C Supplementary

Terry Marris September 2012

Command Line Arguments

We see how the user, usually a person who is not a programmer, can supply options and filenames to a C program for processing. We look at pointers, the increment and decrement operators, arrays, functions, and command line arguments (*argc* and *argv*). But first we see what a command line is.

Command Line

A command line interface is a program development environment. It enables a user (often a person) to communicate with a computer system by entering lines of text via a keyboard, and viewing the response on a monitor. It is a primitive system: no windows, no mice, no touch screens. I say *no windows*, but in today's computers the command line interface is launched in a window. For example, in Windows XP Classic View you launch the command line window by choosing *Start*, *Programs*, *Accessories*, *Command Prompt*.

Fig 1 Microsoft Windows Command Prompt

C:\ represents the root directory on disk drive *C*. Directories are known as folders in Windows. *Terry* is the sub-folder within the *Documents and Settings* folder. The final > is the MS-DOS command line prompt which tells you that the system is waiting for your commands.

What commands can you issue? Well, some of them are shown if you enter the command line *help* at the MS-DOS command line prompt.

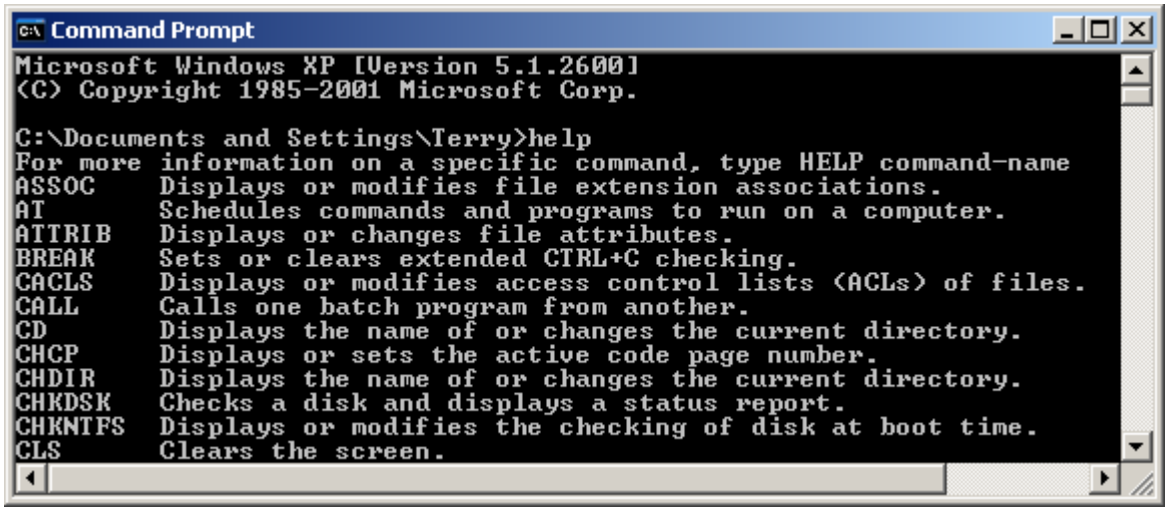


Fig 2 Microsoft windows Command Lines

So, if you enter the command line *CLS,* your command line window is emptied of text. *CLS* is a mnemonic for *clear screen*.

I prefer to launch my C compiler from the command line prompt, and view the program's output in a command line window.

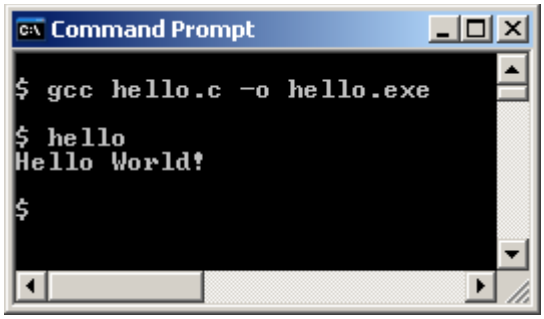


Fig 3 Using the Command Line Interface to compile and execute a C program

Here, my command line prompt is the *$*, and the command lines entered are *gcc hello.c -o hello.exe* and *hello*.

