

面向对象程序语言与C++

授课人：董燎原

中国科学院高能物理研究所

C/C++

2

✚ C语言：20世纪70年代

- ✚ 语言简洁高效，实现汇编级的操作，目标程序效率较高
- ✚ 缺陷：类型检查机制弱，缺少支持代码重用，开发大程序困难

✚ C++语言：Bjarne Stroustrup（Bell Lab.）1983年诞生

- ✚ 保留C的简洁高效，引入“类”（class）的概念，提供面向对象的编程支持，程序的结构更清晰，更易维护和扩充
- ✚ 新增了一些新的运算符，改进了类型系统
- ✚ 引进了运算符重载、引用、虚函数等许多特性
- ✚ 适合于大、中型软件的开发，开发时间费用、可重用性、可扩充性、可维护性和可靠性等方都有很大的优越性

程序设计

3

✚ 程序由两个主要方面组成

- ✚ **算法的集合**：将指令组织成程序来解决某个特定的问题
- ✚ **数据的集合**：算法在数据上操作以提供问题的解决方案

✚ 它们之间的关系就是所谓的程序设计方法(programming paradigm)

✚ 程序设计方法

面向过程的编程 (procedural programming) :

- ✚ “what to do” in each step (Fortran, Pascal, C)

面向对象的编程 (Object-oriented programming) :

- ✚ “what is” (object, their properties, etc)

面向对象编程

4

✚ 面向过程的程序（如 C 语言、PASCAL、FORTRAN ）设计：

程序 = （算法(过程或函数)）+ （数据结构）

缺点：不利于软件的重用，过程与数据相互依赖，无法分开维护，开发过程复杂，效率低。

✚ 面向对象的程序（如C++）设计：

数据结构及其相关算法被封装成类，对象是类的一个实例。

对象 = （算法+数据结构）； 程序 = （对象+对象+...）

优点：提高了程序的重用，简化程序设计，提高开发效率。

对象成为程序的基本构件，编程类似在搭积木。

面向对象编程

5

✚ 通过继承机制和动态绑定机制扩展了抽象数据类型

- ✚ 继承机制是对现有代码的重用
- ✚ 动态绑定是指对公共接口的重用。

✚ 面向对象程序设计的基本特征

- ✚ **封装：**实现了数据隐藏，保护对象的数据不被外界随意改变，使对象成了相对独立的功能模块。
- ✚ **抽象：**是特定属性的事物的一个概括，以隐藏事物固有的复杂性。
- ✚ **继承：**允许一个类继承其它类(称为基类)的属性和方法, 该类称为派生类；是类的层次结构之间共享数据和方法的机制，允许和鼓励类的重用。
- ✚ **多态：**不同类的对象对同一消息作出不同的响应。

BOOKS:

- The C++ Programming Language by Bjarne Stroustrup.
- C++ Primer by Lippman and Lajoie. *Available online.*
- Effective C++ by Scott Meyers. *Available online.*
- More Effective C++ by Scott Meyers. *Available online.*

INTERNET:

- <http://www.cppreference.com/wiki/>
- <http://www.cplusplus.com/reference/>

一个简单的C++程序

7

C++/simple.C

```
// a simple C++ code
#include <iostream>
using namespace std;
int main()
{
    double x,y;
    cout<<"Enter two double number:";
    cin>>x>>y;
    double z=x+y;
    cout<<"x+y="<<z<<endl;
    cout<<__LINE__<< endl;
    return 1;
}
```

- 主函数入口main，可有参数，也可无参数，但返回值需为int
- 头文件(.h)：定义函数原型、类的声明等。
- 源程序文件(.C or .CPP)：定义函数或类成员函数的实现。
- 预处理命令#include：把一个文本的内容包含到该命令处
- 编译： g++ simple.C 运行： a.out 或者 ./a.out

一个简单的C++程序

8

- ✚ 头文件包含有对象和函数说明。

如： `iostream.h` 声明用于输入和输出等的流操作。

- ✚ `using namespace std;` 意思是使用c++的标准命名空间。

`cout`是输出流对象，`cin`是输入流对象。

`cout<<__LINE__;`（显示当前行号，常用于定位错误）`endl`是换行符。

- ✚ “>>” 从输入流中提取数据赋值给一个变量，称为提取操作符

“<<” 称为插入运算符；

- ✚ 注释符： 一种以 “//” 起头，直到行末；

一种是用斜线星号组合 “/*” 和 “*/” 括起的任意文字。

- ✚ 编写程序时, 代码需要书写工整有层次, 必要时添加注释。

PART 1: C++基本语法

C++ 中的数据类型

10

- 支持原有C的基本数据类型: 使用 `sizeof` 函数查看各类型数据变量的长度。C++/sizeof.C

- `void` 空类型

- `int` - 整型 (4字节Byte) Range: ± 2147483648

- `double` - 双精度 (8字节) Range: $\pm 1.7 \cdot 10^{\pm 308}$ 小数点后精度为大约 15 到 16 位

- `Float` - 单精度 (4字节) Range: $\pm 3.4 \cdot 10^{\pm 38}$ 小数点后精度为大约 6 到 9 位

- `char` - 字符型 (1字节)

