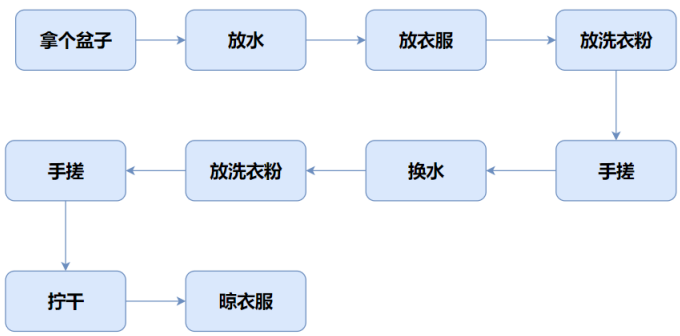
类与对象(上)

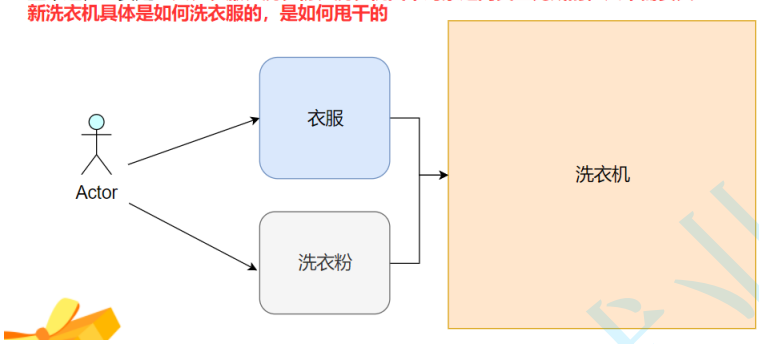
【本节目标】

- 1.面向过程和面向对象初步认识
- 2.类的引入
- 3.类的定义
- 4.类的访问限定符及封装
- 5.类的作用域
- 6.类的实例化
- 7.类的对象大小的计算
- 8.类成员函数的this指针

1.面向过程和面向对象初步认识

> 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。

总共有四个对象：**人、衣服、洗衣粉、洗衣机**，
整个洗衣服的过程：人将衣服放进洗衣机、倒入洗衣粉，启动洗衣机，洗衣机就会完成洗衣过程并且甩干
整个过程主要是：**人、衣服、洗衣粉、洗衣机四个对象之间交互完成的，人不需要关新洗衣机具体是如何洗衣服的，是如何甩干的**



2.类的引入

C语言结构体中只能定义变量，在C++中，结构体内不仅可以定义变量，也可以定义函数。比如：之前在数据结构初阶中，用**C语言方式实现的栈，结构体中只能定义变量**；现在以**C++方式实现，会发现struct中也可以定义函数。**

```
typedef int DataType;
struct Stack
{
    void Init(size_t capacity)
    {
        _array = (DataType*)malloc(sizeof(DataType) * capacity);
        if (nullptr == _array)
        {
            perror("malloc申请空间失败");
            return;
        }

        _capacity = capacity;
        _size = 0;
    }

    void Push(const DataType& data)
    {
        // 扩容

        _array[_size] = data;
```

```
        ++_size;
    }

    DataType Top()
    {
        return _array[_size - 1];
    }

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

    DataType* _array;
    size_t _capacity;
    size_t _size;
};

int main()
{
    Stack s;
    s.Init(10);
    s.Push(1);
    s.Push(2);
    s.Push(3);
    cout << s.Top() << endl;
    s.Destroy();
    return 0;
}
```

上面结构体的定义，**在C++中更喜欢用class来代替。**

3.类的定义

```
class className
{
    // 类体：由成员函数和成员变量组成

};  // 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。**

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。

声明和定义全部放在类体中

```
//人
class Person
{
public:
	//显示基本信息
	void showInfo()
	{
		cout << _name << "-" << _sex << "-" << _age << endl;
	}
public:
	char* _name;	//姓名
	char* _sex;	//性别
	int  _age;	//年龄
};
```

.

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::**

声明放在类的头文件person.h中

```
//人
class Person
{
public:
	//显示基本信息
	void showInfo();
public:
	char* _name;	//姓名
	char* _sex;	//性别
	int  _age;	//年龄
};
```

定义放在类的实现文件person.cpp中

```
#include "person.h"

