C 语言函数指针教程

作者：Lars Haendel
2005 年 1 月，德国波鸿
http://www.newty.de/
email: 请到网站查看

Contents

1 函数指针介绍

函数指针是非常高效有趣和优雅的编程技巧。你可以用它来替代 switch/if 语句，来实现你的后期绑定和函数回调。但是，由于它的语法的复杂程度，它在很多计算机书籍和文档中不受待见。即使提到了，也都是很浅的一笔带过。因为你不需要为它分配和收回内存，它造成错误的概率要比普通指针低很多。所有你需要做的就是搞清楚它的意义和相关的语法。但是要搞清楚：你是否真的需要一个函数指针来完成任务。实现自己的后期绑定很漂亮，但如果使用 C++ 现有的结构和方法或许会让你的代码可读性更高。后期绑定的应用场景在运行时，如果你调用了一个普通函数，你的代码需要知道调用哪个函数。它用一个包含所有可能被调用的函数的 V-Table。每次调用将会消耗一些性能，你可以用函数指针来代替普通函数来解决这个问题，也许不需要[1]

1.1 什么是函数指针

当你在程序中一个叫做标记的点调用一个函数 *DoIt()* 的时候，你只需要把函数的调用放到程序源码的标记处，之后你每次编译代码到这个位置，这个方法将被执行，一切都很正常。但是如果你不想在编译的时候确定哪个函数应该被调用需要怎么做？你可能希望在程序运行的时候再决定什么函数应该被调用。或者你希望使用回调函数或在一个函数池中选择一个合适的来执行。当然，可以使用 *switch* 语句来实现后一种效果，在不同的条件分支中调用你想用的函数。但是还有其它的方法可以实现这个目标：使用函数指针！在下面的例子中，我们将实现一个加减乘除的 4 种数学运算。首先来用一个 *switch 语法*来解决这个问题，然后再用函数指针的方式实现同样的目标[2]。

```
//---------------------------------------------
// 1.2 例子：如何取代一个switch-case
// 任务：用＋、－、、×/ 实现一个数学运算



// 四种运算符 ... 下面的一个函数将被使用
// 在运行时靠一个来选择或使用函数指针switch
float Plus    (float a, float b) { return a+b; }
```

[1]现代的编译器已经非常出色。用我的 Borland，我可以节省。。。
[2]这是个非常简单的例子，所以应该不会有人来用函数指针来实际解决这个问题

```
float Minus   (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide  (float a, float b) { return a/b; }


// 的解决方法switch - 运算符<> 将决定执行哪个操作
void Switch(float a, float b, char opCode)
{
   float result;

   // 执行操作
   switch(opCode)
   {
      case '+' : result = Plus     (a, b); break;
      case '-' : result = Minus    (a, b); break;
      case '*' : result = Multiply (a, b); break;
      case '/' : result = Divide   (a, b); break;
   }

   cout << "Switch: 2+5=" << result << endl;          // display result
}


// 使用函数指针的解决方法 - <pt2Func> 是一个指向接受个浮点数为参数，返回一个浮点数2
// 的函数。这个指针函数将决定哪个操作应该被执行
void Switch_With_Function_Pointer(float a, float b, float (*pt2Func)(float, float))
{
   float result = pt2Func(a, b);     // call using function pointer

   cout << "Switch replaced by function pointer: 2-5=";  // display result
   cout << result << endl;
}


// 执行示例代码
void Replace_A_Switch()
{
   cout << endl << "Executing function 'Replace_A_Switch'" << endl;

   Switch(2, 5, /* '+' specifies function 'Plus' to be executed */ '+');
   Switch_With_Function_Pointer(2, 5, /* pointer to function 'Minus' */ &Minus);
}
```

重要提示：每个函数指针总是指向特定特征的函数！对于一个函数指针，所有使用它的函数必须有相同类型的参数和返回值！

2 函数指针的 C 和 C++ 的语法

根据这 2 个语言的语法，有 2 种类型的函数指针：一种是指向普通的 C 语言的函数，或指向 C++ 的静态成员函数。另外一种是指向 C++ 的非静态成员函数。基本的区别是所有指向非静态成员函数的需要一个隐藏的参数：一个指向某个实例的 *this* 指针[3]。一定要搞清楚，这两种类型的指针是不相容的。

2.1 定义一个函数指针

一个函数指针只不过是一个变量，它必须像其它变量一样先声明。在下面的例子里我们声明 2 个函数指针，分别是 *pt2Function*, *pt2Member* 和 *pt2ConstMember*。他们指向函数，

[3]原文：this-pointer

这个函数以一个浮点数和 2 个字符作为参数并返回一个整数。在 C++ 例子中我们假设我们指向的函数是类 *TMyClass* 中的非静态成员。

```
// 2.1 定义一个函数指针并初始化为NULL
int (*pt2Function)(float, char, char) = NULL;                          // C
int (TMyClass::*pt2Member)(float, char, char) = NULL;                  // C++
int (TMyClass::*pt2ConstMember)(float, char, char) const = NULL;       // C++
```

