

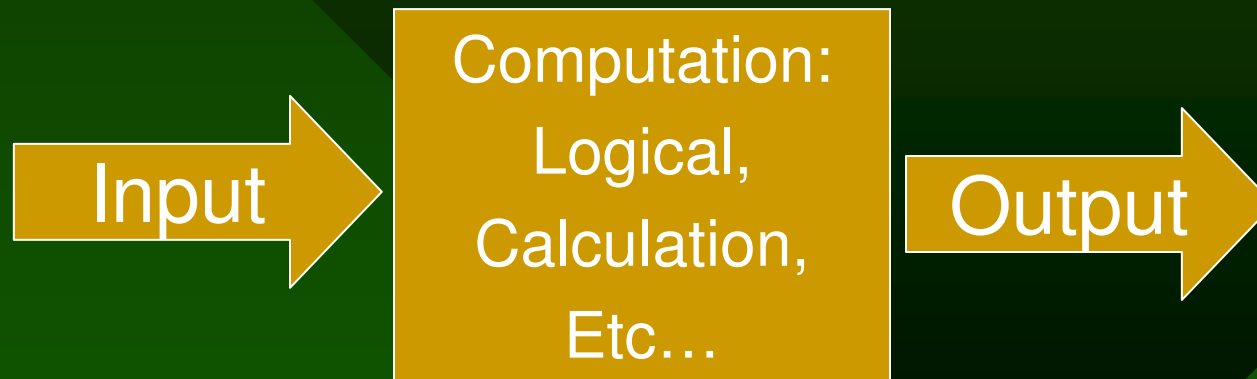
Team Ford First C Programming Workshop

Maurice Tedder

Team Ford First

What is a Computer?

- A computer accepts input; performs some computation on that input and generates output



What is a computer program?

- A list of instructions created by a computer programmer (you) that tell a computer what to do next
- Your job as a computer programmer is to:
 - Decide how to get from point A to point B using a bag of tricks (instructions)
 - For a computer, instructions = reserved keywords & operators arranged in a statement using C syntax

```
( )  
* / %  
+ -  
< <= > >=  
== !=  
=
```

• A (what I want the program to do)

C vocabulary

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while



• B

(program doing what I want it to do)

An Introduction to Structured Programming (or How to get from point A to point B)

- There are many different philosophies regarding how to get from A to B in computer programs here are two possibilities
 - Genetic programming: randomly mix and match operators and keywords until you get a program that does what you want it to do.
 - Can only be done by a computer
 - Requires a lot of computer processing time to generate code
 - The final result may not be clear or understandable by humans
 - This method works and has been used to solve many difficult programming problems but at Team Ford First we prefer another way!
 - Structured programming: the method we want to use

An Introduction to Structured Programming (or How to get from point A to point B)

- Structured programming philosophy states that all programs can be written using only three single entry/single exit control structures
 1. Sequence structure: is built into C and means that statements are executed sequentially one after another as written in the file
 2. Selection structure: chooses between alternative courses of action
 3. Repetition structure: tells which actions are to be repeated while some condition is true
- Programs are built in structured programming by stacking and/or nesting as many control structure building blocks as needed to solve the problem¹

“Hello World” – C program structure with comments

- Create a file:
yourFilename.c
- Create the part of the file where all the action is:
 - `void main(){
 putActionHere;
}`
- Stack and nest your action filled structured programming blocks inside main’s braces
- All C statements end with a semicolon “ ; ”

helloWorld.c file

```
/* Example 1. This is a comment. It is ignored  
by the compiler but it helps us humans  
explain and remember things under stress  
(competition). Also provides a trail of  
information about code. */  
#include "printf_lib.h"  
void main( )  
{  
    /* Displays Hello World text in the IFI  
terminal window */  
    printf("Hello World \n");  
} /* end of main – program ends here */
```

Variables and data types Part I

Variables are “symbols and names that contain values you can store and modify”². To us a variable is a name but to the computer it is a memory location in the computer.

- Variable names can be any series letters, digits, and underscores but cannot begin with a digit or be keywords
- Variable names are case sensitive and only the first 31 characters are recognized in C
- The keyword **extern** tells the compiler that a variable has been declared in another file.
- Use descriptive variable naming convention:
firstwordSecondword

Table of C data types⁴

Type	Size	Minimum	Maximum
char	8 bits	-128	127
signed char	8 bits	-128	127
unsigned char	8 bits	0	255
int	16 bits	-32768	32767
unsigned int	16 bits	0	65535
short	16 bits	-32768	32767
unsigned short	16 bits	0	65535
short long	24 bits	-8,388,608	8,388,607
unsigned short long	24 bits	0	16,777,215
long	32 bits	-2,147,483,648	2,147,483,647
unsigned long	32 bits	0	4,294,967,295
float	32 bits	2^{-126}	2^{128}
double	32 bits	2^{-126}	2^{128}

Variables and data types Part II

It is a good programming practice to initialize all variables you declare with some default value.