- C++定义了两种新的类型:

- `bool` 布尔型: `true`, `false`

- `wchar_t` 宽字符型: 用于扩展字符集, 比如汉字和日语 (2或4字节)

- 扩展更多的类型: `signed`, `unsigned`, `short`, and `long`

`sizeof(long) >= sizeof(int) >= sizeof(short)`

ref: http://www.cppreference.com/wiki/data_types

常量与变量

11

✚ 字符与字符串:

✚ `c = 'a';` `//` `c`是一个字符

✚ `s = "string";` `//` `s`是一个指向7个字符数组的指针

✚ 变量为程序中值可以改变的量: 名字、类型和值

✚ 标识符: 只能由字母数字下划线组成, 非数字开头, 非系统保留字。

✚ 声明: `<类型> <变量名1, 变量名2, ...>;`

✚ 常变量: `const <类型> <变量名1, 变量名2, ...>;`

数组

12

✚ 声明数组:

```
int c[12];    或    const int max = 12; int c[max];
```

✚ 使用: `c[0]`, `c[1]`, ..., `c[max-1]`

✚ 初始化: `int n[5] = {1, 2, 3, 4, 5};`

或 `int n[] = {1, 2, 3, 4, 5};`

✚ 字符型数组: `char string1[20];`

或 `char string2[] = "string literal";`

每个数组元素是一个字符型: `string2[3] = 'i';`

✚ 数组名存储数组的首地址(const), 具有指针的作用

13

□ C/C++中多维数组按行排列

多维数组初始化

14

2个分波分析中用到的数组的初始化:

```
const int G[4][4] = { {1,0,0,0}, {0,-1,0,0}, {0,0,-1,0},  
                      {0,0,0,-1} };  
  
const int E[4][4][4][4] =  
{ { { {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0} },  
    { {0,0,0,0}, {0,0,0,0}, {0,0,0,1}, {0,0,-1,0} },  
    { {0,0,0,0}, {0,0,0,-1}, {0,0,0,0}, {0,1,0,0} },  
    { {0,0,0,0}, {0,0,1,0}, {0,-1,0,0}, {0,0,0,0} } },  
  { { {0,0,0,0}, {0,0,0,0}, {0,0,0,-1}, {0,0,1,0} },  
    { {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0} },  
    { {0,0,0,1}, {0,0,0,0}, {0,0,0,0}, {-1,0,0,0} },  
    { {0,0,-1,0}, {0,0,0,0}, {1,0,0,0}, {0,0,0,0} } },  
  { { {0,0,0,0}, {0,0,0,1}, {0,0,0,0}, {0,-1,0,0} },  
    { {0,0,0,-1}, {0,0,0,0}, {0,0,0,0}, {1,0,0,0} },  
    { {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0} },  
    { {0,1,0,0}, {-1,0,0,0}, {0,0,0,0}, {0,0,0,0} } },  
  { { {0,0,0,0}, {0,0,-1,0}, {0,1,0,0}, {0,0,0,0} },  
    { {0,0,1,0}, {0,0,0,0}, {-1,0,0,0}, {0,0,0,0} },  
    { {0,-1,0,0}, {1,0,0,0}, {0,0,0,0}, {0,0,0,0} },  
    { {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0} } } };
```

指针

15

- ✦ 指针变量的值指向变量的地址(内存位置的地址)
- ✦ 指针也有类型定义：对不同的类型变量的地址分别对待。

- ✦ 操作符&和*

例：int *p; // a pointer to int 定义整型指针变量

int c; // an int variable 整型变量

p = &c; // address of c is given to p

 // 将c在内存的地址给p

指针

16

- ✚ 指向常量的指针: `const int *p;`
 - ✚ `p`指向一个不能修改的常量数据
- ✚ 常指针: `int * const p = &x;`
 - ✚ `p`总是指向变量`x`
- ✚ `const int *const p = &x;`
 - ✚ `p`总是指向一个不能修改的常量数据
- ✚ 指向函数的指针: 函数名可看作为函数入口的地址, 该地址就称为函数的指针
可用指针变量指向函数, 通过该指针变量可调用该函数。
`void (*f)(int);`
 - ✚ `f`是指向形式为`void g(int)`的函数: `f = g;`
 - ✚ `(*f)(5);` // call the function

