面向对象程序设计(OOP)

(第2章：C++基础)

福州大学·软件学院·软件工程系
王灿辉(wangcanhui@fzu.edu.cn)

第2章：C++基础

动态初始化

- **C语言规定：函数体中的所有说明语句必须放在(可)执行语句之前。**而C++突破了这一限制，不限制说明语句和执行语句出现的顺序，允许进行动态初始化(C99也已经取消了这条限制)。实际原因在于C++引入类后必须允许执行类的构造函数(的程序行)。

第2章：C++基础

动态初始化

```
#include <iostream>
using namespace std;
int main() {  //想使用的时候才定义变量！
  double radius = 4.0, height = 5.0;
  cout << "Enter the height :";
  cin >> height;
  // dynamically initializ volume
  double volume = 3.1416 * radius * radius * height;
  cout << "Volume is " << volume;
  return 0;
} //目的是为了配合类的构造函数！
```

第2章：C++基础

动态初始化

C语言for语句的格式:

```
for (expression;condition;expression)
    statement
```


C++语言for语句的格式:

```
for (for-init-statement condition;expression)
    statement
//for-init-statement(语句)本身就会有;
```

第2章：C++基础

数据类型

- 整型[1](short int、int、long int)，实型(float、double、long double)，**布尔型(bool)**、字符型[1](char)、空类型(void)，枚举型(enum)、联合(共用体union)、结构(struct)、**类(class)**，指针(*)、**引用(&)**、数组([])。

注：[1]可以加signed,unsigned修饰符

第2章：C++基础

布尔类型

```
#include <iostream>
void main(){
   bool b=false; //bool为C++新增类型！
   std::cout << b << "\n"; //输出0
   b=7;    //警告！等价于b=true;
   std::cout<<b<<"\n"; //输出1不是true!
   b=3==4;
   std::cout << b << "\n"; //输出0
} //true和false C++新增的保留字！
```

第2章：C++基础

无名参数

```
#include <iostream>
using namespace std;
void func(int x,int) {
  cout<<x<<endl; //第2个参数无名
}
void main() {
  func(11,88); //必须指定第2个参数
}
```

第2章：C++基础

显式类型转换

- 对于内部类型而言，T(expr)等价于(T)expr，有可能出现截断或溢出(长类型转换为短类型)等，所以是不安全。

  如：int i=int(12.3); //=(int)12.3;

- 指针转换不能直接采用T(expr)的形式表示。指针类型转换实例

第2章：C++基础

循环体内声明循环控制变量

```
#include <iostream>
using namespace std;
int main() {
  int sum = 0,fact = 1;
  for(int i = 1; i <= 5; i++) {
    sum += i;  // i is known throughout the loop
    fact *= i;
  }
  // but, i is not known here(ANSI/ISO Standard C++).
  cout << "Sum is " << sum << "\n";
  cout << "Factorial is " << fact;
  return 0;
}
```

第2章：C++基础

复合语句中定义变量

```
#include <iostream>
using namespace std;
void func(int a) {
  cout << a <<endl;    //输出10
  if (a==10) {
    int a=5,i=20;
    cout << a << i << endl;  //输出520
  }
  //cout << i << endl; 错误，i无定义！
}
void main() {
  func(10);
}
```

第2章：C++基础

::限定符和变量作用域

- 作用域小的变量覆盖作用域大的变量。
- 用“::变量名”可以访问具有文件作用域的变量，但不能访问隐藏的局部变量。
- 用“::变量名”访问全局变量实例
- 变量作用域演示实例

第2章：C++基础

数组边界核查

- C++和C语言都不对数组执行边界核查。

  例如：


```
int i=0,crash[10];
for (;i<100;i++) crash[i]=i;
```

- 执行上述程序片段将造成不可预料的结果，并且错误难于定位。

第2章：C++基础

从键盘读入字符串

```
#include <iostream>
using namespace std;
int main() {
  char s[80];
  cout << "Enter a string: ";
  cin >> s; // read string from keyboard
  //应改为：cin.getline(s,80);最多输79个字符
     //否则遇到空白停止，可能输入超过80个字符
  cout << "Here is your string: " << s;
  return 0;
}
```

