View of C Programming

A C program is a collection of

- Declarations
- Definitions

for

- Functions
- Variables
- Data types

A program may be spread over multiples files

A program file may contain preprocessor directives that

- Include other files
- Introduce and expand macro definitions
- Conditionally select certain parts of the source code for compilation

Consider the following Hello World in **Bash** : **hello.sh**

```
#!/bin/bash
echo “Hello World”
exit 0
```

Sample Output

Hello World

Consider the following Hello World in C : hello.c

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        printf("Hello World!\n");
        return EXIT_SUCCESS;
}
```

Assume that it is stored in a file called **hello.c** in the current working directory.

Then:

$ gcc hello.c

(Note: Compiles without warning or error)

Sample run

$ ./a.out
Hello World!

A Closer Look (1)

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        printf("Hello World!\n");
        return EXIT_SUCCESS;
}
```

We are including two **header files** from the **standard library:**

- **stdio.h** contains declarations for buffered stream-based input and output (we include it for the declaration of **printf**). It is header file for **Standard Input Output**

- **stdlib.h** contains declarations for many odds and ends from the standard library (it gives us **EXIT_SUCCESS**). **It is the header of the general-purpose standard library** of C programming language which includes functions involving **memory allocation, process control, conversions, and others.**

- In general, preprocessor directives start with a hash #

Standard C Libraries

Come with gcc and mostly used :

- **stdio.h:** stream and formatted io functions
- **string.h:** string manipulation functions
- **stdlib.h:** general purpose functions
- **math.h:** mathematical functions
- **time.h:** date and time functions
- **threads.h:** thread library

A Closer Look (2)

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

int main(void)
{
        printf("Hello World!\n");
        return EXIT SUCCESS;
}
```

The program consist of one function named main()

– **main()** returns a int (integer value) to its calling environment
– In this case, it takes no arguments (its argument list is void)

– In general, any C program is started by a call to its main() function, and terminates if main() returns.

A Closer Look (3)

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

int main(void)
{
        printf ("Hello World!\n");
        return EXIT SUCCESS;
}
```

The **function body** contains two statements:

- A call to the **standard library function** **printf** () with the argument ”HelloWorld!\n” (a string ending with a newline character)

- A return statement, returning the value of the symbol EXIT_SUCCESS to the caller of main()

A Closer Look (4)

gcc is the GNU C compiler, the standard compiler on most free UNIX system (and often the preferred compiler on many other systems)

- On traditional systems, the compiler is normally called

  cc or gcc takes care of all stages of compiling:

- Preprocessing
- Compiling
- Linking

It automatically recognizes what to do (by looking at the file name suffix).

gcc Important Options:

- **-g :** **include debug info**
- **o <name>:** **Give the name of the output file**
- **-O :** **Use increasing levels of optimization to generate faster executables (O1 thru O6)**
- **-l :** **link with libraries**
- **-ansi:** **Compile strict ANSI-89 C only**
- **-Wall:** **Warn about all dubious lines- All warnings on**
- **-c:** **Don't perform linking, just generate a (linkable) object file**

Assume that you have creates the following C file "foo3.c" , then:

- **gcc -g -o foo3 foo3.c**

  (include debug info , include output file name)

- **gcc -Wall -g -o foo3 foo3.c**

  (include All Warnings , debug info , include output file name)

- **gcc -O3 -g -o foo3 foo3.c**

  (include Level 3 optimization , debug info , include output file name)

Object and Executable

Compile to object only

- gcc -c foo2.c
- gcc -c foo1.c
- **(do not general link – - c means Compile only)**

Link to executable

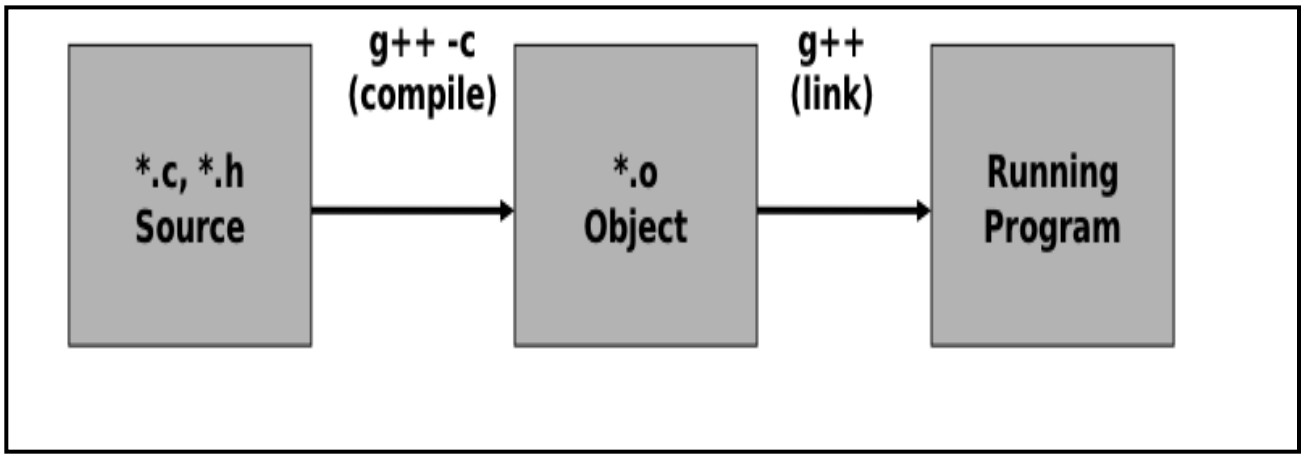
- gcc -o foo2 foo2.c
- -o the name of the output file
  - **Default: a.out (gcc foo2.c)**

For Small Applications :

- gcc -o foo2 foo2.c foo1.c
- Still create *.o files temporarily
- Deleted after linking
- For small applications only typically:
  - All source files are in the same directory.
  - Only one executable is generated.

Compile and Link

C/C++



Another version of Hello.c

```
#include <stdio.h>
int main(int argc, char** argv)
{
    printf("Hello World\n");
    return(0);
}
```

In this case Main Function has :

- int main(int argc, char** argv)
- Arguments

  - **argc** : stands for "**argument count**". **argc** contains **the number of arguments passed to the program.**

  - **argv** : stands for **argument vector**. **argv** is a pointer to a (char) **array of all the parameters that were passed by system to your application**

C vs C++

- Middle-level language.
- File extension in C is .c
- Procedure oriented
- No class, No class derivation, No polymorphic function, No inheritance. No friend functions. That means C does not support object-oriented programming.
- No operator / function overloading
- No template
- No IO Streams / name space
- No new or delete
- No exception
- No reference (type&)
- No built-in Boolean type (true/false)
- No const function
- Different standard libraries

- In C, data can be accessed / communicated between different blocks of code using global declarations.

  Hence, **data is less secure** in C

- C allows **only data members** in its structures. C structs **cannot** have functions.

- C **does not support inline** function definitions by default.

- C is more suitable for stable programs like writing operating system kernels.

Comments

- Comments in C are enclosed in /* and */
- Comments can contain any sequence of characters except for */ (although your compiler may complain if it hits a second occurrence of /* in a comment)
- Comments can be on one line or can span multiple lines

```
// single line comment


/*
Start of Comment
continues
continues
.
.
.
End of Comment
*/
```

Variables

*"**int** Fahrenheit, Celsius;"* declares two **variables** of type ***int*** that can store a signed integer value from a finite range

- By intention, *int* is **the fastest data type available** on any given C implementation

- On most modern UNIX systems, ***int*** is a **32 bit** type and interpreted in 2s complement, giving a range from

  -2 147 483 648 — 2 147 483 647.

In general, a **variable** in a program corresponds to a **memory location** and can store a value of a specific type

- All variables **must be declared**, before they can be used

- Variables can be **local** to a function (like the variables we have used so far), local to a single source file, or **global** to the hole program

A variables value is changed by an **assignment**, an expression of the form "var = expression;"

Variable Declarations

Variable names:

- A valid variable name **starts** with a letter or underscore , and may contain any sequence of letters, underscores, and digits.

- Capitalization is significant : a_variable is different from A_Variable **(Case sensitive)**

- In addition to the language **keywords**, certain other names are reserved (by the standard library or by the implementation). In particular, **avoid** using names that start with an **underscore**!

Variable declarations:

- A (simple) variable declaration has the form ***<type> <varlist>;*** where *<type>* is a type identifier *(e.g. int),* and *<varlist>* is a coma-separated list of variable names.

- In ANSI-89 C, **variables can only be declared outside any blocks** or directly after an open curly brace. The new standard relaxes this requirement.

- A variable declared in a block is (normally) visible just inside that block.

Data Declarations Types

Declarations serve two purposes:

1. They tell the compiler to set aside an **appropriate amount of space in memory** for the program's data (variables).

2. They **enable the compiler to correctly operate on the variables.** The compiler needs to know the data type to correctly operate on a variable. The compiler needs to know how many bytes are used by variables and the format of the bits. The meaning of the binary bits of a variable are different for different data types.

    - **char** is 8 bit (1 byte) ASCII, but can also store numeric data.
    - **int** is 4 byte 2's complement.
    - **short** is 2 byte 2's complement.
    - **long** is 8 byte 2's complement.
    - **unsigned** (int, short, and long) are straight binary
    - **float** is 4 byte IEEE Floating Point Single Precision Standard.
    - **double** is 8 byte IEEE Floating Point Double Precision Standard.

Character - integer relation

The ASCII code is a set of integer numbers used to represent characters.

