C Language Programming for Microcontrollers

Cuauhtémoc Carbajal
ITESM CEM
19/08/2014

Objectives

- After completing this chapter you should be able to:
  - explain the overall structure of a C language program
  - use appropriate operators to perform desired operations in C language
  - understand the basic data types and expressions of C language
  - write program loops
  - write functions and make subroutine calls in C language
  - use arrays and pointers for data manipulation
  - perform basic I/O operations in C language
  - use Code::Blocks to compile your C programs
  - use the Keil uVision integrated development environment

Introduction to C

- C has gradually replaced assembly language in many embedded applications.
- Books on C language
  - Kernighan & Ritchie, "The C Programming Language", Prentice Hall, 1988.
  - Deitel & Deitel, "C: How to Program", Prentice Hall, 1998.
  - Kelly & Pohl, "A Book on C: Programming in C", Addison-Wesley, 1998.
- A C program consists of functions and variables.
  - A function contains statements that specify the operations to be performed.
    - Types of statements:
      - Declaration
      - Assignment
      - Function call
      - Control
      - Null
  - A variable stores a value to be used during the computation.
- The `main()` function is required in every C program.

AC Program Example

```
#include <stdio.h>                          /* causes the file stdio.h to be included */
int main (void)                                    /* program execution begins */
{                                           /* marks the start of the program */
   int a, b, c;                             /* declares three integer variables a, b, and c */
   a = 3;                                   /* assigns 3 to variable a */
   b = 5;                                   /* assigns 5 to variable b */
   c = a + b;                               /* assigns the sum of a and b to variable c */
   printf (" a + b = %d\n", c);             /* prints the string a + b = followed by value of c */
   return 0;                                /* returns 0 to the caller of main( ) */
}                                           /* ends the main( ) function */
```

- Types, operators, and expressions
  - Variables must be declared before they can be used.
  - A variable declaration must include the name and type of the variable and may optionally provide its initial value.
  - The name of a variable consists of letters and digits.
  - The underscore character "_" can be used to improve readability of long variables.

- Data types
  - C has five basic data types: **void**, ***char***, ***int***, ***float***, and ***double***.
  - The ***void*** type represents nothing and is mainly used with function.
  - A variable of type ***char*** can hold a single byte of data.
  - A variable of type ***int*** is an integer that is the natural size for a particular machine.
  - The type ***float*** refers to a 32-bit single-precision floating-point number.
  - The type ***double*** refers to a 64-bit double-precision floating-point number.
  - Qualifiers ***short*** and ***long*** can be applied to integers. Typically ***short*** size is half the integer size and ***long*** *size* is twice the integer size.
  - Qualifiers ***signed*** and ***unsigned*** may be applied to data types ***char*** and ***integer***.
- Declarations
  - A declaration specifies a ***type***, and contains a list of one or more variables of that type.

int_width.c

Constants on my computer

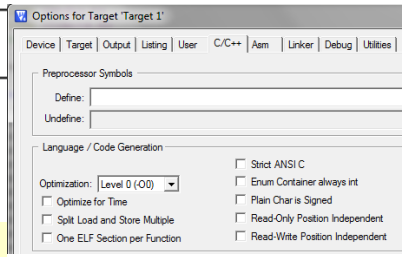
```
#include <stdio.h>
#include <limits.h>
#include <float.h>
int main (void)
{
   printf("bits in char ................. %d\n",CHAR_BIT);
   printf("max value of char ............ %d\n",CHAR_MAX);
   printf("min value of char ............ %d\n",CHAR_MIN);
   printf("max value of int ............. %d\n",INT_MAX);
   printf("min value of int ............. %d\n",INT_MIN);
   printf("max value of long ............ %d\n",LONG_MAX);
   printf("min value of long ............ %d\n",LONG_MIN);
   printf("max value of signed char ..... %d\n",SCHAR_MAX);
   printf("min value of signed char ..... %d\n",SCHAR_MIN);
   printf("max value of short ........... %d\n",SHRT_MAX);
   printf("min value of short ........... %d\n",SHRT_MIN);
   printf("max value of unsigned char ... %u\n",UCHAR_MAX);
   printf("max value of unsigned int .... %u\n",UINT_MAX);
   printf("max value of unsigned long ... %u\n",ULONG_MAX);
   printf("max value of unsigned short .. %u\n",USHRT_MAX);

   printf("min value of a float.......... %e\n",FLT_MIN);
   printf("max value of a float.......... %e\n",FLT_MAX);
   printf("min value of a double......... %e\n",DBL_MIN);
   printf("max value of a double......... %e\n",DBL_MAX);

   system("pause");

   return 0;
}
```

data_width.c

Keil data types for ARM C and C++:

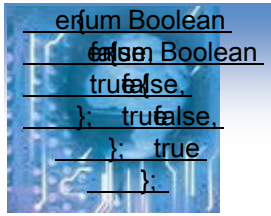
Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)
float	32	4 (word-aligned)
double	64	8 (doubleword-aligned)
long double	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
bool (C++ only)	8	1 (byte-aligned)
_Bool (C only[a])	8	1 (byte-aligned)
wchar_t (C++ only)	16	2 (halfword-aligned)

[a] `stdbool.h` can be used to define the `bool` macro in C.



NOTE: Plain type char is unsigned.

http://www.keil.com/support/man/docs/armccref/armccref_Babfcgfc.htm



enum

enum Color { red, orange, yellow, green, blue, indigo, violet };

- This declaration defines a new type, Color, and seven unalterable variables of that type. For example,

Color c = green;

- Declares the variable c to be of type Color with initial value green. C initializes each name in the enumarator list with an integer number code, starting with 0 and increasing by 1 going left to right.

