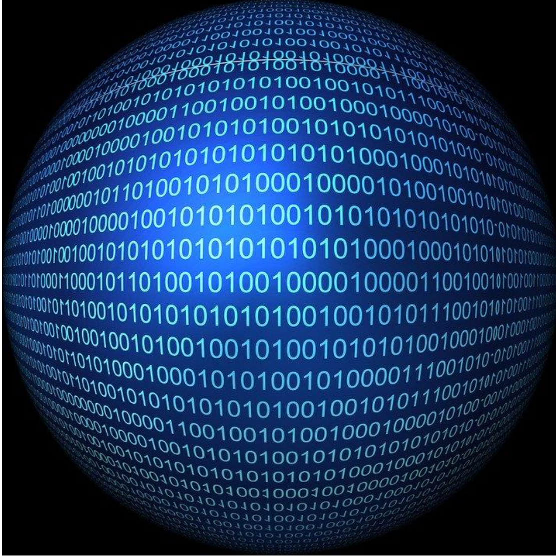


THIN C / C++ REFERENCE

For Embedded Software Development

ABSTRACT

Ed Barker
STEM Leadership Foundation, Inc

Purpose of this guide.

The purpose of this guide is to help support new aspirant and established programmers in learning and using the C and C++ languages for embedded software development.

The information herein follows in the spirit of the original ***The C Programming Language*** (sometimes termed K&R, after its authors' initials) by Brian Kernighan and Dennis Ritchie, and more so like ***The C Programmer's Handbook*** by Morris Bolsky at the Systems Training Center historic AT&T Bell Laboratories.

It is my belief that developing a fundamental understanding of programming must precede all other matters. What will become obvious to someone following this course of study is how little attention we spend on the expansive body of computers science. As you will soon see, we will get lot of robot programming done with a minimal amount of computer science knowledge. By knowing and using the fundamentals of C and C++ we will step into an embedded programming framework and quickly have a fun, functional, and fascinating embedded software project in place to run a FIRST Robotics Competition robot. From there, the world is your oyster, and the road will be wide open for further learning and personal development.

It is not the purpose of this guide or course to 'boil the ocean' but to establish a solid foundation wherein the student understands what they are doing and have developed self-confidence and resilience in this matter.

Contents

Introduction 3
The original 32 reserved keywords of the ‘C’ language. 3
The original *basic data types* of the ‘C’ language. 3
C language Statements 4
There are five types of statements: 4
Labels 4
Compound statements 4
Expression statements 5
Selection statements 5
Iteration statements 5
Jump statements 5
Standard integer types, *aka ‘whole numbers’* 6
Standard fractional number types, *aka real numbers* 6
A simple sample ’C’ program 7
A simple sample ’C++’ program 7
A simple sample ’C++’ program using ’C++’ and ‘C’ libraries 8
#include 8
The main() function 8
Case Sensitivity 8
Functions 9
Return values 9
Variables 9
Mathematical Operators 10
Tests and Comparisons 10
switch 11
break 11
iteration – for, while, and do 12
logical operators 13
Comments 13
Arrays 14
Strings 14
Constants 14
Preprocessor directives 15
Header files 15
Pointers and addresses 15

Introduction

The original 32 reserved keywords of the ‘C’ language.

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

The original ***basic*** *data types* of the ‘C’ language.

‘whole number types’

char	character (one byte)
int	integer (usually one word)
unsigned	non-negative integer (same size as integer)
short	small integer (word or halfword)
long	large integer (word or doubleword)

‘real number types’

float	floating point (single precision
double	floating point (double precision)

‘house keeping’

void	no value (typically to discard the value of a function call)

Notice the lack of specificity of the sizes and ranges of these datatypes. This is because this was left to the designer of the computer chips and compilers, *way back in the day.* This problem will be permanently addressed later in this document.

Advanced datatypes include pointers, arrays, structures, bit fields, unions, and enumerators.

