

# C/C++

## 函数

---

张晓平

武汉大学数学与统计学院

# Table of contents

1. 函数概述
2. 递归
3. 地址运算符：&
4. 改变调用函数中的变量
5. 指针

## 函数概述

## 为什么使用函数？

- 使用函数可以减少代码的重复。若程序需要多次使用某种特定的功能，只需编写一个合适的函数，然后程序可以在任何需要的地方调用该函数。
- 即使某种功能在程序中只使用一次，将其以函数的形式实现也有必要，因为函数使得程序更加模块化，从而有利于程序的阅读、修改和完善。

# 为什么使用函数？

假设你想编写一个程序，以实现如下功能：

- 读入一行数字
- 对数字进行排序
- 求他们的平均值
- 打印出一个柱状图

# 为什么使用函数？

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
    float list[SIZE];
    readlist(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);
    bargraph(list, SIZE);
    return 0;
}
```

## 为什么使用函数？

如何实现这四个函数需要你自行完成。描述性的函数名可以清楚地表明程序的功能和组织结构，然后对每个函数进行独立设计，若这些函数足够通用化，则可以在其他程序中调用它们。

## 为什么使用函数？

- 函数可看做是一个“黑盒子”，你只需关注函数的功能及使用方法，而其内部行为你无需考虑，除非你是该函数的编写者。
- 如我们在使用 `printf()` 时，只需输入一个控制字符串，或者还有其它一些参数，就可以预测 `printf()` 的执行结果，而无须了解 `printf()` 内部的代码。
- 以这种方式看待函数，有助于集中精力投入到程序的整体设计而不是实现细节。



# 对于函数需要了解些什么？

- 如何正确定义函数
- 如何调用函数
- 如何建立函数间的通信

# 一个简单的例子

请打印一个简单的信头：

```
*****  
Wuhan University  
299 Bayi Road Wuchang District,  
Wuhan, PR China 430072  
*****
```

## 一个简单的例子 i

```
// lethead1.c
#include <stdio.h>
#define NAME "Wuhan University"
#define ADDRESS "299 Bayi Road, Wuchang District
, "
#define PLACE "Wuhan, PR China 430072"
#define WIDTH 40
void starbar(void); /* prototype the function */
int main(void)
{
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    printf("%s\n", PLACE);
```

## 一个简单的例子 ii

```
    starbar(); /* use the function */  
    return 0;  
}  
  
void starbar(void) /* define the function */  
{  
    int count;  
    for (count = 1; count <= WIDTH; count++)  
        putchar('*');  
    putchar('\n');  
}
```

starbar() 在不同位置出现了三次:

- 函数原型 (function prototype): 告知编译器 starbar 的函数类型
- 函数调用 (function call): 使函数执行
- 函数定义 (function definition): 实现函数的具体功能

函数同变量一样有多种类型。函数在被使用之前都要声明其类型，故 `main()` 之前出现了代码

```
void starbar(void);
```

- 圆括号表明 `starbar()` 是一个函数名。
- 第一个 `void` 指的是函数类型，表明该函数没有返回值。
- 第二个 `void` 表明该函数不接受任何参数。
- 分号表示该语句是进行函数声明，而不是函数定义。

# 程序分析

函数原型也可以放在 main 函数内变量声明的任何位置，故以下两种写法都正确：

```
...  
void starbar(void);  
int main(void)  
{  
    ...  
}
```

```
...  
int main(void)  
{  
    void starbar(void);  
}
```

程序在 `main()` 中通过使用以下方式调用 `starbar()` :

```
starbar();
```

- 当程序执行到该语句时，它找到 `starbar()` 并执行其中的指令。
- 执行完 `starbar()` 中的代码后，程序将返回到调用函数 (calling function) 的下一条语句继续执行。



- 程序中 `starbar()` 和 `main()` 有相同的定义格式，即首先以类型、名称和圆括号开始，接着是开始花括号、变量声明、函数语句定义以及结束花括号。
- 注意此处的 `starbar()` 后跟花括号，告诉编译器这是在定义函数，而不是调用它或声明其原型。

- 该程序中 `starbar()` 和 `main()` 在同一个文件中，也可以将它们放在不同文件中。
- 单文件形式比较容易编译，而使用多个文件则有利于在不同的程序中使用相同的函数。
- 若使用多文件形式，则每个文件中都必须包含 `#define` 和 `#include` 指令。

- `starbar()` 中的变量 `count` 是一个局部变量，这意味着该变量只在 `starbar()` 中可用。
- 即使你在其它函数中使用名称 `count`，也不会出现任何冲突。

改写以上程序，让信头的文字居中，形如

