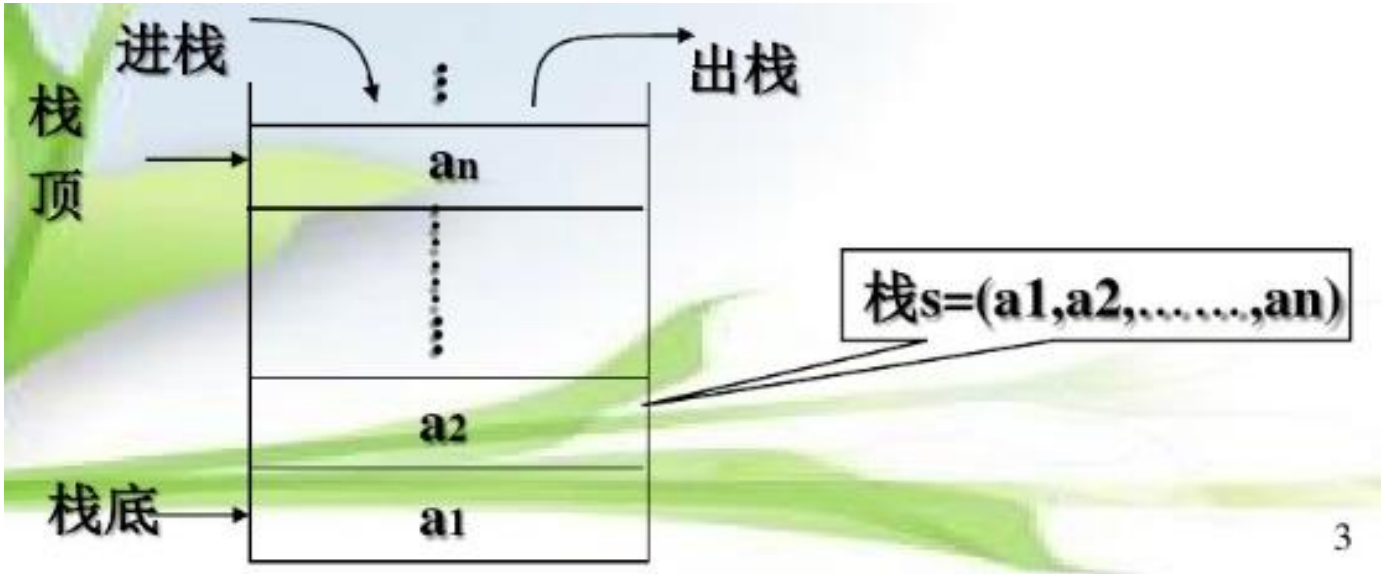
关于栈：定义

返回

- 栈是一种由若干个按线性次序排列的元素所构成的**复合数据**；
- 插入或删除栈元素只能在栈的一端进行，成为栈顶；
  - 栈只能实施两种操作：进栈push和退栈pop；
  - 两个操作必须在栈的同一端（栈顶）进行；

栈的初始化



栈的顺序存储实现——基础

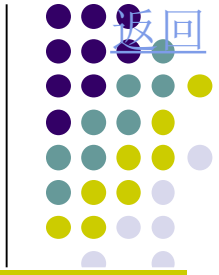
返回

- 栈的存储：依赖数组
  - 数组可以在任何位置插入和删除元素，因此需要在数组基础上做限定（这个限定是由程序员来实现的！）
- 为何把栈考虑成结构体
  - 栈的两个要素：栈顶指针和栈存储体
  - 栈顶指针：不是真指针，它仅指示栈顶位置（元素的处理位置），
  - 栈存储体：使用数组，且使用栈顶指针限定范围

```
1. const int STACK_SIZE=100;
2. struct Stack {
3.     int top;
4.     int buffer[STACK_SIZE];
5. };
```