You may prefer to use a C compiler in a windows-like environment. Such environments may (or may not) provide a means of using a command line interface.

Exercise 1

1 Explain how to reach a command line window on your computer system, and how to edit, compile and execute a C program in that window.

Pointers

A variable is a location in memory. It has a name, a type, a value and an address. A variable's address is its location in memory.

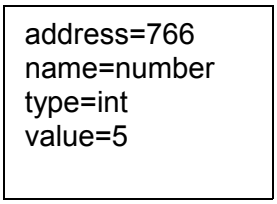


Fig 4 A variable is a location in memory

In C we might write

```
int number = 5;
```

A variable's address is automatically chosen and set by the computer system at run time.

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

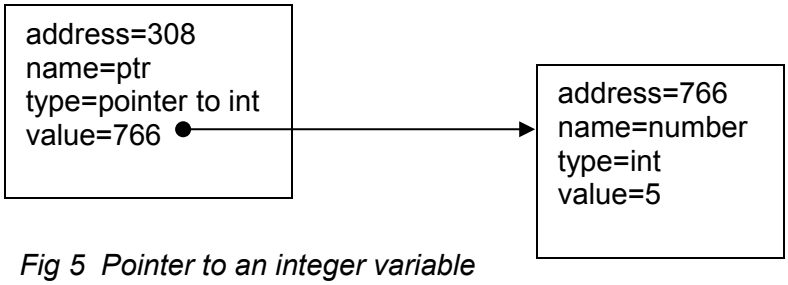


Fig 5 Pointer to an integer variable

In C we might write

```
/* pointer.c - demonstrates pointer fundamentals */

#include <stdio.h>

int main()
{
  int number = 5;
  int *ptr = &number;

  printf("Value stored in number: %d\n", number);
  printf("Value referenced by ptr: %d\n", *ptr);
  return 0;
}
```

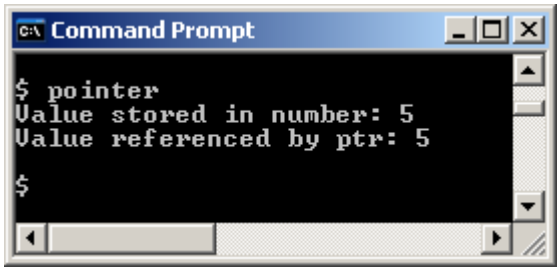


Fig 6 A pointer is a variable that contains the address of another variable

The line

```
int number = 5;
```

defines a variable named *number*, of type *int*, with value *5*.

The line

```
int *ptr = &number;
```

defines a variable named *ptr*, of type *pointer to int*, with value *address of number*. & is the *address of* operator.

We say *ptr* points to *number*. This means the variable *ptr* contains the address of the variable *number*. A pointer variable is always initialised with the address of an object.

The statement

```
printf("Value referenced by ptr: %d\n", *ptr);
```

prints the value of the variable pointed to by *ptr*. The expression **ptr* provides the value of the variable pointed to by *ptr*. We say **ptr* references the data stored at the address held in *ptr*.

Exercise 2

1 Write a C program that: declares and initialises a *char* variable; declares and initialises a pointer to the *char* variable; prints the value stored in the *char* variable; and prints the value referenced by the pointer variable.

Increment and Decrement Operators

++ is the increment operator. It means *add one to*. So

```
int n = 2;
n++;
```

leaves *n* with the value three.

-- is the decrement operator. It means *subtract one from*. So

```
int n = 5;
n--;
```

leaves *n* with the value four.

We can write *n*++ or ++*n*. Both mean add 1 to n. But ...

```
int n = 3;
int x = n++;
```

leaves *n* with the value four, *x* with the value three. *n* is incremented *after* its value is assigned to *x*. In this situation ++ is a *postfix* operator.