```
*****  
                Wuhan University  
    299 Bayi Road, Wuchang District,  
                Wuhan, PR China 430072  
*****
```

## 如何做到？

假设一行是 40 个字符宽度。

## 如何做到？

假设一行是 40 个字符宽度。

1. 打印一行星号很容易做到，直接输出 40 个星号即可。

## 如何做到？

假设一行是 40 个字符宽度。

1. 打印一行星号很容易做到，直接输出 40 个星号即可。
2. 如何让 Wuhan University 居中呢？。

## 如何做到？

假设一行是 40 个字符宽度。

1. 打印一行星号很容易做到，直接输出 40 个星号即可。
2. 如何让 Wuhan University 居中呢？。

在输出文字之前输出若干空格即可。



## 如何做到？

假设一行是 40 个字符宽度。

1. 打印一行星号很容易做到，直接输出 40 个星号即可。

2. 如何让 Wuhan University 居中呢？

在输出文字之前输出若干空格即可。

3. 那到底输出多少个空格呢？

## 如何做到？

假设一行是 40 个字符宽度。

1. 打印一行星号很容易做到，直接输出 40 个星号即可。
2. 如何让 Wuhan University 居中呢？

在输出文字之前输出若干空格即可。

3. 那到底输出多少个空格呢？

设文字长度为  $l$ ，则一行中除文字外还需  $40 - l$  个空格。想要文字居中，左边应输出  $(40 - l)/2$  个空格。

## 程序实现 i

```
// lethead2.c
#include <stdio.h>
#include <string.h>
#define NAME "Wuhan University"
#define ADDRESS "299 Bayi Road, Wuchang District", "
#define PLACE "Wuhan, PR China 430072"
#define WIDTH 40
#define SPACE ' '
void show_n_char(char ch, int num);

int main(void)
{
    int spaces;
```

## 程序实现 ii

```
show_n_char('*', WIDTH);
putchar('\n');
show_n_char(SPACE, 12);      /* use a constant
as arguments */
printf("%s\n", NAME);
spaces = (WIDTH - strlen(ADDRESS))/2;
show_n_char(SPACE, spaces); /* use a variable
as argument */
printf("%s\n", ADDRESS);
show_n_char(SPACE, (WIDTH - strlen(PLACE))/2);
/* use an expression as argument */
printf("%s\n", PLACE);
show_n_char('*', WIDTH);
putchar('\n');
return 0;
```

```
}

/* show_n_char() definition */
void show_n_char(char ch, int num)
{
    int count;
    for (count = 1; count <= num; count++)
        putchar(ch);
}
```

## 定义带参数的函数（形式参数，简称“形参”）

### 函数头

```
void show_n_char(char ch, int num)
```

- 这行代码告诉编译器，`show_n_char()` 使用了两个参数 `ch` 和 `num`，它们的类型分别为 `char` 和 `int`。
- 变量 `ch` 和 `num` 被称为形式参数（formal argument）或形式参量（formal parameter）。
- 形式参量是局部变量，为函数所私有，这意味着可以在其它函数中使用相同的变量名。
- 调用函数时，形式参量会被赋值。

## 定义带参数的函数（形式参数，简称“形参”）

必须在每个形参前声明其类型，不能像通常的变量声明那样使用变量列表来声明同一类型的变量。比如

```
void func1(int x, y, z)    // wrong  
void func2(int x, int y, int z)  // right
```

## 定义带参数的函数（形式参数，简称“形参”）

古老的函数定义方式 1:

```
void show_n_char(ch, num)
char ch;
int num;
{
    ...
}
```



## 定义带参数的函数（形式参数，简称“形参”）

古老的函数定义方式 2:

```
void func1(x, y, z)
int x, y, z;
{
    ...
}
```

## 带参数函数的声明

- 使用函数之前需要用 ANSI 原型声明该函数

```
void show_n_char(char ch, int num);
```

- 当函数接受参数时，函数原型通过使用一个逗号分隔的类型列表指明参数的个数和类型。在函数原型中可根据你的喜好省略变量名：

```
void show_n_char(char, int);
```

- 在原型中使用变量名并没有实际地创建变量。

## 带参数函数的声明

ANSI C 也支持旧的函数声明形式，即圆括号内不带任何参数：