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

int main()
{
    char c = 'a';       /* 'a' has ASCII value 97 */
    int  i = 65;        /* 65 is ASCII for 'A'    */

    printf( "%c\n", c + 1 );  /* b */
    printf( "%d\n", c + 2 );  /* 99 */
    printf( "%c\n", i + 3 );  /* D */

  return 0;
}
```

Integers

A variations of int are stored binary data which is directly translated to it's base-10 value.

- int
- long
- short
- Signed and Unsigned version

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

int main()
{
   int    a = 1;
   short  b = 1;
   long   c = 1;
   long   d = 1;
   unsigned int aa = 1;
   unsigned long bb = 1;
   printf("\n\n%u", a);
   printf("\n%d", b);
   printf("\n%lu", c);
   printf("\n%lld", d);


return 0;
}
```

Floating point data

Variables of type float and double are **stored in three parts**: the sign, the mantissa (normalized value), and an exponent.

sizeof

Because some variables take different amount of memory of different systems, C provides an operator which **returns the number of bytes needed to store a given type of data.**

```
i = sizeof(char);
j = sizeof(long);
k = sizeof(double);
```

The `sizeof` operator is very useful when **manually allocating memory** and dealing with complex data structures.

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

int main()
{

    printf("size of int : %d\n ",sizeof(int));
    printf("size of signed int :
           %d\n ",sizeof(signed int));
    printf("size of unsigned long :
           %d\n ",sizeof(unsigned long));
    printf("size of unsigned long :
        %d\n ",sizeof(long long));

  return 0;
}
```

Sample Run

```
size of int : 4
size of signed int : 4
size of unsigned long : 4
size of unsigned long long: 8
```

Conversions and Casts

Two mechanisms exist to convert data from one data type to another, ***implicit*** conversion and ***explicit*** conversion, which is also called *casting*.

Implicit Conversion

If a arithmetic operation on two variables of differing types is performed, one of the variables is converted or promoted to the same data as the other before the operation is performed.

In general, smaller data types are always promoted to the larger data type.

```
#include<stdio.h>
int main()
{
   int    x = 10;    // integer x
   char y = 'a';  // character c

   // y implicitly converted to int. ASCII
   // value of 'a' is 97
   x = x + y;

   // x is implicitly converted to float
   float z = x + 1.0;

   printf("x = %d, z = %f", x, z);
   return 0;
}
```

Sample Run :

x = 107 , z = 108.00000

Explicit Conversion or Casts

The programmer can tell the compiler what types of conversions should be performed by using a cast.

A cast is formed by putting a data type keyword in parenthesis in front of a variable name or expression.

x = **(float)**(m * j);

i = **(int)**x + k;

Arithmetic Expressions

C supports various arithmetic operators, including the usual
+, - ,* , / , **

Typical precedence levels

- parentheses
- unary operators
- pow
- *, / , %
- +, -

- Subexpressions can be grouped using **parentheses**
- Normal arithmetic operations can be used on both integer and floating point values, with the type of the arguments determining the type of the result

- **Example**: *(fahrenheit-32)*5/9* is an arithmetic expression in C,

implementing the well-known formula
C = 5/9 (F – 32) for converting Fahrenheit to Celsius

- Since all arguments are integer, all intermediate results are also integer (as well as the final result)

- Therefore we have to multiply with 5 first, then divide by nine (multiplying with (5/9) would effectively multiply with 0)

Bit-wise, logical and operator comparison operators also normally also return numeric values

Possible operands include variables, numerical (and other) constants, and function calls

Note: In C, **any** normal statement is an expression and has a value, including the assignment!

Evaluate the following expressions :

3 + 4*5 - 6/3*4/8 + 2*6 - 4*3*2

24/(1 + 2%3 + 4/5 + 6 + 31%8)

Formatted Output

printf() is a function for **formatted output**

It has at least one argument (the **format string**), but may have an arbitrary number of arguments

- The control string may contain various placeholders, beginning with the character **%,** followed by (optional) formatting instructions, and a letter (or letter combination) indicating the desired output format

- Each placeholder corresponds to exactly one of the additional arguments (modern compilers will complain, if the arguments and the control string do not match)

- In particular, **%5d** requests the output of a normal int in decimal representation, and with a width of at least 5 characters

Note*: printf()* is not part of the C language proper, but of the (standardized) C library

Format

Input : %[modifier]type

- %d, %ld, %lld, %f, %lf, %c, %s
- %[c] or %[^c], scan strings of or until c

Output

- **%[flags][width][.precision][modifier] type**
- **%+5d,** print with **sign and 5 chars**
- **%-04ld,** print with **left** aligned and 4 chars
- **d** /* %d int (**signed** decimal integer)
- **u** /* decimal int, **unsigned** base 10 */
- **s** /* **char** *, null terminated **sequence** of characters */
- **c** /* int, **single character** */
- **f** /* **floating point** values (fixed notation) - float, double */
- **e** /* floating point values (**exponential** notation) */

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {

        int a = 10;
        char b = 'a';
        char c[10] = { "ABCDE" };
        double d = 1111.54321567;

        printf("a is %4d\n", a);
        printf("a is %-4d\n", a);
        printf("a is %04d\n", a);
        printf("b is '%c'\n", b);
        printf("c is \"%s\"\n", c);
        printf("d is %4.1lf\n", d);
        printf("d is %.9le\n", d);
        printf("c is \"%*s\"\n", 15,c);
}
```

Sample Run

```
a is     10
a is   10
a is   0010
b is   'a'
c is   "ABCDE"
d is   1111.5
d is   1.111543216e+03
c is "         ABCDE"
```

Statements, Blocks, and Expressions

C programs are mainly composed of **statements.**

In C, a statement is either:

- An expression, followed by a semicolon ';' (as a special case, an empty expression is also allowed, i.e. a single semicolon represents the empty statement)
- A flow-control statement (*if, while, goto,break. . .*)
- A **block** of statements (or **compound statement**), enclosed in curly braces **'{}'**. A compound statement can also include new variable declarations.

Note: The following is actually legal C (although a good compiler will warn you that some of your statements have no effect):

```
#include <stdio.h>
int main() {
        int a;
        10 + 2;                          //  statement with no effect
        10 * (a = printf("Hello\n"));    //  value computed is not used

return 0;
}
```

Flow-Control: if

The primary means for conditional execution in C is the if statement:

```
if(<expr>)
    <statement>
```

- If the expression evaluates to a non-zero ("true") value, then the statement will be executed

- *<statement>* can also be a block of statements – in fact, it quite often is good style to always use a block, even if it contains only a single statement

- An if statement can also have a branch that is taken if the expression is zero ("false"):

```
if(<expr>)
   <statement>
else
  <statement>
```

Example

```
#include <stdio.h>
int main() {
	int number = -8;
	// Test expression is true if number  is less than 0

	if (number < 0) {
		printf("You entered %d.\n", number);
	}

	printf("The if statement is easy.");

	return 0;
}
```

Sample run

```
You entered -8.
The if statement is easy.
```

Example

```
#include <stdio.h>
int main()
{
   int number = 9 ;

   // True if remainder is 0

   if( number%2 == 0 )
      printf("%d is an even integer.",number);
   else
      printf("%d is an odd integer.",number);
   return 0;
}
```

Sample run

```
9 is an odd integer.
```

Dangling else problem :

What is the output of the following code

```
double shippingCharge;

shippingCharge = 5.00;

if ( country equals "USA" )
   if ( state equals "HI" )
      shippingCharge = 10.00;
else
   shippingCharge = 20.00;
```

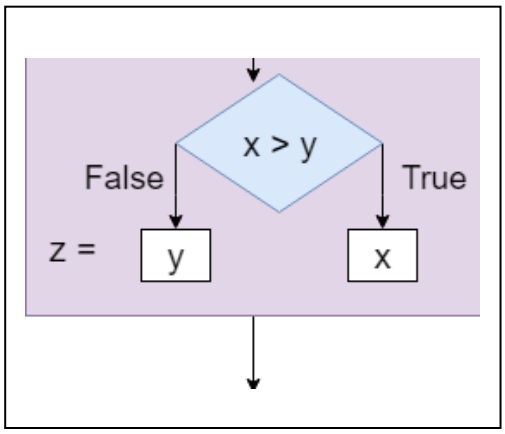
You need to use { } to introduce syntax errors when C tries to associate the else with the second if:

```
double shippingCharge;

shippingCharge = 5.00;
if ( country.equals("USA") )
{
   if ( state.equals("HI") )
      shippingCharge = 10.00;   // Shipping to Hawaii is $10
}
else
   shippingCharge = 20.00;      // Shipping to outside USA is $20
```

Ternary Conditional Operator

Many programming languages also include a special operator, known as a **ternary conditional operator**, sometimes referred to simply as a **ternary operator**, that is effectively a shortcut for a simple **if-else** statement that produces a single value



Example :

// C program to find largest among two numbers **using ternary operator**

```
#include <stdio.h>
int main()
{
   int m = 5, n = 4;

   (m > n) ? printf("m is greater than n that is %d > %d",
                m, n)
         : printf("n is greater than m that is %d > %d",
                n, m);

   return 0;
}
```

Simple Loops

A **while-loop** has the form

while(<expr>)
<body>

where *<body>* either can be a single statement, terminated by a semicolon ';' , or a statement block, included in curly braces '**{}**'
It operates as follows:

- At the beginning of the loop, the **controlling expression** is evaluated
- If it evaluates to a non-zero value, the loop body is executed once, and control returns to the *while*
- If it evaluates to 0, the body is skipped, and the program continues on the next statement after the loop

Note: The body can also be empty (but this is usually a programming bug)

Example

```
/* A program that prints a Fahrenheit -> Celsius
   conversion table */

#include <stdio.h>

int main(void)
{
      int Fahrenheit, Celsius;
      printf("Fahrenheit -> Celsius\n\n");
      Fahrenheit = 0;

      while(Fahrenheit <= 300)
      {
            Celsius = (Fahrenheit-32)*5/9;
            printf("%3d %3d\n", Fahrenheit, Celsius);
            Fahrenheit = Fahrenheit + 10;
      }

return  (0) ;

}
```

The Fahrenheit-Celsius Example

Compilation:

$ gcc -ansi -Wall -W -o Celsius fahrenheit.c

Sample Run

$ ./Celsius

or

$ ./Celsius | more

```
Fahrenheit -> Celsius
0 -17
10 -12
20 -6
30 -1
40 4
50 10
60 15
70 21
80 26
90 32
100 37
--More--
```

Example

```
#include <stdio.h>

int main() {

	/* local variable definition */
	int a = 10;

	/* while loop execution */
	while (a < 20) {
		printf("value of a: %d\n", a);
		a++;
	}

	return 0;
}
```

