Programming with C

Terry Marris November 2010

13 Pointers

In the previous chapter we created a module of useful functions. Now we turn to pointers. Pointers are used extensively in implementing data structures.

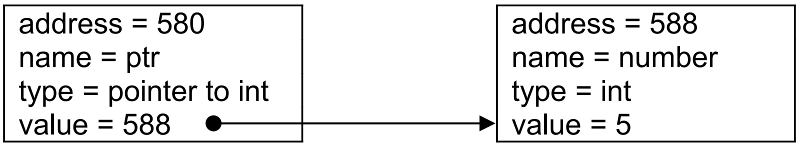
13.1 Pointer

We can think of memory as a set of numbered storage locations. A storage location's number is known as its address.

A variable has an address in memory where it is located, a name, a type, and a value.

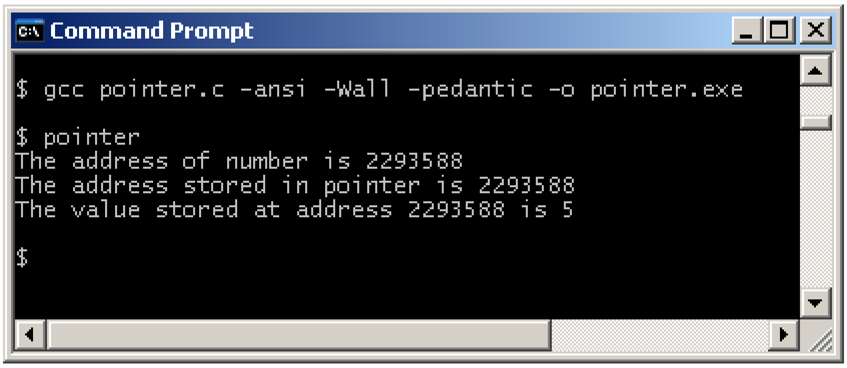
address = 588
name = number
type = int
value = 5

A pointer is a variable whose value is the address of another variable.



The next program, shown below, prints the contents of a pointer variable, and the value of the variable at the address contained in the pointer.

```
/* pointer.c: illustrates the fundamentals of pointers */

#include <stdio.h>

int main()
{
  int number;
  int *ptr;

  number = 5;
  ptr = &number;

  printf("The address of number is %d\n", (int)&number);
  printf("The address stored in pointer is %d\n", (int)ptr);
  printf("The value stored at address %d is %d\n", (int)ptr, *ptr);
  return 0;
}
```



```
$ gcc pointer.c -ansi -Wall -pedantic -o pointer.exe

$ pointer
The address of number is 2293588
The address stored in pointer is 2293588
The value stored at address 2293588 is 5

$
```

First, the declarations

```
int number;
int *ptr;
```

int number; says that *number* is a variable of type *int*. Its address in memory is set by the system. At this point its value is not defined.

*int *ptr*; says that *ptr* is a variable of type *pointer to int*. Its address in memory is set by the system, and its value is undefined. In the context of a variable declaration, * means *pointer to*.

Now, the assignment statements

```
number = 5;
ptr = &number;
```

number = *5*; copies the value 5 into the variable *number*.

ptr = *&number;* copies the address of the variable *number* into the variable *ptr*. & is the *address of* operator. It returns the address of the item immediately following it. So, *&number* is known as a *pointer to number*.

To print the address of the variable *number* we write

```
printf("The address of number is %d\n", (int)&number);
```

Here, the address of number is cast to an *int*. (In some systems where the maximum integer is 32,767, you will need to use *%lu* for the conversion specification and *(unsigned long)* for the cast.)

To print the address stored in the variable *ptr* we write

```
printf("The address stored in pointer is %d\n", (int)ptr);
```

And to print the value stored at that address we write

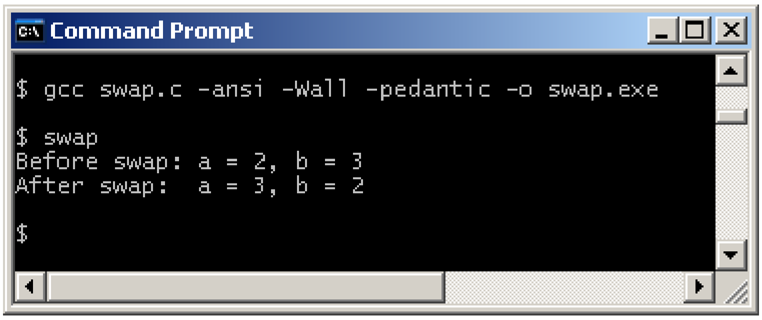
```
printf("The value stored at address %d is %d\n", (int)ptr, *ptr);
```

**ptr* says the value stored at the address contained in *ptr*. Since the address stored is that of the variable *number*, it is *number*'s value, 5, that is displayed. In this context, when the * is used before a pointer variable, the * is known as the *indirection operator*. The indirection operator gives the value of the object being pointed to. We say * de-references the pointer.