```
void show_n_char();
```

该方式请不要使用。了解该形式的主要原因只是为了让你能正确识别并理解以前的代码。

## 调用带参数的函数：实际参数，简称“实参”

函数调用中，通过使用实际参数（actual argument）对 `ch` 和 `num` 赋值。

- 第一次调用中

```
show_n_char(SPACE, 12);
```

实参是空格字符和 12，它们被赋给 `show_n_char()` 中相应的形参：`ch` 和 `num`。

- 实参可以是常量、变量或一个复杂的表达式。
- 但无论何种形式的实参，执行时首先要计算其值，然后将该值赋值给被调函数中相应的形参。

## 调用带参数的函数：实际参数，简称“实参”

实参赋值给形参，被调函数使用的值是从调用函数中复制而来的，故不管在被调函数中对赋值数值进行了什么操作，调用函数中的原数值不受影响。

## 使用 `return` 从函数中返回一个值

- 将实参赋值给形参，实现了从调用函数到被调函数的通信。
- 而想从被调函数往调用函数传递信息，可以使用函数返回值。

## 使用 `return` 从函数中返回一个值

### 例

编写函数，比较两个整数的大小，并返回较小值。同时编制一个驱动程序来测试该函数。

## 使用 `return` 从函数中返回一个值 `i`

```
// lesser.c -- finds the lesser of two integers
#include <stdio.h>
int imin(int, int);
int main(void)
{
    int n1, n2;
    printf("Enter two integers (q to quit):\n");
    while (scanf("%d %d", &n1, &n2) == 2) {
        printf("The lesser of %d and %d is %d.\n",
               n1, n2, imin(n1,n2));
        printf("Enter two integers (q to quit):\n");
    }
    printf("Bye.\n");
    return 0;
}
```



## 使用 `return` 从函数中返回一个值 ii

```
}  
  
int imin(int n,int m)  
{  
    int min;  
    min = (n < m) ? n : m;  
    return min;  
}
```

## 使用 `return` 从函数中返回一个值

```
Enter two integers (q to quit):  
509 333  
The lesser of 509 and 333 is 333.  
Enter two integers (q to quit):  
-9333 6  
The lesser of -9333 and 6 is -9333.  
Enter two of integers (q to quit):  
q  
Bye.
```

## 使用 `return` 从函数中返回一个值

- 关键字 `return` 指明了其后的表达式的值即为该函数的返回值。
- `imin()` 中的变量 `min` 是其私有的，但 `return` 语句将它的值返回给了调用函数。

- 语句

```
lesser = imin(m, n);
```

相当于把 `min` 的值赋给了 `lesser`。

- 能否这么写？

```
imin(m, n);  
lesser = min;
```

## 使用 `return` 从函数中返回一个值

- 关键字 `return` 指明了其后的表达式的值即为该函数的返回值。
- `imin()` 中的变量 `min` 是其私有的，但 `return` 语句将它的值返回给了调用函数。
- 语句

```
lesser = imin(m, n);
```

相当于把 `min` 的值赋给了 `lesser`。

- 能否这么写？

```
imin(m, n);  
lesser = min;
```

# 当然不行!!!

## 使用 return 从函数中返回一个值

返回值不仅可以被赋给一个变量，也可以被用作表达式的一部分。如

```
answer = 2*imin(m, n) + 5;  
printf("%d\n", imin(answer+2, LIMIT));
```

## 使用 return 从函数中返回一个值

返回值可以由任何表达式计算而得到，而不仅仅来自于一个变量。如 `imin` 函数可以改写为

```
int imin(int n,int m)
{
    return ((n < m) ? n : m);
}
```

这里并不要求使用圆括号，但如果想让程序更清晰，可以把添上一个圆括号。

## 使用 return 从函数中返回一个值

观察以下代码：

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z;
}
```

这里，返回值的类型和声明的类型不一致，What will happen?

