

<C++> 二、类和对象

1. 面向对象和面向过程

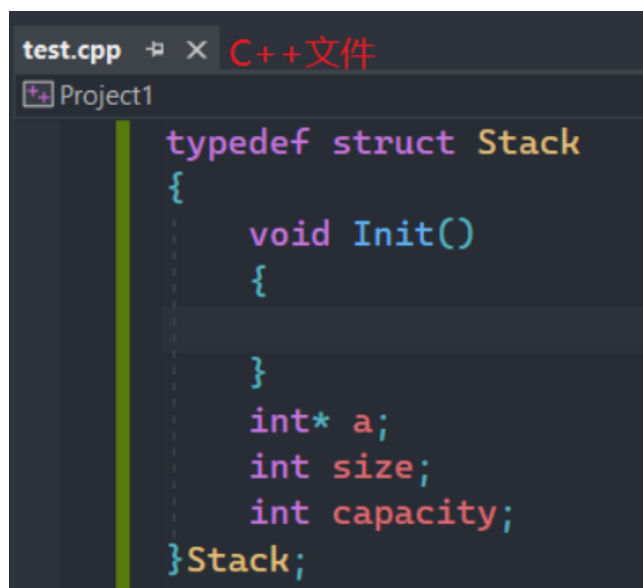
C 语言是**面向过程**的，关注的是过程，分析出求解问题的步骤，通过函数调用逐步解决问题。

C++ 是基于**面向对象**的，关注的是对象，将一件事情拆分成不同的对象，靠对象之间的交互完成。

2. C++ 类

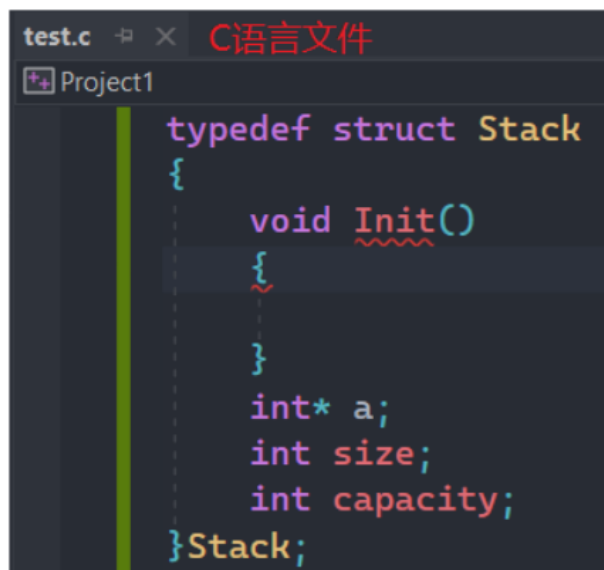
C 语言结构体中只能定义变量，在 C++ 中，结构体不仅可以定义变量，也可以定义函数。

C++ 兼容 C 结构体的语法，C++ 把结构体升级成了类，在 C++ 中更喜欢用 class 代替 struct。



test.cpp C++ 文件

```
Project1
typedef struct Stack
{
    void Init()
    {
    }
    int* a;
    int size;
    int capacity;
}Stack;
```



test.c C语言文件

```
Project1
typedef struct Stack
{
    void Init()
    {
    }
    int* a;
    int size;
    int capacity;
}Stack;
```

在 C 语言文件结构体中定义函数，编译器报错，C++ 编译器不报错

3. 类的定义

C/C++

```
1 class className
2 {
3     // 类体：由成员函数和成员变量组成
4 }; // 一定要注意后面的分号
```

class 为定义类的关键字，**ClassName** 为类的名字，{} 中为类的主体，注意类定义结束时后面分号不能省略。

类的两种定义方式：

1. 声明和定义全部放在类体中，需注意：成员函数如果在类中定义，编译器可能会将其当成内联函数处理。

C/C++

```
1 class Stack
2 {
3     // 成员函数
4     void Init(int n = 4)
5     {
6         a = (int*)malloc(sizeof(int)* n);
7         if (nullptr == a)
8         {
9             perror("malloc申请空间失败");
10            return;
11        }
12
13        capacity = n;
14        size = 0;
15    }
16
17    void Push(int x)
18    {
19        //...
20        a[size++] = x;
21    }
22
23    // 成员变量
24    int* a;
25    int size;
26    int capacity;
27 };
```

2. 类声明放在 .h 文件中，成员函数定义放在 .cpp 文件中，注意：成员函数名前需要加类名 ::