- Assign variables using the assignment operator: “=”
- unsigned char and unsigned int are the typical variable types used in the default code.
- The cast operator can be used to convert from one type to another but it is better not to mix types.
- struct { } is a user defined mixed type

```
/* Example 2. This is a comment. It is ignored by the
   compiler but it helps us humans explain things. */

#include "printf_lib.h"
unsigned char pwm02 = 254;
void main( )
{
    unsigned char pwm01 = 127;
    pwm01 = pwm01 + 1;

    /* Displays the value of pwm01 in the IFL terminal
       window */
    printf("pwm01 equals %d \n", (int)pwm01);

} /* end of main – program ends here */
```

Arithmetic and logical operators Part I

The arithmetic operators (), *, /, %, +, - follow these rules of evaluation

- First: ()
- Second and left to right: *, /, %
- Last and left to right: + or -

Example 1:

$2 + 6 / 2 = 5$; do you want this

$(2 + 6) / 2 = 4$; or this

$2 + 6 / 2 - 1 = 4$; do you want this

$2 + 6 / (2 - 1) = 8$; or this

Logical operators && and | | follow these rules of evaluation

- First: &&
- Second and left to right: | |

Logical AND is represented by the symbol &&

Logical OR is represented by the symbol | |

Bitwise AND is & and is often used for masking bits in a bit string (ex. $10011101 \& 1111000 = 1001000$)

Bitwise OR is | and is often used for forming bit strings (ex. $10010000 | 00001101 = 10011101$)

Logical NOT is represented by the symbol ! and negates a true or false statement. Can be used to create a toggle variable.

Arithmetic and logical operators Part II

Precedence of all operators¹:

()					left to right
++	--	+	-	(type)	right to left
*	/	%			left to right
+	-				left to right
<	<=	>	>=		left to right
==	!=				left to right
&&					left to right
					left to right
?:					right to left
=	+=	-=	*=	/=	right to left

A Logical AND expression is true only if both parts are true

A logical OR expression is true if any part is true

Example 2:

$1 < 2 \ \&\& \ 3 > 5 == \text{false}$

$1 < 2 \ \&\& \ 3 < 5 == \text{true}$

$1 < 2 \ || \ 3 > 5 == \text{true}$

$1 < 2 \ || \ 3 > 5 == \text{false}$

$(1 < 2 \ || \ 3 > 5) \ \&\& \ (1 < 2 \ \&\& \ 3 < 5) == \text{true}$

*DO NOT USE "=" (assignment operator) AS AN EQUALS TO OPERATOR IN A LOGICAL EXPRESSION. YOUR PROGRAM WILL COMPILE BUT YOUR LOGICAL EXPRESSION WILL GIVE RANDOM RESULTS. THIS IS A COMMON SOURCE OF UNDETECTABLE BUGS IN MANY PROGRAMS. USE "==" FOR "EQUALS TO".

Selection structures and logical comparisons

- C provides three types of selection structures that give the computer the ability to make decisions¹

Single-selection

```
if ( condition == true )
{
    perform this;
}
```

- When a logical expression evaluates to **true** inside the if-statement parenthesis the statements inside the braces “{ }” are executed; otherwise, the statements inside the braces are ignored

```
/* Example 3 */
#include "printf_lib.h"
unsigned char pwm02 = 254;
void main( )
{
    unsigned char pwm01 = 130;
    pwm01 = pwm01 + 1;

    if( pwm01 > 115 && pwm01 < 130){
        pwm01 = 127;
    }
    /* Displays the value of pwm01 in the IFI terminal
    window */
    printf("pwm01 equals %d \n", (int)pwm01);

} /* end of main – program ends here */
```

Double selection structures – two choices

Double-selection if-else

```
if ( condition1 == true )
{
    perform this;
} /* brackets indicate optional
statements */
else [if (condition2 == true)]
{
    do this when first if is false;
}
etc....
```