## 使用 return 从函数中返回一个值

观察以下代码：

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z;
}
```

这里，返回值的类型和声明的类型不一致，What will happen?

将把 `double` 型变量 `z` 的值强制转换为 `int` 型。



## 使用 return 从函数中返回一个值

`return` 的另一个作用是终止函数的执行，并把控制返回给调用函数的下一条语句，即使 `return` 语句不在函数尾部。如 `imin()` 可以写成

```
int imin(int n, int m)
{
    if (n < m)
        return n;
    else
        return m;
    printf("Oh my god!\n");
}
```

`return` 语句使得 `printf` 语句永远不会执行。

## 使用 return 从函数中返回一个值

也可以使用语句

```
return;
```

该语句会终止执行函数，并把控制返回给调用函数。此时，`return` 后没有任何表达式，故没有返回值，该形式只能用于 `void` 类型的函数。

- 函数应该进行类型声明，同时其类型应和返回值类型相同。
- 无返回值的函数应该被声明为 `void` 类型。
- 类型声明是函数定义的一部分，该类型指的是返回值类型。如函数头

```
double klink(int a, int b)
```

表示函数使用两个 `int` 型的参数，而返回值类型为 `double`。

# 函数类型

为正确使用函数，程序在首次调用函数之前需要知道该函数的类型。

- 方式一：调用之前给出完整的函数定义。

```
int imin(int n, int m)
{
    ...
}

int main(void)
{
    ...
    n = imin(n1, n2);
    ...
}
```

# 函数类型

- 方式二：对函数进行声明，以便将函数信息通知编译器。

```
int imin(int, int);

int main(void)
{
    int n1, n2, lesser;
    ...
    n = imin(n1, n2);
    ...
}

int imin(int n, int m)
{
    ...
}
```

也可将函数声明放在调用函数内部。

```
int main(void)
{
    int imin(int, int);
    int n1, n2, lesser;
    ...
    n = imin(n1, n2);
    ...
}

int imin(int n, int m)
{
    ...
}
```

在 ANSI C 标准库中，函数被分为几个系列，每一系列都有各自的头文件，这些头文件中包含了本系列函数的声明部分。

```
// stdio.h
int getchar();
int putchar(int c);
int printf(const char *format , ... );
int scanf (const char *format , ... );
```



# 函数类型

```
// math.h
double sin(double);
double cos(double);
double tan(double);
double asin(double);
double acos(double);
double atan(double);
double log(double);
double log10(double);
double pow(double x, double y);
double exp(double);
double sqrt(double);
int abs(int);
double fabs(double);
```

# 递归

- C 允许一个函数调用其自身，这种调用过程被称为递归 (recursion)。
- 递归一般可用循环代替。有些情况使用循环会比较好，而有时使用递归更有效。
  - 递归虽然可使程序结构优美，但其执行效率却没循环语句高。

## 递归 i

```
/* recur.c -- recursion illustration */
#include <stdio.h>
void up_and_down(int);
int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); //
    1
    if (n < 4)
```

```
    up_and_down(n+1);  
    printf("LEVEL %d: n location %p\n", n, &n); //  
    2  
}
```

```
Level 1: n location 0x7fff5fbff7bc  
Level 2: n location 0x7fff5fbff79c  
Level 3: n location 0x7fff5fbff77c  
Level 4: n location 0x7fff5fbff75c  
LEVEL 4: n location 0x7fff5fbff75c  
LEVEL 3: n location 0x7fff5fbff77c  
LEVEL 2: n location 0x7fff5fbff79c  
LEVEL 1: n location 0x7fff5fbff7bc
```

```
Level 1: n location 0x7fff5fbff7bc  
Level 2: n location 0x7fff5fbff79c  
Level 3: n location 0x7fff5fbff77c  
Level 4: n location 0x7fff5fbff75c  
LEVEL 4: n location 0x7fff5fbff75c  
LEVEL 3: n location 0x7fff5fbff77c  
LEVEL 2: n location 0x7fff5fbff79c  
LEVEL 1: n location 0x7fff5fbff7bc
```

& 为地址运算符，&n 表示存储 n 的内存地址，printf() 使用占位符 %p 来指示地址。

- 首先, `main()` 使用实参 1 调用 `up_and_down()`, 打印语句 #1 输出 Level 1。
- 然后, 由于  $n < 4$ , 故 `up_and_down()` (第 1 级) 使用实参 2 调用 `up_and_down()` (第 2 级), 打印语句 #1 输出 Level 2。
- 类似地, 下面的两次调用打印 Level 3 和 Level 4。



- 当开始执行第 4 级调用时， $n$  的值为 4，故 if 语句不满足条件，不再继续调用 `up_and_down()`，接着执行打印语句 #2，输出 Level 4，至此第 4 级调用结束，把控制返回给第 3 级调用函数。
- 第 3 级调用函数中前一个执行过的语句是在 if 语句中执行第 4 级调用，因此，它开始执行后续代码，即执行打印语句 #2，输出 Level 3。
- 当第 3 级调用结束后，第 2 级调用函数开始继续执行，输出 Level 2。以此类推。

## 递归：递归的基本原理

- 每一级的递归都使用其私有变量  $n$ 。
- 每一次函数调用都会有一次返回。当程序执行到某一级递归的结尾处时，它会转移到前一级递归继续执行。

## 递归：递归的基本原理

- 递归函数中，位于递归调用前的语句和各级被调函数具有相同的执行次序。

如打印语句 #1 位于递归调用语句之前，它按递归调用的顺序执行 4 次，即依次为第 1 级、第 2 级、第 3 级和第 4 级。

- 递归函数中，位于递归调用后的语句和各级被调函数具有相反的执行次序。

如打印语句 #2 位于递归调用语句之后，执行次序为：第 4 级、第 3 级、第 2 级和第 1 级。

- 递归函数中，必须包含可以终止递归调用的语句。

最简单的递归方式是**把递归调用语句放在函数结尾，return 语句之前**。这种形式被称为**尾递归 (tail recursion)**。尾递归的作用相当于一条循环语句，它是最简单的递归形式。

分别使用循环和尾递归编写函数计算阶乘，然后用一个驱动程序测试它们。

## 递归：尾递归 i