Sample run

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Flow-Control : for

The for-loop in C is a construct that combines initialization, test, and update of loop variables in one place:

for(<expr1>; <expr2>; <expr3>)
<statement>

- Before the loop is entered, *<expr1>* is evaluated
- Before each loop iteration, *<expr2>* is evaluated
  - If it is true, the body is executed, then *<expr3>* is evaluated and control returns to the top of the loop
  - If it is false, control goes to the first statement after the body
  - In the typical case, both *<expr1> and <expr3>* are assignments to the same variable, while *<expr2>* tests some property depending on that variable

Example

Here is the Fahrenheit/Celsius conversation using ***for:***

```
/* A program that prints a Fahrenheit -> Celsius conversion table */

#include <stdio.h>

int main(void) {
        int Fahrenheit, Celsius;

        printf("Fahrenheit -> Celsius\n\n");

        for (Fahrenheit = 0; Fahrenheit <= 300;
                Fahrenheit = Fahrenheit + 10) {
                Celsius = (Fahrenheit - 32) * 5 / 9;
                printf("%5d %5d\n", Fahrenheit, Celsius);
        }

        return (0);
}
```

Sample run

Fahrenheit -> Celsius

```
  0  -17
 10  -12
 20   -6
 30   -1
 40    4
 50   10
 60   15
 70   21
 80   26
 90   32
100   37
110   43
120   48
130   54
140   60
150   65
      .
      .
      .

300  148
```

Flow-Control : do .. while

- do-while loops are exactly like while loops, except that the test is performed at the end of the loop rather than the beginning.

- This guarantees that the loop will be performed at least once, which is useful for checking user input among other things (see example below.)

- Syntax:

```
do {
    body;
} while( condition );
```

Example

```
#include <stdio.h>

int main() {

      /* local variable definition */
      int a = 10;

      /* do  ..  while loop execution */
      do   {
            printf("value of a: %d\n", a);
            a++;
      }  while ( a < 20 ) ;

      return 0;
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

Simple Character I/O

The C library defines the three I/O streams *stdin, stdout*, and *stderr*, and guarantees that they are open for reading or writing, respectively

Reading characters from stdin: int getchar(void)

- ***getchar()*** returns the numerical (ASCII) value of the next character in the *stdin* input stream

- If there are no more characters available, *getchar()* returns the special value *EOF* that is guaranteed different from any normal character (that is why it returns int rather than char

Printing characters to *stdout: int putchar(int)*

- ***putchar(c)*** prints the character c on the *stdout* steam
- (It returns that character, or EOF on failure)

getchar(), putchar(),* and *EOF* are declared in *<stdio.h>

Example:

```
#include <stdio.h>
#include <ctype.h>
int main() {
        char c;
        printf("Enter some character. Enter  $  to    exit...\n");
        while (c != '$')
        {
                c = getchar();
                printf("\n You Entered : ");
                putchar(c);
        }
        return 0;
}
```

Sample Run

```
Enter some character. Enter $ to exit...
E

 You Entered : E
B

 You Entered : B
X

 You Entered : X
$

 You Entered : $
```

Example: Character Counting

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
        int c;
        long count = 0;
        while ((c = getchar()) != '$') {
                count++;
        }
        printf("Number of characters: %ld\n", count);
        return EXIT_SUCCESS;
}
```

Sample Run

```
ABCDefgh123$
Number of characters: 11
```

Simple Character I/O

scanf

Read data in from standard input (keyboard), call the scanf function.

scanf(format_string, list_of_variable_addresses);

- The format string is **like that of printf**
- But instead of expressions, we need **space to store incoming data**, hence the list of variable addresses

Example

```
#include <stdio.h>

int main()
{
 int i;
 float f;
 char c;

 printf("Enter an integer and a float, then Y or N\n> ");
 scanf("%d%f%c", &i, &f, &c);

 printf("You entered:\n");
 printf("i = %d, f = %f, c = %c\n", i, f, c);

}
```

Sample Run

```
Enter an integer and a float, then Y or N
> 123 123.5Y
You entered:
i = 123, f = 123.500000, c = Y
husain-gholooms-macbook:~ husaingholoom$
```

Expressions: Relational Operators

Relational operators take two arguments and return a truth value (0 or 1)

We already have seen the equational operators. They apply to all basic data types and pointers:

- ***a == b*** (**equal**) evaluates to 1 if the two arguments have the same value, otherwise it evaluates to 0
- ***a != b*** (**not equal**) evaluates to 1 if the two arguments have different values

Note : *a == b == c* is evaluated as *(a == b) == c*, i.e. it compares c to either **0** or **1**!

We can also compare the same types using the greater/lesser relations:

- **>** evaluates to 1, if the first argument is greater than the second one
- **<** evaluates to 1, if the second argument is greater than the first one
- ***a >= b*** evaluates to 1, if either a > b == 1 or (a == b) ==1
- ***a <= b*** evaluates to 1, if either a < b == 1 or (a == b) == 1

Precedence rule: The relational operators have lower precedence than the arithmetic ones (*a+1 < 2*b* makes sense)

Expressions: Logical Operators

Logical operators operate on truth values, i.e. all non-zero values are treated the same way (representing **true**)

The binary logical operators are **||** and ***&&***

- ***a* || *b*** evaluates to 1, if at least one of *a* **or** *b* is non-zero (otherwise it evaluates to 0)

- **a && b** evaluates to 1, if both a **and** b are non-zero

- Both || and *&&* are **evaluated left-to-right**, and the evaluation stops as soon as we can be sure of the result (short-circuit evaluation)

  - Example: If *a!=b*, then *(a==b)&&c* will not evaluate c
  - Similarly: *(a==0 || 10/a >= 1)* will never divide by zero!

! is the (unary) logical negation operator, !a evaluates to **1**, if a has the value **0**, it evaluates to 0 in all other cases

Precedence rules:

- The binary logical operators have **lower** precedence than the relational ones
- || has **lower** precedence than &&
- ! has a **higher** precedence than even arithmetic operators

Example

```
#include <stdio.h>

int main(void) {
        int m = 40, n = 20;
        int o = 20, p = 30;
        if (m > n && m != 0) {
                printf("&& Operator : Both conditions are true\n");
        }
        if (o > p || p != 20) {
                printf("|| Operator : Only one condition is true\n");
        }
        if (!(m > n && m != 0)) {
                printf("! Operator : Both conditions are true\n");
        } else {
                printf("! Operator : Both conditions are true. "
                                "But, status is inverted as false\n");
        }
}
```

Sample Run

```
&&   Operator :  Both conditions are true
||   Operator :  Only one condition is true
!    Operator :  Both conditions are true. But, status is
                 inverted as false
```

Expressions: Assignments

The **assignment operator** is = (a single equal sign)

- *a = b* is an expression with the value of b

As a **side effect**, it will change the value of a to that same value

The expression on the left hand side of an assignment (a) has to be an **lvalue**, i.e. something we can assign to. Legal lvalues are

- Variables
- Dereferenced pointers ("memory locations")
- Elements in a struct, union, or array

The assignment operator is **right-associative** (so you can write
a = b = c = d = 0; to set all for variables to zero)

The assignment operator has extremely low precedence (lower than all other operators we have covered up to now)

Floating Point Numbers

C supports three types of **floating point** numbers, *float, double, and long double*

- **float** is the most memory-efficient representation (typically 32 bits), but has limited range and precision
- **double** is the most commonly used floating point type. In particular, most numerical library functions accept and return **double** arguments. Doubles normally take up 64 bits
- **long double** offers extended range and precision (sometimes using 128 bits) and is a recent addition

Floating point constants are written using a decimal point, or exponential notation (or both):

- 1.0 is a floating point constant
- 1 is an integer constant. . .
- . . . but 1e0 and 1.0E0 are both floating point constants

If we mix integer and floating point numbers in an expression, a value of a "smaller" type is converted to that of the bigger one transparently:

- 9/2 == 4, but 9/2.0 == 4.5 and 9.0/2 == 4.5

Fahrenheit to Celsius – More Exactly

/* A program that prints a Fahrenheit -> Celsius conversion table */

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
        int Fahrenheit;
        double Celsius;
        printf("Fahrenheit -> Celsius\n\n");
        for(Fahrenheit=0; Fahrenheit<=300; Fahrenheit=fahrenheit+10)
        {
                Celsius = (fahrenheit-32.0)*5.0/9.0;
                printf("%3d %7.3f\n", Fahrenheit, Celsius);
        }
        return EXIT_SUCCESS;
}
```

Remark : *The %7.3f* conversion specification prints a *float* or *double* with a total width of 7 characters and 3 fractional digits

Generating Integers

- The **rand()** function is used to generate a random number.
- Every time it is called, it gives a random number.
- If the developers add some logic with it, they can generate the random number within a defined range and if the range is not defined explicitly, it will return a totally random integer value.

Example

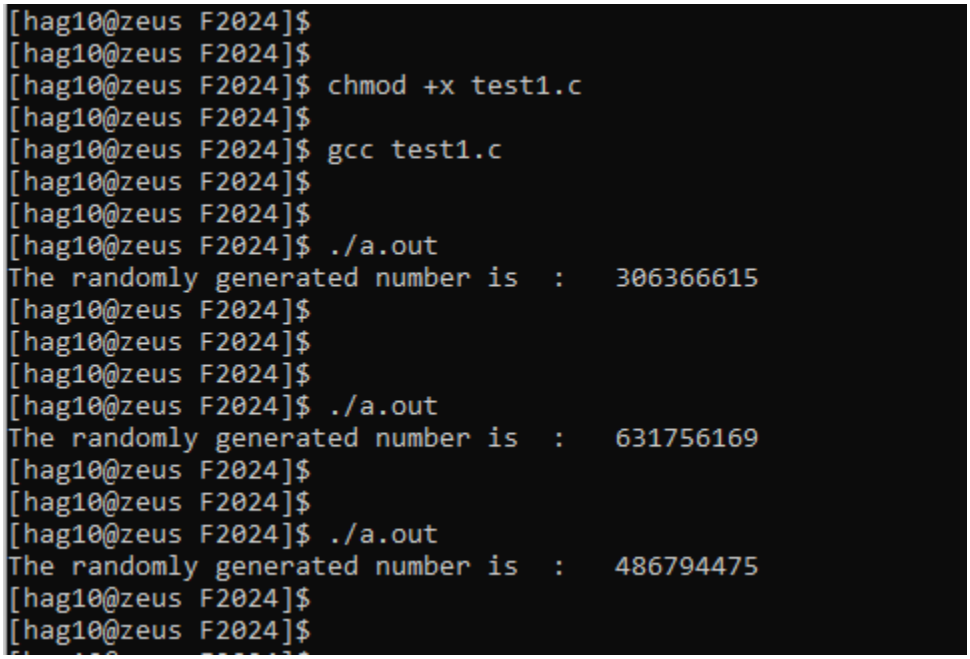
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{
   int rand_num;
   srand(time(NULL));

   printf("The randomly generated number is :  ");
   rand_num = rand() ;
   printf("%d\n", rand_num);
   return 0;
}
```

Sample Run

The randomly generated number is : 23273

Under Zeus :



Switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

How does C switch statement work

1. First, the **<Expression>** inside the switch clause is evaluated to an integral constant.
2. Its result is then compared against the case-value inside each case statement.
3. If a match is found, all the statements following that matching case label are executed, until a break or end of switch is encountered. This is a critical statement.
4. If no match is found, statements under the default case label are executed.