Starting with these keywords and data types, the entirety of the modern computing world can be recreated. Youtube, Google, Snapchat, Facebook, and all the rest of the internet.

C language Statements

Statements are fragments of the C program that are executed in sequence. The body of any function is a compound statement, which, in turn is a sequence of statements and declarations:

```
int main(void)
{
      int m;                    // declaration (not a statement)
      int n = 1;                // declaration (not a statement)
      n = n + 1;                // expression statement
      printf("n = %d\n", n);  // expression statement
      return 0;                 // return statement
}
```

There are five types of statements:

1. compound statements
2. expression statements
3. selection statements
4. iteration statements
5. jump statements

Labels

Any statement can be labeled, by providing a name followed by a colon before the statement itself.

```
identifier : statement                       // target for a goto
case constant_expression : statement         // case label in a switch
statement
default : statement                          // default label in a switch
statement
```

Any statement (but not a declaration) may be preceded by any number of labels, each of which declares identifier to be a label name, which must be unique within the enclosing function (in other words, label names have function scope).

Label declaration has no effect on its own, does not alter the flow of control, or modify the behavior of the statement that follows in any way.

Compound statements

A compound statement, or block, is a brace-enclosed sequence of statements and declarations.

```
{
      expression  | declaration (optional);
}
```

Expression statements

An expression followed by a semicolon is a statement.

```
expression (optional);
```

Selection statements

The selection statements choose between one of several statements depending on the value of an expression.

- **if** (expression)
  {
  }

- **if** (expression)
  {
  }
  else
  {
  }

- **switch** (*expression*)
  {
  }

Iteration statements

The iteration statements repeatedly execute a statement.

- **while** (*expression*)
  {
  }

- do
  {
  } **while** (*expression*) ;

- **for** (*init_clause* ; *expression* (optional) ; *expression* (optional))
  {
  }

Jump statements

The jump statements unconditionally transfer flow control.

- break;
- continue;
- **return** *expression* (optional);
- **goto** identifier;

Standard integer types, *aka ‘whole numbers’*

Type	Storage size	Value range
char signed char int8_t	1 byte	-128 to 127
unsigned char uint8_t	1 byte	0 to 255
short short int unsigned short int int16_t	2 bytes	-32,768 to 32,767
unsigned short unsigned short int uint16_t	2 bytes	0 to 65,535
int int32_t	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int uint32_t	2 or 4 bytes	0 to 4,294,967,295
long int_64_t	8 bytes	−9,223,372,036,854,775,807 to +9,223,372,036,854,775,807
unsigned long uint64_t	8 bytes	0 to 18,446,744,073,709,551,615

Standard fractional number types, *aka real numbers*

Type	**Storage size**	**Value range**	**,Precision**
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.3E-308 to 1.7E+308	15 decimal places