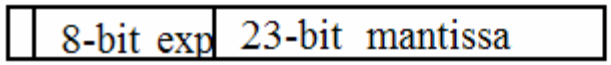
```
enum Boolean
{
  false,
  true
};
```

enum_week.c

Floating-Point Types

Type	Default Format	Default Value Range	
		Min	Max
float	IEEE32	-1.17549435E-38F	3.402823466E+38F
double	IEEE32	1.17549435E-38F	3.402823466E+38F
long double	IEEE32	1.17549435E-38F	3.402823466E+38F
long long double	IEEE32	1.17549435E-38F	3.402823466E+38F

IEEE 32-bit Format (Precision: 6.5 decimal digits)



sign bit

$$value = -1^{S} * 2^{(E-127)} * 1.m$$

- Examples of declarations
  - `int  i, j, k;`
  - `char cx, cy;`
  - `int m = 0;`
  - `char echo = 'y';` /* the ASCII code of letter y is assigned to variable *echo.* */
- Constants
  - Types of constants: integers, characters, floating-point numbers, and strings.
  - A character constant is written as one character within single quotes, such as **'x'**.
  - A character constant is represented by the ASCII code of the character.
  - A string constant is a sequence of zero or more characters surrounded by double quotes, as **"MC9S12C32 is made by Motorola"** or **""**, which represented an empty string. Each individual character in the string is represented by its ASCII code.
  - An integer constant like **1234** is an ***int***. A long constant is written with a terminal ***l*** or **L**, as in **44332211L**.

- A number in C can be specified in different bases.
- The method to specify the base of a number is to add a prefix to the number:

base	prefix	example
decimal	none	1234
octal	0	04321
hexadecimal	0x	0x45
binary	0b	0b11110000

Tip: If you want to have a decimal number, don't write a `'0'` at the beginning.

Arithmetic Operators

+	add and unary plus	
-	subtract and unary minus	
*	multiply	
/	divide	// truncate quotient to integer when both operands are integers.
%	modulus (or remainder)	// cannot be applied to float or double
++	increment (by 1)	
--	decrement (by 1)	

Bitwise Operators (1 of 2)

- C provides six operators for bit manipulations; they may only be applied to integral operands.

&	AND
|	OR
^	XOR
~	NOT
>>	right shift
<<	left shift

- **&** is often used to clear one or more bits of an integral variable to 0.

```
PTH = PTH & 0xBD;      /* clears bit 6 and bit 1 of PTH to 0 (PTH is of type char) */
```

- | is often used to set one or more bits to 1.

```
PTB = PTB | 0x40;  /* sets bit 6 to 1 (PTB is of type char) */
```

- **^** can be used to toggle a bit.

```
abc  = abc ^ 0xF0;     /* toggles upper four bits (abc is of type char) */
```

Bitwise Operators (2 of 2)

- **>>** can be used to shift the involved operand to the right for the specified number of places.

```
xyz  = xyz >> 3;          // shift right 3 places
```

- **<<** can be used to shift the involved operand to the left for the specified number of places.

```
xyz  = xyz << 4;          // shift left 4 places
```

- The assignment operator **=** is often combined with the operator. For example,

```
PTH  &= 0xBD;
PTB  |= 0x40;
xyz  >>= 3;
xyz  <<= 4;
```

biitwise_operators.c

What is the output of this C code?

```
#include <stdio.h>
int main()
{
  int c = 2 ^ 3;
  printf("%d\n", c);
}
```

a) 1
b) 8
c) 9
d) 0

What is the output of this C code?

```
#include <stdio.h>
int main()
{
  signed int a = 10;
  a = ~a;
  printf("%d\n", a);
}
```

a) -9
b) -10
c) -11
d) 10

Relational and Logical Operators (1 of 2)

- Relational operators are used in expressions to compare the values of two operands.
  - The value of the expression is 1 when the result of comparison is true. Otherwise, the value of the expression is 0.
- Relational and logical operators:

==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
&&	and
||	or
!	not

conditional_operators.c

Relational and Logical Operators (2 of 2)

- Examples of relational and logical operators:

```
if (!(ATD0STAT0 & 0x80))
      statement1;      /* if bit 7 is 0, then execute statement1 */
if (i > 0 && i < 10)
      statement2;      /* if 0 < i < 10 then execute statement2 */
if (a1 == a2)
      statement3;      /* if a1 == a2 then execute statement3 */
```

- Control flow
  - The control-flow statements specify the order in which computations are performed.
  - Semicolon is a statement terminator.
  - Braces **{** and **}** are used to group declarations and statements together into a *compound statement,* or *block.*

If-Else Statement

```
if (expression)
   statement1
else                          -- The else part is optional.
   statement2
```

Example,

```
if (a != 0)
   r = b;
else
   r = c;
```

What is the output of this C code?

```
#include <stdio.h>
int main()
{
  if (7 & 8)
    printf("Honesty");
  if ((~7 & 0x000f) == 8)
    printf("is the best policy\n");
}
```

a) Honesty is the best policy
b) Honesty
c) is the best policy
d) No output

What is the output of this C code?

```
#include <stdio.h>
int main()
{
  int i = 1;
  if (i++ && (i == 1))
    printf("Yes\n");
  else
    printf("No\n");
}
```

a) Yes
b) No
c) Depends on the compiler
d) Depends on the standard

Multiway Conditional Statement

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
…
else
    statementn
```

Example,

```
if (abc > 0) return 5;
else if (abc == 0) return 0;
else return -5;
```

Multiway Conditional Statement (Example)

