C-Programming Language

History, Types, Variables and Constants

Brief History

- *C* evolved from two previous languages, *BCPL* and *B*.
- *BCPL* was developed in 1967 by *Martin Richards.*
- *B*, in 1970 by *Ken Thompson*.
- Both *BCPL* and *B* were *“typeless”* languages.
- *C*, in 1972 by *Dennis Ritchie* at Bell Labs.
- *C* is widely used with UNIX operating system.

History …

- Many versions of the language C was created by many people.
- This created problems with *portability*.
- In 1999, ANSI (American National Standards Committee) standardized the language and is called ANSI-C.

C program

- C program is a set of functions.
- Every C program should have a function with name *main( )*
- Execution of the program starts with *main( )*
- Some functions to do common tasks comes along with the C language. These functions are kept in libraries.
  - For e.g. the functions to do input and output are in the library called *stdio.h*

C Standard Library

- Libraries that comes along with C are called *standard libraries*.
- So, you need to learn C language and also how to use the *standard libraries.*

A Simple C Program

```
/* This program Prints Hello word */
#include <stdio.h>

main()
{
  printf(“Hello.\n”);
} /* End of main */
```

test.c

```
/* This program finds the area & perimeter
   of a rectangle */

#include <stdio.h>

main()
{
   int l, b, area, perimeter;
   b = 4;
   l = 6;
   area = l*b;
   perimeter = 2*(l+b);
   printf(“Area = %d \n”, area);
   printf (“Perimeter = %d \n”, perimeter);
} /* End of main  */
```

- Complier ignores comments
- **`#include`** line must not end with semicolon
- Every statement is terminated by semicolon

Output of the program

```
Area = 24
Perimeter = 20
```

`cc test.c` *compile program test.c*

`a.out` *execute the program*

Variables and Constants

- Basic data objects manipulated in a program.
- *Variable* should have a *name* (*identifier*) and a *value*.
- Names are made up of *letters* and *digits*; the first character must be a *letter*.
- E.g.: *total, n, sum1, sum2*
- "_" (underscore) is also a *letter*.
- E.g.: *val_1* is a valid name.
- But don't begin variable names with underscore, since library functions often uses such names.

Names (Identifiers) …

- Upper case and lower case letters are distinct. That is *x* and *X* are two different letters.
- There are reserve words in the C language, like *if, else, int, float*, etc., which should not be used as variable names.

Names …

- Invalid names :
  - *$total*
  - *1st*
  - *no.*
  - *case*
- Are the following valid?
  - *_total*
  - *\new*
  - *end*

Names …

- Invalid names :
  - *$total*
  - *1st*
  - *no.*
  - *case*
- Are the following valid?
  - *_total*
  - *\new*
  - *end*   /* valid, not a keyword in C */

Constants

- a data item whose value doesnot change.
- Constants also can have names following the same rules as that for variable names.
- Simple numbers like 123, 0.24 are of course constants (which are without name).

  - *More about constants follows later …*

Data Types

- There are various categories of data items, like *integers*, *real numbers*, *character strings*, etc.
- Each category is called a *type.*
- There are inbuilt types (basic types) and user defined types.

Basic data types

- character (*char*) → single character
- integer (*int*) → an integer number
- real (*float*) → a real valued number

  (*double*) → a real valued number with higher precision.

Declarations

- An identifier which you want to use should be *declared* first.
- A declaration specifies *type* and identifiers name.
- Optionally a declaration for a variable can contain its initial value also.

Declarations …

- int count; /* declares that *count* is a variable which can hold integer values */
- char s, c; /* s and c are variables which can hold a single character each */
- float sum_1 = 0.5;
- double d, total, sum;
    - Declaration is a *statement* and every *statement* in C should end with a semicolon ;

Declarations …

- Syntax for a declaration is:

  type var_1, var_2, …,var_n;

  E.g: `int  i, j,k;`

  Or, write three separate lines like:

  ```
  int i;
  int j;
  int k;
  ```

- All the variables that you are using should be delared first.

Declarations …

- Declaration specifies type and also the size (the number of bits that should be allocated to store the declared variable).
- These sizes are machine (Hard-ware) dependant.
- Size determines the range of values you can use.