成员变量命名规则的建议：

```
1 // 我们看看这个函数，是不是很僵硬？
2 class Date
3 {
4 public:
5     void Init(int year)
6     {
7         // 这里的year到底是成员变量，还是函数形参？
8         year = year;
9     }
10
11 private:
12     int year;
13 };
14
15 //所以一般建议这样
16 class Date
17 {
18 public:
19     void Init(int year)
20     {
21         _year = year;
22     }
23
24 private:
25     int _year;
26 };
27 // 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。
```

4. 类的访问限定符及封装

4.1 访问限定符

C++ 实现封装的方式：**用类将对象的属性与方法结合在一块，让对象更加完善，通过访问权限选择性的将其接口提供给外部的用户使用。**

访问限定符有三种：

public（公有）、**protected**（保护）、**private**（私有）

访问限定符说明：

1. public 修饰的成员在类外可以直接被访问
2. protected 和 private 修饰的成员在类外不能直接被访问(此处 protected 和 private 是类似的)
3. 访问权限作用域从该访问限定符出现的位置开始直到下一个访问限定符出现时为止
4. 如果后面没有访问限定符，作用域就到 } 即类结束。
5. class 的默认访问权限为 private，struct 为 public(因为 struct 要兼容 C)

注意：访问限定符只在编译时有用，当数据映射到内存后，没有任何访问限定符上的区别

问题：C++ 中 struct 和 class 的区别是什么？

解答：C++ 需要兼容 C 语言，所以 C++ 中 struct 可以当成结构体使用。另外 C++ 中 struct 还可以用来定义类。和 class 定义类是一样的，区别是 struct 定义类默认访问权限是 public，class 定义类默认访问权限是 private。注意：在继承和模板参数列表位置，struct 和 class 也有区别，后序给大家介绍。

4.2 封装

封装：将数据和操作数据的方法进行有机结合，隐藏对象的属性和实现细节，仅对外公开接口来和对象进行交互。

封装本质上是一种管理，让用户更方便使用类。比如：对于电脑这样一个复杂的设备，提供给用户的就只有开关机键、通过键盘输入，显示器，USB 插孔等，让用户和计算机进行交互，完成日常事务。但实际上电脑真正工作的却是 CPU、显卡、内存等一些硬件元件。

对于计算机使用者而言，不用关心内部核心部件，比如主板上线路是如何布局的，CPU 内部是如何设计的等，用户只需要知道，怎么开机、怎么通过键盘和鼠标与计算机进行交互即可。因此**计算机厂商在出厂时，在外部套上壳子，将内部实现细节隐藏起来，仅仅对外提供开关机、鼠标以及键盘插孔等，让用户可以与计算机进行交互即可。**

在 C++ 语言中实现封装，可以**通过类将数据以及操作数据的方法进行有机结合，通过访问权限来隐藏对象内部实现细节，控制哪些方法可以在类外部直接被使用。**

5. 类的作用域

类定义了一个新的作用域，类的所有成员都在类的作用域中。在类体外定义成员时，需要使用 :: 作用域操作符指明成员属于哪个类域。

C/C++

```
1 class Person
2 {
3 public:
4     void PrintPerson();
5
6 private:
7     char _name[20];
8     char _gender[3];
9     int _age;
10 };
11 // 这里需要指定PrintPerson是属于Person这个类域
12 // PrintPerson函数前指明属于哪个类用::连接
13 void Person::PrintPerson()
14 {
15     cout << _name << " " << _gender << " " << _age << endl;
16 }
```

6. 类的实例化

用类类型创建对象的过程，称为类的实例化

1. **类是对对象进行描述的**，是一个模型一样的东西，限定了类有哪些成员，**定义出一个类并没有分配实际的内存空间**来存储它。
2. 一个类可以实例化出多个对象，**实例化出的对象占用实际的物理空间，存储类成员变量**
3. **类实例化出对象就像现实中使用建筑设计图建造出房子，类就像是设计图**，只设计出需要什么东西，但是并没有实体的建筑存在，同样类也只是一个设计，实例化出的对象 能实际存储数据，占用物理空间

C/C++

```
1 // 类 -- 别墅设计图
2 class Date
3 {
4 public:
5     // 定义
6     void Init(int year, int month, int day)
7     {
8         _year = year;
9         _month = month;
10        _day = day;
11    }
12
13 private:
14     //对数据声明，而不是定义，定义是在创建对象的时候
15     int _year;
16     int _month;
17     int _day;
18 };
19
20 int main()
21 {
22     // 类对象实例化 -- 开空间
23     // 实例化 -- 用设计图建造一栋栋别墅
24     Date d1;
25     Date d2;
26
27     //Date.Init(2023, 2, 2); //错误，调用成员函数，要用对象，不能用类
28     d1.Init(2023, 12, 12);
29     d2.Init(2022, 1, 1);
30     //d1, d2中的成员变量占用不同的内存空间
31     return 0;
32 }
```

