我们经常在 **C++**设计时通过使用回调函数可以使有些应用（如定时器事件回调处理、用回调函数记录某操作进度等）变得非常方便和符合逻辑，那么它的内在机制如何呢，怎么定义呢**?**它和其它函数（比如钩子函数）有何不同呢？

使用回调函数实际上就是在调用某个函数（通常是 **API** 函数）时，将自己的一个函数（这个函数为回调函数）的地址作为参数传递给那个函数。

而那个函数在需要的时候，利用传递的地址调用回调函数，这时你可以利用这个机会在回调函数中处理消息或完成一定的操作。至于如何定义回调函数，跟具体使用的 **API** 函数有关，一般在帮助中有说明回调函数的参数和返回值等。**C++** 中一般要求在回调函数前加 **CALLBACK**（相当于 **FAR PASCAL**），这主要是说明该函数的调用方式。

至于钩子函数，只是回调函数的一个特例。习惯上把与 **SetWindowsHookEx** 函数一起使用的回调函数称为钩子函数。也有人把利用 **VirtualQueryEx** 安装的函数称为钩子函数，不过这种叫法不太流行。

也可以这样，更容易理解：回调函数就好像是一个中断处理函数，系统在符合你设定的条件时自动调用。为此，你需要做三件事：

1. 声明；

2. 定义；

3. 设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为一个参数，以便于系统调用。

声明和定义时应注意：回调函数由系统调用，所以可以认为它属于 **WINDOWS** 系统，不要把它当作你的某个类的成员函数。

二，回调函数、消息和事件例程

调用**(calling)**机制从汇编时代起已经大量使用：准备一段现成的代码，调用者可以随时跳转至此段代码的起始地址，执行完后再返回跳转时的后续地址。**CPU** 为此准备了现成的调用指令，调用时可以压栈保护现场，调用结束后从堆栈中弹出现场地址，以便自动返回。借堆栈保护现场真是一项绝妙的发明，它使调用者和被调者可以互不相识，于是才有了后来的函数和构件。

此调用机制并非完美。回调函数就是一例。函数之类本是为调用者准备的美餐，其烹制者应对食客了如指掌，但实情并非如此。例如，写一个快速排序函数供他人调用，其中必包含比较大小。麻烦来了：此时并不知要比较的是何类数据**--**整数、浮点数、字符串？于是只好为每类数据制作一个不同的排序函数。更通行的办法是在函数参数中列一个回调函数地址，并通知调用者：君需自己准备一个比较函数，其中包含两个指针类参数，函数要比较此二指针所指数据之大小，并由函数返回值说明比较结果。排序函数借此调用者提供的函数来比较大小，借指针传递参数，可以全然不管所比较的数据类型。被调用者回头调用调用者的函数（够咬嘴的），故称其为回调（**callback**）。

回调函数使程序结构乱了许多。**Windows API** 函数集中有不少回调函数，尽管有详尽说明，仍使初学者一头雾水。恐怕这也是无奈之举。

无论何种事物，能以树形结构单向描述毕竟让人舒服些。如果某家族中孙辈又是某祖辈的祖辈，恐怕无人能理清其中的头绪。但数据处理之复杂往往需要构成网状结构，非简单的客户/服务器关系能穷尽。

Windows 系统还包含着另一种更为广泛的回调机制，即消息机制。消息本是 **Windows** 的基本控制手段，乍看与函数调用无关，其实是一种变相的函数调用。发送消息的目的是通知收方运行一段预先准备好的代码，相当于调用一个函数。消息所附带的 **WParam** 和 **LParam** 相当于函数的参数，只不过比普通参数更通用一些。应用程序可以主动发送消息，更多情况下是坐等 **Windows** 发送消息。一旦消息进入所属消息队列，便检感兴趣的那些，跳转去执行相应的消息处理代码。操作系统本是为应用程序服务，由应用程序来调用。而应用程序一旦启动，却要反过来等待操作系统的调用。这分明也是一种回调，或者说是一种广义回调。其实，应用程序之间也可以形成这种回调。假如进程 **B** 收到进程 **A** 发来的消息，启动了一段代码，其中又向进程 **A** 发送消息，这就形成了回调。这种回调比较隐蔽，弄不好会搞成递归调用，若缺少终止条件，将会循环不已，直至把程序搞垮。若是故意编写成此递归调用，并设好终止条件，倒是很有意思。但这种程序结构太隐蔽，除非十分必要，还是不用为好。