```
#include <stdio.h>

int main()                                          /* Most important part of the program! */
{
   int age;                                         /* Need a variable... */

   printf( "Please enter your age: " );             /* Asks for age */
   scanf( "%d", &age );                             /* The input is put in age */
   if ( age < 21 ) {                                /* If the age is less than 100 */
      printf ("You are pretty young!\n" );          /* Just to show you it works... */
  }
  else if ( age == 21 ) {                           /* I use else just to show an example */
     printf( "You are old.\n" );
  }
  else {
    printf( "You are really old.\n" );              /* Executed if no other statement is */
  }
  system("pause");
  return 0;
}
```

```
A. !( 1 || 0 )
B. !( 1 || 1 && 0 )
C. !( ( 1 || 0 ) && 0 )
```

For-Loop Statement

```
for (expr1; expr2; expr3)
     statement;
```

where, *expr1* and *expr2* are assignments or function calls and *expr3* is a relational
expression.

Example

```
sum = 0;
for (i = 1; i < 10; i++)
   sum += i * i;

for (i = 1; i < 20; i++)
   if (i % 2) printf("odd");
```

For-Loop Statement (Example)

```
#include <stdio.h>

int main()
{
    int x;
    /* The loop goes while x < 10, and x increases by one every loop*/
    for ( x = 0; x < 10; x++ ) {
    /* Keep in mind that the loop condition checks
      the conditional statement before it loops again.
      consequently, when x equals 10 the loop breaks.
      x is updated before the condition is checked. */
        printf( "%d\n", x );
    }
    getchar();
}
```

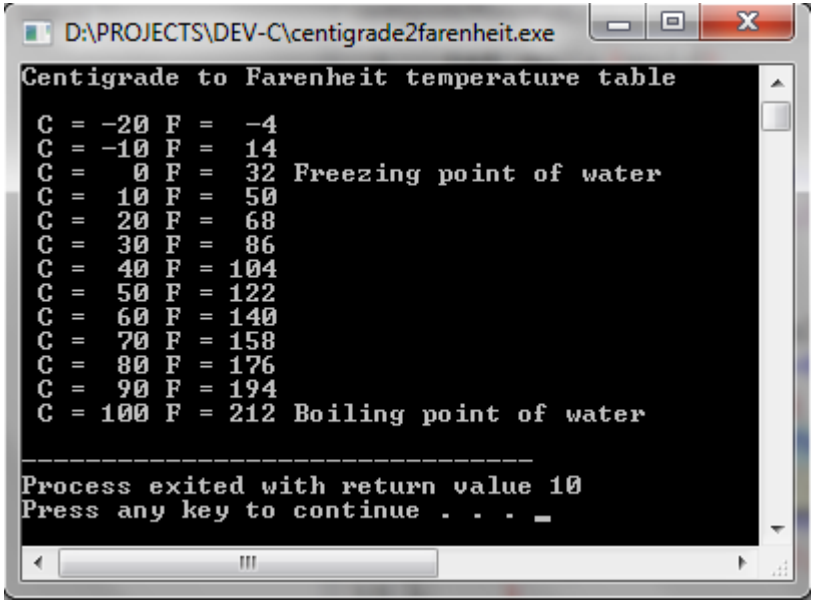
Exercise

- Find the error(s) in the following code snippet:

```
For ( x = 100, x >= 1, x++ )
printf( "%d\n", x );
```

Homework 1

- Create a C program that:
  - generates a list of centigrade and Fahrenheit temperatures and
  - prints a message out at the freezing point of water and another at the boiling point of water.



centigrade2farenheit.c

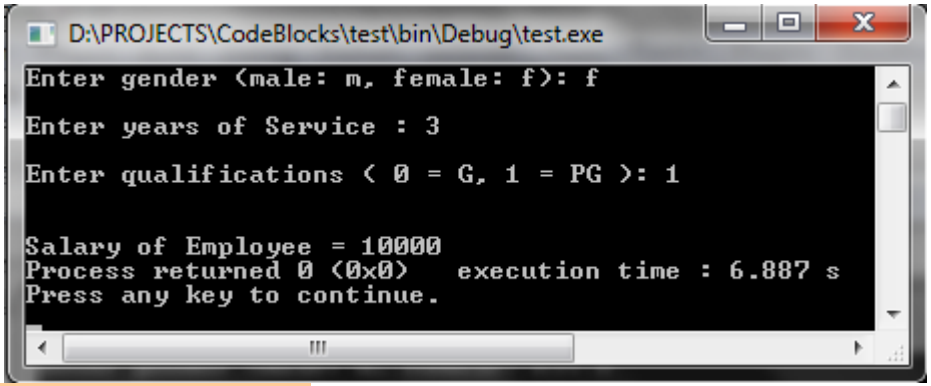
Exercise

- Create a C program to calculate the salary of an employee according to the following table:

Gender	Years of Service	Qualifications	Salary
Male	>= 10	Post-Graduate	15000
	>= 10	Graduate	10000
	< 10	Post-Graduate	10000
	< 10	Graduate	7000
Female	>= 10	Post-Graduate	12000
	>= 10	Graduate	9000
	< 10	Post-Graduate	10000
	< 10	Graduate	6000

Hint:

```
printf ( "Enter gender (male: m, female: f): " ) ;
scanf ( "%c", &g ) ;
```



salary_employee.c

Switch Statement

```
switch (expression) {
   case const_expr1:
      statement1;
      break;
   case const_expr2:
      statement2;
      break;
   …
   default:
      statementn;
}
```

Example