Example

```
#include <stdio.h>
int main()
{
  int x;
  x = 7;
  switch(x)
  {
    case 1:
       printf("\n\nOne\n\n");
       break;
    case 2:
       printf("\n\nTwo\n\n");
       break;
    case 3:
       printf("\n\nThree\n\n");
       break;
    case 4:
       printf("\n\nFour\n\n");
       break;
    case 5:
       printf("\n\nFive\n\n");
       break;
    default:
       printf("\n\nNumber is Out of Range\n\n");
       break;
  }

  return 0;
}
```

Example - 2

```
#include <stdio.h>

int main () {

   /* local variable definition */
   char grade = 'B';

   switch(grade) {
      case 'A' :
         printf("Excellent!\n" );
         break;
      case 'B' :
      case 'C' :
         printf("Well done\n" );
         break;
      case 'D' :
         printf("You passed\n" );
         break;
      case 'F' :
         printf("Better try again\n" );
      default :
         printf("Invalid grade\n" );
   }

   printf("Your grade is  %c\n", grade );

   return 0;
}
```

Arrays

A **array** is a data structure that holds elements of **one type** so that each element can be (efficiently) accessed using an **index**

In C, arrays are always indexed by **integer** values

Indices always run from 0 to some fixed, predetermined value

<type> <var> [<elements>]; defines a variable of an array type:

- <***type***> can be any valid C type, including user-defined types
- <***var***> is the name of the variable defined
- <***elements***> is the number of elements in the array (Note: Indices run from 0 to <elements>-1)

Example:

- char x[10]; defines the variable x to hold 10 elements of type char,

- y = x[5] accesses the 5th element of that array

Initializing Arrays

- Used an **explicit loop** to initialize the array

- List the initial values in the definition of the array:

```
int days per month[12] =
      {31,28,31,30,31,30,31,31,30,31,30,31};
```

- The number of values has to be smaller than or equal to the number of elements in the array

- If given an **explicit initializer**, we can omit the size of the array:

```
int days per month[] =
      {31,28,31,30,31,30,31,31,30,31,30,31};
```

The compiler will automatically allocate an array of sufficient size to hold all the values in the initializer

Example:

```
#include<stdio.h>
#define SIZE 10

int main() {
	int my_arr[SIZE] = { 34, 56, 78, 15, 43, 71, 89, 34, 70, 91 };
	int i, max, min;

	max = min = my_arr[0];

	for (i = 0; i < SIZE; i++) {
		// if value of current element is greater than previous value
		// then assign new value to max
		if (my_arr[i] > max) {
			max = my_arr[i];
		}

		// if the value of current element is less than previous element
		// then assign new value to min
		if (my_arr[i] < min) {
			min = my_arr[i];
		}
	}

	printf("Lowest value = %d\n", min);
	printf("Highest value = %d", max);

	// signal to operating system everything works fine
	return 0;
}
```

Sample Run

```
Lowest value = 15
Highest value = 91
```

No Safety Belts

C **does not** check if the index is in the valid range!

– If you access days per month[13] you might change some critical other data.

– The **operating system** may catch some of these wrong accesses, but do not rely on it!)

This is source of many of the **buffer-overflow** errors exploited by crackers and viruses to hack into systems!

Example

```
#include <stdio.h>
int main() {
	int arr[] = { 1, 2, 3, 4, 5 };

	printf("\n\narr [0] is %d\n\n", arr[0]);
	printf("\n\narr[100] is %d\n\n", arr[100]);

	// allocation memory to out of bound
	// element

	arr[10] = 11;
	printf("arr[10] is %d\n", arr[10]);
	return 0;
}
```

Sample Run

arr [0] is 1

arr[100] is 1869113198

arr[10] is 11

Character Arrays

Character arrays are the most frequent kind of arrays used in C
– They are used for I/O operations
– They are used for implementing string operations in C

To make the use of character arrays easier, we can use string constants to
initialize them.

The following definitions are equivalent:
– char hello[] = {'H','e','l','l','o','\0'};
– char hello[] = "Hello";
– char hello[6] = "Hello";

Notice that the string constant is automatically terminated by a NUL character!

Example

```
#include <stdio.h>
int main()
{
   char str[20];
   printf("Enter characters \n> ");
   scanf("%s", str);
   printf("%s\n", str);
   return 0;
}
```

Sample Run

```
Enter characters
> YXZ
YXZ
```

Question :

What happens if you enter more than 20 chars ???

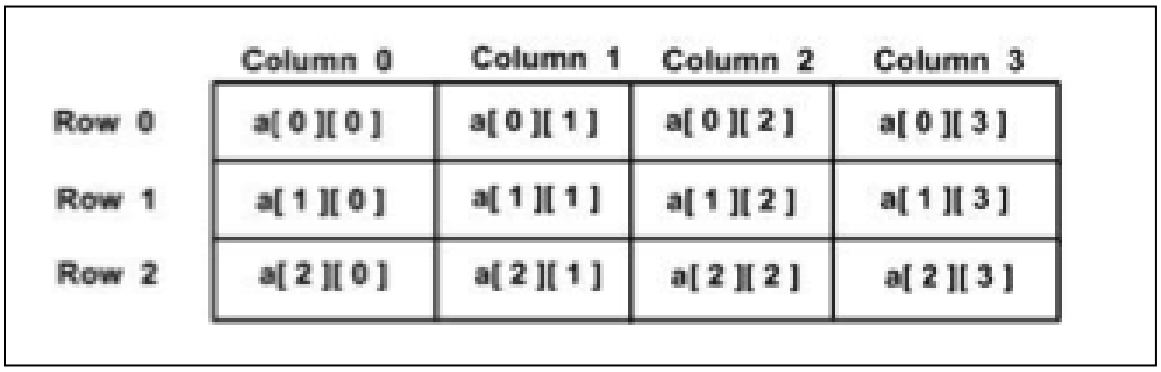
What happens if you enter integers instead of chars ???

Multidimensional Array

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table (matrix) which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −



Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = { { 0, 1, 2, 3 }, /*  initializers for row  indexed by 0 */
                { 4, 5, 6, 7 },  /*  initializers for row   indexed by 1 */
                { 8, 9, 10, 11 } /*  initializers for row   indexed by 2 */
        };
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Example

```
#include <stdio.h>

int main() {

	/* an array with 5 rows and 2 columns*/
	int a[5][2] = { { 0, 0 }, { 1, 2 }, { 2, 4 }, { 3, 6 },    { 4, 8 } };
	int i, j;

	/* output each array element's value */
	for (i = 0; i < 5; i++) {
		for (j = 0; j < 2; j++) {
			printf("a[%d][%d] = %d\n", i, j, a[i][j]);
		}
	}

	return 0;
}
```

Sample Run

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

Functions

Functions are the primary means of structuring programs in C

A function is a named subroutine

- It accepts several numbers of arguments, processes them, and (optionally) returns a result
- Functions also may have **side effects**, like I/O or changes to global data structures
- In C, any subroutine is called a **function**, whether it actually returns a result or is only called for its side effect

Note: A function hides its implementation

- To use a function, we only need to know its interface, i.e. its name, parameters, and return type

- We can improve the implementation of a function without affecting the rest of the program

Function can be reused in the same program or even different programs, allowing people to build on existing code

Function Definitions

A function definition consists of the following elements:

- Return type (or void) if the function does not return a value
- Name of the function
- Parameter list
- Function body

The name follows the same rules as variable names

The **parameter list** is a list of coma-separated pairs of the form <type> <name>

The body is a sequence of statements included in curly braces

Example:

```
int timesX(int number, int x) {

    return x * number;

}
```

Function Calling

A function is **called** from another part of the program by writing its name, followed by a parenthesized list of **arguments** (where each argument must have a type matching that of the corresponding parameter of the function)

If a function is called, control passes from the call of the function to the function itself

- The parameters are treated as local variables with the values of the arguments to the call
- The function is executed normally
- If control reaches the end of the function body, or a return statement is executed, control returns to the caller
- A return statement may have a single argument of the same type as the return type of the function. If the statement is executed, the argument of return becomes the value returned to the caller .

