

# Team 342 Robotics Team

## C Programming Guide

For first time C programmers. Designed to assist students in learning to read the default code in the robot controller.

Written by Chuck Bolin  
November 2004

Requires free editor/compiler from  
<http://www.shed.net/dev/devcpp.html>

Reference Books:

***“C for Dummies – Volume I”***  
by Dan Gookin,  
ISBN 1-878058-78-9

***Workout C: Learn C Through Exercises***  
By David Himmel  
ISBN 1-878739-14-x

## Table ofContents

Introduction	3
Download and Installing Dev-Cpp	3
Creating a C Program Project	3
Project and File Naming Conventions	4
Basic Program that is Autogenerated	4
Printing with printf	5
Variables	6
Placeholders	6
Multiple Placeholders	7
Escape Sequences	8
Math Operations	9
Capturing a Keyboard Keystroke	9
Condition IF and ELSE	10
Condition ELSE-IF	12
AND &&	12
OR	13
Inputting Numbers at the Keyboard	14
Calculator Program	15
Commenting a Program	16
Changing Variable Values	17
Looping With WHILE	18
Never Ending Loop	19
#Define Keyword	20
FOR Loop	21
Passing Parameters to a Function	22
Returning a Value from a Function	23
Receiving a Parameter and Returning a Value	24
Structure	26
An Array of a Structure	26
Variable Scope	27
Static	28
Typedef	28
Enumerations	29
Global Variables	30
Extern	31
Include Files	32
Casting	32
Pointers	33

## Introduction

The 2004 Robot Controllers (RC) are programmed using the C Language. MPLAB is an editor that allows programs to be written. In addition, the program that is written can be compiled and linked to create a single file that is actually loaded into the RC using IFI Loader.

## Download and Installing Dev-Cpp

Practicing programming with a RC, MPLAB and the IFI Loader is not very practical to do at home since expensive hardware and software is required to program.

To facilitate the learning of the C Language, this guide will use a free editor and compiler called Dev-C++. Go to <http://www.bloodshed.net/dev/devcpp.html> to download the file. It is a large file (approximately 7 MB). Click on the download link that reads **Dev-C++ 5.0 beta \*\*\*\*\* with Mingw/GCC \*\*\*\***. The info represented by the asterisks may vary week to week.

Install this to the C:\ drive. On my computer this looks like C:\Dev-Cpp. To run the program, you can open up Windows Explorer and go to the above path. Click on **devcpp.exe** to run the program.

Create a subfolder called C:\Dev-Cpp\Robot. All programs in this guide will be added to this folder.

## Creating a C Program Project

Clicking on devcpp.exe opens up the IDE application. IDE is Integrated Development Environment. The IDE allows us to write C programs, compile, link and to run the created executables. To create a basic C program do the following.

1. Click on File, New, Project...
2. On the Basic Tab, click Console Application.
3. Click the C Project option button.
4. The project name is Project1.
5. Click OK.

The IDE requests that you save the project. Save the project Project1.dev in the C:\Dev-Cpp\Robot folder.

The IDE autogenerates the following C programming code.

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

int main(int argc, char *argv[])
{
    system("PAUSE");
    return 0;
}
```

This code will be explained later. This represents the smallest program that can be written with this IDE. Perform the following steps to compile, link and to run the program.

1. Click on Execute, Compile & Run.
2. The IDE prompts the user to save the source code file. Type the name main1.c and click Save.
3. The program is compiled and a DOS window is displayed that reads "Press any key to continue . . .".
4. Press the spacebar and the DOS window closes.

## Project and File Naming Conventions

For each program in this guide, always create a C project. Increment the number appended to the name to create the next project name. For example: project1, project2, project3, etc. Always save these project files to the C:\Dev-Cpp\Robot path.

Before compiling, it will be necessary to change the name of the default c file. For example: main1.c, main2.c, main3.c, etc. Make sure that the number matches in both the project file and main file. This will allow the user to place hundreds of different programs in the same folder. Do not put spaces in the file names. Although this should not cause problems, it is recommended. It is often difficult to determine if someone has entered one, two or three consecutive spaces in a filename. It matters to the computer.

## Basic Program that is Autogenerated

Refer to the program below. The first two lines are 'include' statements. These lines tell the compiler that information required to compile the program is contained in these two files: stdio.h and stdlib.h.

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

int main(int argc, char *argv[])
{
    system("PAUSE");
    return 0;
}
```

Following the include lines is the function name 'int main(.....)'. It is not necessary to be concerned about the stuff inside these parenthesis. The IDE will always add them and we do not need to be concerned about them. What is important is that every C program must have a function named 'main'.

There are two braces { } that follow the main function. All of the code (program lines) are enclosed inside these braces. Inside are two function calls: system and return.

## Printing

Add the following code (bold print) to the existing program. NOTE: C is case-sensitive. Don't mix upper and lower case letters when you type.

