Applied Signal Processing

An introduction to the C-programing language

Tomas McKelvey

2016

This document will serve the purpose of introducing the structure of the C-programming language, including syntax and a presentation of a few of the basic language elements. For further reading, the seminal book by Kernighan and Richie [1] is highly recommended.

1 Data types

Variables in C can have different types. The language requires the data type of a variable to be defined at the compile time. If the compiler knows the data type at compile time it can produce more efficient code. The C-language supports several primitive data types. The compiler for the Arm based STM32 processor family supports (among other) the following ***`types`***:

- `char` integer (8 bits)
- `int` signed integer (32 bits, range [-2 147 483 648, 2 147 483 647])
- `float` floating point (32 bits)

2 Structure of a C-program

A C-program is based on a set of functions and variables which are defined in source files with extension `.c`. The compiler reads all source files and compiles the source code into object files with extension `.o`. In a final step a linker program assemble all object files of the C-program into one executable binary.

The execution of the program starts by a call to the function `main`. The functions which make up the program are defined in the source file together with definitions of variables and header files. The structure of the c-source file is a result of an ideology where the compiler should be able to compile the code without having to iterate to many times over the source code. Hence the following rules apply:

- All variables need to be explicit defined before used in a function. Definitions can be made global as well as local. A global variable is reachable from all functions while a local variable is only reachable within the function block it is defined in. The *type* of the variable also need to be explicitly given.
- All functions which are called inside a function need to be *declared* by a *function prototype* before being used in the code. The reason for this is that when the compiler reaches the use of the function it can explicitly check that the number of arguments and types of the arguments are correctly given.
- So called header files with the extension ".h" are included before the functions are defined. In the header files special types are defined, constants are defined and function prototypes are declared.

Consider the following C-program,

```
// This is a one line comment
/* This is multiple line comment
...
...
end of comment
*/
// Inclusion of header files
#include <stdio.h>
// Definitions of constants
#define ACONST 10
#define VECSIZE 4
// Declaration of a function prototype
int fmult(int a, int b);
// This is a definition of global variables
int r1;
int vector[VECSIZE]={1,2,3,4}; // This defines a vector of size VECSIZE.
```

```
// Definition of the main function
int main(){
  int a=1, b=2, c; // Definition of local variables
  c = a + b + ACONST;
  r1 = fmult(12, 13);
  printf("a + b + ACONST is %d \nr1 = %d \n", c , r1);
  return(0); // Return 0
}
// Definition of the function fmult
int fmult(int a, int b){
  int c;
  int i;
  c = a * b;
  for (i=0; i<VECSIZE; i++){
    c += vector[i]*vector[i];
  };
  return(c);
}
```

Comments in the code which the compiler should disregard can appear in two forms.

- The one line version. The compiler disregards all characters on which appear after `//`.
- The multi line version. The compiler disregards all characters between the start of comment `/*` and the end of comment `*/`. Comments cannot be nested.

To facilitiate the programing task the program code in the source files are first processed by the preprocessor. The preprocessor follows directives which are given in the format `#`***`name`*** with one or several arguments. Library header files are included with `#include` <***`name.h`***> while header files local to the project (within the same folder) are included with `#include "`***`name.h`***`"`. Numerical constants can be defined using the format `#define` ***`constantname value`***. All occurrences of the symbol ***`constantname`*** will be replaced by the ***`value`*** within all source files. The entity ***`value`*** can be an numerical expression including other previously defined constants and should then be enclosed by parenthesis.