指针和数组

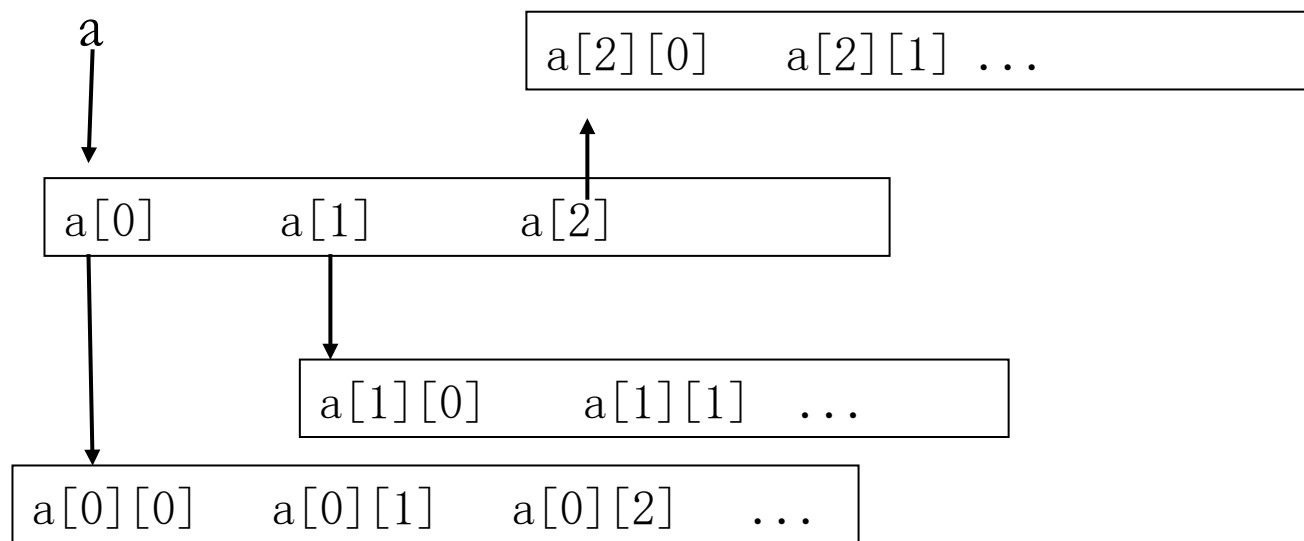
17

表达式 $*(a + k)$ 等同于 $a[k]$

对于多维数组: `int a[3][7];`

a 是指向指针数组 $a[0]$, $a[1]$, $a[2]$ 的指针;

$a[i]$ 是指向数组 $a[i][j]$ 的指针; $**a$ 等同于 $a[0][0]$



指针和字符数组

18

- ✚ `char *suit[4] = { " Hearts ", " Diamonds ", "Clubs", "Spade"};`
- ✚ `suit[0][0]` refers to 'H',
- ✚ `suit[3][4]` refers to 'e' in "Spade",
- ✚ `suit[2][20]` is an illegal access.

引用

19

- ✦ **另一标识符的别名**，C++的一种新的变量类型，对引用的改动即为对目标的改动。
- ✦ **引用运算符“&”**，引用在声明时就初始化，并且无法修改

float & ra=a; //ra引用a，ra等同于a

float *pa=&ra; //ra等同于a，因此 pa指向a

float *p;

float * &rp=p; //rp引用指针p，rp等同于p

float m=6.0;

rp=&m; //对rp的访问，就是对p的访问

引用

20

- 引用声明为T&，它必须用T类型的变量或对象，或能够转换成T类型的对象进行初始化。如果初始值不可获得地址，那么将建立临时变量目标去初始化。double& rr=1; 合法。等同于：

```
double temp;  
//类型转换，并放在临时变量temp中  
temp=double(1);  
double& rr=temp;
```

- void & a=3; //错误

因为 **void** 本质不是类型

```
int a[10];  
int& ra[10]=a; //错误
```

不能引用数组，数组是数据类型的集合，代表该集合空间的起始地址，不是数据类型。

函数的引用类型返回值

21

- ✚ 函数返回值时，要生成一个值的副本；
- ✚ 引用返回值时，不生成值的副本。

C++/func_ref.C

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

float temp;
float fn1(float r){ temp = r*r*3.14; return temp;}
float& fn2(float r){ temp = r*r*3.14; return temp;}
int main(){
    float a=fn1(5.0);          //cond:1
    float& b=fn1(5.0);          //cond:2(error)
    float c=fn2(5.0);          //cond:3
    float& d=fn2(5.0);          //cond:4
    cout<<a<<endl<<b<<endl<<c<<endl<<d<<endl;
    return 1;
}
```

变量**temp**是全局数据，驻留在全局数据区**data**，
函数**main()**、函数**fn1()**和函数**fn2()**驻留在栈区**stack**。

函数的引用类型返回值

22

- ✚ cond 1: a获得全局变量temp的临时变量副本的复制值
- ✚ cond 2: b为全局变量temp的临时变量副本的别名。C++标准中，临时变量生命期在一个完整的语句表达式结束后结束，即在“float& b=fn1(5.0);”之后，引用b的值是无法确定。编译错误。修改为：