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

int main(int argc, char *argv[])
{
    printf("Robots");
    system("PAUSE");
    return 0;
}
```

The function 'printf' should print the word "Robots" to the DOS window. Add two more lines of code as indicated below. Compile and run.

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

int main(int argc, char *argv[])
{
    printf("Robots");
    printf("are");
    printf("cool!");
    system("PAUSE");
    return 0;
}
```

Notice that the three words "Robots, are, cool" are placed next to each other without any spacing. Make the following modifications. Compile and run. NOTE: Only the code affected is displayed below.

```
printf("Robots\n");
printf("are\n");
printf("cool!\n");
```

As a result of adding "\n" the words are displayed on different rows in the DOS window. The "\n" is called a 'new line' character. It instructs the compiler to move the cursor to the very next row and first column. This new line is like pressing the 'enter' key.

Add this change. Compile and run.

```
printf("Robots\n\n");
```

The new line character adds an additional blank row.

Try this change.

```
printf("\nR\no\nb\no\nr\ns\n\n");
```

There are numerous ways that a new line character can be added to the words inside the quotation marks.

Incidentally, these words, letters, numbers inside quotation marks are often referred to as strings.

Delete the printf lines with 'are' and 'cool!'. Modify the remaining line as follows.

```
printf("Robots\nare\ncool!\n");
```

## Variables

Variables allow us to store numbers for further use in the program. There are a few basic variable types generally used with the Robot Controller so only those will be covered here. The variable 'type' refers to what information can be stored and what range of values are acceptable.

<b>Signed</b>	<b>Range</b>	<b>Unsigned</b>	<b>Range</b>
<b>char</b>	-128 to 127	<b>unsigned char</b>	0 to 255
<b>int</b>	-32768 to 32768	<b>unsigned int</b>	0 to 65535
<b>long</b>	-2147483648 to 2147483647	<b>unsigned long</b>	0 to 4294967295

*Table 1*

Note the difference between a signed and unsigned variable. The signed variable refers to positive and negative values. Unsigned indicates no negative values. All of the variable types are valid. It is only necessary to determine what type of variable to use in your program. It should also be noted that the above types are 'whole numbers'

There are two variable types that allow a decimal point: float and double. The range for a float is  $3.4 \times 10^{-38}$  to  $3.4 \times 10^{38}$ . The range for a double is significantly greater and is used when a significant amount of precision is required. The range is  $1.7 \times 10^{-308}$  to  $1.7 \times 10^{308}$ .

## Placeholders

Create a new project named Project2. Add the following code (bold print), compile and run.

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

int main(int argc, char *argv[])
{
    unsigned int age = 12;
    printf("Age: %i \n", age);

    system("PAUSE");
    return 0;
}
```

An unsigned variable is created named 'age'. Its purpose is to store someones age. A signed variable was not chosen because it was not necessary to work with negative

ages. In addition to creating the variable 'age' it was also 'initialized' to a value of 12. It is good programming practice to always initialize a variable to some value.

'%i' in the above code is a 'placeholder'. It tells the compiler to place the actual value of the variable 'age' into the string "Age: ".

Modify the code as follows.

```
printf("Age: %i %i %i \n",age, age, age);
```

The number 12 is displayed three times with space between them. Note also that the variable 'age' is listed three times after the string and separated by commas.

The placeholder '%d' can be used in lieu of '%i'. '%d' is a signed decimal integer. '%u' can be used for an unsigned decimal integer.

### Multiple Placeholders

Create a new project named Project3. Add the following code (bold print), compile and run.

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

int main(int argc, char *argv[])
{
    int a = 2;
    int b = 3;

    printf("%i + %i = %i\n",a,b, a+b);

    system("PAUSE");
    return 0;
}
```

Two variables are created of type integer: a and b. They are initialized to 2 and 3 respectively. Look closely at the printf function. Note that there are three placeholders. The compiler will replace the first two placeholders with values stored in variables a and b. The third placeholder will be replaced by the mathematical expression 'a+b'.

Add the following lines of code, compile and run.

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

int main(int argc, char *argv[])
{
    int a = 2;
    int b = 3;

    printf("%i + %i = %i\n",a,b, a+b);
}
```

```
int c = a + b;
printf("%i + %i = %i\n", a, b, c);

system("PAUSE");
return 0;
}
```

A third integer variable 'c' is created to store the sum of a + b. Either printf method above is acceptable.

## Escape Sequences

The '\n' newline character is also called an 'escape sequence'. The compiler endeavors to print all characters in a string. However when it reads a back slash '\' this alerts the compiler to escape from blindly printing characters to paying attention to the next character in the sequence. Here are some escape sequences that may be useful in programming.

Sequence	Represents
\b	Moves cursor backwards, no erasing.
\n	Newline, like pressing Enter key.
\t	Tab
\\	Backslash character \.
\'	Apostrophe.
\"	Quotation mark.
\?	Question mark.

**Table 2**

Write, compile and run the following program. Save as Project4.

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

int main(int argc, char *argv[])
{
    printf("\t Tab \n");
    printf("Backslash = \\ \n");
    printf("Apostrophe = \' \n");
    printf("Quotation = \" \n");
    printf("Question = \? \n");
    system("PAUSE");
    return 0;
}
```

This program demonstrates various escape sequences.

## Math Operations

Create a new project named Project5. Add the following code (bold print), compile and run.