Function prototypes are declared like

`type name (type1 var1, ...,typek vark);`

Variable definitions have the form like

```
type name1, name2, ..., namek;
```

which would reserve space for k variables of type *type*. Alternatively the variable can also be initialized at the same time like

```
type name1=value1, name2=value2, ..., namek=valuek;
```

Arrays of variables are defined in the following way

```
type name1[size1], name2[size2], ..., namek[sizek];
```

The number within the square brackets determines the number of storage elements which will be allocated in the memory. In C the reference to the elements of the array starts with index 0 and run to index *size*-1. The first element in an array `a` will thus be referred to as `a[0]`. Arrays can also be initialized at the definition. The syntax for this is

```
type name[k] = {value1,value2,..., valuek};
```

which would reserve space for an array of size k initialized with the values given in the list enclosed with curly brackets.

Functions are defined according to the following form

```
type name (type1 var1, ...,typek vark){
     local variable definitions
     function body
     return( expression );
}
```

When the function executes the `return` command the function will return to the calling function and the value given in the expression will be returned to the calling function as the value returned.

3 The basic language elements

The body or block of a C-program is composed of a sequence of *statements*. Each simple statement is concluded by a semicolon `;` A statement can be of many forms like an assignment, an `if` statement or a `for` loop statement. A block which can contain variable definitions and several statements is also a statement. A block is enclosed by curly brackets {...} and is called a *compound statement*. A block can define local variables which then are only accessible from within the block. An example of a compound statement is given below.

```
{
   int a,b,c;
   b = -a;
   c = a*a;
}
```

Statements are composed of *identifiers*, for example, numbers, names of variables or other reserved words. Identifiers are separated by so-called white spaces which are one or several spaces, tabs or new line characters. The C-language disregard the visual formatting of a program and one or many white spaces separates the identifiers from each other.

Arithmetic expressions are formed as as expected using variables and constants and the operators `+,-,*,/,%`. The modulus operator `%` gives the reminder of integer division. e.g. `5 % 2` evaluates to 1 since $2 * 2 + 1$ is 5. Assignments has the following form

variable=*expression*;

which will assign the value of expression to the variable *variable*. The following relational operators (*relop*) exists:

- `==` equality
- `!=` inequality
- `<,>,<=,>=` less than, greater than, less than or equal, greater than or equal

A relational expression looks like

expression relop expression

and results in a boolean value which is either 1 (true) or 0 (false). Boolean operators operate on boolean expressions

- (`&&`) logical and
- (`||`) logical or
- (`!`) logical negation

and the result is again a boolean value. Important to know is that the boolean operators have a higher precedence than the relational operators so it is often necessary to enclose the relational expressions with parenthesis to get the right associations. For example a test to check that two variables `a,b` are grater than 10, then a correct logical expression would be be

```
(a>10) && (b>10)
```

Control structures control the flow of the program the most common of them is the `if` statement.

```
if (expression)
     statement
else
     statement
```

Looping is accomplished by the `while,` `do` and `for` statements. The `while` statement has the form

```
while (expression) statement
```

As long as the expression evaluates to a non-zero value the statement will be repeated. Normally the statement is a block statement. The `do` statement has the syntax

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

Here the order is exchanged. First the statement is executed and then the expression is evaluated. The looping continues until the expression becomes 0 (false). The `for` statement is commonly used when dealing with arrays (or vectors). The syntax is

```
for (exp1; exp2; exp3) statement
```

The semantics are as follows. First expression *exp1* is evaluated only once. Normally the expression contains initialization of the loop variable. Then *exp2* is evaluated. If *exp2* is non-zero (true) *statement* is executed. Finally *exp3* is evaluated. The looping continues until the stop criterion in *exp2* evaluates to false. The following example shows how to add two arrays of size 4:

```
for (i=0; i<4 ;i++){
  c[i] = a[i] + b[i];
}
```

Here we used the special increment C-construction `i++` which is equivalent to `i=i+1`. Also `i--` exists which is equivalent to `i=i-1` Finally we mention the special statement `i += j` which is equivalent to `i = i + j` . Similar constructions exist for the other arithmetic operators `-,*,/,%`.

