# 2.类和对象(上)

## 1. 类的定义

### 1.1 类定义格式

- class为定义类的关键字，Stack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。类体中内容称为类的成员：类中的变量称为类的属性或成员变量；类中的函数称为类的方法或者成员函数。
- 为了区分成员变量，一般习惯上成员变量会加一个特殊标识，如成员变量前面或者后面加_ 或者 m开头，注意C++中这个并不是强制的，只是一些惯例，具体看公司的要求。
- C++中struct也可以定义类，C++兼容C中struct的用法，同时struct升级成了类，明显的变化是struct中可以定义函数，一般情况下我们还是推荐用class定义类。
- 定义在类面的成员函数默认为inline。

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

class Stack
{
public:
	// 成员函数
	void Init(int n = 4)
	{
		array = (int*)malloc(sizeof(int) * n);
		if (nullptr == array)
		{
			perror("malloc申请空间失败");
			return;
		}

		capacity = n;
		top = 0;
	}

	void Push(int x)
	{
		// ...扩容
		array[top++] = x;
	}
```

```
26
27          int Top()
28          {
29                  assert(top > 0);
30
31                  return array[top - 1];
32          }
33
34          void Destroy()
35          {
36                  free(array);
37                  array = nullptr;
38                  top = capacity = 0;
39          }
40
41  private:
42          // 成员变量
43          int* array;
44          size_t capacity;
45          size_t top;
46  }; // 分号不能省略
47
48  int main()
49  {
50          Stack st;
51          st.Init();
52          st.Push(1);
53          st.Push(2);
54
55          cout << st.Top() << endl;
56
57          st.Destroy();
58
59          return 0;
60  }
```

```
1  class Date
2  {
3  public:
4          void Init(int year, int month, int day)
5          {
6                  _year = year;
7                  _month = month;
8                  _day = day;
9          }
```

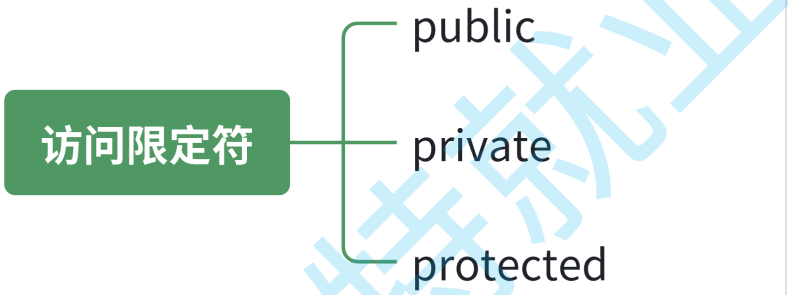
```cpp

private:
	// 为了区分成员变量，一般习惯上成员变量
	// 会加一个特殊标识，如_ 或者 m开头
	int _year; // year_  m_year
	int _month;
	int _day;
};

int main()
{
	Date d;
	d.Init(2024, 3, 31);

	return 0;
}
```

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

// C++升级struct升级成了类
// 1、类里面可以定义函数
// 2、struct名称就可以代表类型

// C++兼容C中struct的用法
typedef struct ListNodeC
{
	struct ListNodeC* next;
	int val;
}LTNode;

// 不再需要typedef，ListNodeCPP就可以代表类型
struct ListNodeCPP
{
	void Init(int x)
	{
		next = nullptr;
		val = x;
	}

	ListNodeCPP* next;
	int val;
};

int main()
```

```
{

        return 0;
}
```

## 1.2 访问限定符

- C++一种实现封装的方式，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。
- public修饰的成员在类外可以直接被访问；protected和private修饰的成员在类外不能直接被访问，protected和private是一样的，以后继承章节才能体现出他们的区别。
- 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，如果后面没有访问限定符，作用域就到 }即类结束。
- class定义成员没有被访问限定符修饰时默认为private，struct默认为public。
- 一般成员变量都会被限制为private/protected，需要给别人使用的成员函数会放为public。