We can only call functions that have already been declared or defined at that point in the program

Example: Printing Character Frequencies

```
#include <stdio.h>

int print_freq(char c, int freq) ;  //  Function Prototype

int main()
{
   int freg;
   freg = print_freq('x',80) ;
   printf("\n\n%d\n", freg);
   return 0;
}

int print_freq(char c, int freq) {       //  Function Definition
        int i;
        printf("%c :", c);
        if (freq < 75) {
                for (i = 0; i < freq; i++)
                     putchar('#');

        } else {
                        printf("#....(%d)...#", freq);
                        freq = -1;
                };

        printf("\n", c);
        return freq;
}
```

Sample Run When call the function with x and 80

x :#....(80)...#

-1

Sample Run When call the function with x and 10

x :##########

-1

Rewrite the Fahrenheit! Celsius Program to use a function for the actual conversion

```
/* The original Program */
/*  A program that prints a Fahrenheit -> Celsius   conversion
    table */

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

int main(void) {
        int Fahrenheit;
        double Celsius;
        printf("Fahrenheit -> Celsius\n\n");
        for (   Fahrenheit = 0; Fahrenheit <= 300;
                Fahrenheit = Fahrenheit + 10)
                {
                        Celsius = (Fahrenheit - 32.0) * 5.0 / 9.0;
                        printf("%3d %7.3f\n", Fahrenheit, Celsius);
                }

        return EXIT_SUCCESS;
}
```

Solution

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

double F2C(int Fahrenheit);

int main(void) {
        int Fahrenheit;
        double Celsius;
        printf("Fahrenheit -> Celsius\n\n");
        for (   Fahrenheit = 0; Fahrenheit <= 300;
                Fahrenheit = Fahrenheit + 10)
                {
                        Celsius =F2C(Fahrenheit);
                        printf("%3d %7.3f\n", Fahrenheit, Celsius);
                }

        return EXIT_SUCCESS;
}


double F2C(int Fahrenheit)
{
    double Celsius;
    Celsius = (Fahrenheit - 32.0) * 5.0 / 9.0;
     return Celsius ;


}
```

Example : Passing Arrays to functions

```
#include <stdio.h>

/* function declaration */
double getAverage(int arr[], int size);

int main() {

        /* an int array with 5 elements */
        int balance[5] = { 1000, 2, 3, 17, 50 };
        double avg;

        /* pass pointer to the array as an argument */
        avg = getAverage(balance, 5);

        /* output the returned value */
        printf("Average value is: %f ", avg);

}

double getAverage(int arr[], int size) {

        int i;
        double avg;
        double sum = 0;

        for (i = 0; i < size; ++i) {
                sum += arr[i];
        }

        avg = sum / size;

        return avg;
}
```

Sample Run

Average value is: 214.400000

Example : Passing Arrays to functions

```
#include <stdio.h>
float average(float age[]);

const int SIZE = 6;

int main() {
        float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
        avg = average(age); // Only name of an array is passed as an argument
        printf("Average age = %.2f", avg);

}

float average(float age[]) {
        int i;
        float avg, sum = 0.0;
        for (i = 0; i <  SIZE ; ++i) {
                sum += age[i];
        }
        avg = (sum /  SIZE );
        return avg;
}
```

Sample Run

Average age = 27.08

Example : Passing two-dimensional array to a function

```
#include <stdio.h>

void displayNumbers(int num[2][2]);

int main() {
        int num[2][2], i, j;
        printf("Enter 4 numbers:\n");
        for (i = 0; i < 2; ++i)
                for (j = 0; j < 2; ++j)
                        scanf("%d", &num[i][j]);

        // passing multi-dimensional array to a function
        displayNumbers(num);
}

void displayNumbers(int num[2][2]) {
        int i, j;
        printf("Displaying:\n");
        for (i = 0; i < 2; ++i)
                for (j = 0; j < 2; ++j)
                        printf("%d\n", num[i][j]);
}
```

Sample Run

```
Enter 4 numbers:
1 2 3 4
Displaying:
1
2
3
4
```

Recursive Functions

As we stated above, functions can call other functions. They can also call themselves recursively

A recursive function always has to handle at least two cases:
– The **base case** handles a simple situation without further calls to the same function
– The **recursive cases** may do some work, and in between make recursive calls to the function for smaller (in some sense) subtasks

Recursion is one of the most important programming principles

Example: Printing Integers

We now want to print positive integer numbers to stdout, using only putchar()

Consider a number in base 10: 421 = 42 *10 + 1
We can split the task into two subtasks:

- Print everything but the last digit (recursively)
- Print the last digit

Base case: There are no digits to print any more

Basic operations:

- **To get the last digit, we use the modulus operator %**
- **To get rid of the last digit, we divide the number by the desired base** (remember, integer division truncates)

Example:

We want to print the number 421 in base 10

– Step 1: 421%10 = 1 and 421/10 = 42. Hence the last number to print is 1 and the rest we still have to print is 42

– Step 2: 42%10 = 2 and 42/10 = 4. The second last digit is 2, the rest is 4

– Step 3: 4%10 = 4 and 4/10 = 0. The next digit is 4

– Step 4: Our rest is 0, hence there is nothing to do but printing the digits in the right order

The same principle applies for other bases (just replace 10 by your base)

The Code

```
/* Write non-zero positive integer in any base to stdout */

#include <stdio.h>

void write_int_b_rekursive(int value, int base) ;

int main() {

        write_int_b_rekursive(421 , 10 );

}

void write_int_b_rekursive(int value, int base) {
        int digit;
        digit = value % base;
        value = value / base;
        if (value != 0) {
                write_int_b_rekursive(value, base);
        }
        putchar(digit + '0');
}
```

Problem: What happens if the input is 0?

Problem: What happens if you change the base to be 2 ?

Example

A recursive function to calculate the factorial value of a number ?

```
#include <stdio.h>

int fact(int n)  ;

int main() {
        int value = fact(5);

        printf("\n\n%d\n\n", value);

        return 0;
}


int fact(int n) {
        if (n <= 1)
                return 1;
        else
                return (n * fact(n - 1));
}
```

Sample Run

```
120
```

Example

A recursive function that will print each digit of a number vertically :

```
#include <stdio.h>

void decompose(int n) {
        if (n >= 10)          // Call decompose if there are   "more" digits in the
                              //  number  to print

                decompose(n / 10);

        printf("\n%d", n % 10);  // print the last   digit of n
}

int main(int argc, char *argv[]) {
        decompose(12345);
        printf("\n\n");
}
```

Sample Run

```
1
2
3
4
5
```

Enumerations in C

Enumeration data types can represent values from a **finite domain** using **symbolic** Names

In C , enumerations are created using the **enum** keyword are of **integer types** followed by a list of identifiers (**enumeration constants**) in curly brackets

The following code describes an enumeration data type for the data safe methods:

enum{ds_register, ds_store, ds_retrieve, ds_delete}

Since enumeration are actually integer types, we can assign specific values to the constants.

– We can even assign the same value to different constants!

Example : (also note preferred form of formatting for enums):

```
#include <stdio.h>
typedef enum {
        ds_register = 1,
        ds_store = 2,
        ds_retrieve = 3,
        ds_delete = 4,
        ds_forget = 4
} DS_operation;

int main() {
DS_operation  operation  =  ds_store;
printf("%d\n", operation);
printf("%d\n", operation + 1);
printf("%d\n", operation + 2);

}
```

Sample Run

```
2
3
4
```

Problem: What happens if the have the following ?

printf("%d\n", operation + 10);

Structure

A **structure** is a datatype that may have any number of **members**

- Members can have different types
- Members can have any other type (including arrays or other structures)
- Members are referred to by their name in the structure

Java analogy: A structure type is a class, but:

- **No** member functions
- **All** members are **public**

Structures are defined using the ***struct*** keyword, followed by an **optional name** and a **list of member definitions** in curly braces

- Each member definition is a normal variable definition, giving type and name of the member

Example :

```
struct key_assoc {
        int  key;
        int  value;  }   key_pair;
```

– It creates a variable key_pair with two members
– They can be referred to by name:

```
key_pair.value = 10;
...
if( key_pair.key == user_key )
{
      count++;
}
```

stuct and typedef

structures are usually used with **typedef:**

```
typedef struct key_assoc {
        int key;
        int value;
}  key_pair_t;

static    key_pair_t      key_value_array[10000];
```

– The first definition defines a new type, key_pair_t
– The second one creates an array of 10000 of these pairs

Using the name (struct key_assoc), we can refer to the array even before we have seen the full definition

– Important for self-referential data types using **pointers**

Examaple

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

typedef struct Books {
        char title[50];
        char author[50];
        char subject[100];
        int book_id;
} Book;

int main() {

        Book  book;

        strcpy(book.title, "C Programming");
        strcpy(book.author, "N B");
        strcpy(book.subject, "C Programming Tutorial");
        book.book_id = 6495407;

        printf("Book title : %s\n", book.title);
        printf("Book author : %s\n", book.author);
        printf("Book subject : %s\n", book.subject);
        printf("Book book_id : %d\n", book.book_id);

        return 0;
}
```

Sample Run

Book title : C Programming
Book author : N B
Book subject : C Programming Tutorial
Book book_id : 6495407

Array of Structures

```
#include<stdio.h>
#include <string.h>
struct student {
        int rollno;
        char name[10];
};
int main() {
        int i;
        struct student st[3];
        printf("Enter Records of 3 students");
        for (i = 0; i < 3; i++) {
                printf("\nEnter Rollno:  ");
                scanf("%d", &st[i].rollno);
                printf("\nEnter Name:  ");
                scanf("%s", &st[i].name);
        }
        printf("\nStudent Information List:");
        for (i = 0; i < 3; i++) {
                printf("\nRollno:%d, Name:%s", st[i].rollno,  st[i].name);
        }
        return 0;
}
```

Sample Run

```
Enter Records of 3 students
Enter Rollno:  1
Enter Name:  Jenny
Enter Rollno:  2
Enter Name:  Angela
Enter Rollno:  3
Enter Name:  Rory


Student Information List:
Rollno:1, Name: Jenny
Rollno:2, Name: Angela
Rollno:3, Name: Rory
```

Passing Struct to Functions - By Value

Example

```
#include <stdio.h>
struct student {
        char name[50];
        int score;
};

// function prototype

void displayDetail(struct student std);

int main() {

        // creating a student structure array variable
        struct student stdArr[3];

        int i;

         // taking user input
         for (i = 0; i < 3; i++) {
          printf("Enter detail of student #%d\n", (i+1));

          printf("Enter Student  Name: ");
          scanf("%s", stdArr[i].name);

          printf("Enter Score: ");
          scanf("%d", &stdArr[i].score);
         }
```

```
 // output
 for (i = 0; i < 3; i++) {
        printf("\nStudent #%d Detail:\n", (i+1));
        displayDetail(stdArr[i]);
 }

 return 0;
}

void displayDetail(struct student std) {

        printf("Student Name : %s\n", std.name);

         printf("Score: %d\n", std.score);
}
```