//显示基本信息，实现：输出 名字 、性别 、年龄
void Person::showInfo()
{
	cout << _name << "-" << _sex << "-" << _age << endl;
}
```

一般情况下，更期望采用第二种方式。**注意：上课为了方便演示使用方式一定义类，大家后序工作中尽量使用第二种。**

成员变量命名规则的建议：

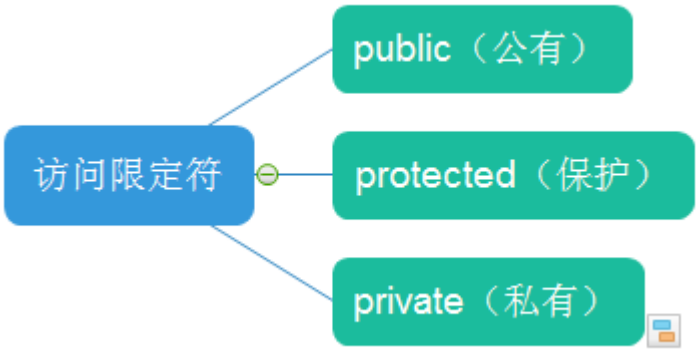
```
// 我们看看这个函数，是不是很僵硬?
class Date
{
public:
```

```
    void Init(int year)
    {
        // 这里的year到底是成员变量，还是函数形参?
        year = year;
    }
private:
    int year;
};

// 所以一般都建议这样
class Date
{
public:
    void Init(int year)
    {
        _year = year;
    }
private:
    int _year;
};

// 或者这样
class Date
{
public:
    void Init(int year)
    {
        mYear = year;
    }
private:
    int mYear;
};

// 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

【面试题】

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定义的类默认访问权限是public，class定义的类默认访问权限是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。

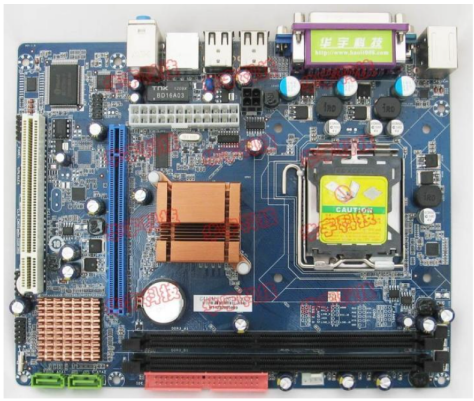
4.2 封装

【面试题】 面向对象的三大特性：**封装、继承、多态。**

在类和对象阶段，主要是研究类的封装特性，那什么是封装呢?