栈实现两种方案比较

返回



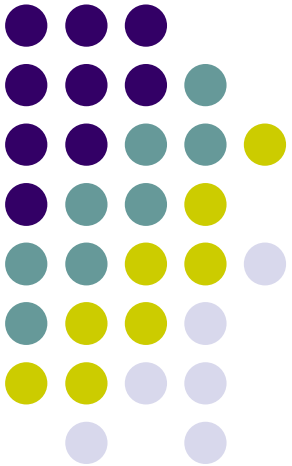
基于过程抽象	数据抽象和封装
侧重：函数	采用类， 将数据和操作封装在一起
函数是独立的	将函数纳为类的成员函数
	成员函数访问私有成员； 对象来调用成员函数

对象生灭

讨论：
构造函数和析构函数

吴清锋

2021年春

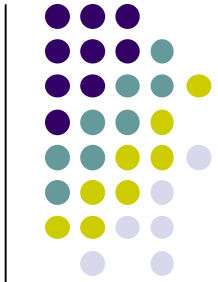


实际：

1、对象是如何初始化的

2、如何把类设计的更好

回忆：结构体的初始化



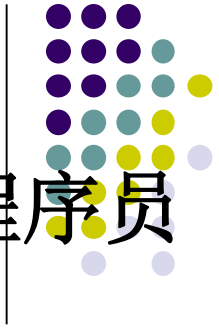
- **什么是初始化？**
  定义对象（变量）的同时给初始值。
- 例：

```
struct Complex {
    double real;
    double image;   };
struct Complex  s1={12,23};
```

- 为何？

  在main函数中，是可以直接访问结构体变量中各个成员

回忆：string的初始化



对用户而言，多种初始化的形式。现在从程序员角度考虑是如何实现“多种初始化”？

string s1;	默认构造函数,s1为空串
string s2(s1);	复制构造函数，将s2初始化为s1的一个副本(s1可以是字符数组或string)
string s3(“value”);	将s3初始化为一个字符串字面值副本
string s4(n,’c’);	将s4初始化为字符’c’的n个副本

- 对象创建过程一定伴随着构造函数的自动调用；
- 鉴于构造函数的运行时间特殊，常用于：对象初始化；
- 多种初始化形式表明构造函数的重载；

导引：对象的生灭与自动执行函数

- 在对象创建（无论显示或是隐式）或是销毁过程，都会有对应函数（构造函数、析构函数）自动执行。

- 构造函数是特殊的成员函数，只要创建类类型的新对象，都要执行构造函数；
- 在对象销毁的时候，会有析构函数自动执行。

提纲

- 对象的创建与初始化
- 构造函数概述——特性与分析
- 如何写构造函数
- 使用默认参数的构造函数
- 默认构造函数
- 构造函数的重载
- 构造函数初始化列表
  - 形式
  - 需要构造函数初始化列表的情形
- 经常被误解的构造函数
- 析构函数
- 构造函数与析构函数的关系

返回

对象的创建与初始化

- C++提供了构造函数，用户可通过构造函数来完成期望在创建对象时需要完成的事情。比如，在创建来处理对象的初始化
  - 什么是初始化？
    定义对象（变量）的同时给初始值。
  - 初始化必要性
  - 初始化方式思考
    1、能否借鉴结构体初始化的模式
    2、之前代码是如何运行的？能否真的实现初始化

构造函数的特性

返回

- 构造函数：
  - 是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用，而是在**建立对象时**自动执行
    - **建立的对象**：有名称的对象或是**未命名的对象**
      显示定义的对象或隐式生成的对象
    - **自动执行：**也无需通过”对象….”
  - 构造函数的名字必须与类名同名，而不能由用户任意命名
  - 它不具有**任何类型，不返回任何值**；