```
// factor.c -- uses loops and recursion to
calculate factorials
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
    int num;
    printf("This program calculates factorials.\n"
);
    printf("Enter a value in the range 0-12 (q to
quit):\n");
    while (scanf("%d", &num) == 1) {
        if (num < 0)
```

## 递归：尾递归 ii

```
    printf("No negative numbers, please.\n");
else if (num > 12)
    printf("Keep input under 13.\n");
else {
    printf("loop:      %d! = %ld\n",
           num, fact(num));
    printf("recursion: %d! = %ld\n",
           num, rfact(num));
}
printf("Enter a value in the range 0-12 (q
to quit):\n");
}
printf("Bye.\n");
return 0;
}
```



## 递归：尾递归 iii

```
long fact(int n) // loop-based function
{
    long ans;
    for (ans = 1; n > 1; n--)
        ans *= n;
    return ans;
}

long rfact(int n) // recursive version
{
    long ans;
    if (n > 0)
        ans = n * rfact(n-1);
    else
        ans = 1;
}
```

```
return ans;  
}
```

## 递归：尾递归

```
This program calculates factorials.  
Enter a value in the range 0-12 (q to quit):  
5  
loop:      5! = 120  
recursion: 5! = 120  
Enter a value in the range 0-12 (q to quit):  
10  
loop:      10! = 3628800  
recursion: 10! = 3628800  
Enter a value in the range 0-12 (q to quit):  
12  
loop:      12! = 479001600  
recursion: 12! = 479001600  
Enter a value in the range 0-12 (q to quit):  
q  
Bye.
```

选用循环还是递归？

选用循环还是递归？一般来说，选择循环更好一些。

选用循环还是递归？一般来说，选择循环更好一些。

- 每次递归调用都有自己的变量集合，需要占用较多的内存。每次递归调用需要把新的变量集合存储在堆栈中。
- 每次函数调用都要花费一定的时间，故递归的执行速度较慢。

那为什么要学习递归呢？

### 那为什么要学习递归呢？

- 尾递归非常简单，易于理解。
- 某些情况下，不能使用简单的循环语句代替递归，所以有必要学习递归。



## 递归：递归与反向计算

**例**

编写程序，将一个整数转换为二进制形式。

### 例

编写程序，将一个整数转换为二进制形式。

对于奇数，其二进制形式的末位为 1；而对于偶数，其二进制形式的末位为 0。于是，对于  $n$ ，其二进制数的末位为  $n\%2$ 。

# 递归：递归与反向计算

## 例

编写程序，将一个整数转换为二进制形式。

对于奇数，其二进制形式的末位为 1；而对于偶数，其二进制形式的末位为 0。于是，对于  $n$ ，其二进制数的末位为  $n\%2$ 。

## 规律

- 在递归调用之前，计算  $n\%2$  的值，在递归调用之后输出。
- 为算下一个数字，需把原数值除以 2。若此时得出的为偶数，则下一个二进制位为 0；若得出的是奇数，则下一个二进制位为 1。

## 递归：递归与反向计算 i