```
int x=fn1(5.0);    int& b=x;
```

- ✚ cond 3:函数fn2()的返回引用，不产生副本，c直接从变量temp中得到复制。提高程序执行效率和空间利用。
- ✚ cond 4:d成为temp的别名。temp是全局变量，这种方式是安全的。如果返回不在作用域范围内的变量或对象的引用，与返回一个局部指针一样，将出现编译错误。

函数参数中的引用和指针

23

- ✦ 函数参数为变量值：只把变量的副本传入函数，不对主程序中的变量起作用。
Swap(int,int)未能达到交换数据的目的。
- ✦ 可用引用swap(int&, int&)或者指针swap(int *,int *):

```
int swap(int &a, int &b);
int main()
{
    int a=1, b=2;
    swap(a,b);
}
int swap(int &a, int &b)
{
    if (b > a) {
        int c = a; a=b; b=c;
    }
}
```

实际传递的就是它们的地址

```
int swap(int *a, int *b);
int main()
{
    int a=1, b=2;
    swap(&a,&b);
}
int swap(int *a, int *b)
{
    *a= ..;
}
```

需要向指针变量传递变量的地址，然后用指针访问变量。

引用具有指针的功效，调用“引用传递”的函数时，可读性比指针好。

运算符

24

✚ $X \text{ op} = Y$ 等同于 $X = X \text{ op} Y$

op 可以是 +, -, *, /, %等

✚ 如: `c += 7;` // same as `c = c + 7;`

`f /= 3;` // same as `f = f / 3;`

✚ ++ 递增操作（加1）；

✚ -- 递减操作（减1）；

✚ ++i 先加1，再引用i的值；

✚ i++ 先引用i的值，再加1；

✚ ~（按位求反） &（按位与） |（按位或） ^（按位异或）；

✚ <<（左移位运算符） >>（右移位运算符）；

算术运算

Operator	Meaning
-a	Sign change
a*b	Multiplication
a/b	Division
a%b	Modulus
a+b	Addition
a-b	Subtraction

逻辑比较

25

== equal to

Be careful! “==” and “=” are different !

!= not equal to

> greater than

< less than

>= greater than or equal to

<= less than or equal to

! logical NOT operator

&& (||) logical AND (OR) operator

在C++中，false为0；true为1（任何非零值都为true）

如：k = 2 < 3; // k=1

流程控制

26

```
if (cond) {  
    statements1 ;  
} else {  
    statements2 ;  
}
```

```
while(cond) {  
    statements;  
}
```

```
do{  
    statements;  
} while(cond)
```

```
switch (x) {  
    case 'a':  
        statements;  
        break;  
    case 'b':  
        ...  
    default:  
        ...  
}
```

```
for( initialize; cond; incre ) {  
    statements;  
}
```

例如：

```
for(int j = 0; j < 10; j++) {  
    ...  
}
```

转向语句：

goto: (尽量不使用)

continue

break

函数 (function)

27

✓合理组织和封装大量程序代码和数据，隐藏具体实现的细节，实现更高级的抽象

✓把一个程序分多个函数来实现

✓把类似的函数放入独立的文件

✓一般结构:

```
Prototypes; //函数声明
```

```
int main() { ... ;} //函数使用； 参数传递
```

```
func1() { ...; } //函数定义
```

```
func2(int i, ...) { ... } //函数定义
```

函数 (function)

28

- ✓ **默认参数**: 可以在函数声明或定义时给一个或多个参数指定默认值。
- ✓ **要求**: 指定了默认值的参数右边不能有没有默认值的参数。
- ✓ 调用有默认参数的函数时, 省略的参数将使用默认值。
- ✓ **函数重载**: 参数个数或类型不同的函数可以使用相同的函数名。

如平方函数 `square (int)` 和 `square (double)`。

两个函数必须分别定义 (函数模板 `template` 省略这个过程)

调用函数是, 编译器会根据参数的类型自动选择对应的函数

C++ 系统函数库

29

- 数学库函数 `#include <math>` // to use math library

Linux 中需要编译时加参数：

`g++ file.C -lm`

函数： `ceil(x)`, `cos(x)`, `exp(x)`, `fabs(x)`, `floor(x)`,
`fmod(x,y)`, `log(x)`, `log10(x)`, `pow(x,y)`, `sin(x)`,
`sqrt(x)`, `tan(x)` （说明见下页）

- 字符串库函数： `#include <string>`
`strlen(s)` 长度, `strcpy(s1,s2)` 复制,
`strcat(s1,s2)` 附加, `strcmp(s1,s2)` 比较, 等等

数学函数

30

Function	Meaning
<code>sin(x)</code>	Sine
<code>cos(x)</code>	Cosine
<code>tan(x)</code>	Tangent
<code>asin(x)</code>	Arc sine
<code>acos(x)</code>	Arc cosine
<code>atan(x)</code>	Arc tangent
<code>atan2(x,y)</code>	Arc tangent (x/y)
<code>exp(x)</code>	Exponential
<code>log(x)</code>	Natural logarithm
<code>log10(x)</code>	Logarithm, base 10
<code>abs(x)</code>	Absolute value
<code>sqrt(x)</code>	Square root
<code>pow(x,y)</code>	x to the power of y

作用域

31

- ✓ 标识符只能在说明它或定义它的范围内可见。
- ✓ 不同标识符定义在不同范围内有不同的作用域：
 - ✓ program scope 整个程序可见
 - ✓ file scope 只在某个C++程序文件中可见
 - ✓ block scope 只在某语句块中可见
 - ✓ function scope 只在某函数中可见
 - ✓ class scope 只在某类中可见