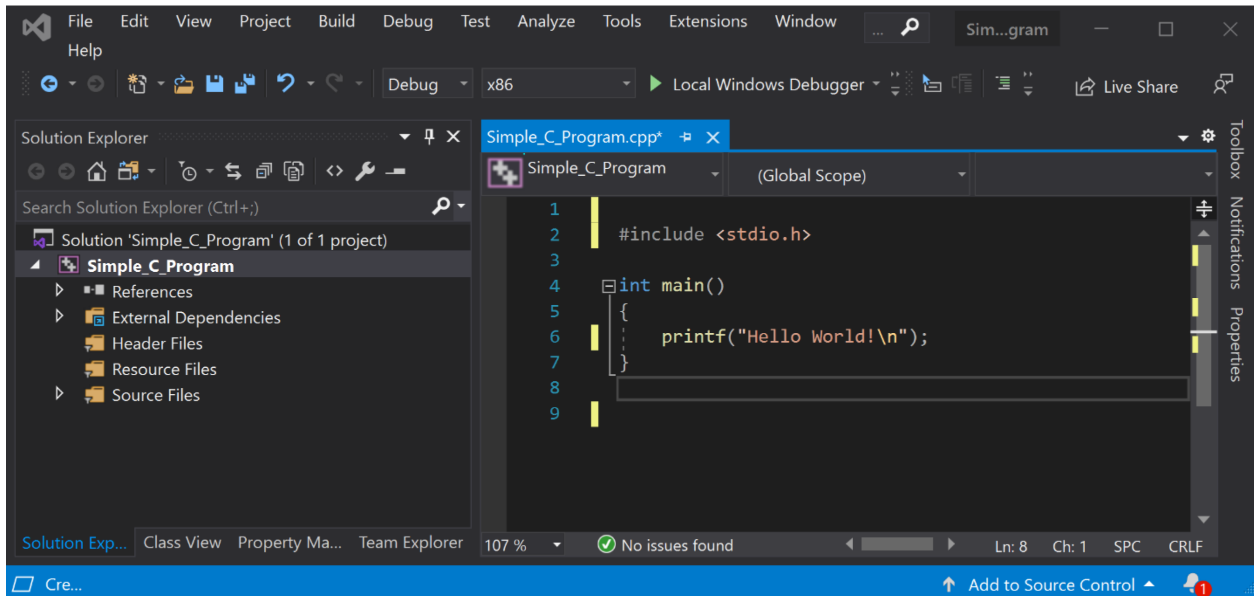
The historic usage of keywords such as char, int, and short are ‘implementation and architecture dependent’. Going forward, we will only use the **int*xx*_t** and **uint*xx*_t** declarations for integer types.

Add the following to the top of your program:

```
#include <inttypes.h>
```

A simple sample 'C' program

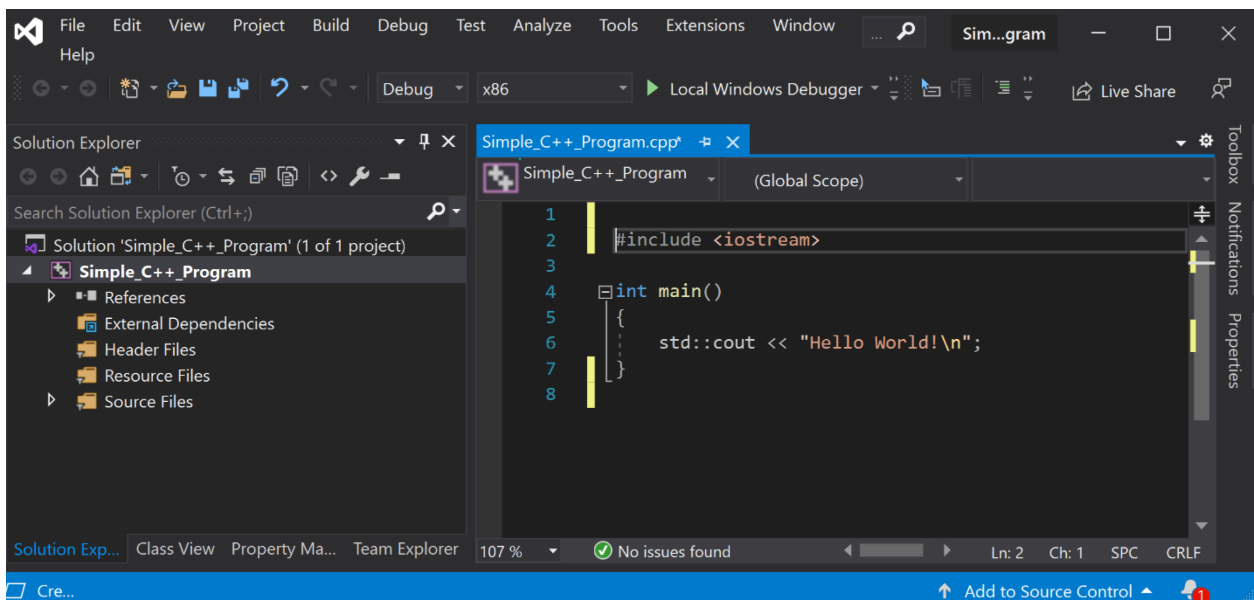
Printing using **'C'** libraries.