Sample Run

Enter detail of student #1
Enter Student Name: ABC
Enter score: 30

Enter detail of student #2
Enter Student Name: EFG
Enter score: 20

Enter detail of student #3
Enter Student Name: WXY
Enter score: 100

Student #1 Detail :
Enter Student Name: ABC
Enter score: 30

Student #2 Detail :
Enter Student Name: EFG
Enter score: 20

Student #3 Detail :
Enter Student Name: WXY
Enter score: 100

Passing Struct to Functions - By Reference

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

struct Lecturer
{
  char Lecturer_Name[50];
  int Total_Experience;
  int Experience_In_This_College;
};

void PassBy_Reference(struct Lecturer *Lecturers);

int main()
{
  struct Lecturer Lecturer1;

  printf("\nPlease Enter the Lecturer Name \n");
  scanf("%s",&Lecturer1.Lecturer_Name);

  printf("Please Enter Lecturers Total Years of Experience\n");
  scanf("%d",&Lecturer1.Total_Experience);

  PassBy_Reference(&Lecturer1);

  printf("\n Lecturer Name = %s", Lecturer1.Lecturer_Name);
  printf("\n Lecturers Total Years of Experience = %d",
Lecturer1.Total_Experience);

  return 0;
}
```

```
void PassBy_Reference(struct Lecturer *Lecturers)
{
  strcpy(Lecturers->Lecturer_Name, "XYZ");
  Lecturers->Total_Experience = 5;
}
```

Sample Run

```
Please Enter the Lecturer Name
ABC
Please Enter Lecturers Total Years of Experience
25

Lecturer Name =  XYZ
Lecturers Total Years of Experience = 5
```

Nested Structure

```
#include<stdio.h>

/* Declaration of structure */
struct address
{
    int houseno;
    char street[20];
    int stateno;
};

/* Declaration of structure */
struct student
{
    char name[30];
    int roll;
    struct address adrs;   /* Nested structure */
};
int main()
{
 struct student stud;
```

```
 printf("Enter name and roll number of student:\n");
 scanf("%s%d",stud.name, &stud.roll);
 printf("Enter street name, house number and state number:\n");
 scanf("%s%d%d",stud.adrs.street, &stud.adrs.houseno,
&stud.adrs.stateno);
 printf("Student detail is:\n");
 printf("Name: %s\tRoll: %d\n", stud.name, stud.roll);
 printf("Address:%s, House no. -%d, state: %d",stud.adrs.street,
stud.adrs.houseno, stud.adrs.stateno);
  return 0;
}
```

Sample Run

```
Enter name and roll number of student:
ABC
17
Enter street name, house number and state number:
Yellow
78
3
Student detail is:
Name: ABC                Roll: 17
Address: Yellow, House no. - 78, state: 3
```

Basic Pointer Operations in C

The most basic operations on pointers are:

- Given an object, return a pointer to it
- Given a pointer, give the object it points to (**dereference** the pointer)

C **uses** the **unary *** operator **for both pointer definition and pointer dereferencing.**

C **uses** **&** for **getting the address** of an existing object

- int var; int *p; defines **var** to be a **variable of type int** and **p** to be a variable of type **pointer to int**

- p = &var makes **p point to var** (i.e. p now stores the **address** of var)

- *p = 17; **assigns 17 to the int object that p points to** (in our example, it would **set var to 17**)

Pointers - A simple Example

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *x, int *y) {
        int z;
        z = *x;
        *x = *y;
        *y = z;
}
int main(void) {
        int var1 = 7, var2 = 42;
        printf("var1: %d var2: %d\n", var1, var2);
        swap(&var1, &var2);
        printf("var1: %d var2: %d\n", var1, var2);
        return EXIT_SUCCESS;
}
```

Sample Run :

```
var1: 7      var2: 42
var1: 42     var2: 7
```

Note that this technique is an example of a frequent way to simulate **call by reference** in C

- Instead of passing an object, we **pass a reference** to it
- Allows changes to the object inside the function
- Often cheaper (especially for big objects)

The C Preprocessor

The C preprocessor performs a **textual rewriting** of the program text before it is ever seen by the compiler proper

- **It includes the contents of other files**
- **It expands macro definitions**
- **It conditionally processes or removes segments of the program text**

Preprocessor directives **start with a hash #** and traditionally are written starting in the very **first column of the program text**

After preprocessing, the program text is free of all preprocessor directives

Normally, **gcc** will transparently run the preprocessor.

Run **gcc -E <file>** if you want to see the preprocessor output

Including Other Files: #include

The #include directive is **used to include other files**
(the contents of the named file replaces the #include directive)

Form 1: #include "file"

- The preprocessor will **search for file** in the current directory
- What happens if file is not found in the current directory, is **implementation-defined**
  - UNIX compilers will typically treat **file** as a pathname (that may be either absolute or relative)
  - If the file is not found, the compiler prints an error message and aborts

Form 2: #include <file>

- file will be **searched for in an implementation-defined** way
- UNIX compilers will typically treat file as a file name relative to the **system include directory**, /usr/include on the lab machines
- You can add to the list of directories that will be searched Using **gcc -I<includedir>**

Example: Include

myfile.c

```
A Poem

#include "mary"
```

mary

```
Mary had a little lamb,
Its fleece was white as snow;
And everywhere that Mary
went The lamb was sure to go.
```

$ gcc -E myfile.c

```
#include "myfile.c"
A Poem

Mary had a little lamb,
Its fleece was white as snow;
And everywhere that Mary went
The lamb was sure to go.
```

Include Discussion

Include directives are typically used for **sharing common declarations between different program parts**

Libraries (including the standard library) come with header files that define their **interface** by

- Defining data types and constants
- Declaring functions (and defining macros)
- Declaring variables

Note that included files can contain further #include statements (that will be automatically expanded by the preprocessor)

Simple Macro Definitions: #define

The **#define** directive is used to **define macros**

Simple Form : **#define <name> <replacement text>**

– This will define a macro for <name>, which has to follow the common rules for C identifiers (alphanumeric characters and underscore, should not start with a digit)

– Any normal occurrence of <name> after the definition will be replaced by

<replacement text>

– Replacement will not take place in strings!

– The macro definition normally ends at the end of the line, however, it can be extended to the next line by appending \ as the very last character of the line

Note that macro expansion even takes place within further macro definitions!

Most common use: Symbolic constants (e.g. **EOF**)

Simple #define Example

reality.c

```
#define      true   1
#define      false  0
void reality_check(void)
{
        if(true == false)
        {
                printf("Reality is broken!\n");
        }
}
```

$ gcc -E reality.c

```
# 4 "reality.c"
void reality_check(void)
{
        if(1 == 0)
        {
                printf("Reality is broken!\n");
        }
}
```

Example

```
#include <stdio.h>
#define PI  3.1415

int main() {
	float radius, area;
	printf("Enter the radius: ");
	scanf("%f", &radius);

	// Notice, the use of PI
	area = PI * radius * radius;

	printf("Area=%.2f", area);
	return 0;
}
```

Macros with Arguments

Macro definitions can also contain **formal arguments**

#define <name>(arg1,...,arg1) <replacement text>

If a macro with arguments is expanded, any occurrence of a formal argument **in the replacement text** is replaced with the actual value of the arguments in the macro call.

This allows a more efficient way of implementing small "functions"

- But: Macros cannot do recursion
- Macro calls have slightly different semantics from function calls
- Therefore, macros are usually only used for very simple tasks

By convention, preprocessor defined constants and many macros are written in **ALL CAPS** (using underscores for structure)

#define Examples

```
#define XOR(x,y)   ((!(x)&&(y))||((x)&&!(y)))       /* Exclusive or */
#define EQUIV(x,y)  (!XOR(x,y))

void test_macro(void)
{
        printf("XOR(1,1)     : %d\n", XOR(1,1));
        printf("EQUIV(1,0)  : %d\n", EQUIV(1,0));
}
```

$ gcc -E reality.c

```
# 4 "macrotest.c"
void test_macro(void)
{
       printf("XOR(1,1) : %d\n", ((!(1)&&(0))||((1)&&!(0))));
       printf("EQUIV(1,0): %d\n", (!((!(1)&&(0))||((1)&&!(0)))));
}
```

#define Caveats

Since macros work by textual replacement, **there are some unexpected effects:**

– Consider #define **FUN(x,y) x*y + 2*x**

- Looks innocent enough, but: FUN(2+3,4) expands into 2+3*4+2*2+3 (**and not** (2+3)*4+2*(2+3))
- To avoid this, **always** enclose each formal parameter in parentheses (unless you know what you are doing)

– Now consider **FUN(var++,1)**

- It expands into x++*1 + 2*x++
- Macro arguments may be evaluated more than once!
- Thus, avoid using macros with expressions that have side effects

Other frequent problems:

– Semicolons at the end of a macro definition **(wrong!)**
– "Invisible" syntax errors (run **gcc -E** and check the output if you cannot locate an error)

Conditional Compilation: #if/#else/#endif

We can use preprocessor directives to conditionally include or exclude parts of the program:

- Program parts may be enclosed in **#if** <expr>**/#endif** pairs
- <expr> has to be a **constant integer expression**
- If it evaluates to 0, the text in the **#if** <expr>/**#endif** bracket is ignored, otherwise it is included
- There also is an optional **#else** "branch"

Most frequent use : Test for the definition of macros

- defined(<macro>) evaluates to :
  - **1** if <macro> is defined (even as the empty string) **or**
  - **0** otherwise

- Short form:
  #if defined(<macro>) is equivalent to
  #ifdef <macro>,

  #if !defined(<macro>) is equivalent to
  #ifndef <macro>,

  – E.g.:
  ```
  #ifndef EOF
  #define EOF -1
  #endif
  ```

Example: #ifdef

cond preproc.c

```
#define hallo
#define fred barney
#define test 2+2
#if defined(hallo)
"Hallo"
#else
#ifdef fred
"Fred"
#endif
#endif
#if test
"test"
#endif
```

$ gcc -E cond preproc.c

"Hallo"

"test"

More on Preprocessor Definitions

You can use **#undef** <name> to get rid of a definition.

