C++引用10分钟入门教程

< 上一页 下一页 >

我们知道，参数的传递本质上是一次赋值的过程，赋值就是对内存进行拷贝。所谓内存拷贝，是指将一块内存上的数据复制到另一块内存上。

对于像 char、bool、int、float 等基本类型的数据，它们占用的内存往往只有几个字节，对它们进行内存拷贝非常快速。而数组、结构体、对象是一系列数据的集合，数据的数量没有限制，可能很少，也可能成千上万，对它们进行频繁的内存拷贝可能会消耗很多时间，拖慢程序的执行效率。

C/C++ 禁止在函数调用时直接传递数组的内容，而是强制传递数组指针，这点已在《C语言指针变量作为函数参数》中进行了讲解。而对于结构体和对象没有这种限制，调用函数时既可以传递指针，也可以直接传递内容；为了提高效率，我曾建议传递指针，这样做在大部分情况下并没有什么不妥，读者可以点击《C语言结构体指针》进行回顾。

但是在 C++ 中，我们有了一种比指针更加便捷的传递聚合类型数据的方式，那就是**引用（Reference）**。

> 在 C/C++ 中，我们将 char、int、float 等由语言本身支持的类型称为基本类型，将数组、结构体、类（对象）等由基本类型组合而成的类型称为聚合类型（在讲解结构体时也曾使用复杂类型、构造类型这两种说法）。

引用（Reference）是 C++ 相对于C语言的又一个扩充。引用可以看做是数据的一个别名，通过这个别名和原来的名字都能够找到这份数据。引用类似于 Windows 中的快捷方式，一个可执行程序可以有多个快捷方式，通过这些快捷方式和可执行程序本身都能够运行程序；引用还类似于人的绰号（笔名），使用绰号（笔名）和本名都能表示一个人。

引用的定义方式类似于指针，只是用 `&` 取代了 `*`，语法格式为：

```
type &name = data;
```

type 是被引用的数据的类型，name 是引用的名称，data 是被引用的数据。引用必须在定义的同时初始化，并且以后也要从一而终，不能再引用其它数据，这有点类似于常量（const 变量）。

下面是一个演示引用的实例：

```
01.  #include <iostream>
02.  using namespace std;
03.
04.  int main() {
05.      int a = 99;
06.      int &r = a;
07.      cout << a << ", " << r << endl;
08.      cout << &a << ", " << &r << endl;
09.
10.      return 0;
11.  }
```

运行结果：
99, 99
0x28ff44, 0x28ff44

本例中，变量 r 就是变量 a 的引用，它们用来指代同一份数据；也可以说变量 r 是变量 a 的另一个名字。从输出结果可以看出，a 和 r 的地址一样，都是 `0x28ff44`；或者说地址为 `0x28ff44` 的内存有两个名字，a 和 r，想要访问该内存上的数据时，使用哪个名字都行。

注意，引用在定义时需要添加 `&`，在使用时不能添加 `&`，使用时添加 `&` 表示取地址。如上面代码所示，第 6 行中的 `&` 表示引用，第 8 行中的 `&` 表示取地址。除了这两种用法，`&` 还可以表示位运算中的与运算。

由于引用 r 和原始变量 a 都是指向同一地址，所以通过引用也可以修改原始变量中所存储的数据，请看下面的例子：

```
01.  #include <iostream>
02.  using namespace std;
03.
04.  int main() {
05.      int a = 99;
06.      int &r = a;
07.      r = 47;
08.      cout << a << ", " << r << endl;
09.
10.      return 0;
11.  }
```

运行结果：
47, 47

最终程序输出两个 47，可见原始变量 a 的值已经被引用变量 r 所修改。

如果读者不希望通过引用来修改原始的数据，那么可以在定义时添加 const 限制，形式为：

```
const type &name = value;
```

也可以是：

```
type const &name = value;
```

这种引用方式为常引用

C++引用作为函数参数

在定义或声明函数时，我们可以将函数的形参指定为引用的形式，这样在调用函数时就会将实参和形参绑定在一起，让它们都指代同一份数据。如此一来，如果在函数体中修改了形参的数据，那么实参的数据也会被修改，从而拥有“在函数内部影响函数外部数据”的效果。

至于实参和形参是如何绑定的，我们将在下节《C++引用在本质上是什么，它和指针到底有什么区别？》中讲解，届时我们会一针见血地阐明引用的本质。

一个能够展现按引用传参的优势的例子就是交换两个数的值，请看下面的代码：

```
01.  #include <iostream>
02.  using namespace std;
03.
04.  void swap1(int a, int b);
05.  void swap2(int *p1, int *p2);
06.  void swap3(int &r1, int &r2);
07.
08.
09.  int main() {
10.      int num1, num2;
11.      cout << "Input two integers: ";
12.      cin >> num1 >> num2;
13.      swap1(num1, num2);
14.      cout << num1 << " " << num2 << endl;
15.
16.      cout << "Input two integers: ";
17.      cin >> num1 >> num2;
18.      swap2(&num1, &num2);
19.      cout << num1 << " " << num2 << endl;
20.
21.      cout << "Input two integers: ";
22.      cin >> num1 >> num2;
23.      swap3(num1, num2);
24.      cout << num1 << " " << num2 << endl;
25.
```

```
    return 0;
}

//直接传递参数内容
void swap1(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

//传递指针
void swap2(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

//按引用传参
void swap3(int &r1, int &r2) {
    int temp = r1;
    r1 = r2;
    r2 = temp;
}
```