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

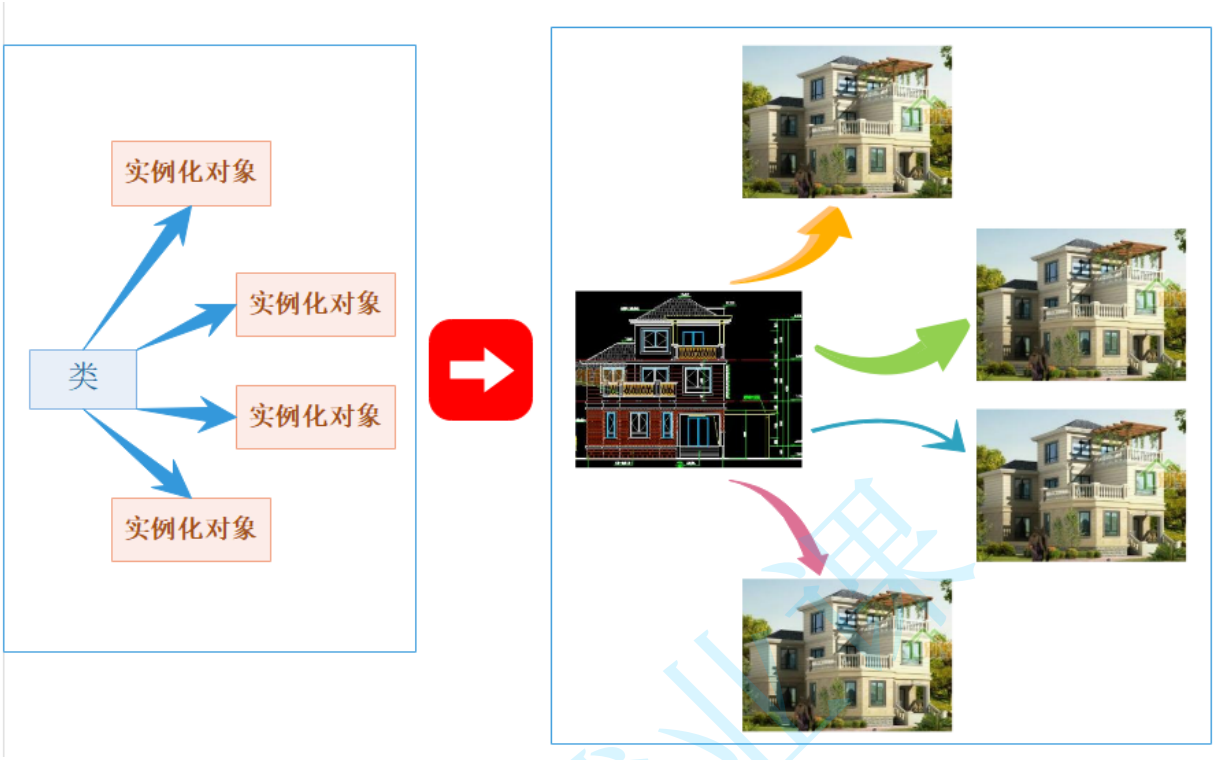
> **类定义了一个新的作用域**，类的所有成员都在类的作用域中。在类体外定义成员时，需要使用 :: 作用域操作符指明成员属于哪个类域。

```cpp
class Person
{
public:
    void PrintPersonInfo();
private:
    char _name[20];
    char _gender[3];
    int  _age;
};

// 这里需要指定PrintPersonInfo是属于Person这个类域
void Person::PrintPersonInfo()
{
    cout << _name << " "<< _gender << " " << _age << endl;
}
```

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；比如：入学时填写的学生信息表，表格就可以看成是一个类，来描述具体学生信息。
>
>   类就像谜语一样，对谜底来进行描述，谜底就是谜语的一个实例。
>
>   谜语："年纪不大，胡子一把，主人来了，就喊妈妈" 谜底：山羊
>
> 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**
>
>    ```cpp
>    int main()
>    {
>        Person._age = 100;   // 编译失败：error C2059: 语法错误:“.”
>        return 0;
>    }
>    ```
>
>   Person类是没有空间的，只有Person类实例化出的对象才有具体的年龄。

3. 做个比方。**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**，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，实例化出的对象才能实际存储数据，占用物理空间




```
//人
class Person
{
public:
    //显示基本信息
    void showInfo();
public:
    char* _name;    //姓名
    char* _sex;     //性别
    int   _age;     //年龄
};
```

```
void Test ()
{
    Person man ;
    man._name= "jack";
    man._age = 10;
    man._sex = "男 ";
    man.showInfo();
}
```

7.类对象模型

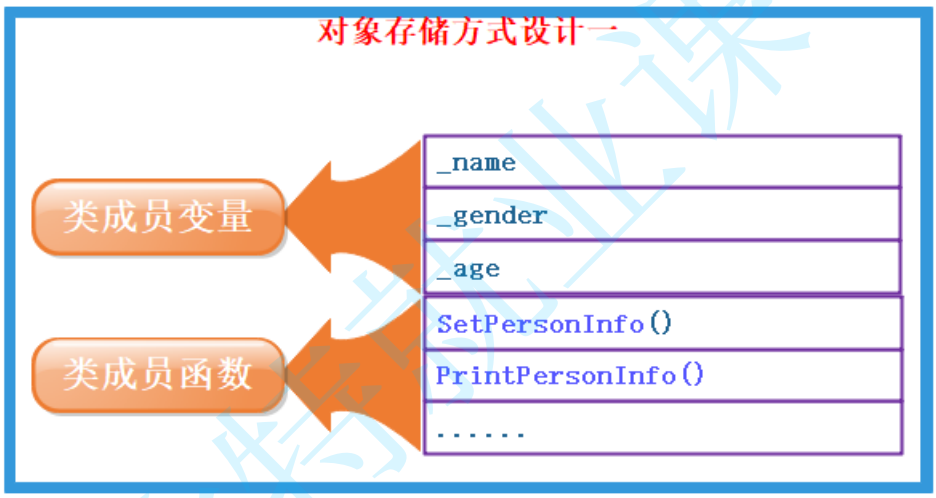
7.1 如何计算类对象的大小

```cpp
class A
{
public:
  void PrintA()
  {
      cout<<_a<<endl;
  }
private:
  char _a;
};
```