```
/* binary.c -- prints integer in binary form
*/
#include <stdio.h>
void to_binary(unsigned long n);
int main(void)
{
    unsigned long number;
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1) {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
}
```

## 递归：递归与反向计算 ii

```
    printf("Done.\n");  
    return 0;  
}  
  
void to_binary(unsigned long n)  
{  
    int r;  
    r = n % 2;  
    if (n >= 2)  
        to_binary(n / 2);  
    putchar('0' + r);  
    return;  
}
```

## 递归：递归与反向计算

```
Enter an integer (q to quit):
```

```
9
```

```
Binary equivalent: 1001
```

```
Enter an integer (q to quit):
```

```
255
```

```
Binary equivalent: 11111111
```

```
Enter an integer (q to quit):
```

```
1024
```

```
Binary equivalent: 10000000000
```

```
Enter an integer (q to quit):
```

```
q
```

```
Done.
```

## 递归：递归的优缺点

- 优点：  
为某些编程问题提供了最简单的解决办法。
- 缺点：  
一些递归算法会很快地耗尽计算机的内存资源，同时递归程序难于阅读和维护。

### 例

编写程序，计算斐波那契数列。

$$F_1 = F_2 = 1,$$

$$F_n = F_{n-1} + F_{n-2}, \quad n = 3, 4, \dots.$$



## 递归：递归的优缺点

```
long Fibonacci(int n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

该函数使用了双重递归 (double recursion)，即函数对本身进行了两次调用。这会导致一个弱点。

## 递归：递归的优缺点

每级调用的变量数会呈指数级增长：

表 1: 每级调用中变量  $n$  的个数

| Level    | number of $n$ |
|----------|---------------|
| 1        | 1             |
| 2        | 2             |
| 3        | $2^2$         |
| 4        | $2^3$         |
| $\vdots$ | $\vdots$      |
| $l$      | $2^{l-1}$     |

**地址运算符：&**

## 地址运算符：&

C 最重要的、也是最复杂的一个概念是指针 (pointer)，即用来存储地址的变量。

## 地址运算符：&

- `scanf()` 使用地址作为参数。
- 更一般地，若想在无返回值的被调函数中修改调用函数的某个数据，必须使用地址参数。

## 地址运算符：&

& 为单目运算符，可以取得变量的存储地址。

## 地址运算符：&

& 为单目运算符，可以取得变量的存储地址。

设 var 为一个变量，则&var为该变量的地址。

一个变量的地址就是该变量在内存中的地址。

## 地址运算符：&

设有如下语句

```
var = 24;
```

并假定 var 的存储地址为 07BC，则执行语句

```
printf("%d %p\n", var, &var);
```

的结果为

```
24 07BC
```



## 地址运算符：& i

```
/* loccheck.c -- checks to see where variables
are stored */
#include <stdio.h>
void func(int);
int main(void)
{
    int var1 = 2, var2 = 5; // local to main()
    printf("main(): var1 = %2d, &var1 = %p\n",
           var1, &var1);
    printf("main(): var2 = %2d, &var2 = %p\n",
           var2, &var2);
    func(var2);
    return 0;
}
```

## 地址运算符：& ii

```
void func(int var2)
{
    int var1 = 10; // local to func()
    printf("func(): var1 = %2d, &var1 = %p\n",
           var1, &var1);
    printf("func(): var2 = %2d, &var2 = %p\n",
           var2, &var2);
}
```

## 地址运算符：&

```
In main(), var1 = 2 and &var1 = 0x7fff5fbff7d8  
In main(), var2 = 5 and &var2 = 0x7fff5fbff7d4  
In func(), var1 = 10 and &var1 = 0x7fff5fbff7a8  
In func(), var2 = 5 and &var2 = 0x7fff5fbff7ac
```

- 两个 var1 变量具有不同的地址，两个 var2 变量也是如此。
- 调用 func 函数时，把实参（main() 中的 var2）的值 5 传递给了形参（func() 中的 var2）。但这种传递只是进行了数值传递，两个变量仍是独立的。