## 1.3 类域

- 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。
- 类域影响的是编译的查找规则，下面程序中Init如果不指定类域Stack，那么编译器就把Init当成全局函数，那么编译时，找不到array等成员的声明/定义在哪里，就会报错。指定类域Stack，就是知道Init是成员函数，当前域找不到的array等成员，就会到类域中去查找。

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

class Stack
{
public:
    // 成员函数
    void Init(int n = 4);
```

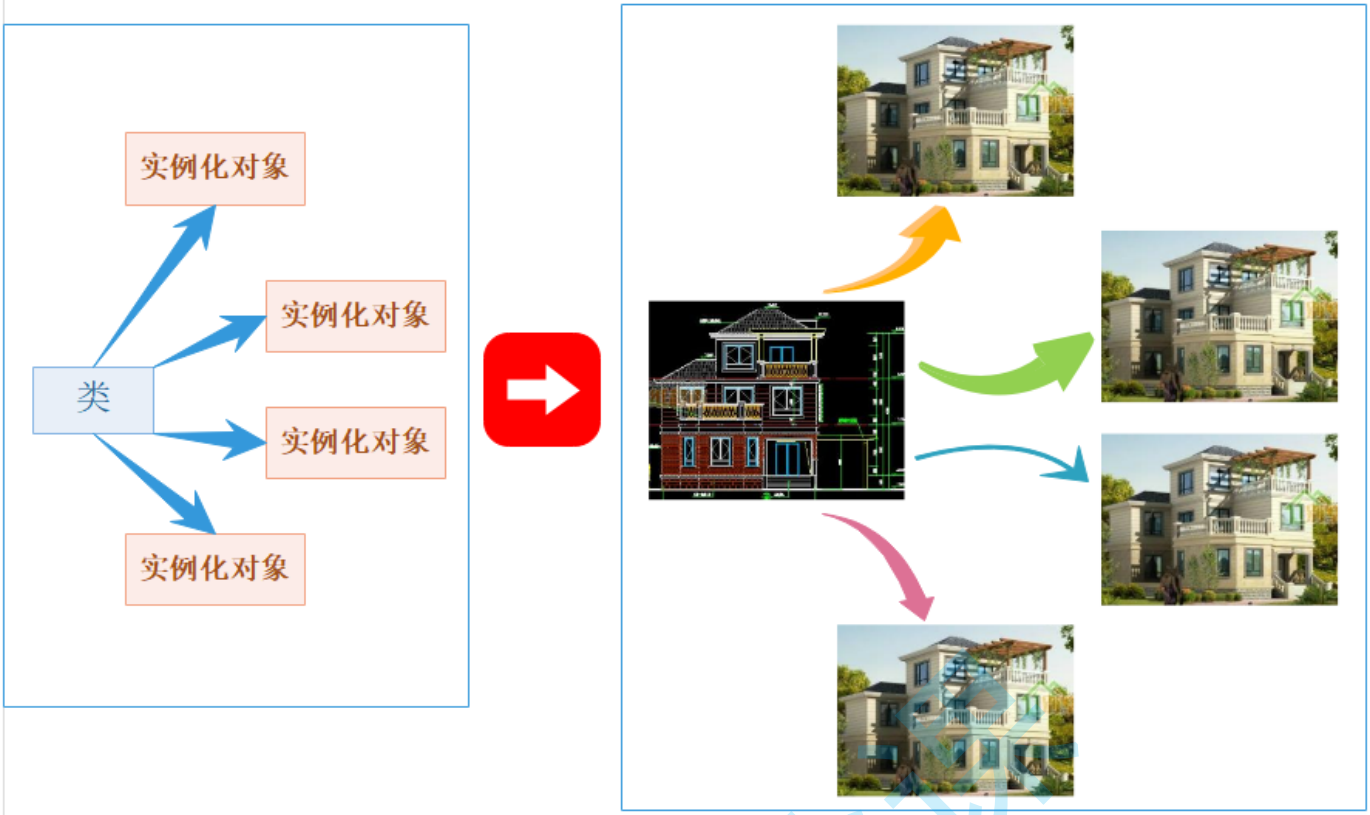
```
 9 private:
10     // 成员变量
11     int* array;
12     size_t capacity;
13     size_t top;
14 };
15
16 // 声明和定义分离，需要指定类域
17 void Stack::Init(int n)
18 {
19     array = (int*)malloc(sizeof(int) * n);
20     if (nullptr == array)
21     {
22         perror("malloc申请空间失败");
23         return;
24     }
25
26     capacity = n;
27     top = 0;
28 }
29
30
31 int main()
32 {
33     Stack st;
34     st.Init();
35
36
37     return 0;
38 }
```



## 2. 实例化

### 2.1 实例化概念

- 用类类型在物理内存中创建对象的过程，称为类实例化出对象。
- 类是对象进行一种抽象描述，是一个模型一样的东西，限定了类有哪些成员变量，这些成员变量只是声明，没有分配空间，用类实例化出对象时，才会分配空间。
- 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。打个比方：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，设计图规划了有多少个房间，房间大小功能等，但是并没有实体的建筑存在，也不能住人，用设计图修建出房子，房子才能住人。同样类就像设计图一样，不能存储数据，实例化出的对象分配物理内存存储数据。

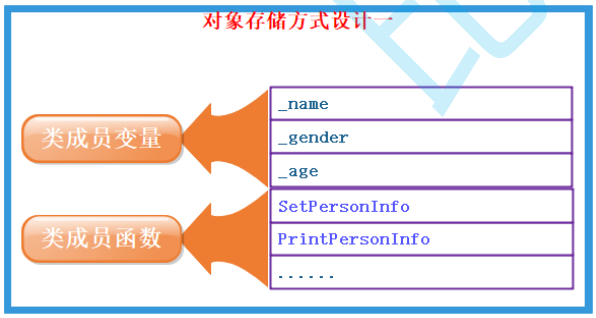