运行结果：

```
Input two integers: 12 34↙
12 34
Input two integers: 88 99↙
99 88
Input two integers: 100 200↙
200 100
```

本例演示了三种交换变量的值的方法：

1) swap1() 直接传递参数的内容，不能达到交换两个数的值的目的。对于 swap1() 来说，a、b 是形参，是作用范围仅限于函数内部的局部变量，它们有自己独立的内存，和 num1、num2 指代的数据不一样。调用函数时分别将 num1、num2 的值传递给 a、b，此后 num1、num2 和 a、b 再无任何关系，在 swap1() 内部修改 a、b 的值不会影响函数外部的 num1、num2，更不会改变 num1、num2 的值。

2) swap2() 传递的是指针，能够达到交换两个数的值的目的。调用函数时，分别将 num1、num2 的指针传递给 p1、p2，此后 p1、p2 指向 a、b 所代表的数据，在函数内部可以通过指针间接地修改 a、b 的值。我们在《C语言指针变量作为函数参数》中也对比过第 1)、2) 中方式的区别。

2) swap3() 是按引用传递，能够达到交换两个数的值的目的。调用函数时，分别将 r1、r2 绑定到

num1、num2 所指代的数据，此后 r1 和 num1、r2 和 num2 就都代表同一份数据了，通过 r1 修改数据后会影响 num1，通过 r2 修改数据后也会影响 num2。

从以上代码的编写中可以发现，按引用传参在使用形式上比指针更加直观。在以后的 C++ 编程中，我鼓励读者大量使用引用，它一般可以代替指针（当然指针在C++中也不可或缺），C++ 标准库也是这样做的。

C++引用作为函数返回值

引用除了可以作为函数形参，还可以作为函数返回值，请看下面的例子：

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

int &plus10(int &r) {
    r += 10;
    return r;
}

int main() {
    int num1 = 10;
    int num2 = plus10(num1);
    cout << num1 << " " << num2 << endl;

    return 0;
}
```

运行结果：
20 20

在将引用作为函数返回值时应该注意一个小问题，就是不能返回局部数据（例如局部变量、局部对象、局部数组等）的引用，因为当函数调用完成后局部数据就会被销毁，有可能在下次使用时数据就不存在了，C++ 编译器检测到该行为时也会给出警告。

更改上面的例子，让 plus10() 返回一个局部数据的引用：

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

int &plus10(int &r) {
    int m = r + 10;
    return m;  //返回局部数据的引用
}

int main() {
    int num1 = 10;
```

```
    int num2 = plus10(num1);
    cout << num2 << endl;
    int &num3 = plus10(num1);
    int &num4 = plus10(num3);
    cout << num3 << " " << num4 << endl;

    return 0;
}
```

在 Visual Studio 下的运行结果：

```
20
-858993450 -858993450
```

在 GCC 下的运行结果：

```
20
30 30
```

在 C-Free 下的运行结果：

```
20
30 0
```

而我们期望的运行结果是：

```
20
20 30
```

plus10() 返回一个对局部变量 m 的引用，这是导致运行结果非常怪异的根源，因为函数是在栈上运行的，并且运行结束后会放弃对所有局部数据的管理权，后面的函数调用会覆盖前面函数的局部数据。本例中，第二次调用 plus10() 会覆盖第一次调用 plus10() 所产生的局部数据，第三次调用 plus10() 会覆盖第二次调用 plus10() 所产生的局部数据。

关于函数调用的内部实现，我已在《C语言内存精讲》专题中讲到。

< 上一页 下一页 >

所有教程

C语言入门 C语言编译器 C语言项目案例 数据结构 多线程 链接库

C++ STL C++11 socket GCC GDB Makefile OpenCV

Qt教程 Unity 3D UE4 游戏引擎 Python Python并发编程
TensorFlow Django NumPy Linux Shell Java教程 设计模式
Java Swing Servlet教程 JSP教程 Struts2 Maven Nexus Spring
Spring MVC Spring Boot Spring Cloud Hibernate Mybatis
MySQL教程 MySQL函数 NoSQL Redis常用命令手册 HBase MongoDB
Go语言 C# MATLAB JavaScript Bootstrap HTML CSS PHP
汇编语言 TCP/IP vi命令 Android教程 区块链 Docker 大数据
云计算 推荐阅读 编程笔记 资源下载 VIP视频 一对一答疑 关于我们