问题：类中既可以有成员变量，又可以有成员函数，那么一个类的对象中包含了什么？如何计算一个类的大小？

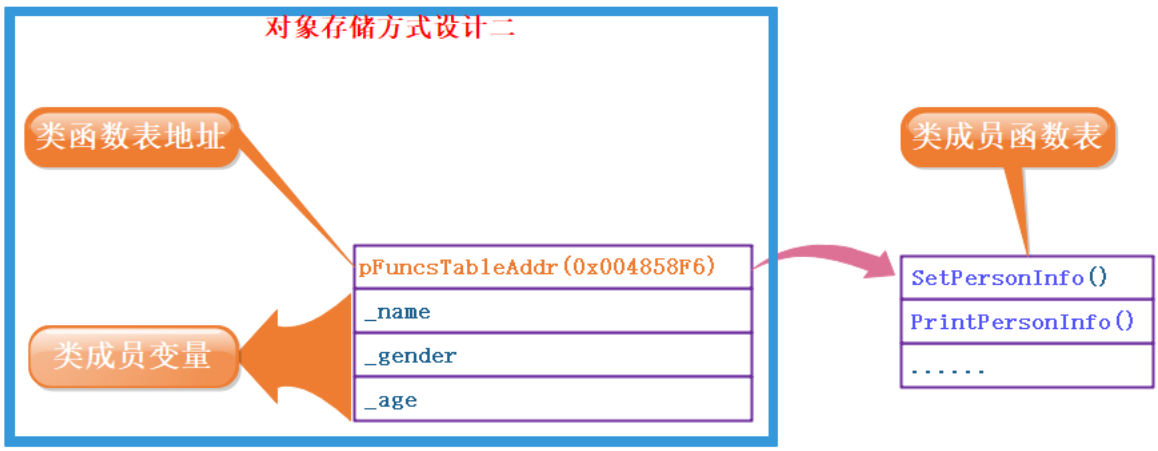
7.2 类对象的存储方式猜测

- **对象中包含类的各个成员**

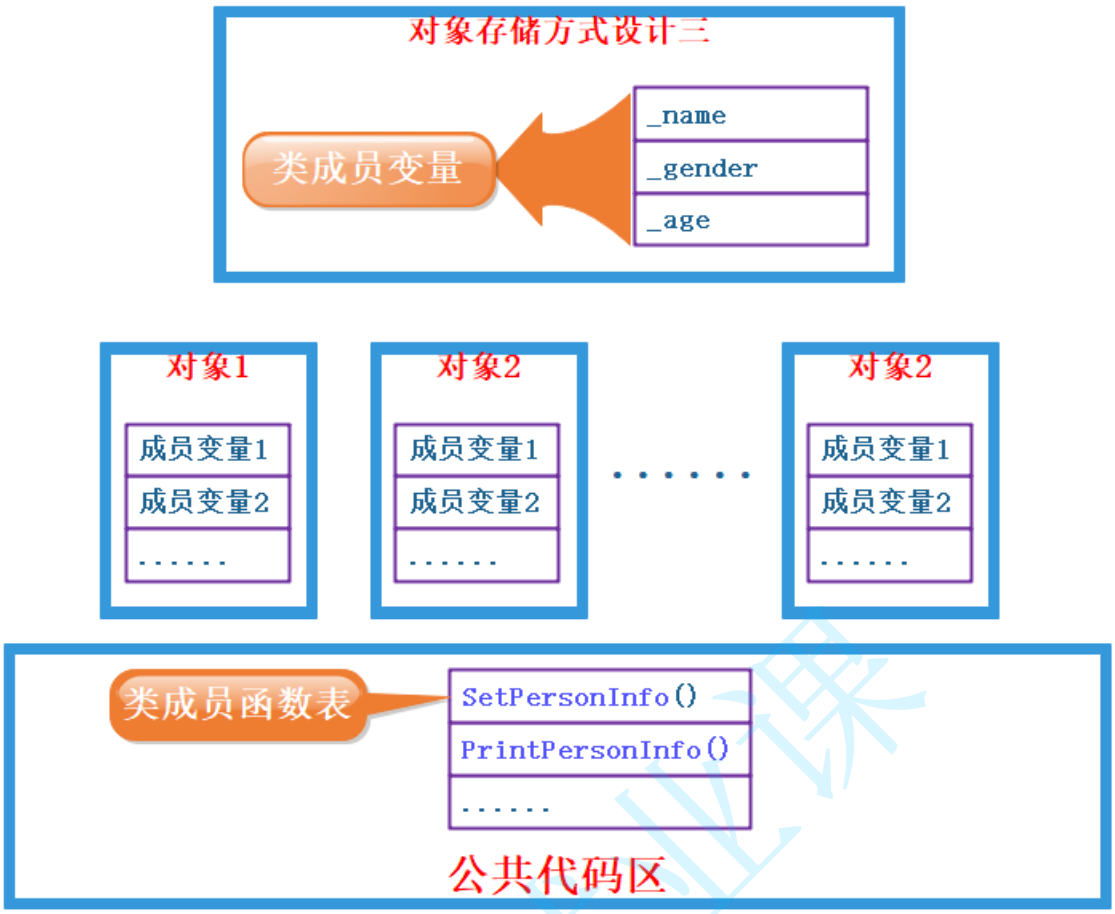


缺陷：每个对象中成员变量是不同的，但是调用同一份函数，如果按照此种方式存储，当一个类创建多个对象时，**每个对象中都会保存一份代码，相同代码保存多次，浪费空间。那么如何解决呢?**

- **代码只保存一份，在对象中保存存放代码的地址**



- **只保存成员变量，成员函数存放在公共的代码段**



问题：对于上述三种存储方式，那计算机到底是按照那种方式来存储的?

我们再通过对下面的不同对象分别获取大小来分析看下

```
// 类中既有成员变量，又有成员函数
class A1 {
public:
    void f1(){}
private:
    int _a;
};

// 类中仅有成员函数
class A2 {
public:
   void f2() {}
};

// 类中什么都没有---空类
class A3
{};
```

sizeof(A1) : _____ sizeof(A2) : _____ sizeof(A3) : _____

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。**

7.3 结构体内存对齐规则

> 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。