```cpp
#include<iostream>
using namespace std;

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:
    // 这里只是声明，没有开空间
    int _year;
    int _month;
    int _day;
};

int main()
```

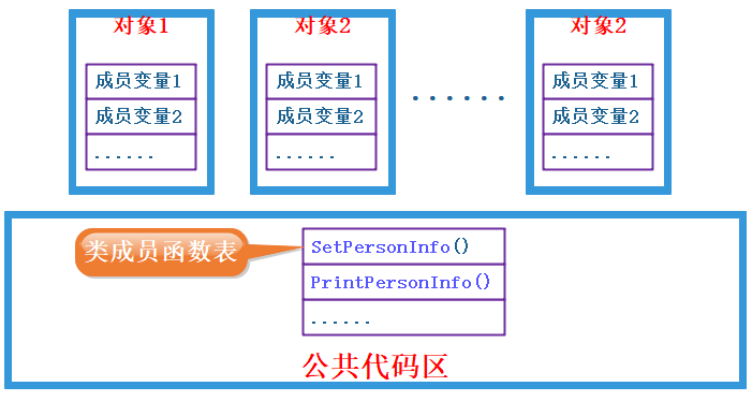
```cpp
{
    // Date类实例化出对象d1和d2
    Date d1;
    Date d2;

    d1.Init(2024, 3, 31);
    d1.Print();

    d2.Init(2024, 7, 5);
    d2.Print();