第2章：C++基础

const修饰符

- const修饰符在C++中很常用。
- 在C中，下述语句是错误的：

```
const int N=10;
//必须改为：#define N 10
int a[N];
```

- 但在C++中，上述语句是允许的。

第2章：C++基础

const修饰符

- const修饰符是不变的意思(在C++中很常用)。

  1、const double PI=3.14; //常量定义，默认是静态的(仅用于本文件)，比用#define PI 3.14好（有类型便于检查，可计算初值表达式）！

  2、int strcmp(const char *s1,const char *s2);

  //函数不允许修改s1和s2指向的2个字符串！

  3、const int &mnin(int &x,int &y);

  //表示不能将函数的返回值做为左值使用

  例：min(a,b)=5; //错误，无const时允许做左值

第2章：C++基础

const修饰符

- const修饰符定义“只读函数”（使用是一种好习惯，不用不会导致错误）：
  例如：int getMember() const;
  表示该函数是一个“只读函数”，即函数不能改变对象的成员变量的值；也不能调用一个非const的成员函数（否则就可能通过它间接修改数据成员）。
- **在函数原型和定义处均必须加上const。**

第2章：C++基础

const修饰符

```
#include <iostream>
int main() {
    const char c1=std::cin.get();
    const char c2=c1-32;
    //c1='a'会报错!
    std::cout<<c1<<' '<<c2<<'\n';
    return 0;
}
```

第2章：C++基础

const修饰符

- int i=1,k=2,*p; //p正常指针
- p=&i;*p=88;p=&k;//这些语句都正确
- 但p不能指向const变量ci：
  const int ci=2; //ci是常量
- p=&ci; //'=':cannot convert from 'const int *' to 'int *'
- 如果允许这样赋值的话，则将可以用形如*p=value;来修改ci的值，显然有问题！

第2章：C++基础

const修饰符

- const int *cp;//等价于:int const *cp;可认为是指向常量的指针，**不能通过该类指针来修改所指向变量的值**！
- cp=&ci; //很明显应该是允许的赋值
- cp=&i; //p可以指向其他普通变量，但无法用*cp=value;来修改i的值！
- *cp=value;//l-value specifies const object
- i=value;//是正常赋值，当然是允许的！

第2章：C++基础

const修饰符

- int * const pc; //const object must be initialized if not extern
- int * const pc=&i; //指针常量，即该指针不能被修改(不能指向其他地址)。
- *pc=8; //指向的内容是可改的
- pc=&k; //l-value specifies const object，但指针本身不可改，很少用！
- int * const pc=&ci; //cannot convert from 'const int *' to 'int

第2章：C++基础

const修饰符

- const int * const cpc1=&ci;
- const int * const cpc2=&i;
- *cpc1=value; //l-value specifies const object
- cpc1=&i; //l-value specifies const object

“演示const修饰符”的完整的程序

第2章：C++基础

其他修饰符和C含义一致

- Volatile修饰符
- 存储类说明符：

  auto（一般不会被用）

  extern

  static(全局变量、局部变量)

  register
- static 函数原型;则该函数仅用于本文件。

第2章：C++基础

枚举类型

- enum weekday {sun, mou, tue, wed, thu, fri, sat};
  //定义枚举类型名为weekday
  //枚举名在编译期间成为关键字！
- enum weekday a, b, c;
  //说明三个变量为该类型（C/C++支持）
- weekday a, b, c; //weekday也是一种类型
  //说明三个变量为该类型（仅C++支持）

第2章：C++基础

结构类型

- struct Date {int y,m,d;};
  //定义结构类型，名为Date。
- struct Date a,b,c;
  //说明三个变量为该类型（C/C++支持）
- Date a,b,c; //Date也是一种类型
  //说明三个变量为该类型（仅C++支持）