```
int number;  
static int value;  
void main( )  
{  
    a=1;  
    number=0;  
    value=number+1;  
}  
int a;
```

左边的程序定义了三个变量，**number**和**value**是文件作用域变量，**a**是局部作用域变量。由于变量**a**未定义就使用，因此在编译时就会出错。

I/O流控制：iomanip.h

32

- **#include <iomanip>**
- setw(*n*)设置域宽为*n*
- setfill(char *c*): 在预设宽度中空的位置用设字符*c*填充
- setprecision(*n*): 控制输出浮点数的数字个数，默认是6
- setiosflags(ios::fixed)是用定点方式表示实数；
setiosflags(ios::scientific)是用指数方式表示实数
- setprecision(*n*)与setiosflags(ios::fixed)合用可控制小数点右边的数字个数。
与setiosflags(ios::scientific)合用，可控制指数表示的小数位。

I/O流控制：iomanip.h

33

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    double amount = 22.0/7;
    cout << amount << endl;
    cout << setprecision(0) << amount << endl;
    cout << setprecision(1) << amount << endl;
    cout << setprecision(2) << amount << endl;
    cout << setprecision(3) << amount << endl;
    cout << setiosflags(ios::fixed) << amount << endl;
    cout << setprecision(8) << amount << endl;
    cout << setiosflags(ios::scientific) << amount << endl;
    cout << setprecision(6) << amount << endl;
    return 1;
}
```

3.14286
3
3
3.1
3.14
3.143
3.14285714
3.1428571
3.14286

结果

精度为x位的浮点数

用固定小数点表示的浮点数

科学表示法表示浮点数

- ```
#ifndef HEAD_H
#define HEAD_H
#include "head.h"
#endif
```

- 宏定义命令：

```
#define PI 3.1415026
```

```
#define ADD(a,b) a+b
```

# const类型声明

35

- 一个变量被声明为const类型后，就为只读，而且该变量必须在声明时初始化在任何地方的修改都将会导致编译错误。

```
const int i;
double * const pd;
```

错误！没有初始化

- 代替用#define定义的常量，因为const型的常量要灵活的多，它可以创建有类型的常量。

```
const int arraySize=6000;
float array[arraySize];
```

- C++中所有const对象的缺省存储类型为static
- const常被应用于函数的参数声明中：在函数体中不改变的参数最好设计成为const类型。

# const类型声明

36

*C++/const\_test.C:*

去掉注释语句，尝试  
编译，理解警告/出错  
输出信息。

**注意：C++ 中类  
型的说明需使  
用从里向外读**

```
char str1[5]="abcd"; char str2[5]="ABCD";
const char * p1; //正确
 p1=str1; //正确
 p1[1]='2'; //错误
char* const p2; //错误
char* const p2=str1; //正确
 p2[1]='2'; //正确
 p2=str2; //错误
const char *const p3; //错误
const char *const p3= str1; //正确
 p3[1]='2' //错误
 p3=str2; //错误
```

ref: <http://unixwiz.net/techtips/reading-cdecl.html>

# 枚举类型

37

- enum定义的枚举类型的定义：

```
enum <枚举名>{标识符1,标识符2,标识符3...}; 。
```

- 枚举实际上定义了一组常量。如：

```
enum Booler{FALSE,TRUE};
```

实际上定义了由两个整型常量组成的枚举。默认从0开始逐一递增。

等价于：`const int FALSE=0; const int TRUE=1;`

- 可在声明中显式地赋值：

```
enum TEXT_MODES {
 LASTMODE= -1, BW40=0, C40, BW80, C80, MONO=7, C4350=64
};
```

// C40、BW80、C80的值分别是1、2、3

# 动态地分配和撤销内存空间

38

- C中是利用库函数malloc和free实现的，但是使用malloc函数时必须指定需要开辟的内存空间的大小，其调用形式是malloc(size)

- C++中提供了较简便而功能较强的运算符new 和delete

来取代malloc和free函数(为和C兼容，C++仍保留该两函数)。

例如：

**new int;** 可开辟一个存放整数的空间，并返回一个指向整形数据的指针

- new运算符的一般格式为 new 类型[初值]
- delete运算符的一般格式为 delete[] 指针变量
- new申请存储空间必须用delete操作释放