```
switch (i) {
   case 1: printf("*");
      break;
   case 2: printf("**");
      break;
   case 3: printf("***");
      break;
   case 4: printf("****");
      break;
   case 5: printf("*****");
      break;
}
```

While Statement

```
while (expression)
   statement
```

Example

```
int_cnt = 5;
while (int_cnt); /* do nothing while the variable int_cnt ≠ 0 */
```

Do-While Statement

```
do
   statement
while (expression);
```

Example

```
int  digit = 9;
do
  printf("%d ", digit--);
while (digit >= 1);
```

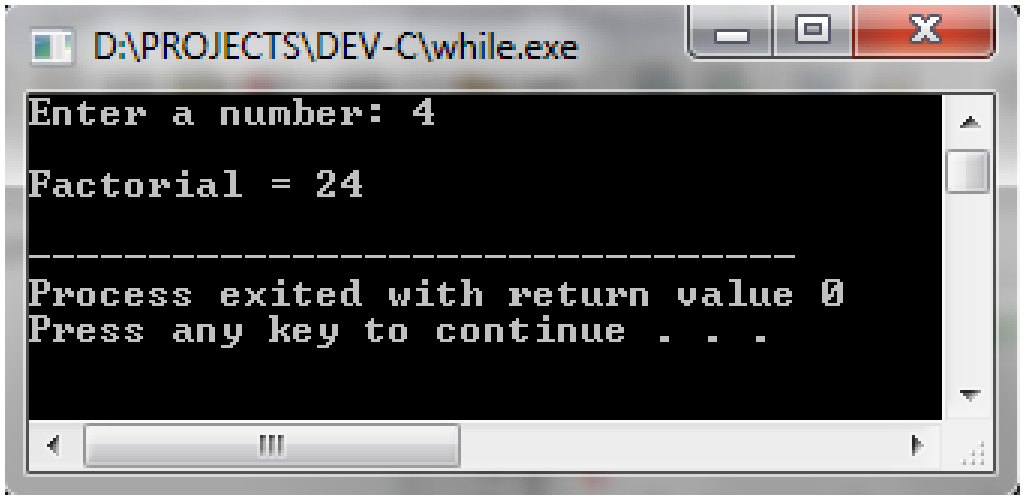
While Statement (Example)

```
#include <stdio.h>

int main()
{
  int x = 0;                    /* Don't forget to declare variables */

  while ( x < 10 ) {            /* While x is less than 10 */
      printf( "%d\n", x );
      x++;                      /* Update x so the condition can be met eventually */
  }
  getchar();
}
```


```
#include <stdio.h>

int main()
{
  int x;

  x = 0;
  do {
  /* "Hello, world!" is printed at least one time
     even though the condition is false*/
    printf( "%d\n", x );
  } while ( x != 0 );
  getchar();
}
```

Exercise

- Create a C program to find the factorial of a number.
- Note: the number is entered by user.
- Hint: factorial of n = 1*2*3*...*n



while.c


The 5th Wave
By Rich Tennant
©RICHTENNANT
Oh come on- how fatal can it be?
FATAL ERROR

Input and Output

Examples

- Not part of the C language itself.
- Four I/O functions will be discussed.

1. **int *getchar*** ( ).
   // returns a character when called

```
char xch;
xch = getchar ();
```

2. **int *putchar*** (int).
   // outputs a character on a standard output device

```
putchar('a');
```

3. **int *puts*** (**const char** *s).
   // outputs the string pointed by **s** on a standard output device

```
puts("Viva Mexico! \n");
```

4. **int *printf*** (*formatting string, arg1, arg2, …).*
   // converts, formats, and prints its arguments on the standard output device

Input and Output (Example)

```
#include <stdio.h>      /* Inserts stdio.h header file into the Pgm */
int main ( )            /* Beginning of main function.*/
{
   char in;             /* character declaration of variable in. */
   printf ("Please enter one character "); /* message to user */
   in = getchar ( ) ;   /* assign the keyboard input value to in. */
   puts("The character is ");
   putchar (in);        /* out put 'in' value to standard screen. */
   putchar ('\n');
   system ("pause");
   return 0;
}
```

Formatting String for Printf

- The arguments of ***printf*** can be written as constants, single variable or array names, or more complex expressions.
- The formatting string is composed of individual groups of characters, with one character group associated with each output data item.
  - The character group starts with **%**.
  - The simplest form of a character group consists of the percent sign followed by a *conversion character* indicating the type of the corresponding data item.
  - Multiple character groups can be contiguous, or they can be separated by other characters, including whitespace characters. These "*other characters*" are simply sent to the output device for display.
- Examples

```
printf ("This is a challenging course! \n");
printf("%d %d %d", x1, x2, x3); /* outputs x1, x2, and x3 using minimal number
                                   of digits with one space separating each value */
printf("Today's temperature is %4.1f\n", temp);
```

Escape sequences

- In printf() we can use many escape sequences and format specifiers.
- *Escape sequences* are special notations through which we can display our data

Escape Sequence	Description
\a	alert (bell) character
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab

Escape Sequence	Description
\\	backslash
\?	question mark
\’	single quote
\”	double quote

Rules for Conversion String

- Between the **%** and the conversion character there may be, in order:
  - A minus sign, -- specify left adjustment
  - A number that specifies the minimum field width
  - A period that separates the field width from precision
  - A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point, or the minimum of digits for an integer
  - An **h** if the integer is to be printed as a ***short***, or ***l*** (letter ell) if as a **long**

Table 5.3 Commonly used conversion characters for data output

Conversion character	Meaning
c	data item is displayed as a single character
d	data item is displayed as a signed decimal number
e	data item is displayed as a floating-point value with an exponent
f	data item is displayed as a floating-point value without an exponent
g	data item is displayed as a floating-point value using either e-type or f-type conversion, depending on value; trailing zeros, trailing decimal point will not be displayed
i	data item is displayed as a signed decimal integer
o	data item is displayed as an octal integer, without a leading zero
s	data item is displayed as a string
u	data item is displayed as an unsigned decimal integer
x	data item is displayed as a hexadecimal integer, without the leading 0x