第2章：C++基础

引用(&)变量

```
#include <iostream>
int main() {
   int j=10,k=11;
   int &i=j; //independent reference
   //变量i成为变量j的一个别名
   std::cout<<j<<' '<<i<<'\n'; //输出：10 10
   std::cout<<&j<<' '<<&i<<'\n';
   //输出：0018FF44 0018FF44
   i=k; //i获得k的值或i成为k的别名？
   k=12;
   std::cout<<i<<' '<<j<<' '<<k<<'\n';
   //输出：11 11 12
}
```

第2章：C++基础

引用调用

- 用值调用方式实现两个变量的值交换？
- 用指针方式实现两个变量的值交换
- 用引用调用方式实现两个变量的值交换

（引用调用的实在参数只能是变量！）

- 注意：int &i和int& i有区别吗？

  int* a,b;的含义是什么？

第2章：C++基础

引用调用

- 函数想返回2个及以上的值的时候就要使用指针或引用来实现。
- 例如：用一个函数同时求数组的最大值和最小值。

```
void extremum(const int *a,int n,
int &max,int &min);
```

第2章：C++基础

引用调用

```
void extremum(const int *a,int n,int &max,int
   &min) {
   max=min=a[0];
   for (int i=1;i<n;i++) {
      if (max<a[i]) max=a[i];
      if (min>a[i]) min=a[i];
   }
}
int main() {
   int max,min;                          //输出：5 89
   int a[]={12,80,78,62,63,89,5,9,88,23,19};
   extremum(a,sizeof(a)/sizeof(a[0]),max,min);
   std::cout<<min<<" "<<max<<std::endl;
}
```

第2章：C++基础

独立引用&返回引用

- 非参数的独立引用很少使用，因为它们很容易混淆并会破坏程序的结构。
- 常引用(const 类型 &引用名=变量名;)实例
- 利用返回引用创建有界数组

输出结果为：

10 20 30

Bounds Error(Put) !

Bounds Error(Get) !

第2章：C++基础

返回引用变量

```
#include <iostream>
using namespace std;
int &func(void) {
  int x=2;  //局部变量
  return x; //编译可以通过，但不能使用！！
  //warning C4172: returning address of local
   variable or temporary
}
void main() {
  cout<<(func()=5)<<func()<<endl;  //输出52
}
```

第2章：C++基础

引用变量

- 引用变量定义时必须进行初始化。在程序中对引用的存取都是对它所引用的变量的存取。**也不能再指向别的变量！**
- 引用与指针不同。指针的内容或值是某一变量的内存单元地址，而引用则与初始化它的变量具有相同的内存单元地址。

第2章：C++基础

引用变量

- 返回引用必须保证返回的引用变量有合法的内存空间，并且不在函数的运行栈中。一般只能返回全局变量和静态变量。
- 使用const常引用的目的就是提高性能(不需要制作数据的副本)。使用引用参数而不加const就应该认为该参数将被修改！
- **当函数需要返回多于1个值，或希望修改实参的值时必须使用引用调用。**

第2章：C++基础

引用变量

- 如果条件允许，就应将引用形参声明为const形参，这样可以避免无意间修改数据而导致编程错误。此外使用const形参使得函数能够接受const和非const实参，否则只能处理非const数据。

第2章：C++基础

引用变量：对比与总结

- void func1(int n) {
  //可以修改n,但值不会返回调用者
  } //可用值或变量调用
- void func2(int &n) {
  //可以修改n,值会返回调用者
  } //只能用同类型的变量调用该函数
- void func3(const int &n) {
  //根本就不允许修改n
  } //可用值或变量调用

第2章：C++基础

引用变量：对比与总结

```
#include <iostream>
int f1(int n) { //2个n的地址是不一样的！
    int fac;
    for (fac=1;n>0;n--) fac*=n;
    return fac;
} //可以修改n,但值不会返回调用者
int main() {
    std::cout<<"4!="<<f1(4)<<' ';
    int n=5,fac=f1(n);
    std::cout<<n<<"!="<<fac<<std::endl;
} //可用值4或变量n调用，输出：4!=24 5!=120
```