# 动态分配和撤销内存空间

39

- `new int;` //开辟一个存放整数的空间，返回一个指向整形数据的指针
- `char *str = new char[10];` //开辟一存放字符数组的空间，该数组有10个元素，  
//返回一个指向字符数据的指针  
`delete[] str;`
- `int *num_p = new int[5][4];` //开辟一个存放二维整形数组的空间  
//该数组大小为5乘4  
`delete []num_p;` //删除数组空间，  
//在指针变量前加一对[]，表示对数组空间的操作
- `float *p=new float(3.14159);` //开辟一个存放实数的空间，  
//并指定该实数的初值为3.14159，将返回的指向实型数据的指针赋给指针变量p  
`delete p;` //撤销上面用new开辟的实数空间

## PART 2: 类与对象



# 类与对象的概念

41

- 客观世界中任何一个事物都可以看成一个对象，对象之间通过一定的渠道相互联系
- 从计算机角度来看，一个对象应包括两个要素：  
一是数据(即活动主体)，二是需要进行的操作。

**对象就是一个包含数据以及与这些数据有关的操作的集合。**

- 每一个实体都是对象，有一些对象具有相同的结构和特性。  
在C++中，对象的类型就称为“类”(class),即类代表了某一批对象的共性和特征。
- **类是对某一类对象的抽象，对象是某一种类的实例。**  
C++中，可先声明类类型，然后去定义若干不同类型的对象，  
即对象就是一个类类型的变量，而类类型相当于产生对象的模板

# 类与对象的概念

42

- C++语言通过类将数据结构和与之相关的操作封装起来，形成一个整体，具有良好的外部接口，防止对数据结构内部未经授权的访问，提高了程序模块之间的独立性
- C中结构struct可把相关联的数据元素组成一个单独的统一体：只有数据，没有操作  
C++的类既包含数据成员，又含有操作(称为成员函数)。

```
struct Student
{
 int num; //学号
 float sum; //总分
};
```

```
class Student
{
public:
 void set_num(int); //成员函数
 void set_sum(int); //成员函数
private:
 int num; //数据成员
 float sum;
};
```

# 限制成员函数与数据

43

- 限制类与外部访问的接口：公共的public和受限制的（private和protected）
- 只有类本身才能访问该类受限制的数据，类本身负责在内部维护
- 用户通过授权访问和修改数据成员的成员函数访问/修改数据
- 减少类与其它代码的关联程度，做到功能独立
- 类的保护机制使得程序更加可靠和易于维护
- 类定义中，不写访问控制说明符时，默认为private
- protected和private的区别，体现在类的继承中

# 类class

44

- 成员函数对数据成员的引用可在其声明之前：C++编译器先扫描类的数据成员的说明后，再处理成员函数代码体。
- 关键字public、private和protected表示存储控制类型。

private 声明的数据成员和成员函数是“私有的”，外部不能调用

public 方式声明的成员是“公有的”，外界可以调用；

protected 成员同私有成员相似，不能被外部调用，但可被派生类的成员函数调用

- 存储控制类型的声明出现的次数及顺序是任意的。

```
void abc() {
 Student a; //定义类对象
 a.sum=181.5; //错误，不能访问sum，因为它是私有成员
 a.set_sum(100); //正确，使sum赋值为100
}
```

# 类的定义格式

定义类的一般形式如下：

```
class 〈类名〉
{
public:
 //公有数据成员和成员函数
[protected:
 //保护的数据和函数]
private:
 //私有数据成员和成员函数
};
 //各个成员函数的实现
```

2. 成员函数的实现的一般形式：

```
〈返回类型〉 〈类名〉 :: 〈成员函数名〉
 (〈形参表列〉)
{
 函数体;
}
```

“::”是C++中新引入的运算符，称为“作用域运算符”，“类名::”用来标识成员函数所属的类。

由于不同的类可有相同的成员函数名，因此在定义成员函数时，必须指出类名。

# 作用域运算符 ::

46

- C++语言规定每一个变量都有其有效的作用域，只能在变量的作用域内使用该变量。例如：

```
float a=13.5;
main()
{
 int a=5;
 cout<<a<<endl;
 cout<<::a<<endl;
}
```

- 在main函数中局部变量将屏蔽全局变量，因此第一行输出是局部变量a的值5。如果要输出全局实型变量的值，则要用作用域运算符::。

# 类的数据成员和成员函数

47

- 类中的数据成员可以是任意的类型  
(整型、浮点型、字符型、数组、枚举, 指针, 引用, 类类型等)
- 类不能定义自身实例作为它的数据成员, 即不能嵌套,  
但类可以包含指向自己实例的指针或引用。
- 不能声明为auto(自动存储变量)、register(寄存器变量)或extern(外部变量)。
- 在类体内通常先说明公有成员(用户关心), 后说明私有成员
- 人们将类定义的说明部分或整个定义及实现部分放在头文件中
- 成员函数定义在类的外面时, 但分两种情况:  
使用inline的内联函数和不使用inline的外联函数