Example shown is in MS Visual Studio 2019

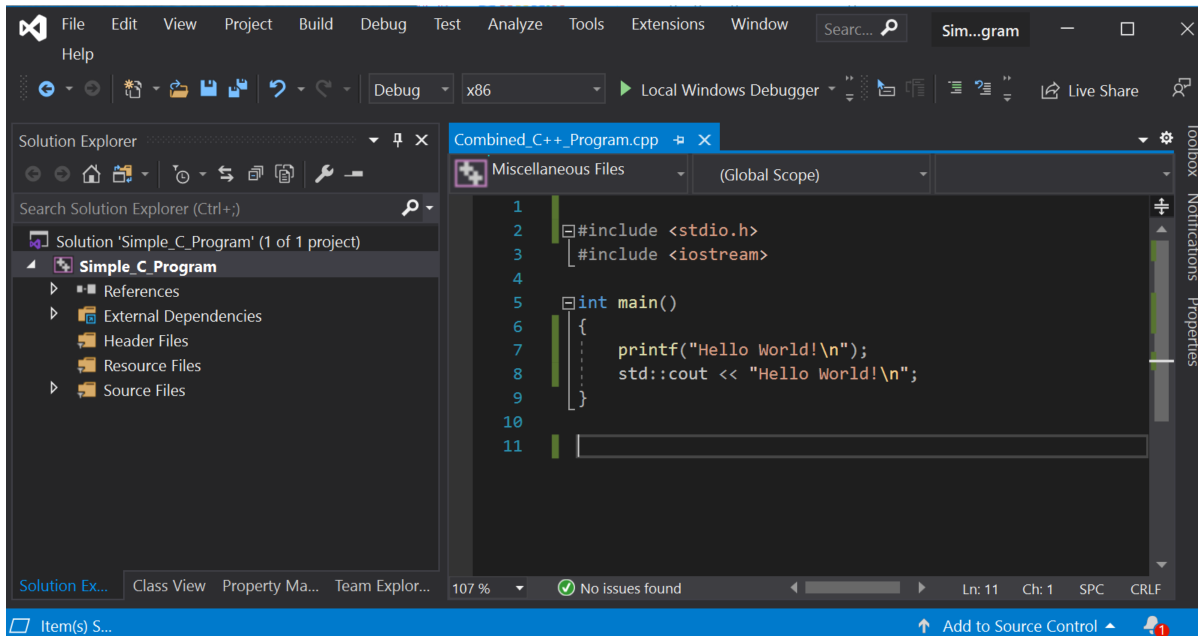
A simple sample 'C++' program

Printing using **'C++'** libraries.



Example shown is in MS Visual Studio 2019

A simple sample 'C++' program using 'C++' and 'C' libraries



Example shown is in MS Visual Studio 2019

#include

#include <some filename> will read a description of an existing library or a library you create

#include <stdio.h> looks up a C library full of 'standard, common file I/O operations

#include <iostream.h> is basically the C++ version of standard common file I/O operations.

In these examples, it is how we tell the compiler to access the library so that we can print information to the 'console'.

The main() function

All C / C++ programs have a '**main()**' function. This is the where the operating system such as windows or Linux begins program execution.

For any generic program that you write, you will start with '**main()**'.

For our purposes, using the FIRST Robotics Competition WPI implementation of C++, you will ***never*** deal with the **main()** function. It is being handled within the robot project framework.

We will work downstream from there.

Case Sensitivity

The language is case-sensitive.