为什么成员变量在对象中，成员函数不在对象中呢？

每个对象成员变量是不一样的，需要独立存储

每个对象调用成员函数是一样的，放在共享公共区域（代码段）

7. 类对象的大小和内存对齐

7.1 类对象大小

C/C++

```
1 class Date
2 {
3 public:
4     // 定义
5     void Init(int year, int month, int day)
6     {
7         _year = year;
8         _month = month;
9         _day = day;
10    }
11
12 //private:
13     int _year;
14     int _month;
15     int _day;
16 };
17
18 class A2
19 {
20 public:
21     void f2() {}
22 };
23
24 // 类中什么都没有---空类
25 class A3
26 {
27
28 };
29
30 int main()
31 {
32
33     Date d1;
34     cout << sizeof(d1) << endl;    //输出结果为12， 只计算了类对象中成员变量的
    大小
35
36     A2 aa2;
37     A3 aa3;
38     cout << &aa2 << endl;    //打印一个无成员变量的类地址，结果0x61fe07
```

```

39     cout << &aa3 << endl;    //打印一个空类的地址 结果0x61fe06
40     cout << sizeof(aa2) << endl;    //1
41     cout << sizeof(aa3) << endl;    //1
42     // 为什么一个没有成员函数或者一个空类占用内存空间1字节呢？
43     // 大小是1，这1byte不存储有效数据
44     // 占位，标识对象被实例化定义出来了。
45     return 0;
46 }

```

结论：一个类的大小，实际就是该类中”成员变量”之和，当然要注意内存对齐注意空类的大小，空类比较特殊，编译器给了空类一个字节来唯一标识这个类的对象。

7.2 结构体(类)的内存对齐规则

1. 第一个成员在与结构体偏移量为 0 的地址处。
 2. 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。
注意：对齐数 = 编译器默认的一个对齐数与该成员大小的较小值。VS 中默认的对齐数为 8
 3. 结构体总大小为：最大对齐数（所有变量类型最大者与默认对齐参数取最小）的整数倍。
 4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的最大对齐数的整数倍处，结构体的整体大小就是所有最大对齐数（含嵌套结构体的对齐数）的整数倍。
- 这个在 C 语言阶段学过了，不用多说

8.this 指针

8.1 this 指针的引出

上面的 Date 类中 Init 与 Print 两个成员函数，函数体中没有关于不同对象的区分，那当 d1 调用 Init 函数时，该函数是如何知道应该设置 d1 对象，而不是设置 d2 对象呢？

C++ 中通过引入 this 指针解决问题，即：**C++ 编译器给每个“非静态的成员函数”增加了一个隐藏的指针参数，让该指针指向当前对象(函数运行时调用该函数的对象)，在函数体中所有“成员变量”的操作，都是通过该指针去访问。只不过所有的操作对用户是透明的，即用户不需要来传递，编译器自动完成。**

8.2 this 指针的特性

- 1.this 指针的类型：类类型 * const，即成员函数中，不能给 this 指针赋值。
2. 只能在“成员函数”的内部使用
- 3.this 指针本质上是“成员函数”的形参，当对象调用成员函数时，将对象地址作为实参传递给 this 形参。所以对象中不存储 this 指针。
- 4.this 指针是“成员函数”第一个隐含的指针形参，一般情况由编译器通过 ecx 寄存器自动传递，不需要用户传递

| | | |
|---|---|--|
| <pre>void Init(int year, int month, int day) { _year = year; _month = month; _day = day; } int main() { Date d1; Date d2; d1.Init(2022, 2, 2); d2.Init(2023, 2, 2); return 0; }</pre> | <p>编译器的处理</p>  | <pre>void Init(Date* this, int year, int month, int day) { this->_year = year; this->_month = month; this->_day = day; } int main() { Date d1; Date d2; d1.Init(&d1, 2022, 2, 2); d2.Init(&d2, 2023, 2, 2); return 0; }</pre> |
|---|---|--|