2.2 调用约定

通常你不需要关心函数的调用约定：如果你没有特别的指定另外的约定，编译器会将 *__cdecl* 作为默认约定。如果你相了解更多，继续往下读。调用约定会告诉编译器怎样传递参数和怎样产生一个函数的名字。其它约定的例子有 *__stdcall*, *__pascal*, *__fastcall*。调用约定属于一个函数的特征：不同调用约定的函数和函数指针互相间是不兼容的！对于 Borland 和微软的编译器，你指定一个特别的调用约定在返回类型和函数或函数指针名之间。对于 GNU GCC 来说，使用 *__attribute__* 关键字：定义函数时要接关键字 *__attribute__* 并用双大括号描述。[4]

```
// 2.2 define the calling convention
void __cdecl DoIt(float a, char b, char c);                            // Borland and Microsoft
void         DoIt(float a, char b, char c)  __attribute__((cdecl));    // GNU GCC
```

2.3 给函数指针分配一个地址

给函数指针分配一个地址是非常简单的。你只需要选择一个合适的名字并且知道函数或成员函数的名字。对于大部分编译器来说地址符 & 是可选的。你需要知道函数的名称和类名，并且你可以访问函数内部。

```
// 2.3 assign an address to the function pointer
//     Note: Although you may ommit the address operator on most compilers
//     you should always use the correct way in order to write portable code.

// C
int DoIt  (float a, char b, char c){ printf("DoIt\n");   return a+b+c; }
int DoMore(float a, char b, char c)const{ printf("DoMore\n"); return a-b+c; }

pt2Function = DoIt;      // short form
pt2Function = &DoMore;   // correct assignment using address operator


// C++
class TMyClass
{
public:
   int DoIt(float a, char b, char c){ cout << "TMyClass::DoIt"<< endl; return a+b+c;};
   int DoMore(float a, char b, char c) const
         { cout << "TMyClass::DoMore" << endl; return a-b+c; };

   /* more of TMyClass */
};

pt2ConstMember = &TMyClass::DoMore; // correct assignment using address operator
pt2Member = &TMyClass::DoIt; // note: <pt2Member> may also legally point to &DoMore
```

[4]如果你想了解更多：告诉我。如果你想知道函数调用在钩子下是如何工作的，你可以看 Paul Carter 的 PC 汇编教程的子程序（Subprograms）那个章节。

2.4 比较函数指针

你可以像往常一样使用比较符（==, !=）。下面的例子将会查检 *pt2Function* 和 *pt2ConstMember* 是否包含 *DoIt* 和 *TMyClass::DoMore* 的地址。如果相等会输出一段文字。

```
// 2.4 comparing function pointers

// C
if(pt2Function >0){                          // check if initialized
   if(pt2Function == &DoIt)
      printf("Pointer points to DoIt\n"); }
else
   printf("Pointer not initialized!!\n");


// C++
if(pt2ConstMember == &TMyClass::DoMore)
   cout << "Pointer points to TMyClass::DoMore" << endl;
```

2.5 用函数指针调用函数

在 C 语言中可以通过 * 号和一个函数指针来调用函数。你也可以直接使用函数指针来替代函数的名字。在 C++ 中的 2 种操作符 “.*”，“->*” 在类的实例上来调用一个非静态成员方法。如果调用发生在另一个成员函数内部你还需要使用 *this*。

```
// 2.5 calling a function using a function pointer
int result1 = pt2Function    (12, ' ' a, ' ' b);
int result2 = (*pt2Function) (12, ' ' a, ' ' b);
TMyClass instance1;
int result3 = (instance1.*pt2Member)(12, ' ' a, ' ' b);
int result4 = (*this.*pt2Member)(12, ' ' a, ' ' b);
// C short way
// C
// C++
// C++ if this-pointer can be used
TMyClass* instance2 = new TMyClass;
int result4 = (instance2->*pt2Member)(12, ' ' a, ' ' b);  // C++, instance2 is a pointer
delete instance2;
```

2.6 如果把函数指针作为一个函数参数

你可以把函数指针作为一个函数参数来使用。比如，你可能需要把函数指针当作回调函数来使用。下面代码将示例如何把函数指针传入另一个函数，并将一个 float 和 2 个 char 作为参数，一个 int 作为返回值：

```
//------------------------------------------------------------------------------
// 2.6 How to Pass a Function Pointer
// <pt2Func> is a pointer to a function which returns an int and takes a float and two char
void PassPtr(int (*pt2Func)(float, char, char))
{
       int result = (*pt2Func)(12, ' ' a, ' ' b);     // call using function pointer
          cout << result << endl;
}
// execute example code - ' ' DoIt is a suitable function like defined above in 2.1-4
void Pass_A_Function_Pointer()
{
       cout << endl << "Executing ' ' Pass_A_Function_Pointer" << endl;
          PassPtr(&DoIt);
}
```