第2章：C++基础

引用变量：对比与总结

```
#include <iostream>
int f2(int &n) {
  std::cout<<&n<<std::endl;
  int fac;
  for (fac=1;n>0;n--) fac*=n;
  return fac;
} //可修改n，修改后的值会返回调用者
//f2的n和main的n的地址一致！
```

第2章：C++基础

引用变量：对比与总结

```
#include <iostream>
int main() {
  //std::cout<<"4!="<<f2(4)<<std::endl;
  int n=5,fac=f2(n);
  std::cout<<n<<"!="<<fac<<std::endl;
  std::cout<<&n<<std::endl;
} //只能用同类型的变量调用该函数
//输出：0!=120
```

第2章：C++基础

引用变量：对比与总结

```
#include <iostream>
int f3(const int &n) {int fac,m;
   for (fac=1,m=n;m>0;m--) fac*=m;
   return fac;
} //根本就不允许修改n
int main() {
   std::cout<<"4!="<<f3(4)<<' ';
   int n=5,fac=f3(n);
   std::cout<<n<<"!="<<fac<<std::endl;
} //可用值4或变量n调用
//输出：4!=24 5!=120
```

第2章：C++基础

内联函数

```
#include <iostream>
using namespace std;
inline int add(int x,int y)
  {return x+y;}
void main()
  {cout << "3+5=" << add(3,5) << endl;}
```


- inline只是一种要求，通常应该是**比较简单**的函数。取代C的“宏定义”。
- 内联函数默认是静态的(仅用于本文件)

第2章：C++基础

内联函数

- inline(内联)函数一般直接放到头(.h)文件中。下列3种情况编译器不会将函数处理成内联函数：

  1、函数内有循环语句；

  2、函数太复杂(取决于编译器)；

  3、程序中有取该函数地址的语句。

第2章：C++基础

内联函数

- inline(内联)函数一般直接放到头(.h)文件中。因为内联函数要在调用点展开，所以编译器必须随处可见内联函数的定义，要不然就成了非内联函数的调用了。所以，这要求每个调用了内联函数的文件都出现了该内联函数的定义。

第2章：C++基础

内联函数

- inline必须与函数定义体放在一起才能使函数成为内联，仅将inline放在声明前面不起任何作用。所以说inline是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义前面都加了inline 关键字，但inline最好不出现在函数的声明中。

第2章：C++基础

内联函数

- 如下风格的函数Foo 不能成为内联函数：
- inline void Foo(int x, int y);
  //inline仅与函数声明放在一起
- void Foo(int x, int y){}
- 而如下风格的函数Foo则成为内联函数：
- void Foo(int x, int y);
- inline void Foo(int x, int y) {}
  //inline与函数定义体放在一起

第2章：C++基础

默认的函数参数

- 在C++中，可以为函数参数指定默认值，默认参数必须在函数第一次声明中进行声明，通常是在函数原型中声明
- 带默认参数的函数实例
- 输出结果：

```
x: 1, y: 2
x: 10, y: 100
x: 0, y: 100
```

带默认参数的函数实例

第2章：C++基础

函数重载

- 函数重载是C++实现多态性的一种方法
- 函数重载的实例
- 输出结果：

```
In f(int), i is 10
In f(int, int), i is 10, j is 20
In f(double), k is 12.23
```

- 每个重载函数的参数类型和（或）数量必须不同；正常执行相似的操作；返回值可以不同。

第2章：C++基础

函数重载

- 在形参和实参之间类型不能直接匹配时进行类型自动转换。
- 自动类型转换和函数重载的实例
- 输出结果：

```
Inside f(int): 10
Inside f(double): 10.1
Inside f(short): 99
Inside f(double): 11.5
```