利用消息也可以构成狭义回调。上面所举排序函数一例，可以把回调函数地址换成窗口 **handle**。如此，当需要比较数据大小时，不是去调用回调函数，而是借 **API** 函数 **SendMessage** 向指定窗口发送消息。收到消息方负责比较数据大小，把比较结果通过消息本身的返回值传给消息发送方。所实现的功能与回调函数并无不同。当然，此例中改为消息纯属画蛇添脚，反倒把程序搞得很慢。但其他情况下并非总是如此，特别是需要异步调用时，发送消息是一种不错的选择。假如回调函数中包含文件处理之类的低速处理，调用方等不得，需要把同步调用改为异步调用，去启动一个单独的线程，然后马上执行后续代码，其余的事让线程慢慢去做。一个替代办法是借 **API** 函数 **PostMessage** 发送一个异步消息，然后立即执行后续代码。这要比自己搞个线程省事许多，而且更安全。

如今我们是活在一个 **object** 时代。只要与编程有关，无论何事都离不开 **object**。但 **object** 并未消除回调，反而把它发扬光大，弄得到处都是，只不过大都以事件（**event**）的身份出现，镶嵌在某个结构之中，显得更正统，更容易被人接受。应用程序要使用某个构件，总要先弄清构件的属性、方法和事件，然后给构件属性赋值，在适当的时候调用适当的构件方法，还要给事件编写处理例程，以备构件代码来调用。何谓事件？它不过是一个指向事件例程的地址，与回调函数地址没什么区别。

不过，此种回调方式比传统回调函数要高明许多。首先，它把让人不太舒服的回调函数变成一种自然而然的处理例程，使编程者顿觉气顺。再者，地址是一个危险的东西，用好了可使程序加速，用不好处处是陷阱，程序随时都会崩溃。现代编程方式总是想法把地址隐藏起来（隐藏比较彻底的如 **VB** 和 **Java**），其代价是降低了程序效率。事件例程（？）使编程者无需直接操作地址，但并不会使程序减速。
（例程似乎是进程的台湾翻译。）

三，精妙比喻:

回调函数还真有点像您随身带的 **BP** 机：告诉别人号码，在它有事情时 **Call** 您。

回调用于层间协作，上层将本层函数安装在下层，这个函数就是回调，而下层在一定条件下触发回调，例如作为一个驱动，是一个底层，他在收到一个数据时，除了完成本层的处理工作外，还将进行回调，将这个数据交给上层应用层来做进一步处理，这在分层的数据通信中很普遍。其实回调和 **API** 非常接近，他们的共性都是跨层调用的函数。但区别是 **API** 是低层提供给高层的调用，一般这个函数对高层都是已知的；而回调正好相反，他是高层提供给底层的调用，对于低层他是未知的，必须由高层进行安装，这个安装函数其实就是一个低层提供的 **API**，安装后低层不知道这个回调的名字，但它通过一个函数指针来保存这个回调，在需要调用时，只需引用这个函数指针和相关的参数指针。 其实：回调就是该函数写在高层，低层通过一个函数指针保存这个函数，在某个事件的触发下，低层通过该函数指针调用高层那个函数。

四 无题

软件模块之间总是存在着一定的接口，从调用方式上，可以把他们分为三类：同步调用、回调和异步调用。同步调用是一种阻塞式调用，调用方要等待对方执行完毕才返回，它是一种单向调用；回调是一种双向调用模式，也就是说，被调用方在接口被调用时也会调用对方的接口；异步调用是一种类似消息或事件的机制，不过它的调用方向刚好相反，接口的服务在收到某种讯息或发生某种事件时，会主动通知客户方（即调用客户方的接口）。回调和异步调用的关系非常紧密，通常我们使用回调来实现异步消息的注册，通过异步调用来实现消息的通知。同步调用是三者当中最简单的，而回调又常常是异步调用的基础。

对于不同类型的语言（如结构化语言和对象语言）、平台（**Win32**、**JDK**）或构架（**CORBA**、**DCOM**、**WebService**），客户和服务的交互除了同步方式以外，都需要具备一定的异步通知机制，让服务方（或接口提供方）在某些情况下能够主动通知客户，而回调是实现异步的一个最简捷的途径。

对于一般的结构化语言，可以通过回调函数来实现回调。回调函数也是一个函数或过程，不过它是一个由调用方自己实现，供被调用方使用的特殊函数。

在面向对象的语言中，回调则是通过接口或抽象类来实现的，我们把实现这种接口的类成为回调类，回调类的对象成为回调对象。对于象 **C++**或 **Object Pascal** 这些兼容了过程特性的对象语言，不仅提供了回调对象、回调方法等特性，也能兼容过程语言的回调函数机制。