>
> **【面试题】**
>
> 1. 结构体怎么对齐？ 为什么要进行内存对齐？
> 2. 如何让结构体按照指定的对齐参数进行对齐？能否按照3、4、5即任意字节对齐？
> 3. 什么是大小端？如何测试某台机器是大端还是小端，有没有遇到过要考虑大小端的场景

8.this指针

8.1 this指针的引出

> 我们先来定义一个日期类`Date`

```
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;     // 年
    int _month;    // 月
    int _day;      // 日
    int a;
};

int main()
{
    Date d1, d2;

    d1.Init(2022,1,11);
```

```
    d2.Init(2022, 1, 12);
    d1.Print();
    d2.Print();
    return 0;
}
```

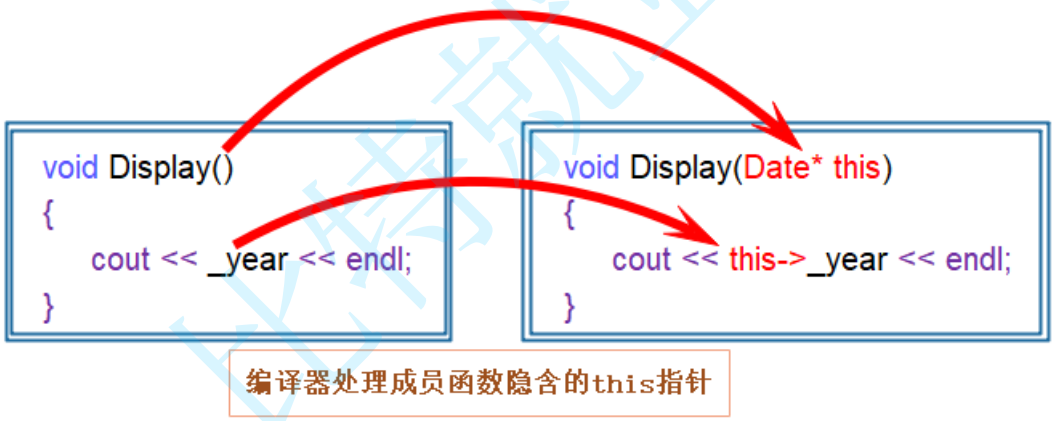
对于上述类，有这样的一个问题：

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 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指针解决该问题，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**



【面试题】

1. this指针存在哪里?
2. this指针可以为空吗?