面试题：

- 1.this 存在哪里？
2. 空指针问题
3. 判断下面三种结果是运行崩溃，还是编译错误，还是正常运行

C/C++

```
1 class Date
2 {
3 public:
4     // 定义
5     void Init(int year, int month, int day)
6     {
7         /*_year = year;
8         _month = month;
9         _day = day;*/
10        cout << this << endl;
11        this->_year = year;
12        this->_month = month;
13        this->_day = day;
14    }
15
16    void func()
17    {
18        cout << this << endl;
19        cout << "func()" << endl;
20    }
21
22 //private:
23     int _year; // 声明
24     int _month;
25     int _day;
26 };
27
28 // 1、this存在哪里？-- 栈，因为他是隐含形参 / vs下面是通过ecx寄存器
29 // 2、空指针问题
30 int main()
31 {
32     Date d1;
33     Date d2;
34     d1.Init(2022, 2, 2);
35     d2.Init(2023, 2, 2);
36
37     //3.判断下面三种结果是运行崩溃，还是编译错误，还是正常运行
38     Date* ptr = nullptr;
39     //ptr为一个对象指针，ptr->表示访问对象成员
```

```

40     ptr->Init(2022, 2, 2);
41     //运行崩溃, 进入Init函数体, this指针为nullptr, this指针访问对象成员变量, 出现空指针解引用。
42
43     ptr->func();
44     // 正常运行, func函数中, 没有this指针访问成员变量, 不会发生空指针解引用
45     (*ptr).func();
46     // 正常运行, (*ptr)等价于ptr,
47
48     return 0;
49 }

```

9.C 语言和 C++ 实现 Stack 的对比

C++

```

class Stack
{
public:
    // 成员函数
    void Init(int n)
    {
        a = (int*)malloc(sizeof(int)* n);
        if (nullptr == a)
        {
            perror("malloc申请空间失败");
            return;
        }

        capacity = n;
        size = 0;
    }

    void Push(int x)
    {
        //...
        a[size++] = x;
    }

    //...
private:
    // 成员变量
    int* a;
    int size;
    int capacity;
};

```

1、数据和方法都封装到类里面
2、控制访问方式。愿意给你访问公有，不愿意给你访问私有。

C

```

typedef int STDataType;
typedef struct Stack
{
    STDataType* a;
    int capacity;
    int top; // 初始为0, 表示栈顶位置下一个位置下标
}ST;

void StackInit(ST* ps);
void StackDestroy(ST* ps);
void StackPush(ST* ps, STDataType x);
void StackPop(ST* ps);
STDataType StackTop(ST* ps);

bool StackEmpty(ST* ps);
int StackSize(ST* ps);

void StackInit(ST* ps)
{
    assert(ps);

    //ps->a = NULL;
    //ps->top = 0;
    //ps->capacity = 0;

    ps->a = (STDataType*)malloc(sizeof(STDataType)*4);
    if (ps->a == NULL)
    {
        perror("malloc fail");
        exit(-1);
    }

    ps->top = 0;
    ps->capacity = 4;
}

void StackPush(ST* ps, STDataType x)
{
    assert(ps);

    // 扩容
    if (ps->top == ps->capacity)
    {
        STDataType* tmp = (STDataType*)realloc(ps->a, ps->capacity*2);
        if (tmp == NULL)
        {
            perror("realloc fail");
            exit(-1);
        }

        ps->a = tmp;
        ps->capacity *= 2;
    }

    ps->a[ps->top] = x;
    ps->top++;
}

STDataType StackTop(ST* ps)
{
    assert(ps);
    assert(!StackEmpty(ps));

    return ps->a[ps->top - 1];
}

```

1、数据和方法是分离的
2、数据访问控制是自由的，不受限制的

可以看到，在用 C 语言实现时，Stack 相关操作函数有以下共性：

每个函数的第一个参数都是 Stack*

函数中必须要对第一个参数检测，因为该参数可能会为 NULL

函数中都是通过 Stack* 参数操作栈的

调用时必须传递 Stack 结构体变量的地址

结构体中只能定义存放数据的结构，操作数据的方法不能放在结构体中，即数据和操作数据的方式是分离开的，而且实现上相当复杂一点，涉及到大量指针操作，稍不注意可能就会出错。

C++ 中通过类可以将数据以及操作数据的方法进行完美结合，通过访问权限可以控制那些方法在类外可以被调用，即封装，在使用时就像使用自己的成员一样，更符合人类对一件事物的认知。而且每个方法不需要传递 Stack* 的参数了，编译器编译之后该参数会自动还原，即 C++ 中 Stack * 参数是编译器维护的，C 语言中需用用户自己维护。