And

```
int n = 3;
int x = ++n;
```

leaves *n* with the value four, *x* with the value four. *n* is incremented *before* its value is assigned to *x*. In this situation ++ is a *prefix* operator.

Exercise 3

1 Design, write and test a program that shows how the increment and decrement operators work in both their postfix and prefix forms.

Arrays

All the data items stored in an array are of the same type. That type might be *int*, *double*, *char*, *pointer*, ... In the array named *bike* pictured below, each data item is of type *char*.

bike

0	1	2	3	4	5	6	7	8	9
D	u	c	a	t	i	\0			

Fig 7 An array of char

An array is a numbered sequence of storage locations. The numbers, known as indices, always start from zero and there are no gaps between the indices. We say an array is indexed from zero.

The value at index 0 is *D*, the value at index 5 is *i*, and the value at index 7 is not defined. We write

bike[0] == 'D'
bike[5] == 'i'
bike[7] == undefined

A sequence of characters is a string and an array of characters implements a string. Every string should be terminated by the null character, shown here as *\0*. Many functions rely on the null character to locate the end of a string.

One way to visit each element in an array is to use a *for ...* loop.

```
/* arrayproc.c - uses a for loop to access each element in an
array of char */

#include <stdio.h>

int main()
{
  char bike[] = "Ducati";
  int i;

  for (i = 0; bike[i] != '\0'; i++)
    printf("%c", bike[i]);

  return 0;
}
```

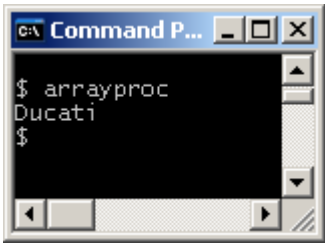


Fig 8 Printing a string

```
char bike[] = "Ducati";
```

declares an array of characters and initialises it with the value *Ducati*. In this case C determines the size of the array automatically. And the terminating null character is also supplied automatically.

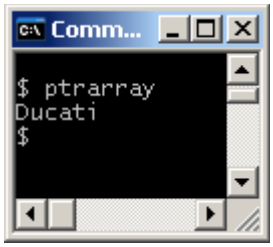
The for loop

```
for (i = 0; bike[i] != '\0'; i++)
  printf("%c", bike[i]);
```

could be replaced entirely with *printf("%s", bike);* But we are just illustrating basic array processing here. For each value of *i*, ranging from zero upwards, we are printing the value of *bike[i]*. So, when *i = 0* we print *bike[0]* i.e. *'D'* When *i = 1* we print *bike[1]* i.e. *'u'*. And so on. When the '\0' character is reached the loop terminates.

You can access array elements with a pointer.

```
/* ptrarray.c - shows how pointers may access a string */

#include <stdio.h>

int main()
{
  char *bike = "Ducati";

  while (*bike != '\0') {
    printf("%c", *bike);
    bike++;
  }
  return 0;
}
```



The line

```
  char *bike = "Ducati";
```

declares a pointer to *char* named *bike*, and assigns to *bike* the address of the first character in *Ducati*.

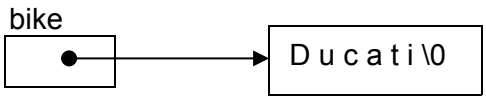


Fig 9 Pointer to a string

The null character, \0, is automatically added to the string by the compiler in the declaration and initialisation.

We loop for as long as the character pointed to by *bike* is not the null character.

```
while (*bike != '\0') {
```

and print the character referenced by **bike*

```
printf("%c", *bike);
```

before incrementing the pointer *bike* to point to (i.e. to contain the address of) the next character with

```
bike++;
```

Exercise 4

1 Create an array of five integers. Use a pointer to visit each element in the array and find the sum of all the values held.

Ok. We have briefly looked at one-dimensional arrays, and the connection between a pointer and an array. We now look at two-dimensional arrays. Two-dimensional arrays are also known as tables. If we need to store a list of string values ...