第2章：C++基础

函数重载

- 以下实例中的函数func不是重载：

```
void func(int);
void g() {
    void func(double); //很容易引起错误！
    //……
    func(1); //调用func(double);
    //……
}
```

第2章：C++基础

函数重载

- 但下述实例的函数func是重载：

```
void func(int);
void func(double);
void g() {
  //……
  func(1); //调用func(int);
  //……
}
```

第2章：C++基础

默认的函数参数和函数重载

- 默认参数的应用之一就是函数重载的简写形式。例如：要编写strcat函数的两个版本：

```
void mystrcat(char *,char *,int);
void mystrcat(char *,char *);
```

可以使用函数重载，但用默认参数实现更简单（只要编写一个版本就可以）

- 函数mystrcat的实现（有函数原型时）
  函数mystrcat的实现（无函数原型时）

第2章：C++基础

函数重载和多义性

- 多义性：编译器不能在两个或多个正确重载的函数之间作出选择。

- 第1个实例
- 第2个实例
- 第3个实例

第2章：C++基础

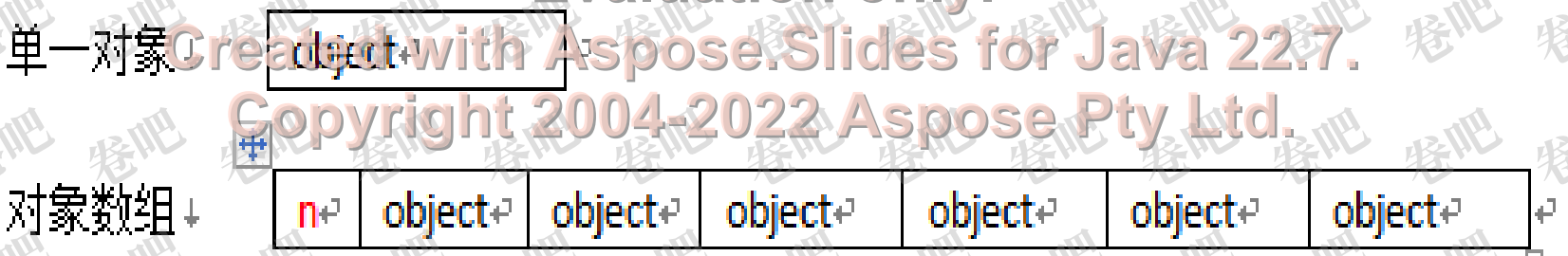
动态分配内存

- 对于动态分配内存,C使用malloc和free,C++使用new和delete,这两个操作的主要区别在于new和delete能够自动调用类里的构造函数和析构函数。定位new不做介绍(自学)。
- 如果new一个单一的对象,如int *p=new int;释放时要使用delete p格式,不能使用delete[] p;而对于new多个对象内存时,如int *p=new int [10];释放内存时要使用delete[] p格式;不能使用delete p;否则只会释放第一个int类型内存,无法释放后面9个。

第2章：C++基础

动态分配内存

- new int和new int[10]这两种方式的内存布局（大多数编译器都使用这样的内存布局）



- 当分配一个对象内存时，仅仅分配一块对象类型的内存块，当分配对象数组内存时，会在内存中添加数组大小的标识n，表示分配了多少个内存块，所以释放对象数组内存使用delete时要加[]

第2章：C++基础

动态分配内存

```
#include <iostream>
#include <new>
using namespace std;
int main() {
  int *p;
  p = new int; // allocate space for an int
  *p = 100;  //可以合并为p = new int(100);
  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";
  delete p; //用delete[] p;不会报告错误！！
  return 0;
} //输出结果：At 004206A0 is the value 100
```

第2章：C++基础

动态分配内存

```
int *ps=new int;
//……
delete ps;//
```

程序结束一般会自动delete

- //只释放ps做指向的内存，但不会释放ps本身占用的内存(将会由系统自动释放)。不能delete不是new来的内存。
- int *ps=NULL;delete ps; //对空指针使用delete是安全的。