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

```
};

int main()
{
    A* p = nullptr;
    p->Print();
    return 0;
}

// 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
class A
{
public:
    void PrintA()
    {
        cout<<_a<<endl;
    }
private:
    int _a;
};

int main()
{
    A* p = nullptr;
    p->PrintA();
    return 0;
}
```

8.3. C语言和C++实现Stack的对比

1. C语言实现

```
typedef int DataType;
typedef struct Stack
{
    DataType* array;
    int capacity;
    int size;
}Stack;

void StackInit(Stack* ps)
{
    assert(ps);
    ps->array = (DataType*)malloc(sizeof(DataType) * 3);
    if (NULL == ps->array)
    {
        assert(0);
        return;
    }

    ps->capacity = 3;
    ps->size = 0;
}
```

```c

void StackDestroy(Stack* ps)
{
    assert(ps);
    if (ps->array)
    {
        free(ps->array);
        ps->array = NULL;
        ps->capacity = 0;
        ps->size = 0;
    }
}

void CheckCapacity(Stack* ps)
{
    if (ps->size == ps->capacity)
    {
        int newcapacity = ps->capacity * 2;
        DataType* temp = (DataType*)realloc(ps->array,
newcapacity*sizeof(DataType));
        if (temp == NULL)
        {
            perror("realloc申请空间失败!!!");
            return;
        }
        ps->array = temp;
        ps->capacity = newcapacity;
    }
}

void StackPush(Stack* ps, DataType data)
{
    assert(ps);
    CheckCapacity(ps);
    ps->array[ps->size] = data;
    ps->size++;
}

int StackEmpty(Stack* ps)
{
    assert(ps);
    return 0 == ps->size;
}

void StackPop(Stack* ps)
{
    if (StackEmpty(ps))
        return;
    ps->size--;
}

DataType StackTop(Stack* ps)

{
```

```
    assert(!StackEmpty(ps));
    return ps->array[ps->size - 1];
}

int StackSize(Stack* ps)
{
    assert(ps);
    return ps->size;
}

int main()
{
    Stack s;
    StackInit(&s);

    StackPush(&s, 1);
    StackPush(&s, 2);
    StackPush(&s, 3);
    StackPush(&s, 4);
    printf("%d\n", StackTop(&s));
    printf("%d\n", StackSize(&s));

    StackPop(&s);
    StackPop(&s);
    printf("%d\n", StackTop(&s));
    printf("%d\n", StackSize(&s));

    StackDestroy(&s);
    return 0;
}
```

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。

2. **C++实现**

```
typedef int DataType;
class Stack
{
public:
    void Init()
    {
        _array = (DataType*)malloc(sizeof(DataType) * 3);
        if (NULL == _array)
        {
            perror("malloc申请空间失败!!!");
            return;
```

```
        }

        _capacity = 3;
        _size = 0;
    }

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

    void Pop()
    {
        if (Empty())
            return;
        _size--;
    }

    DataType Top(){ return _array[_size - 1];}
    int Empty() { return 0 == _size;}
    int Size(){ return _size;}

    void Destroy()
    {
        if (_array)
        {
            free(_array);
            _array = NULL;
            _capacity = 0;
            _size = 0;
        }
    }

private:
    void CheckCapacity()
    {
        if (_size == _capacity)
        {
            int newcapacity = _capacity * 2;
            DataType* temp = (DataType*)realloc(_array, newcapacity *
sizeof(DataType));
            if (temp == NULL)
            {
                perror("realloc申请空间失败!!!");
                return;
            }
            _array = temp;
            _capacity = newcapacity;
        }
    }

private:
```

```
    DataType* _array;
    int _capacity;
    int _size;
};

int main()
{
    Stack s;
    s.Init();

    s.Push(1);
    s.Push(2);
    s.Push(3);
    s.Push(4);

    printf("%d\n", s.Top());
    printf("%d\n", s.Size());

    s.Pop();
    s.Pop();
    printf("%d\n", s.Top());
    printf("%d\n", s.Size());

    s.Destroy();
    return 0;
}
```

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。**