```
/* fprintf example */

#include <stdio.h>

int main (void)
{
  printf ("Characters: %c %c \n", 'a', 65);
  printf ("Decimals: %d %ld\n", 1977, 650000);
  printf ("Preceding with blanks: %10d \n", 1977);
  printf ("Preceding with zeros: %010d \n", 1977);
  printf ("Some different radixes: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
  printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
  printf ("Width trick: %*d \n", 5, 10);
  printf ("%s \n", "A string");
  system ("pause");
  return 0;
}
```

sprintf( )

- Sometimes it's necessary to format a string in an array of chars.
- sprintf( ) works just like printf( ) or fprintf( ), but puts its "output" into the specified character array.
- The character array must be big enough.

```
#include <stdio.h>

int main (void) {
  char message[100];
  int myAge = 4;
  sprintf( &message[0], "I am %d years old :) \n", myAge );
  printf( "%s\n", message);
  return 0;
}
```

Functions and Program Structure

- Every C program consists of one or more functions.
- Definition of a function cannot be embedded within another function.
- A function will process information passed to it from the calling portion of the program, and return a single value.
- Syntax of a function definition:

```
return_type function_name (declarations of arguments)
{
   declarations and statements
}
```

Example

```
char  lower2upper (char cx)
{
   if (cx >= 'a' && cx <= 'z') return (cx - ('a' - 'A'));
   else return cx;
}
```

Functions… (Example)

```
#include <stdio.h>

char  lower2upper (char);

int main (void)
{
   char letra_in;
   char letra_out;
   printf("Escribe una letra: ");
   letra_in = getchar();
   letra_out = lower2upper(letra_in);
   printf ("La mayuscula correspondiente es: %c \n", letra_out);

   system ("pause");
   return 0;
}

char  lower2upper (char cx)
{
 if (cx >= 'a' && cx <= 'z') return (cx - ('a' - 'A'));
 else return cx;
}
```

Example 5.1 Write a function to test if an integer is a prime number.
Solution: A number is a prime if it is indivisible by any integer between 2 and its half.

```
/* this function returns a 1 if a is prime. Otherwise, it returns a 0. */
char test_prime (int a)
{
  int i;
  if (a == 1) return 0;
  for (i = 2; i < a/2; i++)
        if ((a % i) == 0) return 0;
  return 1;
}
```

- A function must be defined before it can be called.
- Function prototype declaration allows us to call a function before it is defined.
- Syntax of a function prototype declaration:

```
return_type function_name (declarations of arguments);
```

Example 5.2 Write a program to find out the number of prime numbers between 0 and 10.

Solution:

```
#include <stdio.h>
char test_prime (int a); /* prototype declaration for the function test_prime */
int main ( )
{
  int i, prime_count = 0;
  for (i = 0; i <= 10; i++) {
     if (test_prime(i))
         prime_count ++;
  }
  printf ("\n The total prime numbers between 100 and 1000 is %d\n", prime_count);
  system ("pause");
  return 0;
}

char test_prime (int n)
{
  int j;
  if (n == 1) return 0;
  for (j = 2; j < n/2; j++)
     if ((n % j) == 0) return 0;
  return 1;
}
```

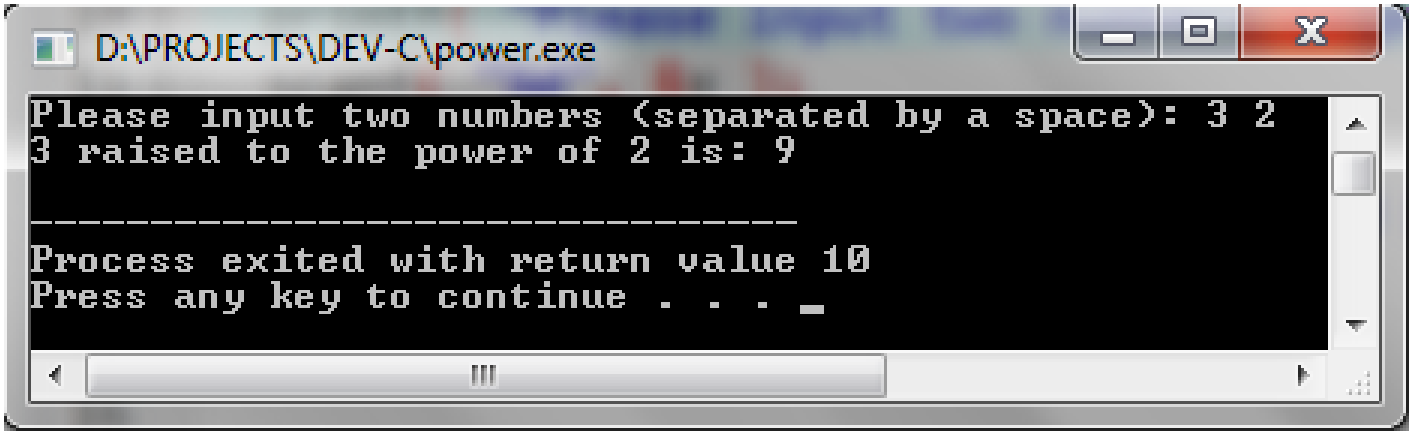
Functions… (Example)