Windows 平台的消息机制也可以看作是回调的一种应用，我们通过系统提供的接口注册消息处理函数（即回调函数），从而实现接收、处理消息的目的。由于 **Windows** 平台的 **API** 是用 **C** 语言来构建的，我们可以认为它也是回调函数的一个特例。

对于分布式组件代理体系 **CORBA**，异步处理有多种方式，如回调、事件服务、通知服务等。事件服务和通知服务是 **CORBA** 用来处理异步消息的标准服务，

他们主要负责消息的处理、派发、维护等工作。对一些简单的异步处理过程，我们可以通过回调机制来实现。

下面我们集中比较具有代表性的语言（**C**、**Object Pascal**）和架构（**CORBA**）来分析回调的实现方式、具体作用等。

四 常见编程语言的 callback 分析

1 N/A

2 过程语言中的回调（C）

2.1 函数指针

回调在 **C** 语言中是通过函数指针来实现的**,**通过将回调函数的地址传给被调函数从而实现回调。因此，要实现回调，必须首先定义函数指针，请看下面的例子：

```
void Func(char *s)；// 函数原型
void (*pFunc) (char *);//函数指针
```

可以看出，函数的定义和函数指针的定义非常类似。

一般的化，为了简化函数指针类型的变量定义，提高程序的可读性，我们需要把函数指针类型自定义一下。

```
typedef void(*pcb)(char *);
```

回调函数可以象普通函数一样被程序调用，但是只有它被当作参数传递给被调函数时才能称作回调函数。

被调函数的例子：

```
void GetCallBack(pcb callback)
{
/*do something*/
}
```

用户在调用上面的函数时，需要自己实现一个 **pcb** 类型的回调函数：

```
void fCallback(char *s)
{
/* do something */
}
```

然后，就可以直接把 **fCallback** 当作一个变量传递给 **GetCallBack,**

GetCallBack（**fCallback**）**;**

如果赋了不同的值给该参数，那么调用者将调用不同地址的函数。赋值可以发生在运行时，这样使你能实现动态绑定。

2.2 参数传递规则

到目前为止，我们只讨论了函数指针及回调而没有去注意 **ANSI C/C++**的编译器规范。许多编译器有几种调用规范。如在 **Visual C++**中，可以在函数类型前加**_cdecl**，**_stdcall** 或者**_pascal** 来表示其调用规范（默认为**_cdecl**）。

C++ Builder 也支持**_fastcall** 调用规范。调用规范影响编译器产生的给定函数名，参数传递的顺序（从右到左或从左到右），堆栈清理责任（调用者或者被调用者）以及参数传递机制（堆栈，**CPU** 寄存器等）。

将调用规范看成是函数类型的一部分是很重要的；不能用不兼容的调用规范将地址赋值给函数指针。例如：

```
// 被调用函数是以 int 为参数，以 int 为返回值
__stdcall int callee(int);

// 调用函数以函数指针为参数
void caller( __cdecl int(*ptr)(int));

// 在 p 中企图存储被调用函数地址的非法操作
__cdecl int(*p)(int) = callee; // 出错
```

指针 **p** 和 **callee()**的类型不兼容，因为它们有不同的调用规范。因此不能将被调用者的地址赋值给指针 **p**，尽管两者有相同的返回值和参数列

2.3 应用举例

C 语言的标准库函数中很多地方就采用了回调函数来让用户定制处理过程。如常用的快速排序函数、二分搜索函数等。

快速排序函数原型：

```
void qsort(void *base, size_t nelem, size_t width, int
(_USERENTRY *fcmp)(const void *, const void *));
```

二分搜索函数原型：

```
void *bsearch(const void *key, const void *base, size_t nelem,
size_t width, int (_USERENTRY *fcmp)(const void *, const void
*));
```

其中 **fcmp** 就是一个回调函数的变量。

下面给出一个具体的例子：

```
#include <stdio.h>
#include <stdlib.h>

int sort_function( const void *a, const void *b);
int list[5] = { 54, 21, 11, 67, 22 };

int main(void)
{
int x;

qsort((void *)list, 5, sizeof(list[0]), sort_function);
for (x = 0; x < 5; x++)
printf("%i\n", list[x]);
return 0;
}
```

```
int sort_function( const void *a, const void *b)
{
return *(int*)a-*(int*)b;
}
```