要从特性反思对编程实现的贡献！

构造函数的分类

默认构造函数

带默认值的构造函数

- 隐式默认构造函数

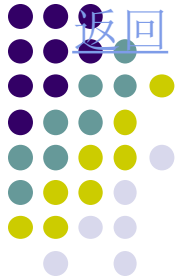
构造函数是**C++提供的**，没有参数，函数体是空的，可能什么都不处理；

- 用户自定义构造函数

函数功能**由程序员定义**：程序员根据初始化要求设计函数体和函数参数,如赋初值,值的有效性校验等。

- 带参数的构造函数
- 不带参数的构造函数

一旦用户自定义了构造函数，隐式默认构造函数就不会执行。



隐式默认构造函数

- 例子：对象创建
  - **Time t1,t2; //对象创建，自动伴随构造函数**
- 类的描述
  - **class Time {**
    private:
    int hour;
    int minute;
    int second;
    public:
    void SetTime(int h,int m,int s);
    void Myprin( );
    };

C++会有隐式默认构造函数：
Time::Time() { }

返回

如何写构造函数：例子及分析

- 带参数的构造函数

- 不带参数的构造函数

思考：

1 函数是否带参数，这取决于什么？

2 函数是否带参数，对初始化的影响？

不带参数的构造函数

- 不带参数的构造函数
  - 若构造函数不带参数，在函数体中对各数据成员赋值，此类方式使该类的每一个对象都得到同一组初值。
  - class Time {
    private:
    int hour;
    int minute;
    int second;
    public:
    Time()　　//不带参数的构造函数
    { hour=11;
    minute=12;
    second=13;
    }
    };//构造函数没有返回值，函数名与类名一致

- Time t1,t2;
t1和t2中的私有成员均为11,12,13

有别于：

```
class Time {
 private:
   int hour=1;
   int minute=1;
   int second=1;
};  //ERROR
```

带参数的构造函数

返回

- 带参数的构造函数
  - 能对不同的对象赋予不同的初值
  - 对象在生成时，通过自动调用构造函数，从外面将不同的数据传递给构造函数，以实现不同的初始化
  - 模型表示

main函数中

```
Date d1(2020,3,15);
```

1 d1对象创建，与此同时，就会自动调用构造函数
2 2020,3,15是实参

```
Date::Date (int x,int y,int z)
{   year=x;
    month=y;
    day=z;
}
```

函数名，即类名

1 设置形参，才能接收用户传递过来的实时的信息
2 函数体，用接收信息的形参处理数据成员。

带参数的构造函数的定义

- 定义过程：构造函数的首部的一般格式：
  构造函数名(**类型1** 形参1,**类型2** 形参2,..)
- 使用过程：实参是在定义对象时给出
  类名 对象名(实参1,实参2,….)

类的声明和构造函数定义

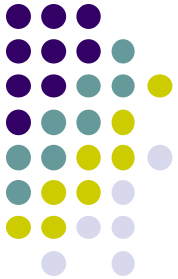
```
class Time {
  private:
     int hour;
     int minute;
     int second;
  public:
     Time(int,int,int);
};
```

```
Time::Time(int x,int y,int z)
{    hour=x;
     minute=y;
     second=z;
 }
```

对象创建

```
Time t1(13,12,20);
```

深入探究问题



- 参数中带有默认值
- 默认构造函数
- 构造函数的重载

使用默认参数的构造函数（1）

返回

- 构造函数中参数的值可以通过实参传递，也可以指定为某些默认值，即用户不指定实参值，编译系统就使用默认值；
- 在调用构造函数时若没有提供实参值，此时将按照默认的参数值对对象进行初始化。尤其适用于对每一个对象都有同样的初始化值时；
- **注意：**
  - 需要遵循函数带默认参数的要求
  - 如果在类定义中仅包含函数的声明，则默认的参数值应该放在声明中，而不能放在函数定义中；

