void指针是什么?

void指针一般被称为通用指针或泛指针，它是C关于“纯粹地址(raw address)”的一种约定。void指针指向某个对象，但该对象不属于任何类型。请看下例：

int \*ip;

void \*p;

在上例中，ip指向一个整型值，而p指向的对象不属于任何类型。

在C中，任何时候你都可以用其它类型的指针来代替void指针(在C++中同样可以)，或者用void指针来代替其它类型的指针(在C++中需要进行强制转换)，并且不需要进行强制转换。例如，你可以把char \*类型的指针传递给需要void指针的函数。

什么时候使用void指针?

当进行纯粹的内存操作时，或者传递一个指向未定类型的指针时，可以使用void指针。void指针也常常用作函数指针。

有些C代码只进行纯粹的内存操作。在较早版本的C中，这一点是通过字符指针(char \*)实现的，但是这容易产生混淆，因为人们不容易判断一个字符指针究竟是指向一个字符串，还是指向一个字符数组，或者仅仅是指向内存中的某个地址。

例如，strcpy()函数将一个字符串拷贝到另一个字符串中，strncpy()函数将一个字符串中的部分内容拷贝到另一个字符串中：

char \*strepy(char'strl，const char \*str2)；

char \*strncpy(char \*strl，const char \*str2，size\_t n)；

memcpy()函数将内存中的数据从一个位置拷贝到另一个位置：

void \*memcpy(void \*addrl，void \*addr2，size\_t n)；

memcpy()函数使用了void指针，以说明该函数只进行纯粹的内存拷贝，包括NULL字符(零字节)在内的任何内容都将被拷贝。请看下例：

#include "thingie.h" /\* defines struct thingie \*/

struct thingie \*p\_src，\*p\_dest；

/\* ... \*/

memcpy(p\_dest，p\_src，sizeof(struct thingie) \* numThingies)；

在上例中，memcpy()函数要拷贝的是存放在structthingie结构体中的某种对象op\_dest和p\_src都是指向structthingie结构体的指针，memcpy()函数将把从p\_src指向的位置开始的sizeof(stuctthingie) \*numThingies个字节的内容拷贝到从p\_dest指向的位置开始的一块内存区域中。对memcpy()函数来说，p\_dest和p\_src都仅仅是指向内存中的某个地址的指针。

# [如何在C语言中实现简单的泛型编程(一）](http://blog.csdn.net/skyline0623/article/details/6164011)

原文：http://blog.csdn.net/skyline0623/article/details/6164011

            鄙人最近在家里听斯坦福大学的开放课程——《编程范式Programming Paradigms》。

附上veryCD的资源下载地址：

<http://www.verycd.com/topics/2838268/>

        这个是英文授课，现在还是没有中文或英文字幕的。就课程来说非常不错。其中Lecture 4中讲到了一种在C语言中不用C++中的template实现泛型编程的方法。我在这里总结了它的笔记后，加入了我自己的一些思考和试验写下了这篇文章分享一些经验给大家。

# 正题

        泛型编程让你编写完全一般化并可重复使用的算法，其效率与针对某特定数据类型而设计的算法相同。所谓泛型(Genericity)，是指具有在多种数据类型上皆可操作的含意。C++通过参数化类型来实现通用的容器。如Java则引入了单根继承的概念。比泛型更加让你熟悉的可能就是STL，Standard template library,标准模板库。STL是一种高效、泛型、可交互操作的软件组件。STL以迭代器 (Iterators)和容器(Containers)为基础，是一种泛型算法(Generic Algorithms)库，容器的存在使这些算法有东西可以操作。STL包含各种泛型算法(algorithms)、泛型指针（iterators）、泛型容器(containers)以及函数对象(function objects)。STL并非只是一些有用组件的集合，它是描述软件组件抽象需求条件的一个正规而有条理的架构。

        而在C语言中，同样也可以通过一些手段实现这样的泛型编程。如通过宏。这里讲的是另一种方法——通过无类型指针void\*。

        看下面的一个实现交换两个元素内容的函数swap，以整型int为例。

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **void** swap(**int**\* i1,**int**\* i2){
2. **int** temp;
3. temp=\*i1;
4. \*i1=\*i2;
5. \*i2=temp;
6. }

当你想交换两个char类型时，你还得重写一个参数类型为char的函数，你会多么想念C++中使用模板。等一下，是不是能用无类型的指针来作为参数呢？看如下改动：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **void** swap(**void** \*vp1,**void** \*vp2){
2. **void** temp=\*vp1;
3. \*vp1=\*vp2;
4. \*vp2=temp;
5. }

这段代码是错误的，是通不过编译的。首先，变量是不能声明为void无类型的。而你不知道调用此函数传进的参数是什么类型的，无法确定一种类型的声明。同时，不能将\*用在无类型指针上，因为系统没有此地址指向对象大小的信息。在编译阶段，编译器无法得知传入此函数参数的类型的。这里要想实现泛型的函数，需要在调用的地方传入相关要交换的对象的地址空间大小size，同时利用在头文件string.h中定义的memcpy（）函数来实现。改动如下：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **void** swap(**void** \*vp1,**void** \*vp2,**int** size){
2. **char** buffer[size];/\*注意此处gcc编译器是允许这样声明的，而我印象中VC编译器是不可以的。\*/
3. memcpy(buffer,vp1,size);
4. memcpy(vp1,vp2,size);
5. memcpy(vp2,buffer,size);
6. }

函数memcpy()