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

int main(int argc, char *argv[])
{
    int a = 8;
    int b = 2;
    int c = a + b;
    int d = a - b;
    int e = a * b;
    int f = a / b;

    printf("%i + %i = %i\n",a,b,c);
    printf("%i - %i = %i\n",a,b,d);
    printf("%i x %i = %i\n",a,b,e);
    printf("%i / %i = %i\n",a,b,f);

    system("PAUSE");
    return 0;
}
```

The four most common math operations are addition (+), subtraction (-), multiplication (\*) and division (/).

### **Capturing a Keyboard Keystroke**

The RC does not allow a keyboard to be used when it is operating in the game field. However, getting inputs from the keyboard is a good way of making programs more user friendly as well as assisting in our understanding of many more C commands.

Create a new project Project6. Write, compile and run the following.

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

int main(int argc, char *argv[])
{
    char key;
    printf("Press a key on the keyboard: ");
    key = getche();
    printf("\nYou pressed the '%c' key. \n",key);

    system("PAUSE");
    return 0;
}
```

A new variable of type 'char' is created named 'key'. Remember a 'char' can store a numerical value between -128 to 127. You may be wondering how a letter is saved as a number.

The 'getche()' function gets a character from the keyboard and 'echos' it to the screen. There are numerous functions for capturing keyboard input.

Each key has a numerical equivalent called an ASCII number (pronounced ASK-KEY). Regular keys on the keyboard go from 0 to 127. For example, the spacebar is 32.

Modify the code as following (bold) and run. Note the ASCII value of the key that is pressed. Run the program several times and observe these values as you press letters, numbers and symbols.

```
char key;  
printf("Press a key on the keyboard: ");  
key = getche();  
printf("\nYou pressed the '%c' key. \n",key);  
printf("ASCII: %i \n",key);
```

You should note that capital letters (upper-case) and smaller letters (lower-case) have different ASCII values.

### Condition IF and ELSE

A very important part of creating programs is to have the program make decisions. The 'if' and 'else' keywords are important. Create a project named Project7. Write, compile and run the following program.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <conio.h>  
  
int main(int argc, char *argv[])  
{  
    char key;  
    printf("Press a key: ");  
    key = getche();  
  
    if (key == 'a')  
        printf("\nYou pressed an 'a'.\n");  
    else  
        printf("\nYou did not press an 'a'.\n");  
    system("PAUSE");  
    return 0;  
}
```

In this program the user is allowed to enter a key. If a lower-case 'a' is pressed a message will be displayed stating so. If an 'a' is not pressed, then an alternate message is displayed.

Following the 'if' keyword are parenthesis containing (key == 'a'). The compiler interprets this as meaning if the key variable contains an ASCII value equivalent to the lower-case 'a' then perform the following line.

If this condition is not satisfied, then the code associated with the 'else' keyword is performed.

It should be noted that only one action (one line of code) was executed for each condition. If it is necessary to do more than one thing such as printing twice, then this code must be placed inside braces '{ }'. Modify the following program (bold).

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

int main(int argc, char *argv[])
{
    char key;
    printf("Press a key: ");
    key = getch();

    if (key == 'a'){
        printf("\nYou pressed an 'a'.\n");
        printf("This is good.\n");
    }
    else {
        printf("\nYou did not press an 'a'.\n");
        printf("This is bad.\n");
    }

    system("PAUSE");
    return 0;
}
```

Remember, braces are very important. Try removing the braces and see what happens.

### Condition ELSE-IF

Often there are multiple possibilities for a single variable. Create a project named Project8. Write, compile and run the following program.

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

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

```
printf("Press a key: ");
key = getche();

if (key == 'a'){
    printf("\nYou pressed an 'a'.\n");
    printf("This is good.\n");
}
else if(key == 'b'){
    printf("\nYou pressed an 'b'.\n");
    printf("This is so-so.\n");
}
else if(key == 'c'){
    printf("\nYou pressed an 'c'.\n");
    printf("This is average.\n");
}
else{
    printf("\nYou did not press an 'a, b or c'.\n");
    printf("This is not very good.\n");
}

system("PAUSE");
return 0;
}
```

The 'else if' allows us to put together lots of different conditions. This will happen in programming the robot. You will have to sort out several sensor signals and control signals to the robot driver in order to make decisions. The 'if, else if and else' are very useful.

## AND &&

It is often essential to check two different conditions prior to performing a specific action. The 'and' is represented by '&&'. Create a project named Project9. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    char key1;
    char key2;

    printf("Press a key on the keyboard: ");
    key1 = getche();
    printf("\nPress another key on the keyboard: ");
    key2 = getche();
    printf("\n");

    if ( key1 == 'a' && key2 == 'a' )
        printf("aa pressed.\n");
}
```

```
    else if ( key1 == 'a' && key2 == 'b')
        printf("ab pressed.\n");
    else if ( key1 == 'b' && key2 == 'a')
        printf("ba pressed.\n");
    else if ( key1 == 'b' && key2 == 'b')
        printf("ab pressed.\n");
    else
        printf("ab combo not pressed.\n");

    system("PAUSE");
    return 0;
}
```

It can be seen that two variables are used: 'key1' and 'key2'. There are four possible configurations using an 'a' and a 'b': 'aa', 'ab', 'ba' and 'bb'. This program decides which of the four configurations if any were selected.

Several conditions can be 'and-ed' together.

## OR ||

Often it is not necessary for all the inputs conditions to be true. It is enough that at least one input condition is true.

Create a new project named Project10. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    char key1;
    char key2;

    printf("Press a key: ");
    key1 = getche();
    printf("\nPress another key: ");
    key2 = getche();

    if (key1 == 'a' || key2 == 'b')
        printf("\n a or b pressed.\n");
    else if (key1 == 'c' || key2 == 'd')
        printf("\n c or d pressed.\n");
    else if (key1 == 'e' || key2 == 'f')
        printf("\n e or f pressed.\n");
    else
        printf("\n Wrong keys.\n");

    system("PAUSE");
    return 0;
}
```

Multiple conditions can be 'OR-d' together. Additionally, conditions can be 'AND-ed' and 'OR-d' together. AND operations take precedence over OR operations. Parenthesis can be added to force some conditions to be evaluated over other conditions.

