造函数
67      // 默认生成构造
68      // 默认生成析构
69      // 默认生成拷贝构造
70
71  private:
72      //默认构造函数初始化 - 默认析构函数
73      Stack _pushST;
74      //默认构造函数初始化 - 默认析构函数
75      Stack _popST;
76      int _size = 0;
77  };
78
79  int main()
80  {
```

```
 81      Date d1(2023, 2, 5);
 82      d1.Print();
 83
 84      Date d2(d1);
 85      Date d3 = d1; //  拷贝构造
 86      d2.Print();
 87
 88      Stack st1;
 89      st1.Push(1);
 90      st1.Push(2);
 91      st1.Push(4);
 92
 93      Stack st2(st1);
 94      cout << "==============================" << endl;
 95
 96      MyQueue q1;
 97      //q1拷贝q2  q1中有两个Stack类和一个size，size直接拷贝，stack类是调用stack
    拷贝构造进行拷贝
 98      MyQueue q2(q1);
 99
100      return 0;
101  }
```