## 改变调用函数中的变量

## 改变调用函数中的变量

有些时候，我们需要用一个函数改变另一个函数的变量。如排序问题中，一个常见的任务是交换两个变量的值。

## 改变调用函数中的变量

以下代码能否交换变量  $x$  和  $y$  的值？

```
x = y;  
y = x;
```

## 改变调用函数中的变量

以下代码能否交换变量  $x$  和  $y$  的值？

```
x = y;  
y = x;
```





# 改变调用函数中的变量

以下代码能否交换变量  $x$  和  $y$  的值？

```
x = y;
```

```
y = x;
```



## 改变调用函数中的变量

那以下代码能否交换变量  $x$  和  $y$  的值？

```
temp = y;  
x = y;  
y = temp;
```

## 改变调用函数中的变量

那以下代码能否交换变量  $x$  和  $y$  的值？

```
temp = y;  
x = y;  
y = temp;
```



## 改变调用函数中的变量

```
// swap1.c
#include <stdio.h>
void swap(int u, int v);
int main(void)
{
    int x = 5, y = 10;
    printf("Before: x = %2d, y = %2d.\n", x, y);
    swap(x, y);
    printf("After : x = %2d, y = %2d.\n", x, y);
    return 0;
}
```

## 改变调用函数中的变量

```
void swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
}
```

## 改变调用函数中的变量

```
void swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
}
```

Before: x = 5, y = 10.

After : x = 5, y = 10.

## 改变调用函数中的变量

```
void swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
}
```

Before: x = 5, y = 10.

After : x = 5, y = 10.

Why not interchanged?

## 改变调用函数中的变量 i

```
// swap2.c:
#include <stdio.h>
void swap(int u, int v);
int main(void)
{
    int x = 5, y = 10;
    printf("Before: x = %2d, y = %2d.\n", x, y);
    swap(x, y);
    printf("After : x = %2d, y = %2d.\n", x, y);
    return 0;
}
```



## 改变调用函数中的变量

```
void swap(int u, int v)
{
    int temp;
    printf("Before: u = %2d, v = %2d.\n", u, v);
    temp = u;
    u = v;
    v = temp;
    printf("After : u = %2d, v = %2d.\n", u, v);
}
```

## 改变调用函数中的变量

```
void swap(int u, int v)
{
    int temp;
    printf("Before: u = %2d, v = %2d.\n", u, v);
    temp = u;
    u = v;
    v = temp;
    printf("After : u = %2d, v = %2d.\n", u, v);
}
```

## 改变调用函数中的变量

Before: x = 5, y = 10.

Before: u = 5, v = 10.

After : u = 10, v = 5.

After : x = 5, y = 10.

## 改变调用函数中的变量

```
Before: x = 5, y = 10.  
Before: u = 5, v = 10.  
After : u = 10, v = 5.  
After : x = 5, y = 10.
```

- 在 `swap()` 中, `u` 和 `v` 的值确实得到了交换。问题出在了把执行结果传递给 `main()` 的时候。
- `swap()` 中的变量独立于 `main()`, 因此交换 `u` 和 `v` 的值对 `x` 和 `y` 的值没有任何影响。

## 改变调用函数中的变量

能否使用 `return` ? 如

```
int main(void)
{
    ...
    x = swap(x, y);
    ...
}

int swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
    return u;
}
```

## 改变调用函数中的变量

此时，`x` 的值得以更新，但 `y` 的值仍未做改变。因为 `return` 语句只能把一个数值传递给调用函数，而现在却需要传递两个数值。

## 改变调用函数中的变量

此时，`x` 的值得以更新，但 `y` 的值仍未做改变。因为 `return` 语句只能把一个数值传递给调用函数，而现在却需要传递两个数值。

怎么办？

## 改变调用函数中的变量

此时，`x` 的值得以更新，但 `y` 的值仍未做改变。因为 `return` 语句只能把一个数值传递给调用函数，而现在却需要传递两个数值。

怎么办？

用指针！



# 指针

指针是一个变量，其值为一个地址。

假如你把某个指针变量命名为 ptr，就可以使用以下语句

```
ptr = &var;
```

即把变量 var 的地址赋给指针变量 ptr，称为ptr “指向” var。

假如你把某个指针变量命名为 ptr，就可以使用以下语句

```
ptr = &var;
```

即把变量 var 的地址赋给指针变量 ptr，称为ptr “指向” var。

ptr 和 &var的区别在于，前者为一变量，后者是一个常量。

ptr 可以指向任何地址，即可以把任何地址赋值给 ptr：

```
ptr = &var1;
```

## 问题

何创建一个指针变量？

## 问题

何创建一个指针变量？

首先需要声明其类型。先介绍一个新运算符 \*。

## 指针：间接运算符或取值运算符：\*

假定 ptr 指向 var，即

```
ptr = &var;
```

就可以用间接运算符 \* 来获取 var 中存放的数值：

```
value = *ptr;
```



## 指针：间接运算符或取值运算符：\*

假定 ptr 指向 var，即

```
ptr = &var;
```

就可以用间接运算符 \* 来获取 var 中存放的数值：

```
value = *ptr;
```

```
ptr = &var;
```

```
value = *ptr;
```