### Inputting Numbers at the Keyboard

The 'getche' function used previously was good at reading in a single character. The following program allows the user to enter a number. Create a new project named Project11. Write, compile and run the following program.

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

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

    printf("Enter a number: ");
    scanf("%i",&num);
    printf("\nYou entered %i.\n",num);

    system("PAUSE");
    return 0;
}
```

The function 'scanf' is what grabs the number. It is completed when the enter key is pressed. Note the use of an ampersand '&' before the variable name 'num'. The function 'scanf' places the number at the address in RAM where the variable is assigned. It is not too important at the moment to understand this. Just remember to place the ampersand before the variable name when using 'scanf' function.

### Calculator Program

Create a new project named Project12. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    float num1;
    float num2;

    printf("Enter the first number: ");
    scanf("%f",&num1);
    printf("Enter the second number: ");
    scanf("%f",&num2);

    printf("%f + %f = %f \n",num1, num2, num1 + num2);
    printf("%f - %f = %f \n",num1, num2, num1 - num2);
    printf("%f * %f = %f \n",num1, num2, num1 * num2);
    printf("%f / %f = %f \n",num1, num2, num1 / num2);
}
```

```
    system("PAUSE");  
    return 0;  
}
```

Note the use of 'float' type variables. This allows the user to enter values with decimal points.

One problem that may not be obvious is that it is possible for the second number to be entered as 'zero'. This is only a problem for division. To anticipate this potential problem, a condition can be added to ensure that the division only occurs if the divisor is a non-zero value. Modify the program as follows (bold).

```
printf("%f + %f = %f \n", num1, num2, num1 + num2);  
printf("%f - %f = %f \n", num1, num2, num1 - num2);  
printf("%f * %f = %f \n", num1, num2, num1 * num2);  
  
if (num2 < 0 || num2 > 0)  
    printf("%f / %f = %f \n", num1, num2, num1 / num2);  
else  
    printf("Cannot divide by zero. \n");  
  
system("PAUSE");
```

The division result will be displayed if num2 is less than zero or greater than zero.

### Commenting a Program

The author of a program can add comments to a program that the compiler will ignore. Since a programmer must often modify his or her program, it is necessary to add comments to remind you in the future what you were trying to do with the code. Another benefit is to other programmers that must modify your code.

Comments can be added by doing one of two things. 1) Preceding the comment with two forward slashes '//'. These comments can be placed above code or to the right side of a line of code. Everything to the right of this comment '/' is ignored by the compiler. 2) Placing a '/\*' before the comment and a '\*' after the comment. This is like placing bookends around a group of books.

In addition to adding comments to a program to add clarity, comments can also be added to temporarily remove some of the program lines from the program. This is often done for testing and debugging the source code.

The meat of Project12 source code is written below with relevant comments.

```
//declare variables  
float num1;  
float num2;  
  
//get two numbers from the user  
printf("Enter the first number: ");  
scanf("%f", &num1);
```

```
printf("Enter the second number: ");
scanf("%f",&num2);

//display the results of calculation
printf("%f + %f = %f \n",num1, num2, num1 + num2);
printf("%f - %f = %f \n",num1, num2, num1 - num2);
printf("%f * %f = %f \n",num1, num2, num1 * num2);

//verify divide by zero error does not occur
if (num2 < 0 || num2 > 0)
    printf("%f / %f = %f \n",num1, num2, num1 / num2);
else
    printf("Cannot divide by zero. \n");
```

Note that a blank row (or line) precedes each comment. This enhances readability. Note also that one relevant comment is added for each section of code. It is important to provide the programmer with comments that serve as an overview of the code. Too much code is unnecessary and can cause tremendous confusion.

Note also in all previous programs the use of indenting code. Code inside braces '{ }' is indented (one tab of 2 to 4 characters). Code following the 'if' and 'else' are indented. All of this is to enhance readability. Remember, the compiler doesn't care if it's readable or not.

## Changing Variable Values

Variables can be incremented (increased) or decremented (decreased) by various amounts. The following program demonstrates various ways to accomplish these operations. You will see that there are several ways to accomplish the same thing.