```
#include <stdio.h>

/*Function to delay about a second */
void wait_a_second()
{
   unsigned long x;
   for (x=0;x<200000000;x++);
}

/*Start of main program */
main()
{
   char i;
   unsigned char porta;
   for(;;)
   {
      porta=~i;            /*invert*/
      i++;
      printf("i=%x\t porta=%x\r",i,porta);
      wait_a_second();     /*wait about a second*/
   }
}
```

This example counts up in binary and displays the result on eight LEDs connected to port A. The data is displayed with about 1 second delay between each output.

Exercise

- Write a program that:
  - ask the user to input two integer numbers *a* and *b*,
  - call a *custom* function (power) that return *a* raised to the power of *b*, and
  - print the result.



power.c

Pointers and Addresses (1 of 3)

- A *pointer* is a variable that holds the address of a variable.
- Pointers provide a way to return multiple data items from a function via function arguments.
- Pointers also permit references to other functions to be specified as arguments to a given function.
- Syntax for pointer declaration:

type_name *pointer_name;
Examples

```
int *ax;
char *cp;
```

- Use the *dereferencing* operator * to access the value pointed by a pointer.

```
int a, *b;
…
a = *b;                    /* assigns the value pointed by b to a */
```

Pointers and Addresses (2 of 3)

• Use the unary operator **&** to assign the address of a variable to a pointer. For example,

```
int  x, y;
int *ip;

ip = &x;        /* ip gets the address of x */
y = *ip;        /* y gets the value of x */
```

If you want to know the size of the various types of integers on your system, running the following code will give you that information.

```
/* PTRTUT1.C   16/09/09 */

#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j has the value %d and is stored at %p\n", j, &j);
    printf("k has the value %d and is stored at %p\n", k, &k);
    printf("ptr has the value %p and is stored at %p\n", ptr, &ptr);
    printf("The value of the integer pointed to by ptr is %d\n", *ptr);
    system("pause");
    return 0;
}
```

There are two “values” associated with the object k. One is the value of the integer stored there (*rvalue*) and the other the value of the memory location (*lvalue*).

Pointers and Addresses (3 of 3)

- Example 5.3 Write a bubble sort function to sort an array of integers.
- Solution:

```
void swap (int *px, int *py); /* function prototype declaration */
void bubble (int a[], int n)  /* n is the array count */
{
  int   i, j;
  for (i = 0; i < n - 1; i++)
     for (j = n - 1; j > i; j--)
        if (a[j - 1] > a[j])
            swap (&a[j - 1], & a[j]);
}

void swap(int *px, int *py)
{
  int  temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

Arrays

- An array consists of a sequence of data items that have common characteristics.
- Each array is referred to by specifying the array name followed by one or more *subscripts,* with each subscript enclosed in brackets. Each subscript is a nonnegative integer.
- The number of subscripts determines the dimensionality of the array. For example,
  - `x[i]` is an element of an one-dimensional array
  - `y[i][j]` refers to an element of a two-dimensional array
- Syntax for one-dimensional array declaration:
  - `array_name [expression];`
- Syntax for two-dimensional array declaration:
  - `data-type array_name [expr1] [expr2];`

Pointers and Arrays

- Any operations that can be achieved by array subscripting can also be done with pointers.
- The pointer version will in general be faster but, somewhat harder to understand.
  - For example,

```
int   ax[20];  /* array of 20 integers */
int   *ip;     /* ip is an integer pointer */

ip = &ax[0];   /* ip contains the address of ax[0] */

ip = ax;       /* ip contains the address of ax[0] */

x = *ip;       /* copy the contents of ax[0] into x */
```

- If **ip** points to ax[0], then **(ip + 1)** points to ax[1], and **(ip + i)** points to ax[i], etc.

Passing Arrays to a Function

- An array name can be used as an argument to a function.
- To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call.
- When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets.
- When declaring a two-dimensional array as a formal argument, the array name is written with two pairs of empty square brackets.

```
int  average (int n, int arr[ ]);
main ( )
{
   int n, avg;
   int arr[50];
   …
   avg = average (n, arr);   /* function call with array name as an argument */
   …
}
int average (int k, int brr [ ])     /* function definition */
{
   …
}
```

Passing Arrays to a Function (Example)

```
#include <stdio.h>
#include <stdlib.h>
int average (int k, int brr[]);     /* average function prototype */
int main(void)
{
    int n = 5;
    int arr[]={10,20,30,40,50};
    int avg;
    int i;
    // printf("arr address: %p\n",&arr[0]);
    avg = average(n,arr);
    printf("\n average=\t%3d\n\n",avg);
    system("pause");
    return 0;
}
int average(int k, int brr[])       /* average function definition */
{
    int sum = 0;
    int i;
    for (i=0;i<k;i++)
      sum += brr[i];
    printf("\n sum=\t\t%3d\n",sum);
    return (sum / k);
}
```