The increment and decrement constructions also results in a numerical value. Hence, the increment operation can be used in an numerical expression. Consider the construction

```
int i=12, j;
j = i++;
```

After execution, variable `i` has value 13 and variable `j` has value 12. The expression `i++` evaluates to the value of the variable `i` *before* the increment. Similarly there exist a construction where the increment is performed first and the expression evaluates to the value *after* the increment. Consider the following

```
int i=12, j;
j = ++i;
```

Now both variables will have the value 13 after execution.

4 Pointers to variables

Normally when functions are called which perform operations on large arrays it is desirable to only pass a reference to the memory location where the array resides instead of transmitting all the numerical values during the function call. This is known as call by reference. Call by reference is also desirable when the called function produce several numerical values as outputs since a C-function can only explicitly return a single numerical value.

In the C langage a reference to variable location are known as a *pointer value* and is an integer value which refers to a specific memory location. Pointer values can also be stored into a variable called a *pointer* variable.

```
int i,j;       // defines the integers
int *pi, *pj; // defines the integer pointer variables
i = 2;
j = 3;
pi = &i;  // pointer variable pi now has the address to variable i
pj = &j;  // pointer variable pj now has the address to variable j
*pj = *pi + 10;  //  This is the same as j = i + 10
```

Pointer variable definitions have the form like

type `*` *name1*, `*` *name2*, ..., `*` *namek*;

which would reserve space for k variables of type *type* `*`, i.e. a pointer to a variable of type *type*. The construction

`*` *variable*

in an expression which is called a pointer dereferencing and results in the numerical value equal to the value stored in the variable which the pointer variable *variable* points to. Note that this expression is valid only if *variable* has been defined as a pointer variable. The assignment statement

$$* \ \mathit{variable} = \mathit{expression}$$

mean that the value of *expression* will be assigned to the variable in the memory which the pointer variable *variable* points to. Finally the expression

$$\& \ \mathit{variable}$$

is the pointer value which corresponds to the memory location for *variable*. When an array is defined like

$$\mathit{type} \ \mathit{name}[\mathit{size}]$$

then the identifier *name* has the data type pointer to *type*. Consider

```
int aarr[4] = {1,2,3,4}       // defines the array of integers
int *pi, // define an integer pointer variables
pi = aarr  // Save the address to the start of the array in pi
*pi = 2;   // assign value 2 to the first element in aarr, i.e. aarr[0]=2
*aarr = 3; // assign value 3 to the first element in aarr, i.e. aarr[0]=3
aarr[1] = pi[2];  // Since aarr and pi has the same value
                  // pi[2] to refer to the same value as aarr[2]
```

For function calls we now can use the call by reference by sending the pointer to the function. See the example below.

```
int funfun(int * arr, int na); // Function prototype declaration
int a[4] = {1,2,3,4}       // defines the array of integers

int main(){
    funfun(a,4);
       // after the function call the array a will have values
       // a[0]=2, a[1]=4, a[2]=8 and a[3]=16
}

int funfun(int * arr, int na){ //definition of function
    int i;
    arr[0] = 2;
    for (i=1; i<na; i++){
      arr[i] = arr[i-1]*2;
    }
}
```

Sometimes it is desirable perform operations on a portion of an array. It is then useful to use the expression `&arr[k]` evaluates to the pointer value of the array element `arr[k]`. In the following example it is shown how the first half of an array is added with the second half.

```
void add_int_array(int * a1, int * a2, int * b, int a_size); // declaration
int a[4] = {1,2,3,4};        // defines the array of integers
int b[2];

int main(){
    add_int_array(&a[0],&a[2], b, 2); //   Send pointers to the array parts
}


void add_int_array(int * a1, int * a2, int * b, int a_size){ //definition of
    int i;
    for (i=0; i<a_size; i++){
      b[i] = a1[i] + a2[i];
    }
}
```

References

[1] B.W. Kernighan and D.M Ritchie. *The C Programming Language.* Prentice-Hall, 2nd edition, 1988.