Introduction To
C Programming Language

Bibliography:

1. B.Kernighan, D.Ritchie. The C Programming Language. ANSI C.
2. Class Lectures.

Working Environment:

1. Linux Ubuntu.
2. C compiler gcc.

History of C Programming language

- ▪ פותחה ותוכננה בתחילת שנות השבעים ע"י דניס ריצ'י במעבדות חברת בל.
- ▪נועדה במקור לפיתוח ליבת מערכת ההפעלה UNIX.
- ▪ה Unix Kernel כתוב בשפת C.
- ▪1978 שנת ההכרזה של C כשפת תכנות.
- ▪שפת C בהתחלה נכתבה ע"י שפת B.
- ▪מהר מאד התפשטה והפכה לשפת תכנות מאד פופלרית.
- ▪מאז ההכרזה על C נולדו מספר גרסאות שונות לשפה עם הבדלים קלים.
  - ✓ נולד הצורך בלקבוע תקן לשפה.
  - ✓ 1989 נולד התקן הראשון ANSI C ואשר גם נודע בשם C89
  - ✓ 1990 אומץ ע"י ISO ונודע בשם תקן C90 עם שינויים מינוריים.
  - ✓ 1999 תקן C99
  - ✓ 2011 נעשה עדכון לתקן אשר נודע בשם C11
  - ✓ הספר אשר נכתב ע"י ממציאי השפה The C Programming Language מתעד את השפה ומהווה קו מנחה לא רשמי לשפה.

Features of C Programming language

- User friendly , Compared to other previous languages

- Low Level Language Support

✓ Closely Related to Lower level Language such as “Assembly Language“.

✓ It is easier to write assembly language codes in C programming.

- Portability

✓ C Programs (source) are portable - can be run on any compiler with very few modification

✓ Compiler and Preprocessor make it possible for C programs to run on different PC.

✓ Compiled programs are machine dependent – they can run only on where they were compiled.

- Powerful – provides rich features

✓ Provides Wide verity of ‘**Data Types**‘

✓ Provides Wide verity of ‘Functions’

✓ Provides useful Control & Loop Control Statements

- Bit Manipulation

✓ It provides wide verity of bit manipulation Operators.

✓ different operations can be performed at bit level.

✓ Memory management at bit level e.g. using Structure to manage Memory at Bit Level.