2.4 面向对象语言中的回调（Delphi）

Dephi 与 C++一样，为了保持与过程语言 Pascal 的兼容性，它在引入面向对象机制的同时，保留了以前的结构化特性。因此，对回调的实现，也有两种截然不同的模式，一种是结构化的函数回调模式，一种是面向对象的接口模式。

附录博文

简介

对于很多初学者来说，往往觉得回调函数很神秘，很想知道回调函数的工作原理。本文将要解释什么是回调函数、它们有什么好处、为什么要使用它们等等问题，在开始之前，假设你已经熟知了函数指针。

什么是回调函数？

简而言之，回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数。

为什么要使用回调函数？

因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为 int）的被调用函数。

如果想知道回调函数在实际中有什么作用，先假设有这样一种情况，我们要编写一个库，它提供了某些排序算法的实现，如冒泡排序、快速排序、shell 排序、shake 排序等等，但为使库更加通用，不想在函数中嵌入排序逻辑，而让使用者来实现相应的逻辑；或者，想让库可用于多种数据类型（int、float、string），此时，该怎么办呢？可以使用函数指针，并进行回调。

回调可用于通知机制，例如，有时要在程序中设置一个计时器，每到一定时间，程序会得到相应的通知，但通知机制的实现者对我们的程序一无所知。而此时，就需有一个特定原型的函数指针，用这个指针来进行回调，来通知我们的程序事件已经发生。实际上，SetTimer() API 使用了一个回调函数来通知计时器，而且，万一没有提供回调函数，它还会把一个消息发往程序的消息队列。

另一个使用回调机制的 API 函数是 EnumWindow()，它枚举屏幕上所有的顶层窗口，为每个窗口调用一个程序提供的函数，并传递窗口的处理程序。如果被调用者返回一个值，就继续进行迭代，否则，退出。EnumWindow()并不关心被调用者在何处，也不关心被调用者用它传递的处理程序做了什么，它只关心返回值，因为基于返回值，它将继续执行或退出。

不管怎么说，回调函数是继续自 C 语言的，因而，在 C++中，应只在与 C 代码建立接口，

或与已有的回调接口打交道时，才使用回调函数。除了上述情况，在 **C++**中应使用虚拟方法或函数符（**functor**），而不是回调函数。

一个简单的回调函数实现

下面创建了一个 **sort.dll** 的动态链接库，它导出了一个名为 **CompareFunction** 的类型 **--typedef int (__stdcall *CompareFunction)(const byte*, const byte*)**，它就是回调函数的类型。另外，它也导出了两个方法：**Bubblesort()**和 **Quicksort()**，这两个方法原型相同，但实现了不同的排序算法。

```
void DLLDIR __stdcall Bubblesort(byte* array,int size,int
elem_size,CompareFunction cmpFunc);

void DLLDIR __stdcall Quicksort(byte* array,int size,int
elem_size,CompareFunction cmpFunc);
```

这两个函数接受以下参数：

·**byte * array**：指向元素数组的指针（任意类型）。

·**int size**：数组中元素的个数。

·**int elem_size**：数组中一个元素的大小，以字节为单位。

·**CompareFunction cmpFunc**：带有上述原型的指向回调函数的指针。

这两个函数的会对数组进行某种排序，但每次都需决定两个元素哪个排在前面，而函数中有一个回调函数，其地址是作为一个参数传递进来的。对编写者来说，不必介意函数在何处实现，或它怎样被实现的，所需在意的只是两个用于比较的元素的地址，并返回以下的某个值（库的编写者和使用者都必须遵守这个约定）：

·**-1**：如果第一个元素较小，那它在已排序好的数组中，应该排在第二个元素前面。

·**0**：如果两个元素相等，那么它们的相对位置并不重要，在已排序好的数组中，谁在前面都无所谓。

·**1**：如果第一个元素较大，那在已排序好的数组中，它应该排第二个元素后面。

基于以上约定，函数 **Bubblesort()**的实现如下，**Quicksort()**就稍微复杂一点：

```
void DLLDIR __stdcall Bubblesort(byte* array,int size,int
elem_size,CompareFunction cmpFunc)
{
  for(int i=0; i < size; i++)
```

```
  {
    for(int j=0; j < size-1; j++)
    {
      //回调比较函数
      if(1 == (*cmpFunc)(array+j*elem_size,array+(j+1)*elem_size))
      {
        //两个相比较的元素相交换
        byte* temp = new byte[elem_size];
        memcpy(temp, array+j*elem_size, elem_size);
        memcpy(array+j*elem_size,array+(j+1)*elem_size,elem_size);
        memcpy(array+(j+1)*elem_size, temp, elem_size);
        delete [] temp;
      }
    }
  }
}
```