Create a project named Project13. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    int a = 0;

    printf("%i \n",a);
    a = a + 1;
    printf("%i \n",a);
    a++;
    printf("%i \n",a);
    a--;
    printf("%i \n",a);
    a = a - 1;
    printf("%i \n",a);
    a = 1;
    printf("%i \n",a);
    a += 5;
```

```
    printf("%i \n",a);
    a -= 5;
    printf("%i \n",a);
    a = 1;
    printf("%i \n",a);
    a *= 15;
    printf("%i \n",a);
    a /= 3;
    printf("%i \n",a);
    a = 1;
    printf("%i \n",a);
    a = a * 15;
    printf("%i \n",a);
    a = a / 3;
    printf("%i \n",a);

    system("PAUSE");
    return 0;
}
```

The 'a++' contains a 'post-fix' operator '++' that automatically increments the value of 'a' by one. 'a--' decrements the value of 'a' by one.

This accomplishes the same code as 'a = a + 1' and 'a = a - 1'.

'a += 1' is the same as 'a = a + 1'. 'a +=12' is the same as 'a = a + 12'. Either methods are acceptable in C. Similar patterns are available for '-=', '\*=' and '/='.

## Looping With WHILE

Sometimes it is necessary for code to repeat over and over and over until a condition is met. The 'while' keyword allows this to happen.

Create a project named Project14. Write, compile and run the following program.

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

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

    while (key != 'x'){
        printf("\nPress a key (x to exit): ");
        key = getch();
    }
    printf("\nThat's all folks!\n");

    system("PAUSE");
    return 0;
}
```

The program will run until the lower-case 'x' is pressed. The exclamation point '!' means 'not'. The 'while' line is read 'while the variable key is not equal to x keep looping'. The code contained inside the braces '{ }' represent the 'loop'.

Create a project named Project15. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    int num = 0;

    printf("Starting...\n");
    while (num < 100){
        num++;
        printf("%i \n",num);
    }
    printf("Done! \n");

    system("PAUSE");
    return 0;
}
```

This program initializes a variable 'num' of type integer. The 'while' loop increments this variable by one each time it executes. When the value is no longer less than 100 the while loop ends.

Try repeating this program without initializing variable 'num'. For example: 'int num;' instead of 'int num = 0;'.

Create a project named Project16. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    unsigned long num = 0;

    printf("Starting...\n");
    while (num < 100000000){ //100 million loops...wow!
        num++;
    }
    printf("Done! \n");

    system("PAUSE");
    return 0;
}
```

This program will loop 100 million times. On my Pentium 4 computer that is about 2 seconds. How about your computer. Try experimenting with larger values. Note that an unsigned long variable is chosen that can count up to over 4 billion.

### Never Ending Loop

Create a project named Project17. Write, compile and run the following program.

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

int main(int argc, char *argv[])
{
    while (1){

    }

    system("PAUSE");
    return 0;
}
```

The '(1)' is always true so this while loop will indefinitely until the program is closed.

### #Define Keyword

Often it is convenient to assign specific numerical values to a constant name. The '#define' keyword allows this to happen. The programmer can use the constant name throughout the program. The compiler will convert this name to an actual value defined at the top of the program.

Create a project named Project18. Write, compile and run the following program.

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

#define PI 3.14159

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

    printf("Enter radius of sphere: ");
    scanf("%f", &radius);
    printf("Volume of sphere is %f \n", 1.33 * PI * radius *
radius * radius);

    system("PAUSE");
    return 0;
}
```

In the above program PI is defined as 3.14159. This allows the user to use PI anywhere in the program. This is particularly useful if the constant is used several times in a program.

## FOR Loop

The 'for' loop is an interesting way to increment a variable beginning at one value all the way to another value.

Create a project named Project19. Write, compile and run the following program.

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

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

    for (num = 0; num < 10; num++){
        printf("Number is %i \n", num);
    }

    system("PAUSE");
    return 0;
}
```

Note that the values printed range from 0 to 9 and counts up. The 'for' loop can also count down. Modify the above program as follows (bold).

```
int num;

for (num = 10; num > 0; num--){
    printf("Number is %i \n", num);
}
```

The program counts down from 10 to 1. Experiment with various values to determine the effect.

## Functions

A function is a section of code that can be written and moved out of the way. It can then be called as often as necessary. Functions can be overwhelming. However, when taken slowly they're not so bad.

Create a project named Project20. Write, compile and run the following program.

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

//here is the "function prototype"
void PrintHello(void);

int main(int argc, char *argv[])
{
    //here is the "function call"
```

```
    PrintHello();

    system("PAUSE");
    return 0;
}

//here is the "actual function"
void PrintHello(void){
    printf("Hello\n");
}
```

All this program does is print the word 'Hello'. However it does it in a cool way. It uses a function. Note that using a function requires three things. First is a function prototype. This line tells the compiler what the function looks like. Note that it ends in a semicolon. Second, the function call is simply the name 'PrintHello' without the 'void' stuff. Third, the actual function contains all the code that must be executed.

The word 'void' means nothing. In this case, nothing is passed to the function and nothing is returned to the function. This will be better understood as more functions are presented.

### Passing Parameters to a Function

Create a project named Project21. Write, compile and run the following program.