原型：void\* memcpy(void \*dest,void \*src,unsigned int n)

作用：由src所指内存区域复制n个字节到dest所指内存区域。

说明：src和dest所指内存区域不能重叠，函数返回指向dest的指针。

之所以将buffer声明为char类型，只是因为char类型的大小是1字节，这样声明可以为buffer指向的空间分配size字节大小的空间。

在调用这个函数时，可以像如下这样调用（同样适用于其它类型的x、y）：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **int** x=27,y=2;
2. swap(&x,&y,**sizeof**(**int**));

其实，你会发现，这样的调用也是没有报错的：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **int** i=44;
2. **short** s=5;
3. swap(&i,&s,**sizeof**(**short**));

两个不同类型的变量同样也可以实现交换。而根据系统的不同，即是小端法还是大端法保存数据，显示的结果是不同的。以Windows或linux下更常见的小端法为例，函数swap将整型i的低16位和短整型的16位进行了交换。

**这样的调用是可行的，也提醒我们C语言下这样实现泛型编程是不安全的，编译器只进行有限的检查，更多的需要程序员的细心。**如下的调用是可以的，但却毫无意义：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **int** i=22;
2. **short** s=11;
3. swap(&i,&s,8);

调用此函数，将交换i的地址向后8字节的内容与s的地址后8字节的内容的交换，而对于大小为2字节的short类型的s，大小为4字节的int类型的i这样的操作一般情况下是没有意义的。在编程时也要注意这个。C语言为我们提供了许多接近机器底层的操作，但也是一把**双刃剑**。

下面看这样的调用：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **char** \*husband=strdup("Fred");
2. **char** \*wife=strdup("Linda");
3. swap(&husband,&wife,**sizeof**(**char**\*));

函数strdup():原型：char\* strdup（char\* s）。作用是复制字符串。

husband与wife是两个字符串，在交换这样的类型时，不能直接将两个字符串的地址传入swap函数，因为两个字符串的大小是不一定的，进行复制肯定会出现错误。这里是将保存两个字符串首地址的指针的地址进行交换，使两个指针指向对方的字符串，实现字符串的交换。sizeof()中写成char\*\*也是可以的，写成double\*也是可以的，因为所有指针的大小都是4字节。但写成char\*，更加明确，在别人读代码会引起更少的误解，因为husband与wife是char类型的一级指针。

下面看另一种功能的函数：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **int** lsearch(**int** key,**int** array[],**int** size){
2. **for**(**int** i=0;i<size;++i)
3. **if**(array[i]==key)
4. **return** i;
5. **return** -1;
6. }

此函数在数组array中查找key元素，找到后返回它的索引，找不到返回-1.如上，也可以实现泛型的函数：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **void**\* lsearch(**void**\* key,**void** \*base,**int** n,**int** elemSize){
2. **for**(**int** i=0;i<n;++i){
3. **void** \*elemAddr=(**char** \*)base+i\*elemSize;/\*重点，认真看这行代码\*/
4. **if**(memcmp(key,elemAddr,elemSize)==0)
5. **return** elemAddr;
6. **return** NULL;
7. }
8. }

定义函数：int memcmp (const void \*s1, const void \*s2, size\_t n);

用法：#include <string.h>或#include<memory.h>

函数说明：memcmp()用来比较s1 和s2 所指的内存区间前n 个字符。

字符串大小的比较是以ASCII 码表上的顺序来决定，次顺序亦为字符的值。memcmp()首先将s1 第一个字符值减去s2 第一个字符的值，若差为0 则再继续比较下个字符，若差值不为0 则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(－33)。

返回值：若参数s1 和s2 所指的内存内容都完全相同则返回0 值。s1 若大于s2 则返回大于0 的值。s1 若小于s2 则返回小于0 的值。

代码第三行：将数组的首地址强制转换为指向char类型的指针，是利用char类型大小为1字节的特性，使elemAddr指向此”泛型“数组的第i-1个元素的首地址。因为之前已经说过，此时你并不知道你传入的是什么类型的数据，系统无法确定此数组一个元素有多长，跳向下个元素需要多少字节，所以强制转换为指向char的指针，再加上参数传入的元素大小信息和累加数i的乘积，即偏移地址，即可得此数组第i-1个元素的首地址。这样使无论传入的参数是指向什么类型的指针，都可以得到指向正确元素的指针，实现泛型编程。

函数memcmp(),原型：int memcmp(void \*dest,const void \*src,int n)，比较两段长度为n首地址分别为dest、src的地址空间中的内容。

此函数在数组base中查找key元素，找到则返回它的地址信息，找不到则返回NULL。

如果使用函数指针，则可以实现其行为的泛型：

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/skyline0623/article/details/6164011)

1. **void** \*lsearch(**void** \*key,**void** \*base,**int** n,**int** elemSize,**int**(\*cmpfn)(**void**\*,**void**\*,**int**)){
2. **for**(**int** i=0;i<n;++i){
3. **void** \*elemAddr=(**char** \*)base+i\*elemSize;
4. **if**(cmpfn(key,elemAddr,elemSize)==0)
5. **return** elemAddr;
6. **return** NULL;
7. }
8. }
9. }

在调用时，可以将指向不同行为的形式相同的函数的指针传入此函数，以实现lsearch函数行为的不同。

函数指针：

返回类型 (\*函数指针名)(形参表)

        综上，C语言也可以实现一定的泛型编程，但这样是不安全的，系统对其只有有限的检查。在编程时一定要多加细心。