Declarations …

- For example, if size of *char* is 1 byte (8 bits) then there can be only 2^8 different characters that you can use.
- Sizes

char → 1 byte

int → 2 bytes or 4 bytes

float → normally 4 bytes or 8 bytes

double→ double the size of float

Declarations …

- For some types, you can increase (or decrease) the size relatively by adding some optional qualifiers to the type.
- For *int* ➔ *short int*

  ➔ *long int*

- For *double* ➔ *long double*

How to find the size?

- sizeof is an operator which you can use.

E.g:

```
---------------------------
#include<stdio.h>
main()
{
    int s;
    s = sizeof( int );
    printf("The size of an int is: %d", s);
}
```

sizeof

- sizeof is an unary operator.
- The operand is either an expression, which is not evaluated, or a paranthesized type name.

- int i,j;
  j = sizeof i; /* but, j = sizeof int; is wrong */

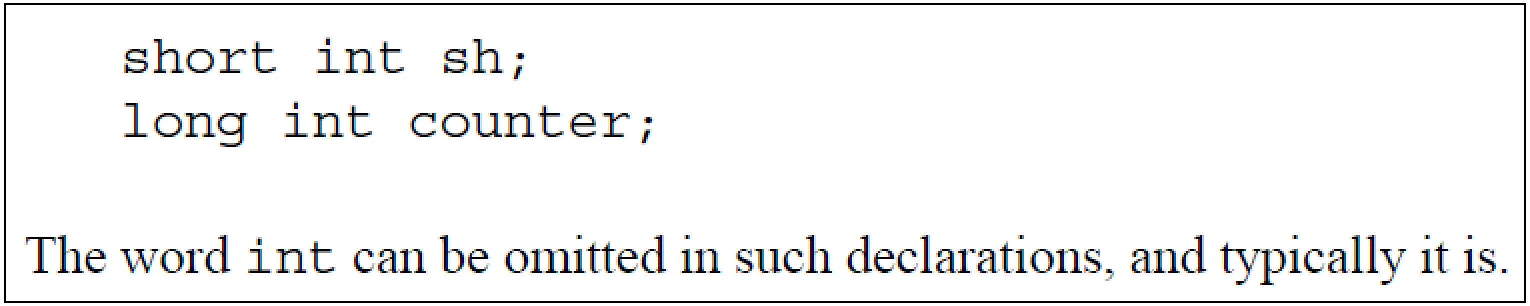
Declarations …

- Some other qualifiers can be used to specify the range …
- E.g: *int* ➔ *signed int*
  unsigned int
- char can also be *signed* or *unsigned*, but this discussion is deferred.

Declarations…

- Some examples:
  - unsigned short int sum;

    if *short int* is of size 1 byte, what is the range of values that *sum* can hold ?
  - long double avg;

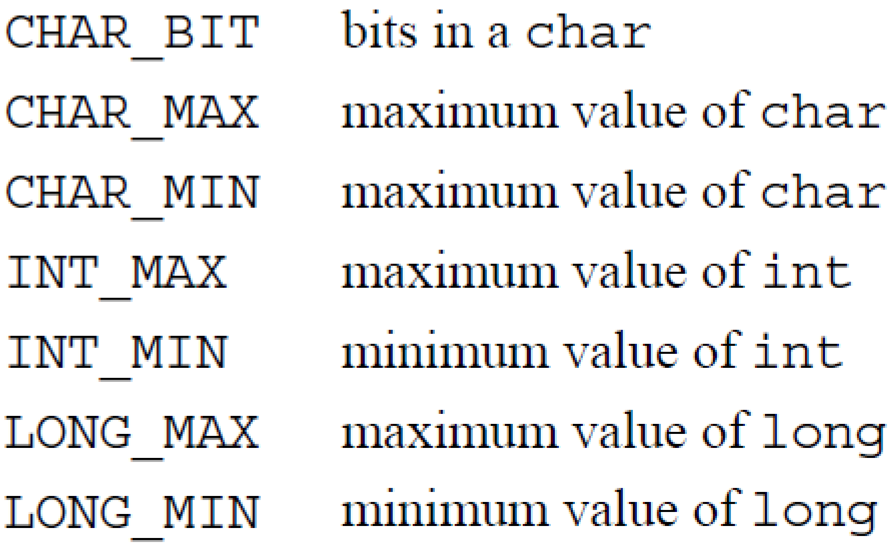


```
    short int sh;
    long int counter;
```