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

#define PI 3.14159

//here is the "function prototype"
void CalcArea(float);

int main(int argc, char *argv[])
{
    //here is the "function call"
    CalcArea(10);

    system("PAUSE");
    return 0;
}

//here is the "actual function"
void CalcArea(float radius){
    float area = PI * radius * radius;

    printf("Area of circle: %f \n", area);
}
```

Remember, all C programs begin with the main function. The first line of code inside the main function is a 'function call' to CalcArea(10); Notice the number 10 in the parenthesis? This is the number that is passed to the function.

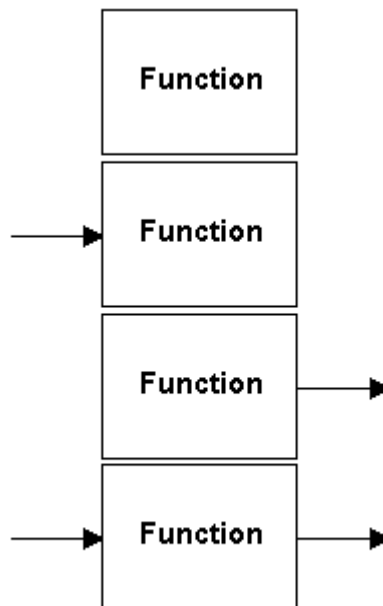
The actual function, placed below the main function, receives this value of '10' and places it into the variable 'radius'. Area of the circle is calculated and then displayed.

Again, the function prototype is placed above the main function.

In the default Robot Controller program the function prototypes are placed in separate files called include files. These files end in '.h'.

It should be noted, that if the actual function is placed physically above the main function then the function prototype is not required.

Now there are four basic patterns for functions. They can be represented as follows graphically. When writing a program with functions it is necessary to determine the pattern of each function.



The above figure illustrates four functions as blocks. Some of the blocks have arrows coming into them. Some blocks have arrows coming out of them.

Arrows coming into the function blocks indicates one or more parameters (numbers, values) are being passed to the function.

The function does work, using parameters if they were passed into the function.

The arrows coming out of the function block represent a value being returned to the function call.

The PrintHello() function is an example of the first type of function block above. It received no parameters and did not return any values. Thus the word void was used.

The CalcArea() function is an example of the second type of function block above. It receives a parameter (radius) but does not return anything.

### Returning a Value from a Function

Create a project named Project22. Write, compile and run the following program.

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

//here is the "function prototype"
int GetNumber(void);

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

    //here is the "function call"
    num = GetNumber();
    printf("Number is %i\n",num);

    system("PAUSE");
    return 0;
}

//here is the "actual function"
int GetNumber(void){
    return 17;
}
```

The above program is an example of the third function block. It receives no value but returns a value. In this case, it returns the number 17 to the calling procedure. This returned value is loaded into a variable.

Again, notice the use and placement of the function prototype, function call and function.

### Receiving a Parameter and Returning a Value

Create a project named Project23. Write, compile and run the following program.

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

//here is the "function prototype"
float CalcVolume (float, float, float);

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

```
float vol, length, height, width;

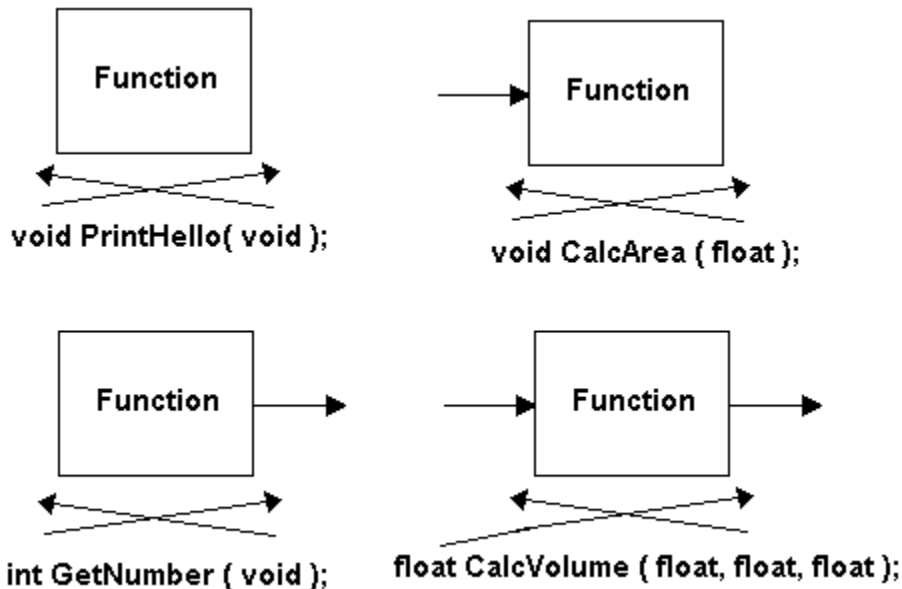
printf("Enter length: ");
scanf("%f",&length);
printf("Enter height: ");
scanf("%f",&height);
printf("Enter width: ");
scanf("%f",&width);

//here is the function call
vol = CalcVolume(length, height,width);
printf ("Volume is %f \n", vol);

system("PAUSE");
return 0;
}

//here is the "actual function"
float CalcVolume (float l, float h, float w){
return l * h * w;
}
```

This program receives three parameters: length, height and width. It calculates the volume and returns this value to the calling function. This returned value is loaded into variable 'vol'.



The above drawing matches a function block diagram with the function prototype. Notice that the parameters are passed in on the left side of the block but are placed inside the parenthesis on the right side of the function. For example, in function CalcArea, an arrow is drawn on the left side of the function block. However, the word 'float' is written inside the parenthesis on the right side of CalcArea.

