第三章 C++中的C

本章目标

- C++是以C为基础的，所以要用C++编程就必须熟悉掌握C的语法。
- 由于C++中的很多语法与Java的语法接近，我们将着重介绍有区别的部分。

程序设计语言

- 一套好的符号系统能够把大脑从所有非必要的工作中解脱出来，集中精力去对付更高级的问题。
- 在引入阿拉伯数字之前计算乘法是困难的，除法(即使只是整数除法)更需要发挥全部的数学才能。
- 在当代社会里，最让一位希腊数学家吃惊的或许是绝大多数西欧人都能完成大整数的除法。

怀特海

程序设计语言

- 结论
  - 编程语言影响程序员的思维
- 高级语言的语句与等效的C代码语句行数之比：

语言	相对于C语言的等级
C	1
C++	2.5
Fortran 95	2
Java	2.5
Perl	6

主要内容

- 函数
- 控制语句
- 运算符
- 数据类型
- 指针/引用*
- 变量
- 运算符
- 类型转换
- 结构*
- 联合(union)*

指针

- 程序中的变量都至少有两个属性：值和存放该值的地址。
- &运算符
  - 取地址操作符
- *运算符
  - 取值操作符

指针示例

```
//YourPets2.cpp

int dog, cat, bird, fish;
void f(int pet) {
     cout << "pet id number: " << pet << endl;
}
int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "dog: " << (long)&dog << endl;
  cout << "cat: " << (long)&cat << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

```
f(): 4199374
dog: 4485128
cat: 4485132
bird: 4485136
fish: 4485140
i: 2293540
j: 2293536
k: 2293532
```

指针声明及引用

int* p;

……

*p = 100;

```
//指针和数组，指针的运算
int main() {

  int *p;

  p = new int[1];

  *p = 0;

  cout << *p << endl;

}
```

指针的运算

```
//MyTest4.cpp
int dog=1, cat=2, bir=3, fis=4;

int main() {
    int i=0, j=1, k=2;
    int *p;

    p = &dog;
    cout << *(p+1) << endl;

    p = &k;
    cout << *(p+1) << endl;

} ///:~
```

指针的常见错误

- 1) 内存分配未成功，却使用了它。
  - 常用解决办法是，在使用内存之前检查指针是否为NULL。
  - 如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。
  - 如果是用malloc或new来申请内存，应该用 if(p==NULL) 或if(p!=NULL)进行防错处理。
- 2) 内存分配虽然成功，但是尚未初始化就引用它。

指针的常见错误

- 3) 内存分配成功并且已经初始化，但操作越过了内存的边界。
  - 例如在使用数组时经常发生下标“多1”或者“少1”的操作。
- 4) 忘记了释放内存，造成内存泄露。
  - 含有这种错误的函数每被调用一次就丢失一块内存直至内存耗尽。动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误（new/delete同理）。

指针的常见错误

- 5) 释放了内存却继续使用它。
  - （a）程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
  - （b）使用free或delete释放了内存后，没有将指针设置为NULL，从而导致错误
    - 用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。

Java中的指针(引用)

- Java中没有指针的概念，但实际上，除了基本数据类型外，Java中所有对象的声明都是引用，而引用实质上就是一个指向内存中某个地址的指针。
- 指针和引用的最大区别是，引用不能够像指针一样运算，也就是说他不能够随意指向任意的内存地址。

函数指针

- 一旦函数被编译并载入计算机中运行，它就会占用一块内存。这块内存有一个地址，因此函数也有地址。
- 下面代码声明了一个指向无参无返回值的函数：
  - void (*funcPtr)();

使用函数指针

- 函数指针在使用前必须给赋一个函数地址。

```
void func() {
  cout << "func() called..." << endl;
}

 void (*fp)();  // Define a function pointer

int main() {
   fp = func;  // Initialize it
    (*fp)();    // calls the function
    void (*fp2)() = func;  // Define and initialize
    (*fp2)();
} ///:~
```

Java

- Java中没有与函数指针对应的语法结构。
  - Java中可以独立存在的语言元素只有类（类型）

指向函数指针的指针数组(1)

- 一个更为有趣的结构是指向函数的指针数组(FunctionTable.c)：

```
// A macro to define dummy functions:
#define DF(N) void N() { \
cout << "function " #N " called..." << endl; }

DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);

void (*func_table[])() = { a, b, c, d, e, f, g };
```

指向函数指针的指针数组(2)