内联函数在调用时不是像一般函数那样要转去执行被调用函数的函数体,  
而是编译时用内联函数的函数体进行替换, 可提高运行效率。

# 定义对象

48

- 对象(object)是类的实例(instance)
- 对象定义的格式为：`<类名> <对象名表>;`
- <对象名表>中有多个对象时以“,”分隔：  
如 `Data d1, *pdate, data [31];     Data &d2=d1;`
- 对象的作用域和生存期：全局对象、静态对象，内部对象和堆对象
  - 运算符new可把对象建立在堆(heap)中，在程序中动态创建，当该对象不再使用时，用运算符delete可以删除它，同时释放内存。
  - 构造函数中使用了new，在析构函数中用delete释放内存



# 调用成员函数

49

- 可用对象和指向对象的指针调用成员函数

```
//Date.h : 类定义
class Date
{
public:
 void set(int,int,int);
 int getleap();
 void print();
private:
 int day, month, year;
};
```

```
//Date.C : 成员函数实现
...
void Date::set(int d,int m,int y){
 day=d;
 month=m;
 year=y;
}
int Date::getleap(){
 return(year%4==0 && year%100 !=
0)|| (year%400 == 0);
}
void Date::print(){
 cout<< month<<"/"
 <<day<<"/"<<year<<endl;
}
```

//判返回值是否为闰年

# 调用成员函数

50

```
//Class_1.C
...
void funca(Date *p){
 p->print(); // p是s对象的指针
 if(p->getleap())
 cout<<"leap year\n";
 else
 cout<<"not leap year\n";
}

int main(){
 Date s;
 s.set(7,3,2024);
 funca(&s); //对象的地址传给指针
}
```

对象和引用型：使用符号点(.)  
对象指针：右箭头(->)

输入今天的日期；

运行结果为：

3/7/2024

leap year

# 重载成员函数

51

- 同一个函数名可以对应多个函数实现。

```
class Student
{
 public:
 float grade(){...}
 float grade(float abc){...}
 private:
 ...
};
class Scope
{
 public:
 float grade(){...}
 private:
 ...
};
char grade(float abc){...}
```

```
void main()
{
 Student s;
 Scope t;
 s.grade(3.2); //Student::grade(float)
 float v=s.grade(); //Student:: grade()
 char c=grade(v); //::grade(float)
 float m=t.grade(); //Scope:: grade()
}
```

每种实现都对应一个函数体，这些函数的名字相同，但是函数的参数个数或类型不同。

# 对象初始化：构造函数

52

Date对象s初始化时，只能通过set函数来实现，比较麻烦。为了增加效率，C++引入构造函数的特殊函数

对象初始化时，只能通过成员函数对私有数据初始化。方便起见，C++规定与类同名的成员函数是构造函数(constructor)，在对象被定义时自动初始化。

- 为成员函数，函数体可在类内或类外
- 由用户创建，在对象创建时自动被调用
- 可重载，即可定义不同参数个数和类型的构造函数
- 一般声明为public，无返回值，不需加void类型声明

# 析构函数

53

- 析构函数(destructor)用来释放一个对象。在对象删除前由系统自动执行它做清理工作。如在对new分配的内存空间，在析构函数中用delete释放。
- 函数名为类名前面加“~”字符。析构函数不指定数据类型，没有参数和返回值。
- 一个类中只能定义一个析构函数，析构函数没有重载。
- 析构函数不能被显式调用。

# Hep3Vector类

54

## 基本构造函数和析构函数

```
class Hep3Vector {
 public:
 inline Hep3Vector(double x = 0.0, double y = 0.0, double z = 0.0);
 inline Hep3Vector(const Hep3Vector &);
 inline ~Hep3Vector();
 inline double x() const;
 inline double y() const;
 inline double z() const;

 private:
 double dx, dy, dz;
};
```

函数名后的const声明，表示该成员函数不修改任何数据成员

# 静态数据成员

55

- 声明为static类的成员能在类的范围内共享，称为静态成员。
- 特点：
  - (1) 节省内存。多少个类的对象，都只有一个静态数据成员的拷贝供所有对象共用
  - (2) 静态数据成员是静态存储的，必须对它进行初始化。
  - (3) 在类体外初始化，在类中不为它分配内存空间。

```
class className{
 public:
 static int abc; //声明，未分配空间
 ...
};
int className::abc = 0;
//在类外分配空间和初始化
```

```
int main()
{
 className::abc=10;
 //访问静态数据成员
};
```

# 静态成员函数

56

- 用static关键字声明的成员函数：类的对象共享
- 可以在建立任何对象之前处理静态函数成员
- 可以在类体内定义，也可以在类体外定义。
- 不直接访问类中非静态成员：通过对象或对象指针

```
class className{
 public:
 static void func(className A);
 private:
 static int abc;
 ...
};
int className::abc = 0;
void className::func(className A){....}
```

```
int main()
{
 className P , Q;
 className::func(P);
 className::func(Q);
 P.func(Q);
}
```



# 静态成员

57

- 使用静态数据成员，可以减少使用全局变量  
全局变量实质上是违背封装原则
- 静态成员：最好放在类的实现部分中定义（.C）
- 要使用静态数据成员必须在main()程序运行之前分配空间和初始化