The function block returns values on the right side but this is indicated on the left side of the function name. For example, the function GetNumber has an arrow coming out of the function block. This is indicated by the word 'int' on the left side of GetNumber.

## Structure

A structure is a neat way of combining two or more variables so that they are easier to track and to use.

Create a project named Project24. Write, compile and run the following program.

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

//defines a structure
struct PERSON
{
    int height;
    int age;
};

//main function
int main(int argc, char *argv[])
{
    //creates a type of PERSON named 'p'
    struct PERSON p;
    p.age = 10;
    p.height = 50;
    printf("Age is %i and height is %i.\n", p.age, p.height);

    system("PAUSE");
    return 0;
}
```

A structure named PERSON is created. It stores two integer variables named height and age. Inside the main function a variable is created of type 'PERSON'. In previous programs, standard variable types were used such as int, float, char, etc. Now, it is possible to define your own variable type for greater efficiency.

PERSON is the 'tag name' for the structure. 'age' and 'height' are members of the structure.

## An Array of a Structure

The above program allows for information about a single person. The following program creates an array of three people. They are referenced as p[0], p[1] and p[2].

Create a project named Project25. Write, compile and run the following program.

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

//defines a structure
struct PERSON
{
    int height;
    int age;
};

//main function
int main(int argc, char *argv[])
{
    //creates a type of PERSON named 'p'
    struct PERSON p[2];
    p[0].age = 10;
    p[0].height = 50;
    p[1].age = 11;
    p[1].height = 52;
    p[2].age = 12;
    p[2].height = 54;

    printf("Age is %i and height is %i.\n", p[0].age,
p[0].height);
    printf("Age is %i and height is %i.\n", p[1].age,
p[1].height);
    printf("Age is %i and height is %i.\n", p[2].age,
p[2].height);

    system("PAUSE");
    return 0;
}
```

Notice how the array of person 'p' is created, how all three person values are initialized (assigned values) and how they are displayed.

## Variable Scope

The scope of a variable is a reference to how long does a variable last and what part of the program can see its value. For example, in the following piece of code a variable 'number' is created inside a function.

```
float GetNumber(void)
{
    float number = 0;

    //blah, blah, blah and more blah
}
```

It is important to note three things. The variable 'number' is created after the open curly brace '{' and it is destroyed at the closing curly brace '}'. After the function has been called and completed, the variable is no longer available. If the function is called a second time the variable 'number' must once again be created and then destroyed. The 'scope' of this variable is limited to inside the function. Outside the function it is not available.

## Static

It is possible to have a variable created in a function that is never destroyed until the program ends. However, the value of the variable is only available inside the function.

```
float GetNumber(void)
{
    static float number;

    //blah, blah, blah and more blah
}
```

The variable 'number' in this case is created the first time the function 'GetNumber' is called. However, the variable is not destroyed when the function is completed. It remains inside the function until the program is ended. Note that the variable 'number' is not initialized to zero.

## Register Variable

In addition to creating variables in the two ways described above, it is also possible to create a variable that would not be loaded into computer memory (RAM). Instead, the variable would be saved to a 'high speed register' in the CPU. The compiler will try to fit the variable to one of these registers. If it does not fit or the register is not available it will place the variable back into RAM.

```
float GetNumber(void)
{
    register int number = 4;

    //blah, blah, blah and more blah
}
```

## Typedef

Often it is useful to rename variable 'types' so that the type name is more clear. For example, an unsigned char stores a value between 0 and 255. A 'byte' is a word used for an 8-bit number. The maximum decimal value to be stored inside an 8-bit number or byte is 255. The lowest value is 0. Therefore, it is possible to 'define a new type' of variable as in the following example.

Create a project named Project26. Write, compile and run the following program.

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

typedef unsigned char BYTE;

int main(int argc, char *argv[])
{
    BYTE input = 23;
    printf("%i ", input);

    system("PAUSE");
    return 0;
}
```

Once a BYTE has been assigned as an unsigned char, it is possible to use the word BYTE throughout the program. It is a convention to use upper-case letters for these names.

## Enumerations

Are data types that allows the user to create useful name values. For example, the following program creates an enumeration that stores some possible responses YES, NO, MAYBE and NO\_WAY. No numbers are assigned by the programmer. However, the compiler actually assigns integer values by default. The four default numerical values associated with the above words are 0, 1, 2 and 3. In this program, we simply don't care what the actual numerical values are assigned to the name.

Create a project named Project27. Write, compile and run the following program.

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

enum responses {YES, NO, MAYBE, NO_WAY };

int main(int argc, char *argv[])
{
    enum responses answer;
    answer = YES;

    if (answer == NO_WAY || answer == NO)
        printf("No way!\n");
    else
        printf("Its possible!\n");

    system("PAUSE");
    return 0;
}
```

Replace the line that reads 'answer = YES' to 'answer = NO'. Compile and run.

## Global Variable

Often it is necessary to have a variable available to all functions within a program file. This is done by declaring the variable outside of a function. Often this is done at the top of the program just after the '#include' statements.

Create a project named Project28. Write, compile and run the following program.