    return 0;
}
```

## 2.2 对象大小

分析一下类对象中哪些成员呢？类实例化出的每个对象，都有独立的数据空间，所以对象中肯定包含成员变量，那么成员函数是否包含呢？首先函数被编译后是一段指令，对象中没办法存储，这些指令存储在一个单独的区域(代码段)，那么对象中非要存储的话，只能是成员函数的指针。再分析一下，对象中是否有存储指针的必要呢，Date实例化d1和d2两个对象，d1和d2都有各自独立的成员变量_year/_month/_day存储各自的数据，但是d1和d2的成员函数Init/Print指针却是一样的，存储在对象中就浪费了。如果用Date实例化100个对象，那么成员函数指针就重复存储100次，太浪费了。这里需要再额外哆嗦一下，其实函数指针是不需要存储的，函数指针是一个地址，调用函数被编译成汇编指令[call 地址]， 其实编译器在编译链接时，就要找到函数的地址，不是在运行时找，只有动态多态是在运行时找，就需要存储函数地址，这个我们以后会讲解。








上面我们分析了对象中只存储成员变量，C++规定类实例化的对象也要符合内存对齐的规则。

**内存对齐规则**

- 第一个成员在与结构体偏移量为0的地址处。
- 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
- 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。
- 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。

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

// 计算一下A/B/C实例化的对象是多大?
class A
{
public:
    void Print()
    {
        cout << _ch << endl;
    }
private:
    char _ch;
    int _i;
};

class B
{
public:
    void Print()
    {
        //...
    }
};

class C
{};

int main()
{
    A a;
    B b;
    C c;
    cout << sizeof(a) << endl;
```

```
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;

    return 0;
}
```

上面的程序运行后，我们看到没有成员变量的B和C类对象的大小是1，为什么没有成员变量还要给1个字节呢？因为如果一个字节都不给，怎么表示对象存在过呢！所以这里给1字节，纯粹是为了占位标识对象存在。

## 3. this指针

- Date类中有 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和Print函数时，该函数是如何知道应该访问的是d1对象还是d2对象呢？那么这里就要看到C++给了一个隐含的this指针解决这里的问题
- 编译器编译后，类的成员函数默认都会在形参第一个位置，增加一个当前类类型的指针，叫做this指针。比如Date类的Init的真实原型为，`void Init(Date* const this, int year, int month, int day)`
- 类的成员函数中访问成员变量，本质都是通过this指针访问的，如Init函数中给_year赋值，`this->_year = year;`
- C++规定不能在实参和形参的位置显示的写this指针(编译时编译器会处理)，但是可以在函数体内显示使用this指针。

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

class Date
{
public:
    // void Init(Date* const this, int year, int month, int day)
    void Init(int year, int month, int day)
    {
        // 编译报错：error C2106: “=”: 左操作数必须为左值
        // this = nullptr;

        // this->_year = year;
        _year = year;
        this->_month = month;
        this->_day = day;
    }

    void Print()
```

```
    {
        cout << _year << "/" << _month << "/" << _day << endl;
    }

private:
    // 这里只是声明，没有开空间
    int _year;
    int _month;
    int _day;
};

int main()
{
    // Date类实例化出对象d1和d2
    Date d1;
    Date d2;

    // d1.Init(&d1, 2024, 3, 31);
    d1.Init(2024, 3, 31);
    d1.Print();

    d2.Init(2024, 7, 5);
    d2.Print();

    return 0;
}
```

下面通过两个选择题测试一下前面的知识学得如何?

**1.下面程序编译运行结果是（）**

A、编译报错 B、运行崩溃 C、正常运行

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

class A
{
public:
    void Print()
    {
        cout << "A::Print()" << endl;
    }
private:
    int _a;
};
```

```
14
15 int main()
16 {
17     A* p = nullptr;
18     p->Print();
19     return 0;
20 }
21
```

**2.下面程序编译运行结果是（）**

A、编译报错 B、运行崩溃 C、正常运行

```
1  #include<iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      void Print()
8      {
9          cout << "A::Print()" << endl;
10         cout << _a << endl;
11     }
12 private:
13     int _a;
14 };
15
16 int main()
17 {
18     A* p = nullptr;
19     p->Print();
20     return 0;
21 }
```



