

C Language Primer

BJ Furman
ME 106
Spring, 1999

Why C?

- ◆ The language of choice for serious programming
 - ❖ Operating systems
 - ❖ Application programs
 - ❖ Device drivers
- ◆ Provides an excellent balance between:
 - ❖ Power
 - Getting the computer to do what you want
 - Down to low levels
 - ❖ Economy
 - Don't have to code everything
 - Easily portable to other processors

Anatomy of a C Program

◆ Program structure:

❖ Simplest form:

```
preprocessor directives  
main()  
{  
  data declaration statements  
  executable statements  
}
```

Example

```
#include <stdio.h> /* include stdio header file */  
#define PI 3.14159 /* PI gets replaced with 3.14159 . PI is  
                  called a symbolic constant*/  
/* blank lines are ok */  
main() /* every program must have a main function */  
{ /* braces delimit compound statements */  
  float D=3.0, r ; /* D is declared as a floating point variable  
                  it is also assigned the value of 3.0. r is also a float*/  
  
  int Area; /* Area is declared an integer variable */  
  r=D/2.0; /* calculate the radius */  
  Area=PI*(D*D)/4; /* calculate the area */  
  printf("The radius is %f \n", r); /* print the radius */  
  printf("The area is %d", Area); /* print the area */  
} /* end brace */
```

Results

```
The radius is _____  
The area is _____
```

Important Ideas

- ◆ Add comments to your programs!
 - ❖ even to your variable declarations
 - ❖ blanks and lines are only for readability
- ◆ All variables must be declared prior to use
 - ❖ there are different data types, so be careful!
 - ❖ case sensitive: Area ≠ area
- ◆ Statements are the fundamental units of execution in C programs
 - ❖ usually terminated by a semicolon
 - ❖ compound statements delimited by braces

Data types and ranges

◆ Data types

Type	Subtype	Size (bytes)	Range
char	unsigned char	1	0 to 255
	char	1	-128 to 127
int	unsigned short	2	0 to 65,535
	short	2	-32,768 to 32,767
	unsigned int	2	same as unsigned short
	int	2	same as short
	unsigned long	4	0 to 4.295×10^9 approx.
	long	4	$\pm 2.147 \times 10^9$ approx.
float	float	4	$\pm 3.4 \times 10^{38}$ approx.
	double	8	$\pm 3.4 \times 10^{308}$ approx.

Numeric Data Types, cont.

◆ Numeric data types, cont.

❖ signed integer

- MSB is called the sign bit
- various ways to represent: two's complement (complement all bits, and add 1)
 - 8 bit --> -128 to 127
 - 16 bit --> -32,768 to 32,767
- need to pay attention that variables values do not exceed the limit of the data types.
 - 1 1 0 0 1 0 0 1 (201)
 - +1 0 1 1 0 0 1 1 (179)
 - [1] 0 1 1 1 1 1 0 0 (380) [] means carry bit set
 - note that as an 8 bit number, the answer is incorrect

Numeric Data Types, cont.

- ◆ Numeric data types, cont.
 - ❖ Floating point numbers (require 4 bytes)
 - range $-3.4E-38$ to $3.4E+38$
 - typically 24 bits to represent mantissa, 8 bits to represent exponent
 - note: floats require more storage space and take longer to process than integers
 - ◆ Representation of numbers
 - ❖ `i=0x6E;` means i has the value of 6E in hex format

Preprocessor Directives

- ◆ `#include <stdio.h>`
- ◆ `#include <hc11.h>`
 - ❖ `#include -->` a preprocessor directive, directs the compiler to take the header file, i.e., `stdio.h`, and insert it into the source code at the point of the include before compilation.
 - ❖ why?
 - “overhead code” (e.g., I/O, libraries of functions, math functions, etc.)
 - common code
 - avoids having to modify every program that may use that piece of code
 - must be a text file

Preprocessor Directives, cont.

- ◆ #define PI 3.14159
 - ❖ Prior to compilation, everywhere the identifier PI appears, it is replaced by 3.14159
 - ❖ PI is called a symbolic constant (it is not a variable)
 - ❖ Convention is to use all uppercase letters to distinguish symbolic constants from variables

Main()

- ◆ main()
 - ❖ C is fundamentally based on the concept of functions.
 - ❖ all C programs need at least one function called main()
 - ❖ main() is the first function to execute in your program. It is customary to place at the beginning of the program.
 - ❖ the body of the function must be enclosed in braces
 - Braces are also used to indicate the beginning and ending of several statements (compound statements)

Main(), cont.

- ◆ main(), cont.
 - ❖ sometimes you may see the main() function written using the keyword 'void' as:
 - void main(void)
 - ❖ functions that return a value must specify what kind of data is returned by the function, e.g. integer, floating point, character, double precision, etc.
 - the first void specifies that the function main() does not *return* a value.
 - the void inside the parentheses is used to indicate that the function main() has no arguments (no values are passed to the function)

Semicolons, Blanks, and Spaces

- ◆ Issues in writing C code
 - ❖ All executable statements must end with a semicolon.
 - printf ("The value of I is %d \n", I);
 - ❖ blanks and extra lines are important only for readability:

```
main()  
{  
    printf ("Hello, world \n");  
}
```

Is equivalent to

```
main(){printf ("Hello, world \n");}
```

- blanks (except those within double quotes) and extra lines are ignored.

Comments in C Code

◆ Issues in writing C code, cont.

❖ Use comments liberally!