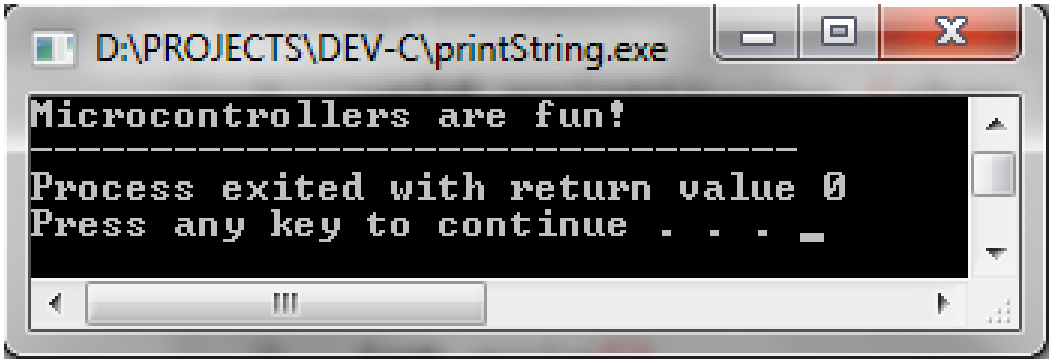
```
/* PTRTUT2.C   01/03/07 */

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

int my_array[] = {1,23,17,4,-5,100};
int *ptr, *ptr2;

int main(void)
{
    int i;
    ptr = &my_array[0];     /* point our pointer to the first
                                       element of the array */
    ptr2 = my_array;

    for (i = 0; i < 6; i++)
    {
      printf("my_array[%d] = %d   ",  i, my_array[i]);       /*<-- A */
      printf("\t (ptr + %d) = %d  ",  i, *(ptr + i));        /*<-- B */
      printf("\t (ptr2 + %d) = %d\n", i, *(ptr2 + i));       /*<-- C */
    }
    system("pause");
    return 0;
}
```

Exercise

- Write a program that:
  - declare an array as follows:

```
char message[] = "Microcontrollers are fun!";
```

  - call a *custom* function that get character by character of the array and print it using the function putchar().



pointer.c
printString.c

Homework 2a

- For each of the following, write a single C language statement that performs the indicated task. Note: value 1 and value2 have been defined as follows:

```
long value1 = 200000; long value2;
```

a. Define the variable lPtr to be a pointer to an object of type long.
b. Assign the address of variable value1 to pointer variable lPtr.
c. Print the value of the object pointed to by lPtr.
d. Assign the value of the object pointed to by lPtr to variable value2.
e. Print the value of value2.
f. Print the address of value1.
g. Print the address stored in lPtr. Is the value printed the same as the address of value1?

Homework 2b

Write a C program to convert a string to uppercase using a pointer.

Template

```
/* Converting a string to uppercase using a pointer */

#include <stdio.h>
#include <ctype.h>

void convert2Uppercase( );

int main( void )
{
  char string[] = "Thomas Alva Edison (1847-1931)";

  printf( "The string before conversion is: %s", string );
  convertToUppercase( string );
  printf( "\nThe string after conversion is: %s\n", string );
  return 0;
}
```

```
void convert2Uppercase( )
{
  while ( )
  { /* current character is not '\0' */

    if ( )
    {  /* if character is lowercase, */
       /* convert to uppercase */
    }  /* end if */

    /* move sPtr to the next character */
  } /* end while */
} /* end function convertToUppercase */
```

```
The string before conversion is: Thomas Alva Edison (1847-1931)
The string after conversion is:  THOMAS ALVA EDISON (1847-1931)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

Hint: use the functions islower and toupper.
http://www.programiz.com/c-programming/library-function/ctype.h/islower
http://www.programiz.com/c-programming/library-function/ctype.h/toupper

convertToUppercase.c

Homework 3

STM32 Programming in C

http://homepage.cem.itesm.mx/carbajal/Microcontrollers/ASSIGNMENTS/homework/STM32%20Programming%20in%20C.docx

Structures

- A group of related variables that can be accessed through a common name
- Each item within a structure has its own data type, which can be different.
- The syntax of a structure is as follows:

```
struct struct_name {                    /* struct_name is optional */
      type1 member1;
      type2 member2;
      …
};
```

The struct_name is optional and if it exists, defines a structure tag that defines a type. Example:

```
struct sample
{
     int a;
     char b;
 }
```

This structure is named *sample*.

The following line declares a structure variable named **s1**. The structure it uses is of the type defined as sample:

```
struct sample s1;
```

Suppose that you're writing a game and need some way to track the characters inside the game. Consider the following structure:

```
struct character
{
        char name[10];
        long score;
        int strength;
        int x_pos;
        int y_pos;
}
```

This structure is named **character**. It contains variables that describe variable attributes of a character in the game: the character's name, score, strength, and location on the game grid.

To define four characters used in the game, the following declarations are needed:

```
struct character g1;
struct character g2;
struct character g3;
struct character g4;
```

Or:

```
struct character g1, g2, g3, g4;
```

Items within the structure are referred to by using *dot notation*. Here's how the name for character **g1** are displayed:

```
printf("Character 1 is %s\n",g1.name);
```

Suppose that character **g2** is decimated by a sword thrust:

```
g2.strength -= 10;
```

This statement subtracts 10 from the value of **g2.strength**, the strength integer in character **g2**'s structure.

Bitfields in a C Struct

C allows you to declare bitfields within a structure, such as:

```
typedef struct {
        unsigned int bit0:1;
        unsigned int bit1:1;
        unsigned int bit2:1;
        unsigned int bit3:1;
        unsigned int bit4:1;
        unsigned int bit5:1;
        unsigned int bit6:1;
        unsigned int bit7:1;
} IOREG;
#define PORTB(*(IOREG *) 0x01)
…
int i = PORTB.bit0; // read
…
PORTB.bit0 = 1      // write
```


```
// Structure example
#include <stdio.h>
#include <conio.h>

struct student {
  int id;
  char *name;
  float percentage;
} student1, student2, student3;

int main() {
  struct student st;
  student1.id=1;
  student2.name = "Angelina";
  student3.percentage = 90.5;
  printf(" Id is: %d \n", student1.id);
  printf(" Name is: %s \n", student2.name);
  printf(" Percentage is: %f \n", student3.percentage);
  getch();
  return 0;
}
```