**3. this指针存在内存哪个区域的 ()**

A. 栈 B.堆 C.静态区 D.常量区 E.对象里面

## 4. C++和C语言实现Stack对比

面向对象三大特性：封装、继承、多态，下面的对比我们可以初步了解一下封装。

通过下面两份代码对比，我们发现C++实现Stack形态上还是发生了挺多的变化，底层和逻辑上没啥变化。

- C++中数据和函数都放到了类里面，通过访问限定符进行了限制，不能再随意通过对象直接修改数据，这是C++封装的一种体现，这个是最重要的变化。这里的封装的本质是一种更严格规范的管理，避免出现乱访问修改的问题。当然封装不仅仅是这样的，我们后面还需要不断的去学习。
- C++中有一些相对方便的语法，比如Init给的缺省参数会方便很多，成员函数每次不需要传对象地址，因为this指针隐含的传递了，方便了很多，使用类型不再需要typedef用类名就很方便
- 在我们这个C++入门阶段实现的Stack看起来变了很多，但是实质上变化不大。等着我们后面看STL中的用适配器实现的Stack，大家再感受C++的魅力。

**C实现Stack代码**

```c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<assert.h>

typedef int STDataType;
typedef struct Stack
{
    STDataType* a;
    int top;
    int capacity;
}ST;

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

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

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

    free(ps->a);
    ps->a = NULL;
    ps->top = ps->capacity = 0;
}

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

```

```c
    // 满了， 扩容
    if (ps->top == ps->capacity)
    {
        int newcapacity = ps->capacity == 0 ? 4 : ps->capacity * 2;
        STDataType* tmp = (STDataType*)realloc(ps->a, newcapacity *
sizeof(STDataType));
        if (tmp == NULL)
        {
            perror("realloc fail");
            return;
        }

        ps->a = tmp;
        ps->capacity = newcapacity;
    }

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

bool STEmpty(ST* ps)
{
    assert(ps);

    return ps->top == 0;
}

void STPop(ST* ps)
{
    assert(ps);
    assert(!STEmpty(ps));

    ps->top--;
}

STDataType STTop(ST* ps)
{
    assert(ps);
    assert(!STEmpty(ps));

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

int STSize(ST* ps)
{
    assert(ps);

```

```c
    return ps->top;
}

int main()
{
    ST s;
    STInit(&s);

    STPush(&s, 1);
    STPush(&s, 2);
    STPush(&s, 3);
    STPush(&s, 4);
    while (!STEmpty(&s))
    {
        printf("%d\n", STTop(&s));
        STPop(&s);
    }

    STDestroy(&s);

   return 0;
}
```

**C++实现Stack代码**

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

typedef int STDataType;
class Stack
{
public:
    // 成员函数
    void Init(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
22      void Push(STDataType x)
23      {
24          if (_top == _capacity)
25          {
26              int newcapacity = _capacity * 2;
27              STDataType* tmp = (STDataType*)realloc(_a, newcapacity *
    sizeof(STDataType));
28              if (tmp == NULL)
29              {
30                  perror("realloc fail");
31                  return;
32              }
33
34             _a = tmp;
35             _capacity = newcapacity;
36          }
37
38          _a[_top++] = x;
39      }
40
41      void Pop()
42      {
43          assert(_top > 0);
44          --_top;
45      }
46
47      bool Empty()
48      {
49          return _top == 0;
50      }
51
52      int Top()
53      {
54          assert(_top > 0);
55
56          return _a[_top - 1];
57      }
58
59      void Destroy()
60      {
61          free(_a);
62          _a = nullptr;
63          _top = _capacity = 0;
64      }
65
66  private:
```

```
67      // 成员变量
68      STDataType* _a;
69      size_t _capacity;
70      size_t _top;
71  };
72
73  int main()
74  {
75      Stack s;
76      s.Init();
77      s.Push(1);
78      s.Push(2);
79      s.Push(3);
80      s.Push(4);
81
82      while (!s.Empty())
83      {
84          printf("%d\n", s.Top());
85          s.Pop();
86      }
87
88      s.Destroy();
89
90      return 0;
91  }
```