If a variable is named HELLO, then referencing hello will not work. Any combination of upper- and lower-case letters is legal, as is using the underscore '_' character.

Functions

Functions are named blocks of code. All C / C++ programs have at least one function called main(). You can create new functions to do work, such as your own add function, as follows.

```
uint32_t my_add(uint32_t number_a, uint32_t number b)
{
        uint32_t the_sum;
        the_sum = number_a + number_b;
        return the_sum;
}
```

Return values

The return statement in a function must return a value of the same type as declared in the function type. In the example above, the statement '`uint8_t my_add(…)`' says that a `uint8_t` will result from calling the function. Therefore, the `return the_sum`; is returning a number of the same type. If a function does not return a value, the function return type should of type *`void`* and a `return` is not needed in the function.

Variables

A variable is container that we store information into. It is called a variable because the contents can vary. Think it as an electronic scratchpad. If we needed to keep track of the air temperature, we can declare a variable as follows.

```
whole number declarations

uint8_t     air_temperature;     // 0  to  255 degrees
uint32_t    air_temperature;     // 0  to  4,294,967,295 degrees
int8_t      air_temperature;     // -128  to  127 degrees
int32_t     air_temperature;     // -2,147,483,648  to  2,147,483,647 degrees

real number declaration

float       air_temperature;     // 1.2E-38 to 3.4E+38 degrees
```

It is obvious that the choice of **datatype** can dramatically influence the ability to read a thermostat and adequately store and retain the information in a meaningful way.

Mathematical Operators

A simple math operator can look like this:

```
a = a + b;
```

A compound operator can look like this:

```
a += b;
```

Both of the statements above are equivalent.

Common examples:

operator	example	equivalent to
+=	a += b	a = a + b
-=	a -= b	a = a – b
*=	a *= b	a = a * b
/=	a /= b	a = a / b

Adding 1 to a variable simplifies to: `i++`

subtracting 1 simplifies to : `i--`

example of using ‘i’ and then add 1 after the use

```
if (i++ > 10)
{
        // do something
}
```

Example of adding 1 to ‘I’ before use

```
if (++i >= 13)
{
        // do something
}
```

Tests and Comparisons

The two ‘if’ statement above are examples of logical tests and comparison. Commonly used comparisons are:

```
==    // equals
!=    // not equals
>     // greater than
<     // less than
```

NOTE: == is the comparison operator
= is the assignment operator

This will be a common source of programming errors for the novice.

switch

The switch statement is a convenient way to construct a 1 of many selections, in lieu of writing a long stack of if then else statements. The switch (expression) examines the expression and compares it to each case test. The case that matches the(expression) begins execution until the break statement is encountered.

```
switch (day_of_week)
{
case 1:
      day = Monday;
      break;

case 2:
      day = Tuesday;
      break;

case 3:
      day = Wednesday;
      break;

case 4:
      day = Thursday;
      break;

case 5:
      day = Friday;
      break;

default:
      day = weekend;
      break;
}
```

break

In the switch statement, all cases will be executed after the first matching case until execution is ended with a break keyword is encountered. That is why the break is preceding each case. There may instances where the break is not desired.

iteration – for, while, and do

There are three iteration statements, for, while, and do while.

The while, and do while are the simplest. The statement block is repeatedly executed as long as (*expression*) evaluates true. The while case evaluates the condition at the beginning of the statement, and the do while evaluates the condition at the end of the statement.