2.7 怎样返回一个函数指针

函数指针同样可以作为函数的返回值。下面的例子有 2 种方式演示如何返回函数指针。第一个返回的函数指针需要 2 个 float 参数，并返回一个 float。如果你想返回一个指向成员函数的指针，你需要修改所有函数指针的定义声明。

```
//------------------------------------------------------------------------------
// 2.7 How to Return a Function Pointer
// ' '      Plus and ' ' Minus are defined above. They return a float and take two float
// Direct solution: Function takes a char and returns a pointer to a
// function which is taking two floats and returns a float. <opCode>
// specifies which function to return
float (*GetPtr1(const char opCode))(float, float){
   if(opCode == ' ' +)
      return &Plus;
   else
      return &Minus;} // default if invalid operator was passed
// Solution using a typedef: Define a pointer to a function which is taking
// two floats and returns a float
typedef float(*pt2Func)(float, float);
// Function takes a char and returns a function pointer which is defined
// with the typedef above. <opCode> specifies which function to return
pt2Func GetPtr2(const char opCode)
{
   if(opCode == ' ' +)
      return &Plus;
   else
      return &Minus; // default if invalid operator was passed
}
// Execute example code
void Return_A_Function_Pointer()
{
   cout << endl << "Executing ' ' Return_A_Function_Pointer" << endl;
   // define a function pointer and initialize it to NULL
   float (*pt2Function)(float, float) = NULL;
   pt2Function=GetPtr1' ' (+);   // get function pointer from function ' ' GetPtr1
   cout << (*pt2Function)(2, 4) << endl;   // call function using the pointer
   pt2Function=GetPtr2' ' (-);   // get function pointer from function ' ' GetPtr2
   cout << (*pt2Function)(2, 4) << endl;   // call function using the pointer
}
```

2.8 怎样使用一个数组的函数指针

操作一个数组的函数指针非常有意思。它使你可以使用数组的索引来使用函数。语法看上去比较复杂，经常会引起混乱。下面你将看到 2 种方式来定义和使用一个数组的函数，分别由 C 和 C++ 来实现。第一种方式使用 *typedef* 来实现，第二种方式直接定义数组。你可以根据自己的喜好来选择实现方式。

```
//------------------------------------------------------------------------------
// 2.8 How to Use Arrays of Function Pointers
// C ---------------------------------------------------------------------------
// type-definition: ' ' pt2Function now can be used as type
typedef int (*pt2Function)(float, char, char);
// illustrate how to work with an array of function pointers
void Array_Of_Function_Pointers()
{
   printf("\nExecuting ' ' Array_Of_Function_Pointers\n");
   // define arrays and ini each element to NULL, <funcArr1> and <funcArr2> are arrays
   // with 10 pointers to functions which return an int and take a float and two char
   // first way using the typedef
   pt2Function funcArr1[10] = {NULL};
```

```
   // 2nd way directly defining the array
   int (*funcArr2[10])(float, char, char) = {NULL};
   // assign the ' functions address - ' ' DoIt and ' ' DoMore are suitable functions
   // like defined above in 2.1-4
   funcArr1[0] = funcArr2[1] = &DoIt;
   funcArr1[1] = funcArr2[0] = &DoMore;
   /* more assignments */
   // calling a function using an index to address the function pointer
   printf("%d\n", funcArr1[1](12, ' ' a, ' ' b));         //  short form
   printf("%d\n", (*funcArr1[0])(12, ' ' a, ' ' b));      // "correct" way of calling
   printf("%d\n", (*funcArr2[1])(56, ' ' a, ' ' b));
   printf("%d\n", (*funcArr2[0])(34, ' ' a, ' ' b));
}
// C++ ----------------------------------------------------------------------------
// type-definition: ' ' pt2Member now can be used as type
typedef int (TMyClass::*pt2Member)(float, char, char);
// illustrate how to work with an array of member function pointers
void Array_Of_Member_Function_Pointers()
{
   cout << endl << "Executing ' ' Array_Of_Member_Function_Pointers" << endl;
   // define arrays and ini each element to NULL, <funcArr1> and <funcArr2> are
   // arrays with 10 pointers to member functions which return an int and take
   // a float and two char
   // first way using the typedef
   pt2Member funcArr1[10] = {NULL};
   // 2nd way of directly defining the array
   int (TMyClass::*funcArr2[10])(float, char, char) = {NULL};
   // assign the ' functions address - ' ' DoIt and ' ' DoMore are suitable member
   //  functions of class TMyClass like defined above in 2.1-4
   funcArr1[0] = funcArr2[1] = &TMyClass::DoIt;
   funcArr1[1] = funcArr2[0] = &TMyClass::DoMore;
   /* more assignments */
   // calling a function using an index to address the member function pointer
   // note: an instance of TMyClass is needed to call the member functions
   TMyClass instance;
   cout << (instance.*funcArr1[1])(12, ' ' a, ' ' b) << endl;
   cout << (instance.*funcArr1[0])(12, ' ' a, ' ' b) << endl;
   cout << (instance.*funcArr2[1])(34, ' ' a, ' ' b) << endl;
   cout << (instance.*funcArr2[0])(89, ' ' a, ' ' b) << endl;
}
```