- Powerful Pointer support

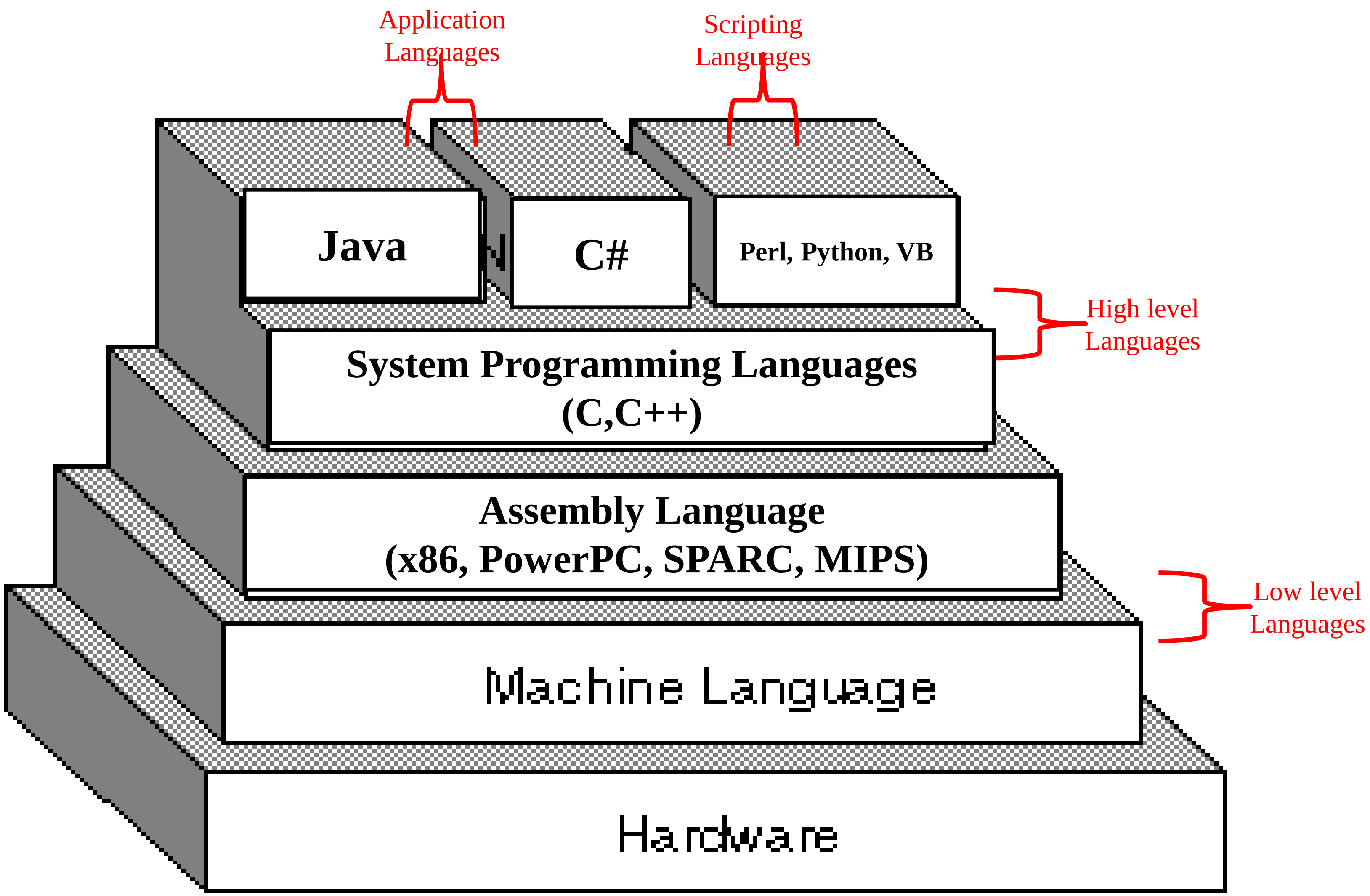
✓ Pointers has direct access to memory.