注意：因为实现中使用了 **memcpy()**，所以函数在使用的数据类型方面，会有所局限。

对使用者来说，必须有一个回调函数，其地址要传递给 **Bubblesort()**函数。下面有二个简单的示例，一个比较两个整数，而另一个比较两个字符串：

```
int __stdcall CompareInts(const byte* velem1, const byte* velem2)
{
  int elem1 = *(int*)velem1;
  int elem2 = *(int*)velem2;

  if(elem1 < elem2)
    return -1;
  if(elem1 > elem2)
    return 1;

  return 0;
}

int __stdcall CompareStrings(const byte* velem1, const byte* velem2)
{
  const char* elem1 = (char*)velem1;
  const char* elem2 = (char*)velem2;
  return strcmp(elem1, elem2);
}
```

下面另有一个程序，用于测试以上所有的代码，它传递了一个有 **5** 个元素的数组给 **Bubblesort()**和 **Quicksort()**，同时还传递了一个指向回调函数的指针。

```
int main(int argc, char* argv[])
{
  int i;
  int array[] = {5432, 4321, 3210, 2109, 1098};

  cout << "Before sorting ints with Bubblesort\n";
  for(i=0; i < 5; i++)
    cout << array[i] << '\n';

  Bubblesort((byte*)array, 5, sizeof(array[0]), &CompareInts);

  cout << "After the sorting\n";
  for(i=0; i < 5; i++)
    cout << array[i] << '\n';

  const char str[5][10] = {"estella","danielle","crissy","bo","angie"};

  cout << "Before sorting strings with Quicksort\n";
  for(i=0; i < 5; i++)
    cout << str[i] << '\n';

  Quicksort((byte*)str, 5, 10, &CompareStrings);

  cout << "After the sorting\n";
  for(i=0; i < 5; i++)
    cout << str[i] << '\n';

  return 0;
}
```

如果想进行降序排序（大元素在先），就只需修改回调函数的代码，或使用另一个回调函数，这样编程起来灵活性就比较大了。

调用约定

上面的代码中，可在函数原型中找到**__stdcall**，因为它以双下划线打头，所以它是一个特定于编译器的扩展，说到底也就是微软的实现。任何支持开发基于 **Win32** 的程序都必须支持这个扩展或其等价物。以**__stdcall** 标识的函数使用了标准调用约定，为什么叫标准约定呢，因为所有的 **Win32 API**（除了个别接受可变参数的除外）都使用它。标准调用约定的函数在它们返回到调用者之前，都会从堆栈中移除掉参数，这也是 **Pascal** 的标准约定。但在 **C/C++**中，调用约定是调用者负责清理堆栈，而不是被调用函数；为强制函数使用 **C/C++**调用约定，可使用**__cdecl**。另外，可变参数函数也使用 **C/C++**调用约定。

Windows 操作系统采用了标准调用约定（**Pascal** 约定），因为其可减小代码的体积。这点对早期的 **Windows** 来说非常重要，因为那时它运行在只有 **640KB** 内存的电脑上。

如果你不喜欢**__stdcall**，还可以使用 **CALLBACK** 宏，它定义在 **windef.h** 中：

```
#define CALLBACK __stdcallor

#define CALLBACK PASCAL //而 PASCAL 在此被#defined 成__stdcall
```

作为回调函数的 C++方法

因为平时很可能会使用到 **C++**编写代码，也许会想到把回调函数写成类中的一个方法，但先来看看以下的代码：

```
class CCallbackTester
{
  public:
  int CALLBACK CompareInts(const byte* velem1, const byte* velem2);
};

Bubblesort((byte*)array, 5, sizeof(array[0]),
&CCallbackTester::CompareInts);
```

如果使用微软的编译器，将会得到下面这个编译错误：

```
error C2664: 'Bubblesort' : cannot convert parameter 4 from 'int (__stdcall
CCallbackTester::*)(const unsigned char *,const unsigned char *)' to 'int (__stdcall
*)(const unsigned char *,const unsigned char *)' There is no context in which this
conversion is possible
```

这是因为非静态成员函数有一个额外的参数：**this** 指针，这将迫使你在成员函数前面加上 **static**。当然，还有几种方法可以解决这个问题，但限于篇幅，就不再论述了。