The for loop, upon entry, initializes the by executing the *init_clause* exactly once. At the top of the loop it evaluates the 2nd *expression* and if true, then executes the block. At the end of the execution of the block, it executes the 3rd *expression.* The loop terminates when the 2nd *expression* no longer evaluates to true.

```
while (expression)
{
}


do
{
} while (expression);


for (init_clause; expression(optional); expression(optional))
{
}
```

```
// three ways to do the identical same thing.
#include <stdio.h>

int main()
{
	uint32_t i;

	// while loop example
	i = 0;
	while (i < 10)
	{
		printf("%d \n", i);
		i++;
	}

	// do-while loop example
	i = 0;
	do
	{
		printf("%d \n", i);
		i++;
	} while (i < 10);

	// for loop example, prints the numbers from 0 through 9
	for (i = 0; i < 10; i++)
	{
		printf("%d \n", i);
	}
```

logical operators

Logical operators can test multiple *expressions* to form a compound logic statement. For example:

```
if (day == Saturday) && (grass_needs_cutting = true)
      cut_the_grass = true;


if ((i_am_thirsty) || (i_am_hungry))
      take_a_break = true;
```

The && operator is a logic AND.

The || operator is a logic OR.

Comments

You can document your source code by adding comments that are ignored by the C compiler. Traditional C comments are placed between the delimiters /* and */ and may span many lines. The following code is surrounded by the /* and */, thereby disabling the code.

```
/*
      if (day == Saturday) && (grass_needs_cutting = true)
      cut_the_grass = true;


      if ((i_am_thirsty) || (i_am_hungry))
      take_a_break = true;
*/
```

You can comment a single line by using // at the beginning of the line.

```
// sum = a + b;
```

Arrays

An array is a linear collection of elements of a specified data type. Each element has an index. The first element is at index 0. The last element is at the index given by the length of the array (that is, the number of elements) minus 1. An array can be declared with its type followed by its capacity between square brackets. This is a declaration of an array containing 5 integers:

```
int my_array[5];
```

You can index into an array using square brackets to obtain an existing element or assign a new value at a specified index. Here I assign 100 to index 2 of my array:

```
int my_array[5];
my_array[2] = 100;
```

An array can be initialized when it is declared by placing empty square brackets after the variable name and then assigning a comma-delimited list of values between curly brackets, as in this example:

```
int my_array[] = { 1,2,3,4,5 };
```

Strings

There is no string data type in C. What we call a string in C is really an array of characters terminated by a null '\0' character. When you initialize a string variable, as shown below, C automatically adds a null at the end.

```
int my_string[] = “Hello World!!”;
```

Constants

If you need a variable-like identify with a value that never changes, you should use a constant. The older traditional way is to use a #define directive.

```
#define PI 3.14
```

The modern way is to define it as follow:

```
const double PI = 3.14159;
```

Preprocessor directives

The C compiler's preprocessor can interpret special directives. The directives are preceded by the hash # character. Commonly used directives are `#include` to include header files and `#define` to define constants.

Header files

A file with the extension '.h' is called a 'header file'. A header file typically contains definitions of functions and constants. The header file does not contain the implementation of functions – only their declarations.

The implementations are generally contained in source code files that end with the extension '.c' for 'C' or '.cpp' for 'C++'.

The declarations in a header file enable the compiler check that all the data types used by the function-calls in your program.

When you include a header file, its contents are inserted into your code just as though you had cut and pasted them.

The `#include` file name is enclosed in angle brackets when it is a standard C library file, or double quote when using headers that you created.

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

Pointers and addresses

Just as homes and businesses have street addresses, every variable, every object in a computer resides at an address.

A pointer variable or pointer is a 'signpost' that points to the object or variable. It is not the item, it just points to where it is located, or if configured erroneously, to somewhere probably dangerously. Pointer management not done well will create many problems.

Pointers are one of the most difficult-to-understand features of the C language. Most modern object-oriented languages make minimal, if any, explicit use of pointers. All our usage is implicit, and 'behind-the-scenes'. There are good reasons to use pointers in certain instances but little reason for us currently.

We will make very little explicit use of pointers in our programming exercises, in favor of referencing our objects and variables by name. This concept will make more sense as we progress into the exercises.

Appendix

https://ide.geeksforgeeks.org/

https://en.cppreference.com/w/c

https://en.cppreference.com/w/cpp

http://www.cplusplus.com/doc/tutorial/