- /* put comments between forward slashes and asterisks, like this */
 - Can go anywhere and extend over multiple lines
 - /* For example, this comment:
 - it is still a comment even though it covers two lines */

Variables

◆ Variable declaration

- ❖ Every variable used in the program must be declared (data type followed by variable name(s)):
 - Ex:

```
int i; short q, x; unsigned p1;
float k, temp;
char j;
```
 - there are other data types, but not all are supported by the compiler we will use in the lab.
- ❖ Assign values to variables using a single equal sign:

```
temp=4.2*2.0;
i=temp; /* would mean i=8 since i is an integer variable,
so be careful! */
```

Relational Operators

◆ Relational operators

- ❖ < less than
- ❖ > greater than
- ❖ <= less than or equal to
- ❖ >= greater than or equal to
- ❖ != not equal to
- ❖ == equal to (note the double equal sign to test for equality! This is very different from *assignment*.)

❖ example:

```
int i, c=2, d=5;
i=c<d;          /* what is the value of i? */
```

Logical Operators

◆ Logical operators (conditional operation)

- ❖ && AND
- ❖ || OR
- ❖ ! NOT
- ❖ Precedence order !, &&, ||
- ❖ ex: what gets done here?
 - int k=7, p=3;
 - if(k>12 && p<6) /* be careful with && and || "short circuiting" */
 - { do this stuff ... ;}
 - else
 - { do other stuff ... ;}

Bit-Wise Logical Operations

- ◆ Bit-wise logical operators. These perform the logical operation bit to bit in the numbers on either side of the symbol.
 - ❖ & bit-wise AND
 - ❖ | bit-wise OR
 - ❖ ^ bit-wise XOR
 - ❖ << n shift left (fast multiply by 2 raised to nth power)
 - ❖ >> n shift right (fast divide by 2 raised to nth power)
 - ❖ ex. (take a few minutes, then share with neighbor)
 - int z, q, w, r, l, x=0x2B, y=0xAB, u=0x0C;
 - z=(x & 0x80); q=(y & 0x80); /* what happens here? */
 - w=y^ 0x20; r=u<<2;

Control Structures: Looping

- ◆ FOR loops
 - ❖ have 3 parts:
 - for(*initializing statement*; *test condition*; *index adjustment*)
 - the initializing statement usually initializes a variable as an index
 - the test condition tests the variable. If true, the loop will be executed
 - the index adjustment expression either increases or decreases the index variable at the end of the loop
 - example:

```
int i;
for (i=2; i<4; i++)    /* i++ implies i= i+1 */
{
    printf("Just a test %d \n", i);
}
```

Control Structures: Selecting Among Alternatives

◆ IF statements

❖ form

- if (condition) statement1;
- else statement2; /*the “else” is optional*/

❖ example

```
if (timer < 100) /*Test for time out. If true, then enter for loop, else print Time's up! */
{
    for (i=0; i<10; i++)
    {
        :
        dacout(i, ch); /*output value of i */
        :
    }
}
else
    printf ("Time's up! \n");
```

While Loops

◆ WHILE(relational test) loops

❖ continues as long as the condition stays true

❖ ex.

- int i, b, c;
- while((i !=6 || (c>b))
 > {do stuff while the above condition is true... ;}

❖ ex.

- while(1)
 > {do stuff ... ;} /*what will this do? why is it useful? */

❖ ex.

- while(1);
 > {do stuff ... ;} /* what will happen? */

Functions

◆ Functions

- ❖ functions are like subroutines, i.e., blocks of code packaged as a separate program
 - makes your program modular
 - function modules are reusable
- ❖ can pass data by value or by address
- ❖ can return a value via “return”
- ❖ prototype your function before main()
 - this tells the compiler what to expect
 - argument names are not needed, only data types, but it's just as easy to cut-and-paste from the function call line to create the prototype

Functions, cont.

Example using a function

```
....
void dacout(float volts, int ch);    /* this is the function prototype. Note that it has two arguments, volts and ch.
                                     What is the return data type? */

void main(void)
{ int ch; float x;
  ch = 2;
  x = 2.7;
  dacout(x, ch);                    /* here is the call to the function. Note that the VALUES of x and ch are passed to
                                     function. The variables x and ch are not changed.*/
.. }
....
void dacout(float volts, int ch)    /* This is the function header, note, no semicolon. Here we've declared that the
                                     function will be passed a floating point value, "volts" and an integer value ,
                                     "ch". */
{ int x;
  .....
  .....
  x=volts/2048;
  .....
}
```

Functions, cont.

Returning a value from a function

```
....
int divide(int); /* note that only the data type of the arguments are needed in function prototype */
int divisor=2; /* note that divisor is a global variable */

void main(void)
{ int z, x=8;
  ....
  ....
  z=divide(x); /* the value of x is passed to the function, and the function's return value is assigned to z */
}
....
int divide(int y) /* function definition header, the one argument is an integer variable*/
{
  int ans;
  ans=y/divisor; /* remember divisor was declared to be a global variable */
  return (ans); /* the value of ans is returned by the function */
}
```

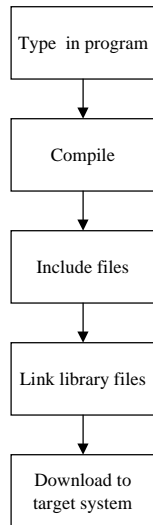
Comments on Functions

◆ Comments on Functions

❖ Local and global variables

- variables declared outside of main() are **global**, which means they are known throughout the program and functions included therein.
 - Use global variables **very** sparingly, e.g., for reserved variables like time.
 - If you inadvertently re-assign a global variable somewhere, you may have big problems!
- variables declared within a block of code (between braces, i.e., { }) are “visible” only in that block of code. They are called **local** variables.

The C Environment



C References

- ◆ Antonakos, J. L., Mansfield Jr., K. C., *Reference Guide to C and C++*, Prentice-Hall, NJ, 1999. ISBN: 0-13-956376-8
- ◆ Overland, B., *C In Plain English*, MIS Press, NY, 1995. ISBN: 1-55828-430-3