⇔

```
value = var;
```

## 指针：指针声明

能否如以下方式声明一个指针？

```
pointer ptr;
```

## 指针：指针声明

能否如以下方式声明一个指针？

```
pointer ptr;
```



## 指针：指针声明

能否如以下方式声明一个指针？

```
pointer ptr;
```



原因在于，仅声明一个变量为指针是不够的，还需说明指针所指向变量的类型。

- 不同的变量类型占用的存储空间大小不同，而指针需要知道变量类型所占用的存储空间。
- 程序也需要知道地址中存储的是何种数据。

## 指针：指针声明

```
// 正确的指针声明方式  
int * pi;           // pi是指向一个整型变量的指针  
char * pc;          // pc是指向一个字符变量的指针  
float * pf, * pg;   // pf和pg是指向浮点变量的指针
```

## 指针：指针声明

```
// 正确的指针声明方式
int * pi;           // pi是指向一个整型变量的指针
char * pc;          // pc是指向一个字符变量的指针
float * pf, * pg;   // pf和pg是指向浮点变量的指针
```

- 类型标识符表明了被指向变量的类型，\*表示该变量为一个指针。
- 声明 `int * pi;` 的含义是：pi 是一个指针，且 \*pi 是 `int` 类型的。
- \* 与指针名之间的空格可选。通常在声明中使用空格，在指向变量时将其省略。

## 指针：指针声明

- pc 所指向的值 (\*pc) 是 char 类型的，而 pc 本身是“指向 char 的指针”类型。
- pc 的值是一个地址，在大多数系统中，它由一个无符号整数表示。但这并不表示可以把指针看做是整数类型。
- 一些处理整数的方法不能用来处理指针，反之亦然。如两个整数可以相乘，但指针不能。
- 指针是一种新的数据类型，而不是一种整数类型。



## 指针：使用指针在函数间通信

这里将重点介绍如何通过指针解决函数间的通信问题。

## 指针：使用指针在函数间通信 i

```
// swap3.c
#include <stdio.h>
void swap(int *, int *);
int main(void)
{
    int x = 5, y = 10;
    printf("Before: x = %2d, y = %2d.\n", x, y);
    swap(&x, &y);
    printf("After : x = %2d, y = %2d.\n", x, y);
    return 0;
}
```

```
void swap(int * u, int * v)
{
    int temp;
    temp = *u;
    *u = *v;
    *v = temp;
}
```

# 指针

Originally:  $x = 5, y = 10.$

Now :  $x = 10, y = 5.$



## 指针：使用指针在函数间通信

- 函数调用语句为

```
swap(&x, &y);
```

故函数传递的是  $x$  和  $y$  的地址，而不是它们的值。

- 函数声明为

```
void swap(int * u, int * v);
```

也可简化为

```
void swap(int *, int *);
```

## 指针：使用指针在函数间通信

- 函数体中声明了一个临时变量

```
int temp;
```

- 为了把  $x$  的值存在 `temp` 中，需使用以下语句

```
temp = *u;
```

因  $u$  的值为  $\&x$ ，即  $x$  的地址，故  $*u$  代表了  $x$  的值。

- 同理，为了把  $y$  的值赋给  $x$ ，需用以下语句

```
*u = *v;
```

## 指针：使用指针在函数间通信

该例中，用一个函数实现了  $x$  和  $y$  的数值交换。

- 首先函数使用  $x$  和  $y$  的地址作为参数，这使得它可以访问  $x$  和  $y$  变量。
- 通过使用指针和运算符  $*$ ，函数可以获得相应存储地址的数据，从而就可以改变这些数据。

## 指针：使用指针在函数间通信

通常情况下，可以把变量的两类信息传递给一个函数，即传值与传址。



## 指针：传值

- 调用方式为

```
function1(x);
```

- 定义方式为

```
int function1(int num)
```

- 适用范围：使用函数进行数据计算等操作。

## 指针：传址

- 调用方式为

```
function2(&x);
```

- 定义方式为

```
int function2(int * ptr)
```

- 适用范围：改变调用函数中的多个变量的值。