第2章：C++基础

动态分配内存

- 下述程序编译正确，但执行时出错：

```
#include <iostream>
using namespace std;
void main() {
   int *p1=new int;
   int *p2=new int;
   delete p1;  //第1次释放p1正确！
   delete p1;  //第2次释放p1错误！
} //而p2又没有释放！
```

第2章：C++基础

动态分配内存

- 为数组动态分配内存：

```
p = new array type[size];
delete[] p; //用delete p;不报错！
```

- 例如：int * const p=new int[10];

```
//这样定义更像数组
//注意和p=new int(10);的区别
delete[] p;    //不能用delete p;
```

- 不能给动态(分配内存的)数组赋初值！

第2章：C++基础

动态分配内存

```
#include <iostream>
void main() {
  int *p1=new int(10);
  int *p2=new int[10];
  delete[] p1; //不会报告错误！
  delete   p2; //不会报告错误！
  std::cout<<"程序正常结束！\n";
}//但导致的结果是不确定的！
```

第2章：C++基础

动态分配内存

- 像C一样，C++不区分指向单个对象的指针和指向某个数组起始元素的指针，因此程序员必须明确说明要删除的是单个对象还是数组。
- 混用delete在delete[]在某些编译器上可能获得警告，但更多的是在运行时造成极其危险的后果。
- new和delete最好放在同一个函数。

第2章：C++基础

动态分配内存

```
#include <iostream>
using namespace std;
void main() {
   int i=0;
   int *p=&i;
   cout<<*p<<endl;
   delete p;
}//程序运行出现什么结果？
```

第2章：C++基础

动态分配内存

```
#include <stdio.h>
const int N=10;
int main() {
    int i,*p;
    p=new int[N];
    for (i=0;i<N;i++) *p++=i;
    p-=N;
    for (i=0;i<N;i++) printf("[%d]",*p++);
    p-=N;
    //如无该语句或改为p-=N-1;等均会导致运行时错
    delete[] p;
    //改为delete p;运行正常，但会导致内存泄漏。
}
```

第2章：C++基础

动态分配内存：二维数组

```
#include <iostream>
using namespace std;
void main() {
   int (*a)[5];
   a=new int[3][5]; //二维数组a[3][5]
   //……其他代码
   delete[] a;
} //为二维数组动态申请内存！
```

第2章：C++基础

动态分配内存：二维数组

```
#include <iostream>
using namespace std;
void main() {
 int i,**a;
  a=new int *[3];
  for (i=0;i<3;++i) a[i]=new int[5];
  //……其他代码
  for (i=0;i<3;++i) delete[] a[i];
  delete[] a;
} //为二维数组a[3][5]动态申请内存!
```

第2章：C++基础

处理指针参数

```
#include <iostream>
using namespace std;
void func(int *a) {
  a=new int[2];  a[0]=7;
  cout<<"[func]"<<a<<':'<<a[0]<<endl;
}
void main() {
  int *a=new int[4];  a[0]=5;
  cout<<"[main]"<<a<<':'<<a[0]<<endl;
  func(a);
  cout<<"[main]"<<a<<':'<<a[0]<<endl;//输出?
}
```

第2章：C++基础

动态分配内存

- 自定义的对象必须用delete释放，否则将导致运行时错误。


```
int main() {
   cout<<"Begin..."<<endl;
   Circle *c=new Circle;
   delete c;
   //delete[] c; 该语句将导致运行时错误!
   cout<<"End..."<<endl;
   return 0;
}
```

第2章：C++基础

动态分配内存

- 自定义的对象数组必须用delete[]释放，否则将导致运行时错误。


```
int main() {
   cout<<"Begin..."<<endl;
   Circle *r=new Circle[10];
   delete[] r;
   //delete r; 该语句将导致运行时错误！
   cout<<"End..."<<endl;
   return 0;
}                                    //完整的程序
```

本章内容讲授到此结束！

福州大学·软件学院·软件工程系
王灿辉(wangcanhui@fzu.edu.cn)