例子

返回

```
#include <iostream>
using namespace std;
class Tc {
  public:
    Tc(int a=4,int b=18);
    int Acout( ) {return A;}
    int Bcout( ) {return B;}
 private:
    int A,B;
};
```

```
Tc::Tc(int a,int b) {
  A=a;B=b;
} //带默认参数的构造函数，不给出默认值
```

main函数中：

```
Tc  c1,c2(9),c3(17,22);
```

思考：

c1、c2和c3私有成员值？

使用默认参数的构造函数（2）

- 问题更复杂：带默认参数的构造函数与构造函数重载
  - 没有带参数的构造函数与全部为默认参数的构造函数 ×

    原因：定义了多个默认构造函数（或对重载函数调用不明确）
  - 没有带参数的构造函数与部分为默认参数的构造函数 √

例子

返回

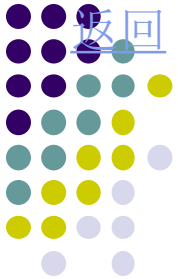
```
#include <iostream>
using namespace std;
class Tc {
  public:
    Tc(int a=4,int b=18);
    Tc();
    int Acout( ) {return A;}
    int Bcout( ) {return B;}
 private:
    int A,B;
};
```

```
Tc::Tc(int a,int b) {
  A=a;B=b;
} //带默认参数的构造函数，不给出默认值
Tc::Tc( ) {
  A=40; B=180;
} //两个构造函数的重载
```

main函数中：

Tc c1;

编译器抉择困难



默认构造函数（1）

- C++有一个定义：默认构造函数
  - 隐式默认构造函数
  - 自定义的构造函数
    - 不带参数的自定义构造函数
    - 带默认参数的自定义构造函数

默认构造函数定义的必要性

返回

- 如果为类已经定义了一个带参数的构造函数【C++免费送的隐式默认构造函数就不起作用】，若想要使用Time t1，则必须定义自己的“默认构造函数”；
- 途径1：不带参数
- 途径2：全部默认值

```
class Time
{  private:
       int hour;
       int second;
   public:
       Time(int x=1,int y=100):hour(x) { }  //默认构造函数
       Time(double y):hour(y)  { }
        void display(){ cout<<hour; }
};
```

Time(int x,int y):hour(x) { }
那么，执行：
Time X; //调用默认构造函数，注意：不需要添加括号
会提示：
no appropriate default constructor available

再次强调：程序员定义了构造函数，那么C++“送的”默认构造函数就不会起作用；为了支撑Time t1【此时，对象的创建需要一个默认构造函数】，就需要自行定义

构造函数的重载：
Time(int,int);
Time(double);

构造函数的重载(1)

- 回顾一个现象：string对象的多种初始化
  思考,为何能支持这么多种的初始化形式?
  构造函数重载
- 回顾：什么是重载？
  多个功能目标相同的函数，其函数名字一致。
- 【编程思考】通过函数重载，提供不同的构造函数。这就允许**用户指定不同的方式来初始化数据成员**;
  构造函数的重载：具有相同的函数名字,而参数的个数或参数的类型或位置不相同;

构造函数的重载(2)

返回

- 构造函数重载的注意事项：
  - 尽管构造函数同名，**系统根据函数调用的形式（实参）**去确定将选用的构造函数；
  - 应当警惕：编译器选择的困惑
    没有带参数的构造函数与全部为默认参数的构造函数 同时出现（重载）
  - 若在建立对象时选用的是无参构造函数，应注意正确书写定义对象的语句，尤其避免写括号！
    - 例子：Sales_item **empty**; //使用默认构造函数

构造函数的初始化列表

- 在构造函数中，可以用显式的赋值语句来初始化对象的成员，还可以使用“构造函数初始化列表”
  - 这是一种值得推荐的方式
- 例子：

  Time::Time(int h,int m,int s):hour(h),minute(m),second(s) { }
- 形式
  - 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，每个数据成员后面跟一个放在圆括号中的初始化式；
  - 构造函数初始化式只在构造函数的定义中而不是声明中指定；

成员初始化的次序

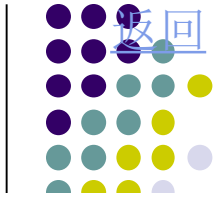
返回

- 构造函数初始化列表，仅**给出**用于初始化成员的各个值，并**未对初始化成员执行的次序进行指定；**
- **成员被初始化的次序依赖于成员在类中定义的次序；**
- 例子

```
class Time
{  private:
      int second;  //定义的顺序
      int hour;
   public:
      Time():hour(1),second(hour) { } //会有莫名的值
      void display(){cout<<second<<"   "<<hour;}
};
```