```
// Define struct overlay
typedef struct
{
   unsigned int count;          // Offset 0x00
   unsigned int max;            // Offset 0x02
   unsigned int _reserved;      // Offset 0x04
   unsigned int flags;          // Offset 0x06
} Counter;

// Create pointer to chip base address Counter
Counter volatile * const pCounter = 0x10000000;

// Start counting
...
pCounter->max = 5000;       // eq. to (*pCounter).max = 5000;
                            // eq. to *((unsigned int  *)(0x10000000+2) = 5000;
pCounter->flags |= GO;
...
// Poll timer state
if (pCounter->flags &= DONE)
{
   ...
}
```

Unions

- A variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements.
- The syntax is as follows:

```
union  union_name {
       type-name1 element1;
       type-name2 element2;
       …
       type-namen elementn;
};
```

where, union_name is optional. When exists, it defines a *union-tag*.

- The current temperature can be represented as follows:

```
union u_tag {
       int i;
       char c[4];
} temp;
```

- A member of a union can be accessed as
  union-name.member

A union might be used to group different record layouts from the same file, or to handle a single field that could contain, for example, either numeric or character data.

```
typedef struct transaction
   {
   int       amount;
   union
     {
     int     count;
     char    name[4];
     } udata;
   char      discount;
   } Transaction;
```

The fields in this structure are referred to as follows:

```
Transaction trans;

trans.amount = 0;
trans.udata.count = 0;
trans.discount = 'N';
```

Unions can contain any types of variables, including structures.

```
typedef struct twoshorts
   {
   short smallamount1;
   short smallamount2;
   } TwoShorts;

typedef union udata
   {
   TwoShorts smallamounts;
   int       bigamount;
   } Udata;
```

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

typedef union
{
        int Wind_Chill;
        char Heat_Index;
} Condition;

typedef struct
{
        float temp;
        Condition feels_like;
} Temperature;

int main()
{
        Temperature *tmp;

        tmp = (Temperature *)malloc(sizeof(Temperature));

        printf("\nAddress of Temperature = %u", tmp);
        printf("\nAddress of temp = %u, feels_like = %u",
           &(*tmp).temp,  &(*tmp).feels_like);
        printf("\nWind_Chill = %u, Heat_Index= %u\n",
           &((*tmp).feels_like).Wind_Chill,&((*tmp).feels_like).Heat_Index);
        getchar();
        return 0;
}
```

External Variables

- A variable declared inside a function is called an *internal variable.*
- A variable defined outside of a function is called an *external variable.*
- An external variable is available to many functions.
- External variables provide an alternative to function arguments and return values for communicating data between functions.
- Any function may access an external variable by referring to it by name if the name has been declared somewhere.
- The use of **static** with a local variable declaration inside a block or a function causes a variable to maintain its value between entrances to the block or function.

```
/*****************************************************************
SOURCE FILE ONE
*****************************************************************/
#include <stdio.h>
extern int i;                 // Reference to i, defined below
void next( void );            // Function prototype

int main() {
   i++;
   printf( "%d\n", i );       // i equals 4
   next();
}

int i = 3;                    // Definition of i

void next( void ) {
   i++;
   printf( "%d\n", i );       // i equals 5
   other();
}

/*****************************************************************
SOURCE FILE TWO
*****************************************************************/
#include <stdio.h>
extern int i;                 // Reference to i in
                              // first source file

void other( void ) {
   i++;
   printf_s( "%d\n", i );     // i equals 6
}
```


```
#include <stdio.h>

void foo()
{
      int a = 10;
      static int sa = 10;

      a += 5;
      sa += 5;
      printf("a = %d, sa = %d\n", a, sa);
}

int main()
{
   int i;

   for (i = 0; i < 10; ++i)
      foo();

   getchar();
   return 0;
}
```

Scope Rules (1 of 2)

- The functions and external variables that make up a C program can be compiled separately.
- The source text of the program may be kept in several files.
- The scope of a name is the part of the program within which the name can be used.
- For a variable declared at the beginning of a function, the scope is the function in which the name is declared.
- Local (internal) variables of the same name in different functions are unrelated.
- The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled.

Scope Rules (2 of 2)

In the following program segment

```
…
void f1 (…)
{
  …
}

int a, b, c;
void f2(…)
{
  …
}
```

Variables *a*, *b*, and *c* are accessible to function f2 but not f1.

The use of external variables are illustrated in the following 2-file C programs

in file 1

```
extern int xy;
extern long arr [ ];
main ( )
{
   …
}
void foo (int abc) { … }
long soo (void) { … }
```

in file 2

```
int xy;
long arr [100];
```

Type Casting (1 of 2)

- Type casting forces a variable of some type into a variable of different type.
- The format is **(type) variable**.
- It is often used to avoid size mismatch among operands for computation.

```
long result;
int x1, x2;
…
result = x1 * x2;
```

- The value of the product of x1 and x2 would be truncated when it exceeds 16 bits.
- The solution to the problem is to use type casting:

```
result = ((long)x1) *((long)x2);
```

- Type casting also allows one to treat a variable of certain type as a variable of different type to simplify the computation.

Type Casting (2 of 2)

For the following declaration:

```
struct personal {
        char name[10];
        char addr [20];
        char sub1[5];
        char sub2[5];
        char sub3[5];
        char sub4[5];
} ptr1;
char *cp;
```

One can use type casting technique to treat the variable **ptr1** as a string:

```
cp = (char *)&ptr1;
```