13.2 Arguments and Parameters

The classic example featuring pointers and function parameters and arguments is the *swap()* function. But first we see how to exchange the values of two variables.

```
/* swap.c: exchanges the contents of two variables */
#include <stdio.h>

int main()
{
  int a = 2;
  int b = 3;
  int temp;

  printf("Before swap: a = %d, b = %d\n", a, b);

  temp = a;
  a = b;
  b = temp;

  printf("After swap:  a = %d, b = %d\n", a, b);
  return 0;
}
```



```
$ gcc swap.c -ansi -Wall -pedantic -o swap.exe

$ swap
Before swap: a = 2, b = 3
After swap:  a = 3, b = 2

$
```

Initially, *a* is 2, *b* is 3 and the value of *temp* is undefined.

Then the value of *a* is preserved in *temp*, the value of *a* is overwritten by the value of *b*, and the value of *b* is replaced with the value stored in *temp*.

	a	**b**	**temp**
initially	2	3	
temp = a			2
a = b	3		
b = temp		2	

Now we look at the *swap()* function.

```
/* swap: exchanges the values pointed to by ptrA and ptrB */
int swap(int *ptrA, int *ptrB)
{
  int temp;

  temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
  return 0;
}
```

The function parameters are *(int *ptrA, int *ptrB)*. *int *ptrA* says *ptrA* is a pointer to an *int*. *int *ptrB* says *ptrB* is a pointer to an *int*.

*temp = *ptrA* says copy the contents of the variable whose address is in *ptrA* into *temp*.

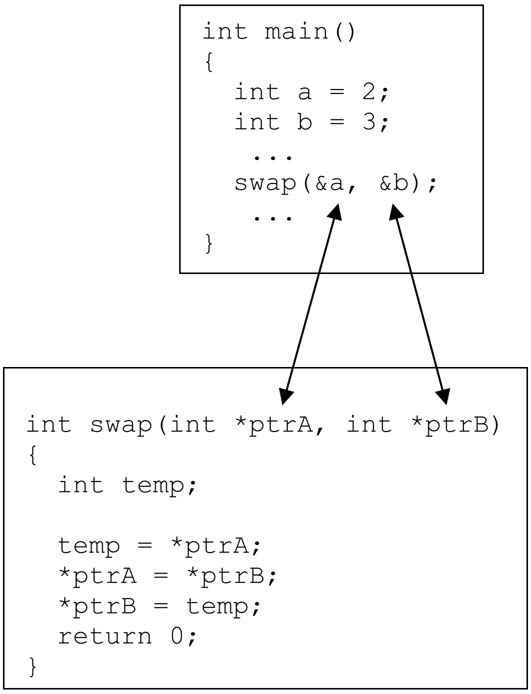
**ptrA = *ptrB* says copy the contents of the variable whose address is in *ptrB* into the variable whose address is in *ptrA*.

**ptrB = temp* says copy the contents of *temp* into the variable whose address is in *ptrB*.

How do *ptrA* and *ptrB* get their addresses? *ptrA* and *ptrB* get their addresses from argument values that are pointers.

```
int a = 2;
int b = 3;
...
swap(&a, &b);
```

It is the address of *a* that is copied to *ptrA*. It is the address of *b* that is copied to *ptrB*.