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	D	u	c	a	t	i	\0								
1	M	o	t	o		G	u	z	z	i	\0				
2	L	a	v	e	r	d	a	\0							

Fig 10 A table is a 2-dimensional array

In C we might write

```
/* table.c - illustrates the fundamentals of 2-dimensional
arrays */

#include <stdio.h>

int main()
{
  char table[3][15] = {
      { "Ducati" },
      { "Moto Guzzi" },
      { "Laverda" }
  };

  int row, col;

  for (row = 0; row < 3; row++) {
    for (col = 0; table[row][col] != '\0'; col++) {
      printf("%c", table[row][col]);
    }
    printf("\n");
  }
  return 0;
}
```

First of all,

```
char table[3][15]
```

defines a table with three rows and 15 columns. Each row can hold up to 15 characters, including the null string terminating character. The array is initialised with

```
char table[3][15] = {
    { "Ducati" },
    { "Moto Guzzi" },
    { "Laverda" }
};
```

Rows and columns are always indexed from zero - see Fig 10 above. The order for referencing a location in the table is always *[row][column]* in that order. So *bike[1][5]* refers to the *G* in *Guzzi*. The standard way for processing a table is: for each row, look at each column in turn.

```
for (row = 0; row < 3; row++) {
  for (col = 0; table[row][col] != '\0'; col++) {
    printf("%c", table[row][col]);
  }
  printf("\n");
}
```

The problem with character arrays is that we must ensure they are large enough for the longest string, but, in doing so, space is often wasted: strings are not usually all the same length. A solution is to use an array of pointers, where each pointer refers to a string.

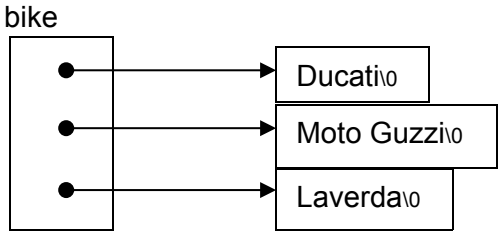


Fig 11 Array of pointers to strings of different lengths

Here is the C program.

```
/* arrayofptrs.c - array of pointers */

#include <stdio.h>

int main()
{
  char *bike[] = { "Ducati", "Moto Guzzi", "Laverda" };
  int i;
  char *ptr;
```

```
  for (i = 0;i < 3; i++) {
    for (ptr = bike[i]; *ptr != '\0'; ptr++) {
      printf("%c", *ptr);
    }
    printf("\n");
  }

  return 0;
}
```

The line

```
char *bike[] = { "Ducati", "Moto Guzzi", "Laverda" };
```

declares and array of pointers to *char*, and assigns to each pointer in the array the address of the first element in each string.

In

```
int i;
char *ptr;
```

i indexes the array and is used to refer to each element in the array. *ptr* is a pointer to *char* used to refer to each character in each string.

The outer loop looks at each pointer in the array in turn.

```
  for (i = 0;i < 3; i++) {
     ...
    printf(" ");
  }
```

In the inner loop, the *ptr* is initialised with an array pointer, so *ptr* refers to a string. Then we march *ptr* along the string printing out each character in turn, until we hit the null character.

```
  ...
    for (ptr = bike[i]; *ptr != '\0'; ptr++) {
      printf("%c", *ptr);
    }
  ...
```

So you can see that programs *table.c* and *arrayofptrs.c* both do exactly the same job: store and print a string.