应该修改成：
int hour;
int second;

只能使用构造函数的初始化列表



- 很多时候，初始化列表可以**转化为**通过在函数体内对数据成员赋值来实现。
- 但是，在某些情况下，只能使用初始化列表：
  - 某个类中的类成员，没有默认构造函数；
    - 反思：若没有为类成员提供初始化式，则编译器会隐式地使用类成员的默认构造函数。而该类成员若恰好没有默认构造函数，那么编译器尝试使用默认构造函数就会失败；
  - 某个类中有const成员；
  - 某个类中有引用类型的成员；
  - **如果类存在继承关系，派生类必须在其初始化列表中调用基类的构造函数**
- 例子

某个类中有const成员

```
#include <iostream>
using namespace std;
class A  {
    public:
        A(int size) : SIZE(size) {    }
    private:
        const  int SIZE;  //const意味着什么呢？
};
int main()  {
    A a(100);
}
```

- 在类中声明成员为**const**类型，但是不可以初始化
- **const**常量的初始化必须在构造函数初始化列表中初始化，而不可以在构造函数函数体内初始化

某个类中有引用类型成员

```
class A {
   public:
      A(int &v) : i(v), p(v), j(v) {   }
      void print_val() {
          cout << "hello:" << i << " " << j << endl;}
   private:
      const int i;
      int p;
      int &j; //j与谁绑定？
};
 int main( ) { int r= 45; A b(r); b.print_val(); }
```

类的特殊成员:类成员

返回

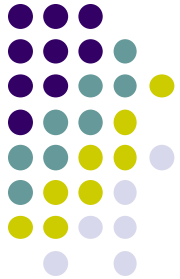
- 类定义中有数据成员和成员函数，数据成员可以是内部数据类型的变量实体，也可以是对象实体；
  - 若类的数据成员部分有其他类的对象实体的声明。此时，构造函数是如何工作的？
- 例子

```
#include <iostream>
class Base {
   public: Base(int a) : val(a) {   }
   private: int val;
 };
class A {
  public:
     A(int v) : p(v), b(v) {  }
      void print_val() {cout << "hello:" << p << endl;}
  private:
    int p;
   Base b;
};
```

此时**b**是个对象，创建的时候，要调用**Base**的构造函数，将**v**作为实参给**a**

```
int p1 = 45;
A f(p1);
f.print_val();
```

经常被误解的构造函数



- 构造函数经常用来做初始化工作
- 但是：

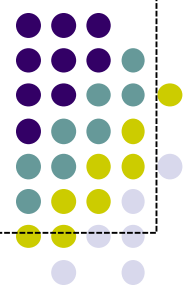
  构造函数的功能≠初始化工作
- **还可以有：new创建空间**

  即：数据成员中指针出现

  （这是一个神奇的问题！）

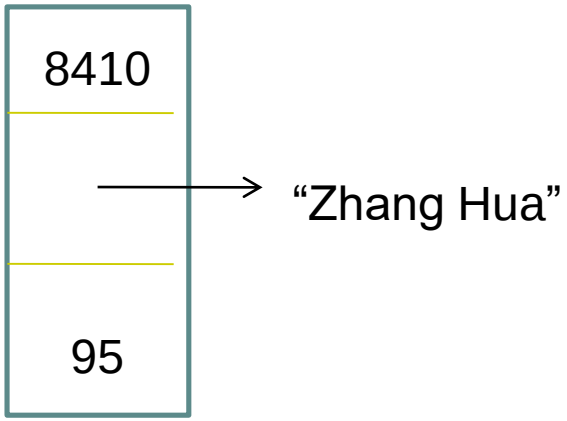
```
class Student {
  public:
    Student(int pid,char *pname,float s);
    Student();
    void modify(float s);
    void display();
  ~Student();
private:
   int id;
   char *name;
   float score;
};
```

```
Student s1;
Student s2(8410,"Zhang hua",95);
s2.modify(90);
```



```
Student::Student(int pid,char
    *pname,float s) {
  id=pid;
  name=new char[strlen(pname)+1];
  strcpy(name,pname);
  score=s;
}
Student::Student( ) {
   id=0;
   name=new char[11];
   strcpy(name,"No name");
   socre=0;
}
void Student::modify(float s) {score=s; }
```