- When a logical expression evaluates to **false** inside the if-statement parenthesis the statements inside the braces “{ }” are ignored and statements following else block are executed. A maximum of one block will be executed.

```
/* Example 4 */
#include "printf_lib.h"
unsigned char pwm02 = 254;

void main( )
{
    unsigned char pwm01 = 130;
    pwm01 = pwm01 + 1;

    if( pwm01 > 115 && pwm01 < 130){
        pwm01 = 127;
    }
    else{
        /* This is the post increment operator it means pwm01 =
        pwm01 +1 */
        pwm01++;
    } /*end of if/else block*/
    /* Displays the value of pwm01 in the IFI terminal window */
    printf("pwm01 equals %d \n", (int)pwm01);

} /* end of main – program ends here */
```

Multiple selection structures

In C FALSE = 0 (zero) and TRUE = 1 (or non-zero value)

Multiple-selection

```
switch (integer constant) {  
  case constant1:  
    do something here;  
    break;  
  Case constant2:  
    do something here;  
    break;  
  etc....  
  Default:  
    do this as a default case;  
    break;  
} /* end switch */
```

- All, some or none of the case blocks will be executed
- Keyword **case** begins the block and **break** ends it.

```
/* Example 5 */  
void main( ){  
  unsigned char pwm01 = 127;  
  unsigned char pwm02 = 127;  
  unsigned char robotState = 1;  
  /* Does Case 2 ever get executed? */  
  switch (robotState){  
    case 1:  
      dig_out04 = 1;  
      dig_out05 = 1;  
      robotState = 2;  
    break;  
    case 2:  
      pwm01 = 127;  
      pwm02 = 200;  
      if(dig_in01 == 1) robotState = 3;  
    break;  
    case 3:  
      pwm01 = 200;  
      pwm02 = 127;  
    break;  
    default;  
      pwm01 =127;  
      pwm02 = 127;  
    break;} /* end switch block */  
} /* end of main – program ends here */
```

While loop repetition structures

- C also provides three types of repetition structures that repeat (loop) the specified actions while some condition is true.¹

While loop

```
while ( condition == true )  
{  
    perform this;  
}
```

- Statements between the while loop braces will be executed until the logical expression in parenthesis proves false²
- Avoid infinite loops caused by logical expression that never become false. This will cause your program to “freeze”
- The Robot Controller will also “freeze” if a section of code takes longer than 26 ms

/ Example 6 */*

```
void main( )
```

```
{
```

```
    unsigned char pwm03 = 127;
```

```
    unsigned char pwm04 = 127;
```

```
    while( dig_in01 == 0 ){ /* loop until this part proves false */
```

```
        pwm03 = 200;
```

```
        pwm04 = 200;
```

```
    } /* end while loop */
```

```
} /* end of main – program ends here */
```

Do while loop repetition structure

- Remember to use standard C indentation format to make your code easier to read and follow.

Do while

```
do
{
    perform this;
} while (condition == true );
```

- Do while is similar to a while loop except that statements between the do-while loop braces will be executed a least once before evaluating the while condition.

```
/* Example 7 */
```

```
void main( )
```

```
{
```

```
    unsigned char pwm03 = 127;
```

```
    unsigned char pwm04 = 127;
```

```
    do{
```

```
        pwm03 = 200;
```

```
        pwm04 = 200;
```

```
    } while( dig_in01 == 0 ); /* loop until this part proves false */
```

```
        pwm03 = 127;
```

```
        pwm04 = 127;
```

```
    } /* end of main – program ends here */
```

For loop repetition structure

For loop

```
for ( counter = initial value; counter <= someNumber; ++counter )  
{  
    perform this;  
}
```

- A for loop handles all the details of counter-controlled repetition automatically¹
 1. The first thing the computer does when it enters a for loop is evaluate the first expressions “counter = initial value”
 2. Then the loop condition “counter <= someNumber” is evaluate. If false the for loop exits.
 3. The expression “++counter” is evaluated last before the loop condition is checked again

In structured programming you can stack and/or nest selection and repetition structures to form your program

```
/* Example 8 */  
#include "printf_lib.h"  
void main( )  
{  
    unsigned char x = 0;  
  
    for(x = 0; x < 10; x++){  
        /* Displays the value of x in the IFI terminal window */  
        printf("x equals %d \n", (int) (x * 2) );  
    } /* end for loop */  
  
} /* end of main – program ends here */
```

Functions();

- A segment of code that you write once; use many times in many different places in your program².

```
returnType functionName( parameterType parameter, ...)  
{  
    do something;  
    return valueToReturn;  
}
```

- Functions usually return a value but this is not a requirement
- You can pass parameters to functions as arguments
- Functions make a program modular, reusable, and more manageable (divide and conquer!)
- Functions must either be defined before `main()` or defined somewhere else and provide a function prototype before `main`. This gives the function global scope (see Example 9).