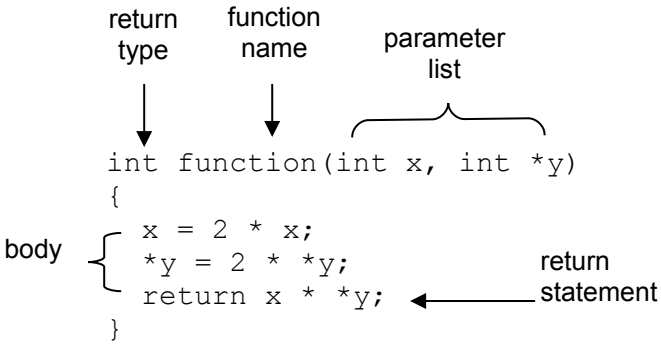
Exercise 5

1 Create a table of three rows and five columns that stores integers. Find the sum of all the integers.

Functions

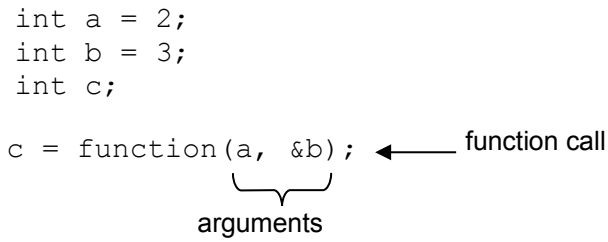
Functions are the fundamental building block of all C programs. A function usually performs one small task.

A function has a name, a return type, a parameter list, a body and a return statement.



Data is passed to a function for processing via its parameter list. A parameter acts like a local variable. The function shown above has two parameters: an integer, *x*, and a pointer to an integer, ***y***.

Parameters receive their values from arguments supplied when the function is called.

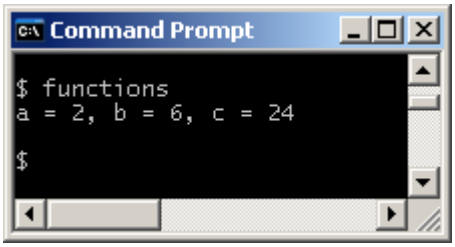


Arguments must match their parameters, both in their order and their type. The first parameter is an *int* and so the first argument must be an *int*. The second parameter is a pointer to an *int* and so the second argument must be the address of an *int*.

The value returned by a function is returned in the function name. Here, the value returned in the function name is assigned to the *int* variable *c*.

Here is the entire program.

```
/* functions.c - illustrates the basics of functions */

#include <stdio.h>

int function(int x, int *y)
{
  x = 2 * x;
  *y = 2 * *y;
  return x * *y;
}


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

  c = function(a, &b);
  printf("a = %d, b = %d, c = %d\n", a, b, c);
  return 0;
}
```



The argument values supplied to the function are *a* = 2 and the address of *b*. The value of variable *b* is 3. 2 is passed to *x*. The address of *b* is passed to *y*.

In the function the value of *x* is multiplied by 2; this has no affect on the value stored in the variable *a*. The value at the address stored in *y* is also multiplied by 2; this has a direct affect on the value stored in variable *b*.

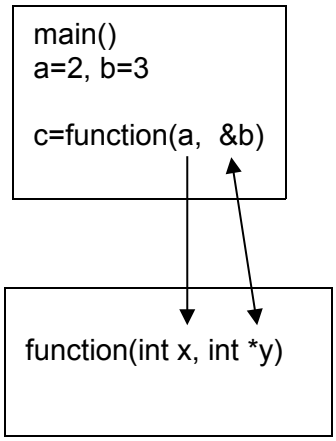


Fig 12 Arguments and Parameters

Exercise 6

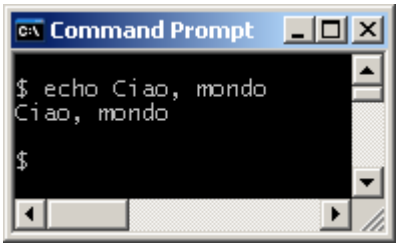
1 Explain each line of *function(int x, int *y)* shown above.
2 Design, write and test a function, named *doubleValue*, that, when given an integer as an argument value, returns the integer doubled both in a parameter and as a return value.

Command Line Arguments

Command line arguments allow the user, often not a programmer, to supply arguments to a program for processing at run time. For example, the user might supply a filename to a program, and the program then creates a back up of the file with the given name. But first we illustrate the basic principles of using command line arguments.

main() is a C function, and, like any other C function, can have a parameter list. The arguments are usually named *argc* and *argv*. *argc* stands for argument count. *argv* stands for argument vector. *argc* is of type *int*. *argv* is of type pointer to an array of strings.