– This is most often used to start from a clean slate:

```
#undef true
#undef false
#define true 1
#define false 0
```

– It is, however, forbidden to undefine implementation-defined names

1. What is the output of the following :

```
#include <stdio.h>
#define MULTIPLY(a, b) a*b
int main()
{
    // The macro is expended as 2 + 3 * 3 + 5
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
```

2. What is the output of the following :

```
#include <stdio.h>
#define MULTIPLY(a, b) (a)*(b)
int main()
{
    printf("%d", MULTIPLY(2+3, 3+5));
    return 0;
}
```

3. What is the output of the following :

```
#include <stdio.h>

#define LIMIT 1000
int main()
{
   printf("%d",LIMIT);
   #undef LIMIT
   int LIMIT=1001;
   printf("\n%d",LIMIT);
   return 0;
}
```

4. What is the output of the following :

```
#include <stdio.h>
#define merge(a, b) a##b
int main()
{
    printf("%d ", merge(12, 34));
}
```

Explicit Declarations

Variables can be declared by adding the **extern** keyword to the syntax of a definition:

- extern int counter;
- extern char filename[MAXPATHLEN];

Scope Example

```
extern int global_count;

int abs_val (double number)
{
   double help = number;

   if(number < 0)
   {
      help = -1 * help;
      global_count++;
   }
}

int main()
{
   printf("\%7f\n", abs_val(-1.0));
}

int global_count;
```

Lifetime and Initialization of Variables

Global variables have **unlimited** lifetime

Most local variables (and function parameters) only have **limited** lifetime

Example: Static and Automatic Variables

```
#include <stdio.h>
#include <stdlib.h>
static int global_count = 0;
void counter_fun(void) {
        static int static_count = 0;
        int auto_count = 0;
        int pseudo_count = global_count;

        global_count++;
        auto_count++;
        static_count++;
        pseudo_count++;
        printf("Global: %3d Auto: %3d Static: %d Pseudo: %d\n",
         global_count, auto_count, static_count,  pseudo_count);
}

int main(void) {
        counter_fun();
        counter_fun();
        global_count = 0;
        counter_fun();
        counter_fun();
        return EXIT_SUCCESS;
}
```

$ gcc -o vartest vartest.c
$ ./vartest

Global: 1 Auto: 1 Static: 1 Pseudo: 1
Global: 2 Auto: 1 Static: 2 Pseudo: 2
Global: 1 Auto: 1 Static: 3 Pseudo: 1
Global: 2 Auto: 1 Static: 4 Pseudo: 2
$

C Standard Library Organization

– **ctype.h**: Character classes (+)
– **errno.h**: Error reporting for library functions
– **float.h**: Implementation limits for floating point numbers
– **limits.h**: Limits for other things
– locale.h: Localization support
– **math.h**: Mathematical functions
– setjmp.h: Non-local function exits
– **signal.h**: Signal handling
– stdarg.h: Support for functions with a variable number of arguments (as e.g. printf())
– stddef.h: Standard macros and typedefs
– **stdio.h**: Input and output (+)
– **stdlib.h**: Miscellaneous library functions (+)
– **string.h**: String (character array) handling
– **time.h**: Functions about time and date

Error Handling: errno.h

Library functions typically signal an error by returning an **out of range** value, i.e. a value that cannot possibly be correct

– For many functions that is **-1** or **NULL**

They communicate the **cause** of the error by setting the global int variable **errno** to a specific value

– At the program start, **errno** is guaranteed to have the value 0

– No library function will ever set errno to 0, but failed library functions will set it to an implementation-defined value encoding the cause of the error

Error codes have symbolic names (with **#define**):

- **EDOM**: (Required by the standard) **Domain** error for some math functions
- **ERANGE**: (Required by the standard) **Range** error for some math functions
- **EAGAIN**: (UNIX) Temporary problem, **Resource temporarily unavailable , try again**
- **ENOMEM**: (UNIX) Out of memory
- **EBUSY**: (UNIX) Some necessary **resource is already** in use
- **EINVAL**: (UNIX) Invalid argument to some function
- **ECOMM :** **Communication** error on send.
- **ECONNABORTED:** Connection aborted .
- **ECONNREFUSED:** Connection refused .
- **ECONNRESET:** Connection reset .
- **EDQUOT:** Disk quota exceeded .
- **EHOSTDOWN:** Host is down**.**

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
int main(int argc, char* argv[])
{
	char *res;
	printf("errno: %d\n", errno);
	res = strdup("Hallo"); /* Allocate space, copy the string to it */
	if(!res)
	{
		printf("Could not copy string, errno: %d = %d\n", errno, ENOMEM);
	}
	else
	{
		printf("All is fine, errno: %d\n", errno);
		free(res);
	}
	return EXIT_SUCCESS;
}
```

Example run :

```
errno: 0

All is fine, errno: 0
```

Example

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

extern int errno;

int main() {

        FILE * pf;
        int errnum;
        pf = fopen("unexist.txt", "rb");   // open file in binary

        if (pf == NULL) {

                errnum = errno;
                fprintf(stderr, "Value of errno: %d\n", errno);
                perror("Error printed by perror");
                fprintf(stderr, "Error opening file: %s\n",
              strerror(errnum));
        } else {

                fclose(pf);
        }

        return 0;
}
```

Sample Run

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Implementation limits for floating point numbers :

float.h

C Standard Library contains a set of various **platform-dependent** constants related to floating point values.

Example

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

int main() {
        printf("The maximum value of float = %.10e\n", FLT_MAX);
        printf("The minimum value of float = %.10e\n", FLT_MIN);

        printf("The number of digits in the number = %.10e\n", FLT_MANT_DIG);
}
```

Sample Run

```
The maximum value of float = 3.4028234664e+38
The minimum value of float = 1.1754943508e-38
The number of digits in the number = 1.1754943508e-38
```

Date and Time : Functions about time and date (time.h)

Example :

```
#include <stdio.h>
#include <time.h>
int main() {
        time_t begin, end;
        long i;
        begin = time(NULL);
        for (i = 0; i < 1500000000; i++)
                ;
        end = time(NULL);
        printf("for loop used %f seconds to complete the execution\n",
                difftime(end, begin));
        return 0;
}
```

Sample Run

```
for loop used 9.000000 seconds to complete the execution
```

The function uses the following structure

```
struct tm {
        int tm_sec; /* seconds,  range 0 to 59        */
        int tm_min; /* minutes, range 0 to 59        */
        int tm_hour; /* hours, range 0 to 23         */
        int tm_mday; /* day of the month, range 1 to 31  */
        int tm_mon; /* month, range 0 to 11          */
        int tm_year; /* The number of years since 1900   */
        int tm_wday; /* day of the week, range 0 to 6    */
        int tm_yday; /* day in the year, range 0 to 365  */
        int tm_isdst; /* daylight saving time          */
};
```

Example

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

int main(void) {

        time_t   rawtime = time(NULL);
        if (rawtime == -1) {
                puts("The time() function failed");
                return 1;    }

        struct tm *ptm = localtime(&rawtime);

        if (ptm == NULL) {

                puts("The localtime() function failed");
                return 1;
        }

        printf("The time is: %02d:%02d:%02d\n", ptm->tm_hour, ptm-
          >tm_min,ptm->tm_sec);

        return 0;
}
```

Sample Run : The time is: 21:18:23

Limit : limits.h

limits specify that a variable cannot store any value beyond these limits, for example an unsigned character can store up to a maximum value of 255.

Example

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

int main() {

        printf("The number of bits in a byte %d\n", CHAR_BIT);

        printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
        printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
        printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

        printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
        printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

        printf("The minimum value of INT = %d\n", INT_MIN);
        printf("The maximum value of INT = %d\n", INT_MAX);

        printf("The minimum value of CHAR = %d\n", CHAR_MIN);
        printf("The maximum value of CHAR = %d\n", CHAR_MAX);

        printf("The minimum value of LONG = %ld\n", LONG_MIN);
        printf("The maximum value of LONG = %ld\n", LONG_MAX);

        return (0);
}
```

Sample Run

```
The number of bits in a byte 8
The minimum value of SIGNED CHAR = -128
The maximum value of SIGNED CHAR = 127
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -2147483648
The maximum value of INT = 2147483647
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -9223372036854775808
The maximum value of LONG = 9223372036854775807
```

Math Library : math.h

```c
/* Example using math functions  */

#include <stdio.h>
#include <math.h>

int main(int argc, const char * argv[]) {
        /* Define temporary variables */
        double value;
        double result;

        /* Assign the value we will find the fabs of */
        value = -2.1;

        /* Calculate the absolute value of value */
        result = fabs(value);

        /* Display the result of the calculation */
        printf("The Absolute Value of %f is %f\n", value, result);
        printf("\n");
        for (int i = 1; i < 5; i++)
                printf("pow(3.2, %d) = %lf\n", i, pow(3.2, i));

        result = sqrt(100);
        // printing the calculated value
        printf("\nCalculated square root value is : %.2lf\n", result);

        result = sin(100);
        // printing the calculated value
        printf("\nvalue in sin is = %.4f\n", result);

        result = cos(100);
        // printing the calculated value
        printf("\nvalue in cos is = %.4f\n", result);

        result = tan(100);
        // printing the calculated value
        printf("\nvalue in tan is = %.4f\n", result);
```