Function() Examples;

- A function prototype is the signature for the function and is just like the definition but without the code block

```
/* Example 9.1 – Function without prototype */
#include "printf_lib.h"
/* Define the function here and it has global scope and you do
not need to provide a prototype */
Unsigned char myFunction(unsigned char myParameter)
{
    unsigned char x = 0;
    for(x = 0; x < 10; x++){
        /* Displays the value of x in the IFI terminal window */
        printf("x equals %d \n", (int) (x * myParameter) );
    } /* end for loop */
    return x;
} /* end function definition */

void main( ) /* Notice that main is also a function */
{
    unsigned char output = 0;

    /* This is how you use a function */
    output = myFunction(2);

    printf("output equals %d \n", (int) output );
} /* end of main – program ends here */
```

```
/* Example 9.2 – Function with prototype */
#include "printf_lib.h"
/* This is the prototype for myFunction */
Unsigned char myFunction(unsigned char myParameter);

void main( ) /* Notice that main is also a function */
{
    unsigned char output = 0;
    /* This is how you call a function */
    output = myFunction(2);
    printf("output equals %d \n", (int) output );
} /* end of main – program ends here */

/* Define the function here and it has global scope only if we
provide a prototype */
Unsigned char myFunction(unsigned char myParameter)
{
    unsigned char x = 0;
    for(x = 0; x < 10; x++){
        /* Displays the value of x in the IFI terminal window */
        printf("x equals %d \n", (int) (x * myParameter) );
    } /* end for loop */
    return x;
} /* end function definition */
```

Preprocessor

Directives listed in your code that are preceded by the # sign tell the compiler what to do before it compiles your program.

The # symbol denotes a pre-processor statement that is evaluated before the compiler takes over².

- # include <filename.h>
 - Inserts the given file name into the current file. Often used for function prototypes and #define constants listed in a .h file. Allows functions and #define constants to be edited from one place (file) for an entire project
- #define
 - Replaces a variable name with with some other string. Also know as a macro. Used in the default code to create aliases
- #ifdef and #ifndef ... #endif
 - Conditional compilation of statements between the #ifndef and #endif
- #pragma
 - Compiler specific preprocessor directives. Refer to compiler documentation description of available pragma directives.

Preprocessor examples

Lets rewrite example 9 using the preprocessor

- Now we have three separate files but the code is now modular because myFunction() can be included in other .c programs with the myFunction.h file without rewriting the function.

```
/* myFunctions.h file */
/* This is the prototype for myFunction */
#ifndef _myFunction_h_ #define _myFunction_h_
Unsigned char myFunction(unsigned char myParameter);
#endif /* prevents this file from being included more than once
```

```
/* myFuction.c file – Code for myFunction */
#include "printf_lib.h"
#include "myFunctions.h"
#define STOP_LOOP 10
Unsigned char myFunction(unsigned char myParameter)
{
    unsigned char x = 0;
    for(x = 0; x < STOP_LOOP; x++){
        /* Displays the value of x in the IFI terminal window */
        printf("x equals %d \n", (int) (x * myParameter) );
    } /* end for loop */
    return x;
} /* end function definition */
```

```
/* Example 9.3 – Header files */
#include "printf_lib.h"
/* The preprocessor will insert myFunctions.h file here */
#include "myFunctions.h "

void main( ) /* Notice that main is also a function */
{
    unsigned char output = 0;

    /* This is how you call a function */
    output = myFunction(2);
    printf("output equals %d \n", (int) output );

    /* Call it again with a different parameter */
    output = myFunction(4);
    printf("output equals %d \n", (int) output );

} /* end of main – program ends here */
```

References

1. Deital & Deital, C How to Program 3rd Ed. New Jersey: Prentice Hall, 2001.
2. David Maxwell, First_C_2004_Full_Presentation Online seminar, 2004. www.usfirst.org.
3. Innovation First, “2004_Programming_Reference_Guide_10-29-2003.pdf.”, 2004. www.innovationfirst.com.
4. MicroChip, Inc., “MPLAB_C18_Users_Guide_51288b.pdf.”, 2003. MPLAB_CBOT CD.

Glossary

Team Ford First