The program shown below echoes its command line arguments to the screen.



Here, the program name is *echo*. *Ciao,* and *mondo* are arguments to the program.

```
/* echo.c - echoes command line arguments */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  for (i = 1; i < argc; i++)
    printf("%s ", argv[i]);
  printf("\n");
  return 0;
}
```

There are three argument values altogether: the program name, *echo*, *Ciao* and *mondo*. All three arguments are stored in the array *argv[]* and the argument count is three.

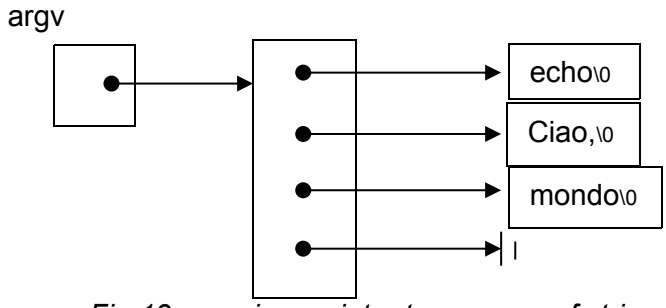


Fig 13 argv is a pointer to an array of strings

Here, *argv[0] == "echo"*, *argv[1] == "Ciao"* and *argv[2] == "mondo"*. *argv[3]* is the null pointer.

An alternative version of *echo.c*, which manipulates pointers, is shown below.

```
/* echo2.c - echoes command line arguments */

#include <stdio.h>

int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", *++argv);
  printf("\n");
  return 0;
}
```

Exercise 7

1 Explain each line of the program *echo2.c* shown above.
2 Design, write and test a C program named *backupfile* that receives a filename as a program parameter and creates a copy of the given file.

Answers

Exercise 2

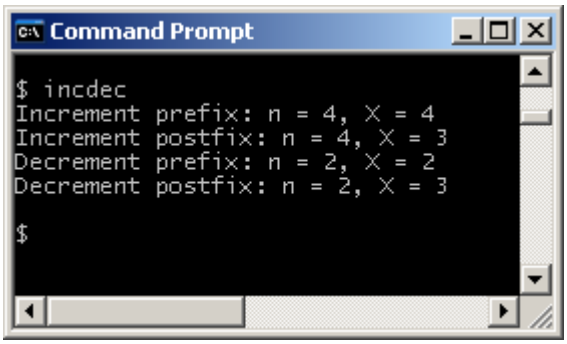
1

```
/* ptrtochar.c */

#include <stdio.h>

int main()
{
  char c = 'X';
  char *cptr = &c;

  printf("Value stored in variable c: %c\n", c);
  printf("Value referenced by pointer cptr: %c\n", *cptr);
  return 0;
}
```

Exercise 3

1

```
/* incdec.c */

#include <stdio.h>

int main()
{
  int n;
  int x;

  printf("Increment prefix: ");
  n = 3, x = ++n;
  printf("n = %d, X = %d\n", n, x);

  printf("Increment postfix: ");
  n = 3; x = n++;
  printf("n = %d, X = %d\n", n, x);

  printf("Decrement prefix: ");
  n = 3, x = --n;
  printf("n = %d, X = %d\n", n, x);

  printf("Decrement postfix: ");
  n = 3, x = n--;
  printf("n = %d, X = %d\n", n, x);

  return 0;
}
```



Exercise 4

1

```
/* intptr.c */

#include <stdio.h>

int main()
{
  int num[] = { 1, 2, 3, 4, 5 };
  int *ptr = num;
  int i;
  int sum = 0;
```

```
  for (i = 0; i < 5; i++) {
    sum = sum + *ptr;
    ptr++;
  }
  printf("Sum 1+2+3+4+5 = %d\n", sum);

  return 0;
}
```