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

//creation of a global variable
int number;

//function prototypes
void Update_A(void);
void Update_B(void);

//main function
int main(int argc, char *argv[])
{
    printf("%i \n", number);
    Update_A();
    Update_B();
    number++;
    printf("%i \n", number);

    system("PAUSE");
    return 0;
}

//first function
void Update_A(void)
{
    number++;
    printf("%i \n", number);
}

//second function
void Update_B(void)
{
    number++;
    printf("%i \n", number);
}
```

Read the program and try to understand the program layout and the displayed results. As a rule, global variables should be avoided as much as possible. It is generally preferred to pass variables or their pointers to functions.

## Extern

It is possible to have multiple files in a program. This means several functions are written in one of several files. This allows for more organization. When functions are spread out like this, it may be necessary to use a global variable across multiple files. To do this, an

'extern' keyword must be added in each module that needs to reference this variable. The module that has the original variable declaration does not need to use 'extern'.

Create a project named Project29. Write the following code. Save this file as 'main29.c'.

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

//creation of a global variable
int number;

//function prototypes
void Update_A(void);
void Update_B(void);

//main function
int main(int argc, char *argv[])
{
    printf("%i \n", number);
    Update_A();
    Update_B();
    number++;
    printf("%i \n", number);

    system("PAUSE");
    return 0;
}
```

Click on the menu Project, New File.... Write the following code. Save this file as 'file29a.c'.

```
extern number;

//first function
void Update_A(void)
{
    number++;
    printf("%i \n", number);
}
```

Click on the menu Project, New File... Write the following code. Save this file as 'file29b.c'.

```
extern number;

//second function
void Update_B(void)
{
    number++;
    printf("%i \n", number);
}
```

Note the use of the keyword 'extern' in the two files above. This means that the variable 'number' is external to this module and is defined somewhere else.

## Include Files

These are files that usually end with the extension '.h' and are often referred to as 'header' files.

Modify the above program as follows.

Add a new file to the current program. Cut the following from 'main29.c' file and paste into this new file.

```
//function prototypes  
void Update_A(void);  
void Update_B(void);
```

Save this file as 'file29.h'. NOTE: Pay attention to the '.h' extension.

Add the include line (bold print) to 'main29.c' file.

```
#include <stdio.h>  
#include <stdlib.h>  
#include "file29.h"
```

Note the use of quotation marks instead of angled brackets. Compile and run the program.

Additional functions can be created and placed in various files as required for good organization. It is only necessary to add the function prototypes to the 'include' file to ensure that the compiler has sufficient information.

## Casting

It is possible to convert a variable value to a different variable type temporarily. This is done through 'casting'. The following example 'casts' an integer to a float.

```
int a = 6;  
float b = (float) a;
```

Create a project named Project30. Write, compile and run the following program.

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main(int argc, char *argv[])  
{  
    int a = 2;  
    float b;
```

```
    //5/2 is converted to 2
    b = a * (5/2);
    printf("a = %i    b = %f \n", a, b);

    //casting forces 5/2 to 2.5
    b = a * ((float)5/2);
    printf("a = %i    b = %f \n", a, b);

    system("PAUSE");
    return 0;
}
```

The second formula casts the value '5/2' to a float so it is seen by the compiler as '2.5' instead of 2 (integer division).

## Pointers

Variables are usually stored in the computer memory called RAM. Each location in RAM can be addressed by the computer and therefore by the programmer. There are two ways to look at each variable. One way is to consider the name of the variable and the value that it contains. Another way to look at this is to consider the address of the variable and what is stored in this address.

Most people can get by fine without referencing the addresses of variables. However, it is often more efficient to use addresses, especially when a significant amount of information is being handled such as an array or file of data.

Assume a variable named 'a' is created of type 'int' and initialized to '5'. This would be written as:

```
int a = 5;
```

This means a value of 5 is placed into the RAM location assigned to the variable name 'a'.

To retrieve the actual address in memory where 'a' is stored, one could write this.

```
printf ( " Address of a is %i \n", &a);
```

The variable 'a' is preceded by the 'address operator' or the ampersand '&'.

To store the address of a variable, the 'indirection operator' or asterisk '\*' is used. It is placed just before the variable name when the variable is created and initialized.

```
*c = &a;
```

The variable 'c' is a 'pointer' meaning it contains the address of 'a'.

Create a project named Project31. Write, compile and run the following program.

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

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int a = 10;
    int b = 12;

    int *ptr_a = &a;
    int *ptr_b = &b;

    printf("Normal Addition: \n");
    printf("%i + %i = %i\n\n", a, b, a + b);

    printf("Address of a is %i\n", ptr_a);
    printf("Address of b is %i\n\n", ptr_b);

    printf("Pointer Addition: \n");
    printf("%i + %i = %i\n\n", *ptr_a, *ptr_b, *ptr_a +
*ptr_b);

    system("PAUSE");
    return 0;
}
```

In the above example, 'a' and 'b' are added and displayed as usual. Following that there are additional lines demonstrating pointer addition. It should be noted that the prefix 'p' or 'ptr' is often used to indicate pointer variable names.

## Conclusion

This programming guide was written to help new C programmers learn the basics of the language. This depth was sufficient to allow the new programmer to recognize about 95% of the C language contained in the Robot Controller default language.

Surprisingly, it is often enough to understand only a few keywords to make the necessary program changes required to add additional sensors or outputs.

Good luck! - Chuck Bolin, Team 342