```
int main() {
  while(1) {
      cout << "press a key from 'a' to 'g' "
           "or q to quit" << endl;
      char c, cr;
      cin.get(c); cin.get(cr); // second one for CR
      if ( c == 'q' )
          break; // ... out of while(1)
      if ( c < 'a' || c > 'g' )
          continue;
      (*func_table[c - 'a'])();
  }
} ///:~
```

JAVA基本数据类型作为参数传递

```
public class test {
    public static void main(String[] args) {
        int i = 1;
        System.out.println("before change, i = "+i);
        change(i);
        System.out.println("after change, i = "+i);
    }
    public static void change(int i){
        i = 5;
    }
}
```

```
before change, i = 1
after change, i = 1
```

JAVA对象作为参数传递

```
public class test {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello ");
        System.out.println("before change, sb is "+sb.toString());
        change(sb);
        System.out.println("after change, sb is "+sb.toString());
    }
    public static void change(StringBuffer stringBuffer){
        stringBuffer.append("world !");
    }
}
```

```
before change, sb is Hello
after change, sb is Hello world !
```

修改外部对象（参数）

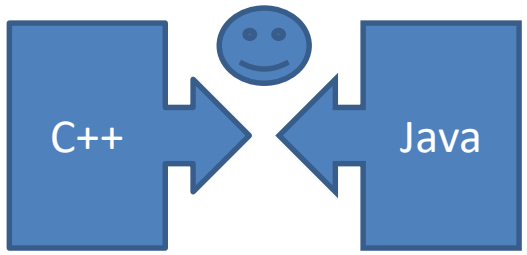
- 有三种传递参数的方式
  - 传值
  - 传指针
  - 传引用

传值(1)

```
//PassByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

传值(2)

```
//PassByValueObject.cpp
class Test
{
	public:
		int a;
		int b;
};
void func(Test test){
		test.a = 2;
		cout << test.a << endl;
}
int main() {
		Test test;
		test.a = 1;
		cout << test.a << endl;
		func(test);
		cout << test.a << endl;
} ///:~
```

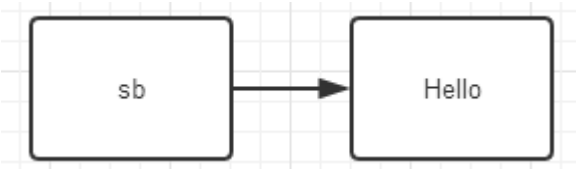


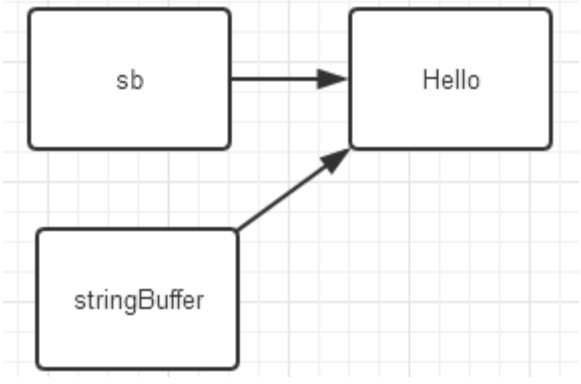
传指针

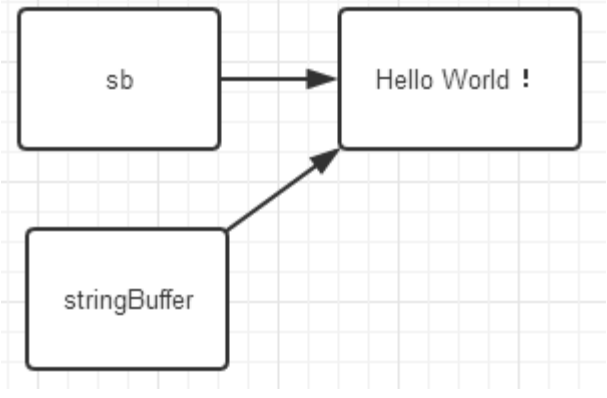
```
//PassByPointer.cpp
void f(int* p) {
   cout << "p = " << p << endl;
   cout << "*p = " << *p << endl;
   *p = 5;
   cout << "p = " << p << endl;
}

int main() {
   int x = 47;
   cout << "x = " << x << endl;
   cout << "&x = " << &x << endl;
   f(&x);
   cout << "x = " << x << endl;
} ///:~
```