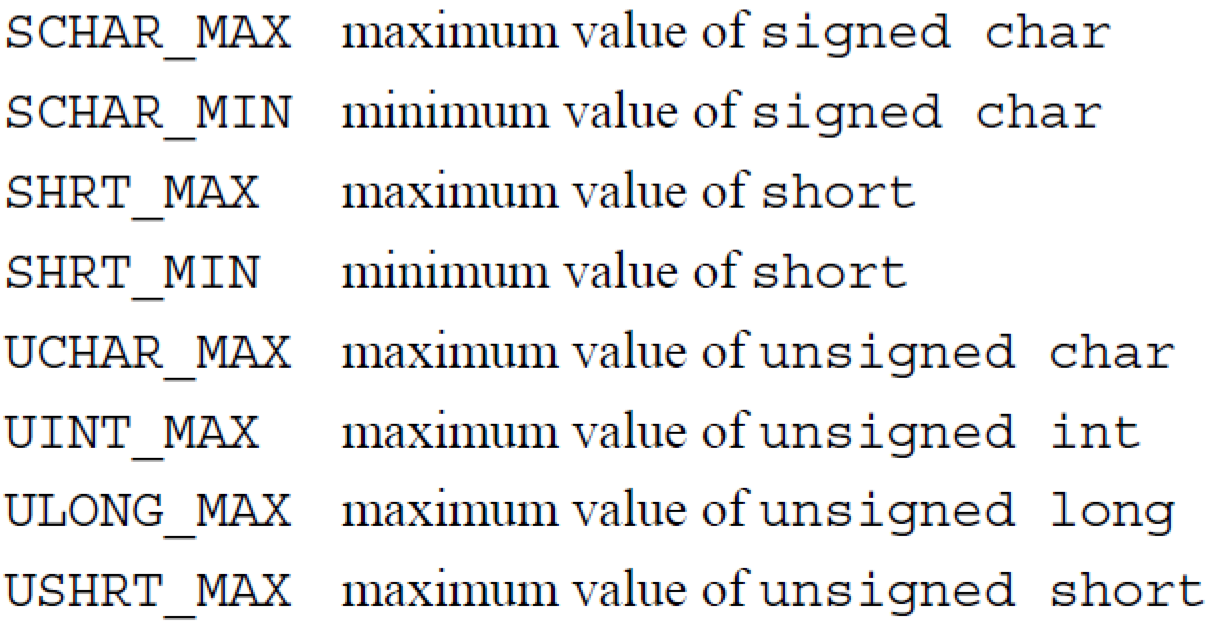
The word `int` can be omitted in such declarations, and typically it is.

How to find sizeof various types

- The header <limits.h> defines constants for the sizes of integral types.



CHAR_BIT	bits in a char
CHAR_MAX	maximum value of char
CHAR_MIN	maximum value of char
INT_MAX	maximum value of int
INT_MIN	minimum value of int
LONG_MAX	maximum value of long
LONG_MIN	minimum value of long



SCHAR_MAX	maximum value of signed char
SCHAR_MIN	minimum value of signed char
SHRT_MAX	maximum value of short
SHRT_MIN	minimum value of short
UCHAR_MAX	maximum value of unsigned char
UINT_MAX	maximum value of unsigned int
ULONG_MAX	maximum value of unsigned long
USHRT_MAX	maximum value of unsigned short

```
#include<limits.h>

…
printf(“%d”, INT_MAX); /*should print max. int value*/
```

- Similarly, <float.h> contains constants related to floating-point arithmetic. Some of them are as follows.

- No need to worry about these, at this point of time.

Constants

- Constants also has type.
- 1234 is an *int*
- A *long int* constant is written with a terminal *l* or *L*. Eg: 123456789L
- An integer too big to fit into an *int* will also be taken as *long*.
- Unsigned constants are written with a terminal *u* or *U*, and the suffix *ul* or *UL* indicates *unsigned long*.