Exercise 5

1

```
/* numtable.c */

#include <stdio.h>

int main()
{
  int num[3][5] = { { 1, 2, 3, 4, 5 },
                    { 1, 2, 3, 4, 5 },
                    { 1, 2, 3, 4, 5 } };
  int row, col;
  int sum = 0;

  for (row = 0; row < 3; row++) {
    for (col = 0; col < 5; col++) {
      sum = sum + num[row][col];
    }
  }
  printf("Expected sum = 45.  Actual sum = %d\n", sum);
  return 0;
}
```

Exercise 6

1 *int function(int x, int *y)* is the function header. Its name is *function*. Its return type is *int* (for integer, a whole number). It has two parameters. The first one, an *int*, is named *x*. The second one, named *y*, is a pointer to an *int*. That means its value is the address of an int variable. The function could be called something like this:

int a = 2;
int b = 3;
c = function(a, &b);

The function body is contained by the outermost pair of braces, { and }.

*x = 2 * x;* means multiply the value held in the parameter *x* by 2 and place the result in *x* (thereby overwriting the original value).

**y = 2 * *y;* means multiply the value stored in variable whose address is in the pointer *y* with 2 and place the result in the variable whose address is stored in *y* (thereby overwriting the original value).

*return x * *y;* means multiply the value stored in the variable whose address is in the pointer variable *y* by the value stored in *x* and return the result.

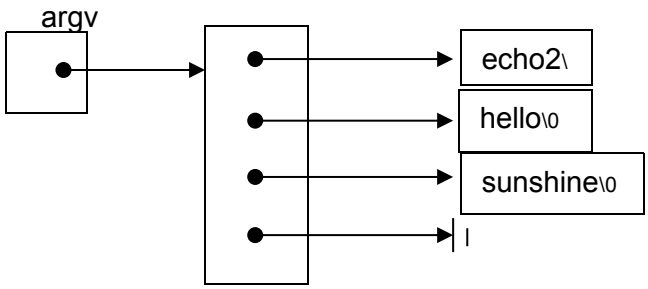
2
```
int doubleValue(int *x)
{
  *x = 2 * *x;
  return *x;
}
```

Exercise 7

1 */* echo2.c - echoes command line arguments */* is a comment written for the benefit of the person who has to read the program listing. It shows the program filename, echo2.c, along with a description of what the program does.

#include <stdio.h> is an instruction to the compiler (a program that translates human readable text into binary form and combines appropriate library code) to include the contents of the file named *stdio.h* with the program. *stdio.h* includes the definition of the *printf()* function.

*int main(int argc, char *argv[])* is the function header. This function is named *main*. Every executable C program has a function named *main*. Program execution starts with the *main()* function. The function returns an *int*, a whole number. The function has two parameters named *argc* and *argv[]*. *argc* is an *int* and contains the number of values held in *argv[]*. *argv[]* is a pointer to an array of strings, where a string is a sequence of characters. If the program is launched with *echo2 hello sunshine*, *argv[]* contains



while *(--argc > 0)* says subtract one from *argc*, then, if *argc* is more than zero, loop. Subtract one from *argc* every time round the loop.

The loop body contains *printf("%s ", *++argv)*. It says point to the next string then print it.

2
```
/* backupfile.c - creates a duplicate copy of the given file.
   Usage: backupfile filename.ext.  Creates file with name
   filename.ext.bak,a duplicate copy of filename.ext.  File
   must be a text file.
*/

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

void error(char *message)
{
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}
```

```
int main(int argc, char *argv[])
{
  FILE *outfile, *infile;
  char *outfilename;
  char c;

  if (argc != 2)
    error("usage: backupfile filename.ext\n");

  infile = fopen(argv[1], "r");
  if (infile == NULL)
    error("cannot open file to be copied from");

  outfilename = strcat(argv[1], ".bak");

  outfile = fopen(outfilename, "w");
  if (outfile == NULL)
    error("cannot open file to be copied to");

  while ((c = fgetc(infile)) != EOF)
    fputc(c, outfile);

  fclose(infile);
  fclose(outfile);
  return 0;
}
```