传引用

```
void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x); // Looks like pass-by-value, is actually pass by reference
  cout << "x = " << x << endl;
} ///:~
```


sb
Hello


sb
Hello
stringBuffer

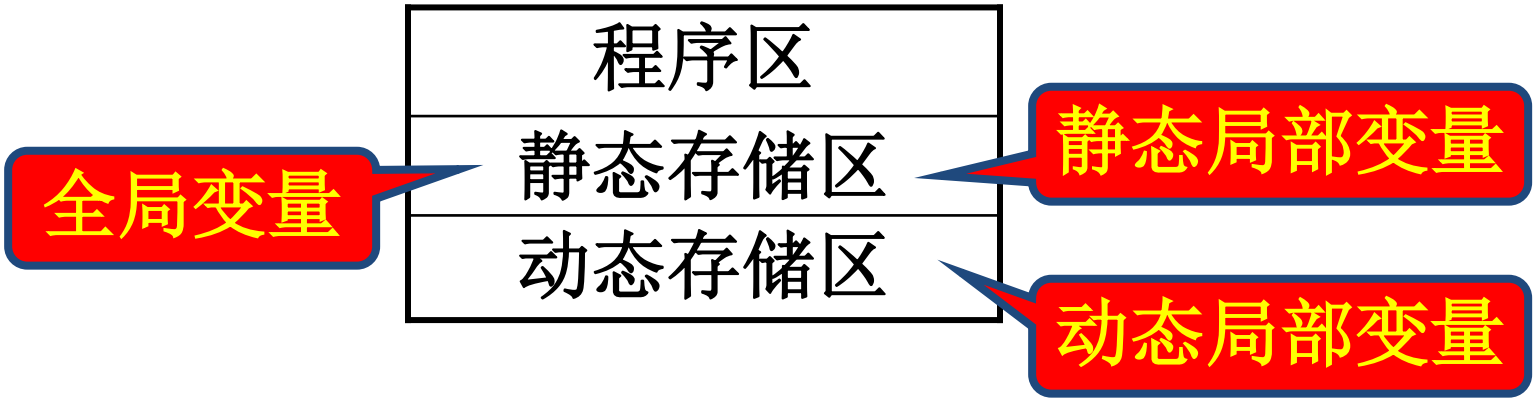

sb
Hello World !
stringBuffer

为变量指定存储空间

- 全局变量
  - 在函数外定义的变量
  - 全局变量的作用域称为文件作用域，即在整个文件中都是可以访问的。缺省的范围是:从定义全局变量的位置开始到该源程序文件结束。
- 局部变量
  - 当在函数作用域内的变量与全局变量同名时，局部变量优先

为变量指定存储空间

- 静态存储
  - 在文件运行期间有固定的存储空间，直到文件运行结束。
- 动态存储
  - 在程序运行期间根据需要分配存储空间，函数结束后立即释放空间。若一个函数在程序中被调用两次，则每次分配的单元有可能不同。



typedef

- 用于给类型定义别名（而不是定义新的类型）
- 例如：

  typedef int* IntPtr;

  IntPtr x,y;
- typedef大量用在结构(struct)的定义中

struct

- struct是把一组变量组合成一个新的数据类型的一种方式。
- struct与class的区别
  - 默认继承权限。如果不明确指定，来自class的继承按照private继承处理，来自struct的继承按照public继承处理；
  - 成员的默认访问权限。class的成员默认是private权限，struct默认是public权限。

指针和结构

- 和其他数据类型一样，可以定义指向结构的指针。
- 使用->符号可以访问结构中的元素。

```
typedef struct Structure3 {
  char c;
  int i;
  float f;
  double d;
} Structure3;
```

指针和结构

```
int main() {
  Structure3 s1, s2;
  Structure3* sp = &s1;
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  sp = &s2; // Point to a different struct object
  sp->c = 'a';
  sp->i = 1;
  sp->f = 3.14;
  sp->d = 0.00093;
  (*sp).d = 0.2;
} ///:~
```

union

- 有时一个程序会使用同一个变量处理不同的数据类型，使用union把所有的数据装进一个单独的空间内，编译器计算出union中最大项所必须的空间。
- 使用union的目的是节省内存

union

```
union Packed { // Declaration similar to a class
    char i;
    short j;
    int k;
    long l;
    float f;
    double d;
    // The union will be the size of a
    // double, since that's the largest element
};  // Semicolon ends a union, like a struct
```

union

```
int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```