Constants …

- Floating point constants contain a decimal point (123.4) or an exponent (1e-2) or both; and their default type is *double*, unless suffixed.
- The suffixes *f* or *F* indicate a float constant;

  l or *L* indicate a long double.

Constants …

- An integer can be specified in *octal* or *hexadecimal* instead of decimal.
- A leading *0* (zero) on an integer constant means octal; a leading *0x* or *0X* means hexadecimal.
- Eg: decimal 31 = 037 (octal)

  = 0x1f = 0X1f (hexa)
- What is 0XFUL in decimal?

Constants …

- An integer can be specified in *octal* or *hexadecimal* instead of decimal.
- A leading *0* (zero) on an integer constant means octal; a leading *0x* or *0X* means hexadecimal.
- Eg: decimal 31 = 037 (octal)
  = 0x1f = 0X1f (hexa)
- What is 0XFUL in decimal? /* 15 */

Constants …

- A character constant is written within a single quotes, like ‘x’
- Actually a character is stored as an integer.
- The integer value of a character is often called as its ASCII value. (In ASCII character set each character is given an integer value.)
- ‘0’ → character zero → ASCII value is 48
- ‘0’ is unrelated to the numeric value 0

Constants

- Some characters can not be written normally, hence they have special codes called escape sequences.
- ‘\n’ → newline; ‘\a’ → bell; ‘\\’ → backslash
  ‘\b’ → backspace; ‘\t’ → tab; ‘\0’ → null
  ‘\’’ → single quote; ‘\”’ → double quote;
- There are some more escape characters which you should read in the book as an exercise.

Constants … (Named constants)

- The qualifier *const* can be applied to the declaration of any variable to specify that its value will not be changed.
- Eg: *const int* sum = 100;
  - Some other way is through #*define* directive

Constants …

- A *constant expression* is an expression that involves only constants.
  - Such expressions may be evaluated during compilation-time rather than run-time, and accordingly may be used in any place that a constant can occur,

  -
    ```
    #define MAXLINE 1000
    char line[MAXLINE+1];
    ```

  -
    ```
    #define LEAP 1 /* in leap years */
    int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
    ```

A *string constant*, or *string literal*, is a sequence of zero or more characters surrounded by double quotes,

"I am a string"

" " /* the empty string; no space between */

"hello, " "world" is equivalent to "hello, world"

This is useful for splitting up long strings across several source lines.

Technically, a string constant is an array of characters.

The internal representation of a string has a null character '\0' at the end,

so the physical storage required is one more than the number of characters written between the quotes

the enumeration constant

There is one other kind of constant, the enumeration constant. An enumeration is a list of constant integer values, as in

enum boolean { NO, YES };

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified.

If not all values are specified, unspecified values continue the progression from the last specified value

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB = 2, MAR = 3, etc. */
```

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

Names in different enumerations must be distinct.

Values need not be distinct in the same enumeration.

Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you.

Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration.

Nevertheless, enumeration variables offer the chance of checking and so are often better than #defines. In addition, a debugger may be able to print values of enumeration variables in their symbolic form.

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday};

int main()
{
    // creating today variable of enum week type
    enum week today;
    today = Wednesday;
    printf("Day %d", today+1);
    return 0;
}
```

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday};

int main()
{
   // creating today variable of enum week type
   enum week today;
   today = Wednesday;
   printf("Day %d", today+1);
   return 0;
}
```

Output

```
Day 4
```

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday}; /* definition of the type enum week*/

int main()
{
    // creating today variable of enum week type
    enum week today; /* declaration of the variable week*/

    today = Wednesday;
    printf("Day %d", today+1);
    return 0;
}
```

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday}; /* definition of the type enum week*/

int main()
{
   // creating today variable of enum week type
   enum week today; /* declaration of the variable week*/

   today = 20; /* try this */
   printf("Day %d", today+1);
   return 0;
}
```

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday}; /* definition of the type enum week*/

int main()
{
    // creating today variable of enum week type
    enum week today; /* declaration of the variable week*/

    today = 20; /* try this */
    printf("Day %d", today+1);
    return 0;
}
```

Output

```
Day 21
```