```
        result = pow(15, 4);
        // printing the result
        printf("\nThe resulted power output is : %.3lf\n", result);


        value = 2.7;
        /* finding log(2.7) */
        result = log(value);
        printf("\nlog(%lf) = %lf", value, result);

        value = 0;
        printf("\n\nThe exponential value of %lf is %lf\n", value,  exp(result));

        printf("\nThe exponential value of %lf is %lf\n", value + 1, exp(value + 1));

        printf("\nThe exponential value of %lf is %lf\n", value + 2, exp(value + 2));

        return 0;
}
```

Sample Run

The Absolute Value of -2.100000 is 2.100000

pow(3.2, 1) = 3.200000
pow(3.2, 2) = 10.240000
pow(3.2, 3) = 32.768000
pow(3.2, 4) = 104.857600

Calculated square root value is : 10.00

value in sin is = -0.5064

value in cos is = 0.8623

value in tan is = -0.5872

The resulted power output is : 50625.000

log(2.700000) = 0.993252

The exponential value of 0.000000 is 2.700000

The exponential value of 1.000000 is 2.718282

The exponential value of 2.000000 is 7.389056

Some C Character Classes

All character class functions accept and return **int** values

– Behavior is only defined if the input is from the range of **unsigned char** or **EOF**
– Each function returns true (non-0) if the character is in the range, **0** otherwise

Character class test functions

– **isdigit(c):** Digits, i.e. **{0-9}**
– **isalpha(c):** Upper and lower case characters (**{a-z,A-Z},** in some locales additional characters, e.g. umlauts like ¨a, ¨O,. . .
– **isalnum(c):** Equivalent to (**isdigit(c)||isalpha(c))**
– **iscntrl(c):** Control characters, i.e. non-printable characters (in ASCII, those are characters with codes 0 to 31 and 127)
– **isxdigit(c):** Hexadecimal digits, {**0-9,a-f,A-F**}
– **islower(c):** Lower case letters
– **isupper(c):** Upper case letters
– **ispunct(c):** Printing characters that are neither letters, digits, nor space
– **isprint(c):** Normal, printable characters

Example: isdigit() function

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

int main() {
        char c;
        c = '5';
        printf("Result when numeric character is passed: %d", isdigit(c));

        c = '+';
        printf("\nResult when non-numeric character is passed: %d", isdigit(c));

        return 0;
}
```

Sample Run

```
Result when numeric character is passed: 1
Result when non-numeric character is passed: 0
```

Example: isalpha() function

```
#include <stdio.h>
#include <ctype.h>
int main()
{
        char c;
        c = 'Q';
        printf("\nResult when uppercase alphabet is passed: %d",   isalpha(c));

        c = 'q';
        printf("\nResult when lowercase alphabet is passed: %d",   isalpha(c));

        c='+';
        printf("\nResult when non-alphabetic character is passed: %d",  isalpha(c));

        return 0;
}
```

Sample Run

```
Result when uppercase alphabet is passed: 1
Result when lowercase alphabet is passed: 1
Result when non-alphabetic character is passed: 0
```

Example: isalnum( ) function return value

```
#include <stdio.h>
#include <ctype.h>
int main() {
        char c;
        int result;

        c = '5';
        result = isalnum(c);
        printf("When %c is passed, return value is %d\n", c, result);

        c = 'Q';
        result = isalnum(c);
        printf("When %c is passed, return value is %d\n", c, result);

        c = 'l';
        result = isalnum(c);
        printf("When %c is passed, return value is %d\n", c, result);

        c = '+';
        result = isalnum(c);
        printf("When %c is passed, return value is %d\n", c, result);

        return 0;
}
```

Sample Run

```
When 5 is passed, return value is 1
When Q is passed, return value is 1
When l is passed, return value is 1
When + is passed, return value is 0
```

Character Class Conversion Functions

There are two functions for converting characters from one class to another:

– **tolower(c)** converts upper case characters to lower case characters
– **toupper(c)** converts lower case characters to upper case characters

Both functions return the character unchanged, if it is not a upper or lower case character, respectively

- iscntrl() function checks whether a character (passed to the function) is a **control** character or not. If character passed is a control character, it returns a non-zero integer. If not, it returns 0

Example: using iscntrl()

```
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(void) {
        int c;
        printf("\nEnter characters     ");

        while (((c = getchar()) != EOF)) {
                if (iscntrl(c)) {
                        printf("<control-character %d\n", c);
                } else {
                        putchar(toupper(c));
                }
        }
        return EXIT_SUCCESS;
}
```

Sample Run

```
Enter characters     abcdeEOF
ABCDEEOF<control-character 10
123
123<control-character 10
abc
ABC<control-character 10
```

Example: C isxdigit() function

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

int main() {
        char c;

        c = '5';
        printf("Result when hexadecimal character %c is passed: %d", c,
                        isxdigit(c));

        c = 'g';
        printf("\nResult when hexadecimal character %c is not passed:
                %d", c,  isxdigit(c));

        return 0;
}
```

Sample Run

Result when hexadecimal character 5 is passed: 1
Result when hexadecimal character g is passed: 0

Example: Program to check punctuation using ispunct()

```
#include <stdio.h>
#include <ctype.h>
int main() {
        char c;
        int result;

        c = ':';
        result = ispunct(c);

        if (result != 0) {
                printf("%c is a punctuation", c);
        } else {
                printf("%c is not a punctuation", c);
        }

        return 0;
}
```

Sample Run

```
: is a punctuation
```

Example: Print all Punctuations

```
#include <stdio.h>
#include <ctype.h>
int main() {
        int i;
        printf("All punctuation marks in C programming are: \n");
        for (i = 0; i <= 127; ++i)
                if (ispunct(i) != 0)
                        printf("%c ", i);
        return 0;
}
```

Sample Run

```
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

Example: isprint() function

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

int main() {
        char c;

        c = 'Q';
        printf("Result when a printable character %c is passed to
                isprint(): %d", c, isprint(c));

        c = '\n';
        printf("\nResult when a control character %c is passed to
          isprint(): %d", c, isprint(c));

        return 0;
}
```

Sample Run

Result when a printable character Q is passed to isprint(): 1

Result when a control character is passed to isprint(): 0

Strings

Strings are not part of the C language proper

– String literals are supported
– Limited support by functions the C standard library

String-handling functions are operating on char* (pointer to char) values

– It is the responsibility of the program to make sure that there is sufficient space for the operations available!

Convention for strings:

– Strings are \0 terminated arrays of character
– Important: Size of the array is not taken into account!

```
char excess[10000] = "a";   /* String length 1, takes up two
                               characters, a and \0 */
```

String Functions from <string.h>

*char *strcpy(char* s, const char *ct)*

– Copy a '**\0**'-terminated string from ct to s
– Returns s
– **s** must point to a sufficiently large area of memory!
– Note: For all string functions that copy strings, source and target areas may not overlap (otherwise, behavior is undefined)

*char *strncpy(char* s, const char *ct, size_t n)*

– As **strcpy()**, but copies at most n characters
– Note: If **ct** is longer than **n, s** will not be **'\0'**-terminated
– If **ct** is shorter than n, then the result will be padded with additional '\0' characters (i.e. s must always have space for n characters, even if **ct** is shorter than **n** characters)

*size_t strlen(const char *cs)*

– Return the **length** of the string at **cs**
– Does not count the trailing **'\0'**

*char *strcat(char *s, const char *ct)*

– **Concatenates ct** at the **end of s**
– Returns s
– Result is always '\0' terminated

*char *strncat(char *s, const char *ct, size_t n)*

– As **strcat**(), but copies at most **n** characters from ct
– Result is always **'\0'** (even if ct is longer than n

strcmp(const char cs, const char* ct)*

– **Compare** two strings in the lexical extension of the natural order on characters
– First differing character decides which string is bigger (including terminating '\0', i.e. a substring is always smaller than a superstring)
– Return value: Integer <0, if cs is smaller, >0, if ct is smaller, or 0 if both are equal

int strncmp(const char cs, const char* ct, size_t n)*

– As **strcmp()**, but compare at most **n** characters

*char *strchr(const char *s, int c)*

– Return pointer to the first occurrence of **c** in cs (or **NULL**, if c is not present in cs)

*char *strrchr(const char *s, int c)*

– Return pointer to the last occurrence of **c** in cs (or **NULL**)

Example :

An algorithm to copy a string without using the ***strcpy()*** function

```
#include <stdio.h>
#include <string.h>
int main()
{
        int i;
        char s[4],t[4];
        s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
        i=0;
        while (s[i] != '\0')
        {
                t[i]=s[i];
                i++;
        }
        t[i]='\0';
        printf("%s\n",t);

        strcpy(t,s);
        printf("%s\n",t);
}
```

Sample Run

```
Sue
Sue
```

Example :

An algorithm to concatenate 2 strings using the ***strcat()*** function

Example:

```
char *t="World";
char s[10] = "Hello";
strncat(s,t,3); /* Ok, t now points to "HelloWor" */
strcat(s,t); /* Error: "HelloWorld" requires 11 character ('\0'!) */
```

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

int main() {
        char src[50], dest[50];

        strcpy(src, "This is source");
        strcpy(dest, "This is destination");

        strcat(dest, src);

        printf("Final destination string : |%s|", dest);

        return (0);
}
```

Sample Run

|This is destination This is source|

Example :

An algorithm to compare two strings without using the ***strcmp()*** function

```
#include <stdio.h>
#include <string.h>
int main()
{
        int i,a;
        char s[4],t[4];
        s[0]='S'; s[1]='u'; s[2]='e'; s[3]='\0';
        t[0]='S'; t[1]='u'; t[2]='n'; t[3]='\0';
        i=0; a=0;
        while (a == 0)
        {
                if (s[i] < t[i]) a=-1;
                if (s[i] > t[i]) a=1;
                if (s[i] == '\0' || t[i] == '\0')
                    break;
                i++;
        }
        printf("%d\n",a);

        a=strcmp(s,t);
        printf("%d\n",a);
}
```

Sample Run

Example

An Algorithm to searches for the first occurrence of the character c (an unsigned char) in the string pointed to by the argument str using ***strchr()***

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

int main() {
        const char str[] = "Unix Systems Programming";
        const char ch = 'm';
        char *ret;

        ret = strchr(str, ch);

        printf("String after |%c| is - |%s|\n", ch, ret);

        return (0);
}
```

Sample Run

String after |m| is - |ms Programming|