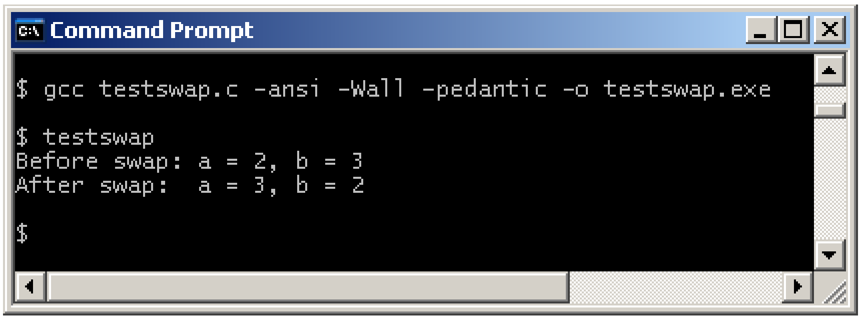
```
int main()
{
  int a = 2;
  int b = 3;
   ...
  swap(&a, &b);
   ...
}
```

```
int swap(int *ptrA, int *ptrB)
{
  int temp;

  temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
  return 0;
}
```

Whatever happens to **ptrA* and **ptrB* in *swap()* directly affects the values stored in *a* and *b* in *main()*.

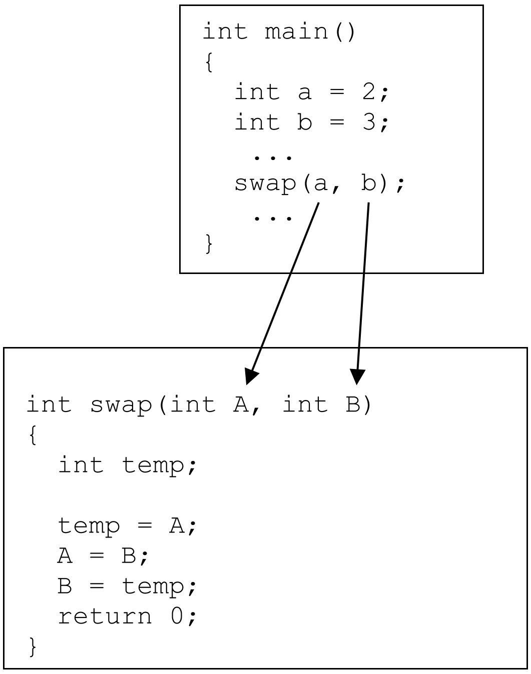
Shown below is the complete program and its run.

```c
/* testswap.c: illustrates use of pointers as arguments */

#include <stdio.h>

/* swap: exchanges the values pointed to by ptrA and ptrB */
int swap(int *ptrA, int *ptrB)
{
  int temp;

  temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
  return 0;
}

int main()
{
  int a = 2;
  int b = 3;

  printf("Before swap: a = %d, b = %d\n", a, b);
  swap(&a, &b);
  printf("After swap:  a = %d, b = %d\n", a, b);
  return 0;
}
```



```
$ gcc testswap.c -ansi -Wall -pedantic -o testswap.exe

$ testswap
Before swap: a = 2, b = 3
After swap:  a = 3, b = 2

$
```

What would happen if you did not bother with addresses and pointer parameters ?



```c
int main()
{
  int a = 2;
  int b = 3;
   ...
  swap(a, b);
   ...
}
```

```c
int swap(int A, int B)
{
  int temp;

  temp = A;
  A = B;
  B = temp;
  return 0;
}
```

The value of *a* is stored in *A*. The value of *b* is stored in *B*. The values of *A* and *B* are exchanged in *swap()*. And that is it. The changes in *A* and *B* have no effect whatsoever on the values stored in *a* and *b*. The communication between arguments and parameters is strictly one way.

If ever we have a choice of having a function return a value directly via a return statement, or return a value through a pointer parameter, we choose to return a value directly because that is simpler and clearer. But there are times when we have no choice but to return values via the pointer argument-parameter pairing.

13.3 Precedence

The indirection and address of operators, * and &, have the same precedence as the increment and decrement operators.

Operator	*Description*	*Precedence*
( )	brackets	highest priority
++ -- * &	increment, decrement, indirection, address	
* / %	times, divide, mod	
+ -	add, subtract	
< <= > >=	relational operators	
== !-	equality operators	
&&	logical and	
||	logical or	
?:	conditional operator	
=	assignment operator	lowest priority

Exercise 13.1

1. Try out the program *pointer.c*, shown above, on your computer system.

2. Write and test a program that declares a variable of type *char* and a variable of type *pointer to char*. Assign the *char* variable a suitable value from 'A'..'Z'. Initialise the pointer variable with the address of the *char* variable. Print the contents of the *char* variable by de-referencing the pointer variable.

3. Try out the *testswap.c* program shown above.

We have looked at pointers.

Next we take a look at arrays. A programmer went to his boss and said: *I want arrays*.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988