对象的模型



存在风险？

内存泄漏

```
Student s1;
Student s2(8410,”Zhang hua”,95);
s2.modify(90);
```

```
Student::Student(int pid,char
    *pname,float s) {
  id=pid;
  name=new char[strlen(pname)+1];
  strcpy(name,pname);
  score=s;
}
Student::Student( ) {
   id=0;
   name=new char[11];
   strcpy(name,”No name”);
   socre=0;
}
void Student::modify(float s) {score=s; }
```

析构函数

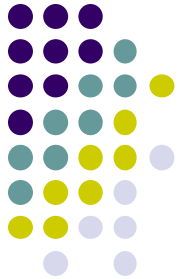
返回

- 当对象消亡时，在系统收回它所占的内存空间之前，对象类的析构函数会被自动调用。
- 析构函数分类
  - 隐式析构函数
  - 用户自定义析构函数
- 例子
  - 执行过程

```
class A {
  private:   int x;
  public:
     A( );
     ~A( ); //析构函数
  };

void f( ) {
   A a;  //自动调用构造函数
   …
} //自动调用析构函数
```

析构函数重点需要考虑的问题



- 对象空间（本体空间）是由操作系统撤销的；
- 若程序执行过程，对象有申请资源，则需要有析构函数来进行资源的回收
  - **例如：若类的声明中有数据成员是指针关系时**

```
Student::~Student( ) {
    delete [ ] name;
}
```

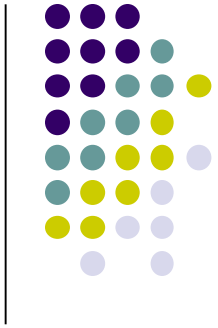
构造函数和析构函数调用顺序关系

- 在对象的生灭过程中，实际是一个函数关于**栈**的调用过程

- 关于局部对象（不是全局对象），调用构造函数与定义对象的顺序相同，而调用析构函数的次序正好与创建的顺序相反。

- 若有类成员出现时，构造函数调用遵循“尊老爱幼”：类成员就是“幼”；析构函数调用遵循栈。

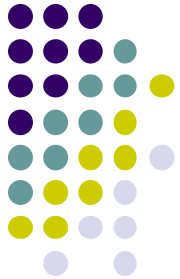
```
1.  #include <iostream>
2.  using namespace std;
3.  class Base {
4.  private:
5.     int b;
6.  public:
7.        Base(){cout<<"Base的构造函数\n";}
8.        ~Base() {cout<<“Base的析构函数\n";}
9.  };
10. class A {
11. private:
12.    int a1;
13.    Base a2;
14. public:
15.       A(){cout<<"大类A的构造函数\n";}
16.       ~A() {cout<<"大类A的xigou函数\n";}
17. };
```



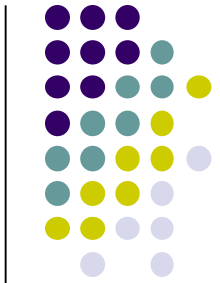
```
主函数：
int main()
{
            A x1;
          return 0;
}
```

Base的构造函数
大类A的构造函数
大类A的析构函数
Base的析构函数

总结



- 通过构造函数和析构函数来描述的是对象的生和灭
- 对构造函数和析构函数的认识，便于设计出更好的类，如：对于构造函数
  A 具有初始化功能
  B 具有多种初始化形式（重载）



感　谢！