# this指针

58

- this指针是隐含于类的成员函数中的特殊指针，指向正在被某个成员函数操作的对象。
- 当一个对象调用成员函数时，编译程序先将对象的地址赋给this指针，然后调用成员函数。
- 成员函数存取数据成员时，隐含使用this指针(不显式地使用this指针)
- 也可以使用\*this来标识调用成员函数的对象。

```
void Date::set(int d,int m,int y){
 this->day=d;
 this->month=m;
 this->year=y;
}
```

## PART 3: 继承与派生

# 继承 (inheritance)

60

- 在既有类（基类或父类）的基础上创建新类（派生类，导出类或子类）
- 从一个或多个既有类中继承所有的数据成员和函数成员，并加上自己的新成员或重新定义由继承得到的成员
- 软件重用和扩展程序的有效手段
- 格式为：

```
class 派生类名: 继承方式 基类名
{
 函数体; //派生类新增成员定义
};
```

# 继承

61

```
class A
{
private:
 int x;
public:
 void Setx(int i){x = i;}
 void Showx(){cout<<x<<endl;}
};
```

```
void main()
{
 B b;
 b.Setx(10); b.Sety(20);
 b.Showx(); b.Showy();
}
```

```
class B:public A
{
private:
 int y;
public:
 void Sety(int i){y = i;}
 void Showy() {
 Showx(); Cout<<y<<endl;}
};
```

运行结果:

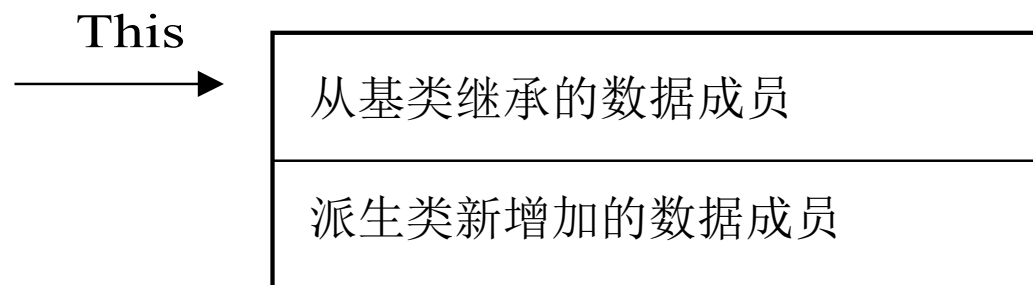
10

10

20

# 继承方式

62



- 继承方式：

- **public(公有)**：基类成员的访问属性在派生类中不变，而基类的private成员在派生类中是不可访问的。
- **protected(保护)**：基类的public(公有)和protected(保护)成员在派生类中的访问属性调整为private(私有)
- **private(私有)**：将基类的public成员和protected成员在派生类中的访问属性调整为protected (保护)

# 继承

63

| 继承方式           | 基类中的访问权限  | 派生类中的访问权限 |
|----------------|-----------|-----------|
| 公有继承 public    | public    | public    |
|                | protected | protected |
|                | private   | 不可访问      |
| 保护继承 protected | public    | protected |
|                | protected | protected |
|                | private   | 不可访问      |
| 私有继承 private   | public    | private   |
|                | protected | private   |
|                | private   | 不可访问      |

# PART4: 其他



# typedef机制

65

- 提供了一种通用的类型定义，用来为内置的或用户定义的数据类型引入助记符号（别名）：`typedef data-type new-name`

```
struct Card {int pip; char *suit;};
Typedef Card *cptr; // cptr 等价于 Card * 定义
cptr deck; // 等同于Card * deck;
```

```
//定义一个相同类型和大小的数组
typedef char Line[255]; // Line 相当于char [255] 定义
Line text, secondline; // 等价于 char text[255], secondline[255];
```

- Typedef 声明有助于代码简化，促进跨平台开发

```
typedef long double REAL; typedef double REAL; typedef float REAL;
```

**//typedef不同于#define**  
**#define**在预编译时进行简单的替换；  
**typedef**当成语句处理。**#define**没有作用域的限制，**typedef**有作用域。

```
Typedef int * pint;
#define PINT int *
const pint p; //p不可更改，p指向的内容可以更改，相当于 int * const p;
const PINT p; //p可以更改，p指向的内容不能更改，相当于 const int *p;
pint s1, s2; //s1和s2都是int型指针
PINT s3, s4; //相当于int * s3, s4; 只有一个是指针。
```

对指针的操作，作用不同

# 处理命令行选项

66

- 如果用户在命令行中指定了选项的话，我们可以通过main()函数的一种扩展原型特征来访问这些选项：

```
int main(int argc, char *argv[]) {...}
```

argc 包含命令行选项的个数； argv 包含argc个C风格字符串，

代表了由空格分隔的命令选项

例如： 命令行prog -d -o ofile data0:

argc为5; argv[0]="prog"; argv[1]="-d";

argv[2]="-o"; argv[3]="ofile"; argv[4]="data0";