Popularity of C language

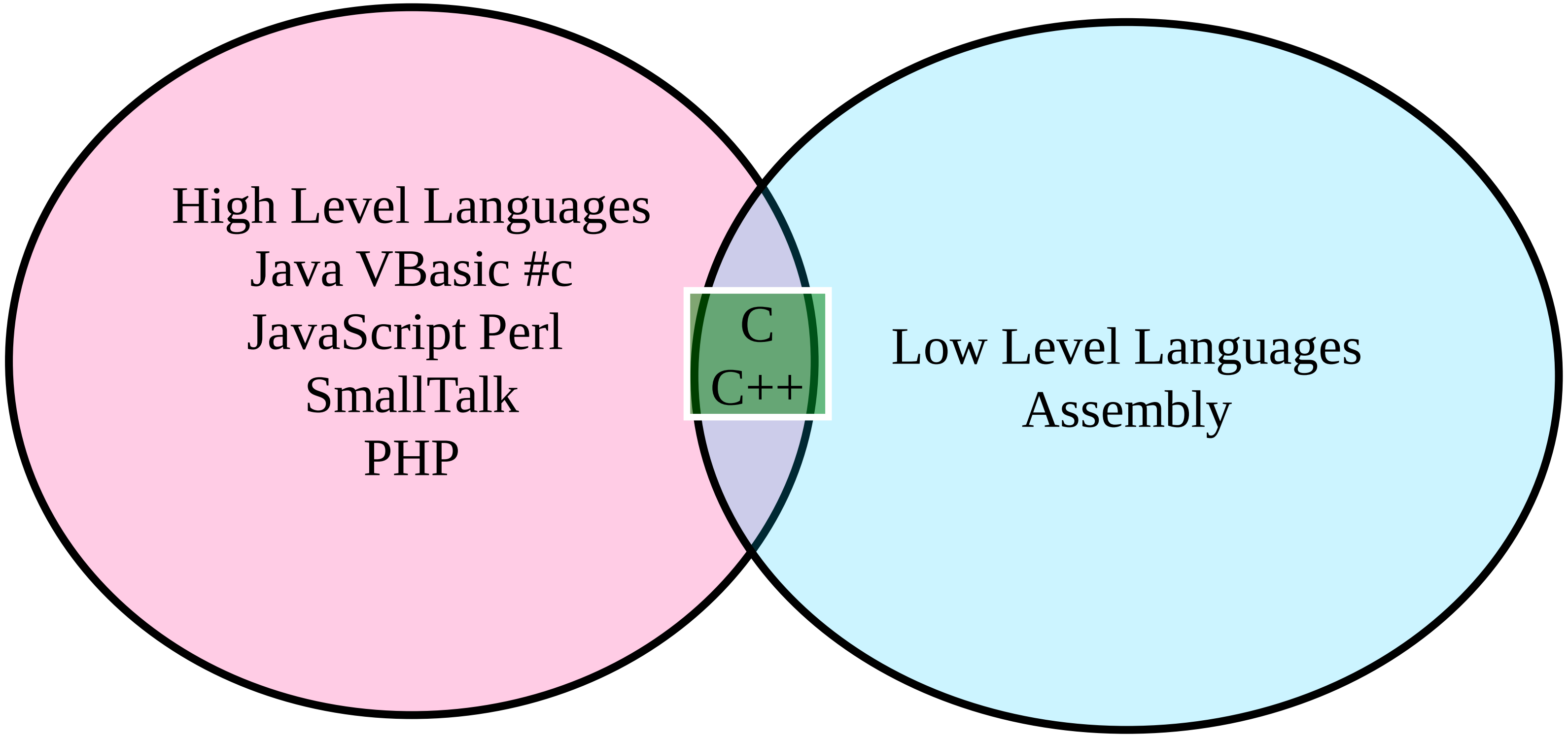
C language is used for creating computer applications. It is widely used in:

- ✓ Writing Embedded software programs.
- ✓ Creating Compilers of other different Languages - By translating input from other language to lower level machine dependent language.
- ✓ Implementing Operating System Operations.
- ✓ UNIX kernel is completely developed in C Language.
- ✓ Firmware of various electronics products - industrial and communications products which use micro-controllers.
- ✓ Developing verification software, test code, simulators etc. for various applications and hardware products.

Layers Of
Programming Languages



C and C++
Can Talk To Hardware



High Level Languages
Java VBasic #c
JavaScript Perl
SmallTalk
PHP

C
C++

Low Level Languages
Assembly

Compiling
And Running C Programs

- C compiler is needed in order to be able to write and compile C programs.
- ✓ variety of C-compilers are distributed in the market.
- ✓ we will use GNU-gcc compiler which supports the ANSI-C standard.
- ✓ We will develop our programs under a Linux based system – Ubuntu.
- ✓ gcc is a complier which allow compiling C source programs.

Compiling
C Programs

The entire mechanism is usually called compilation.
It includes 3 steps:

1) Preprocessor:

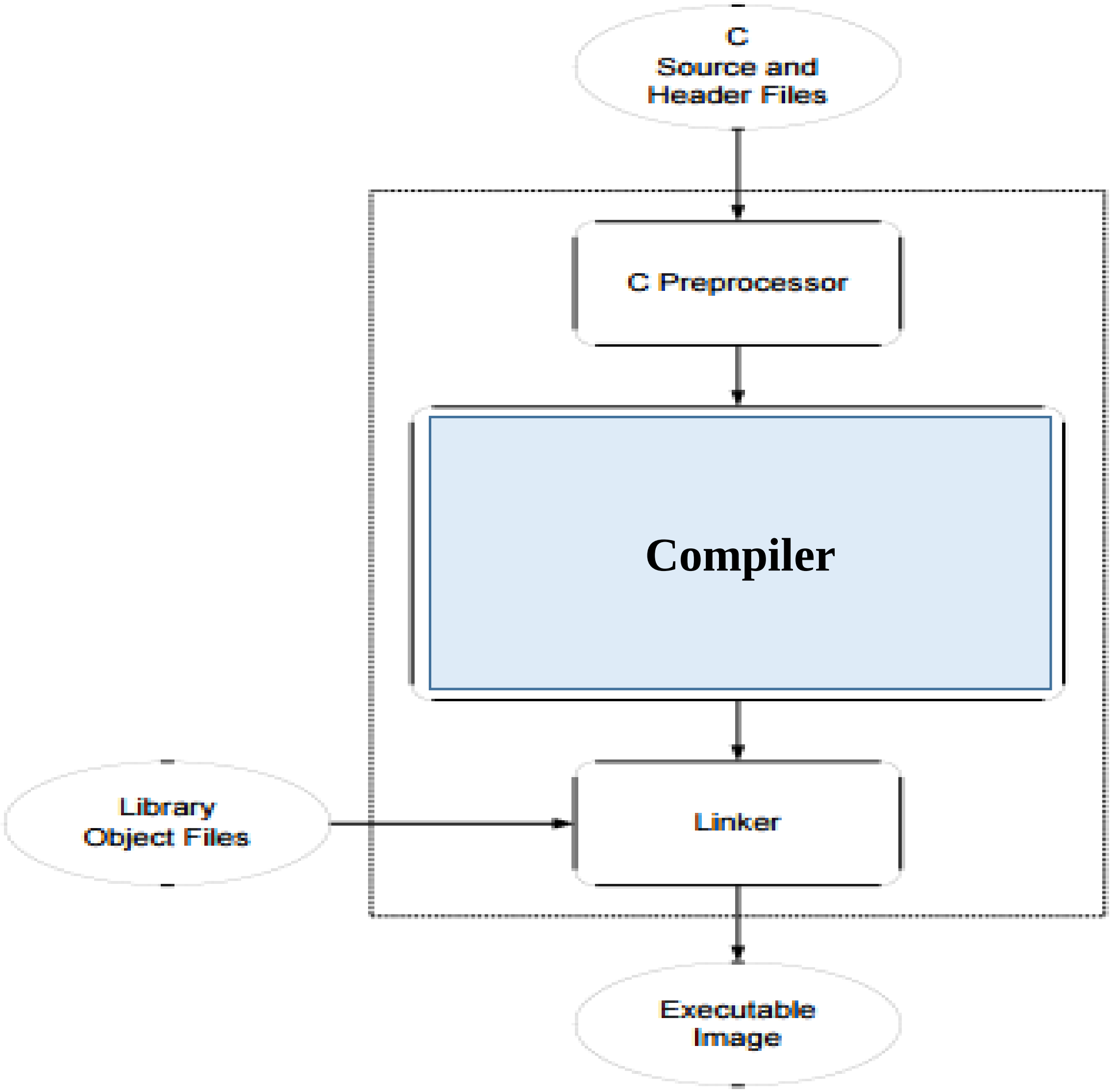
- ✓ Macro substitution.
- ✓ Conditional compilation.
- ✓ Source-level transformations → Output is still C.

2) Compiler:

- ✓ Generates object files →Machine instructions.

3) Linker:

- ✓ Combine object files (including libraries) into executable image. i.e. links together a number of object files to produce a binary file which can be directly executed



C-Preprocessor

1. Source code files are processed by the preprocessor before being complied. A C-Preprocessor is a separate program which can run independently. However, it is invoked automatically by the C-Compiler before the compilation stage.

2. The preprocessor converts the source code file into another source code file i.e. modify and expand the original source code file. That modified file could be stored in memory before being sent to the compiler or even exists as a real file in the file system.

3. Preprocessor commands start with "#“. for example:

- ✓ #define: mainly used to define constants e.g. → #define MAX_ARRAY_SIZE 1000
- ✓ #include: usually used to include header files e.g. → #include <stdio.h>

  this will add the contents of <stdio.h> into the source code file at the location of the #include statement before it gets compiled. This will allow using functions such as printf and scanf, whose declarations are located in the file stdio.h. (include allows re-use of previously written code in C programs).

Files Used
by C Programs.

1. source code files: contain function definitions. Source code files have names which end with ".c" (e.g. hello.c).

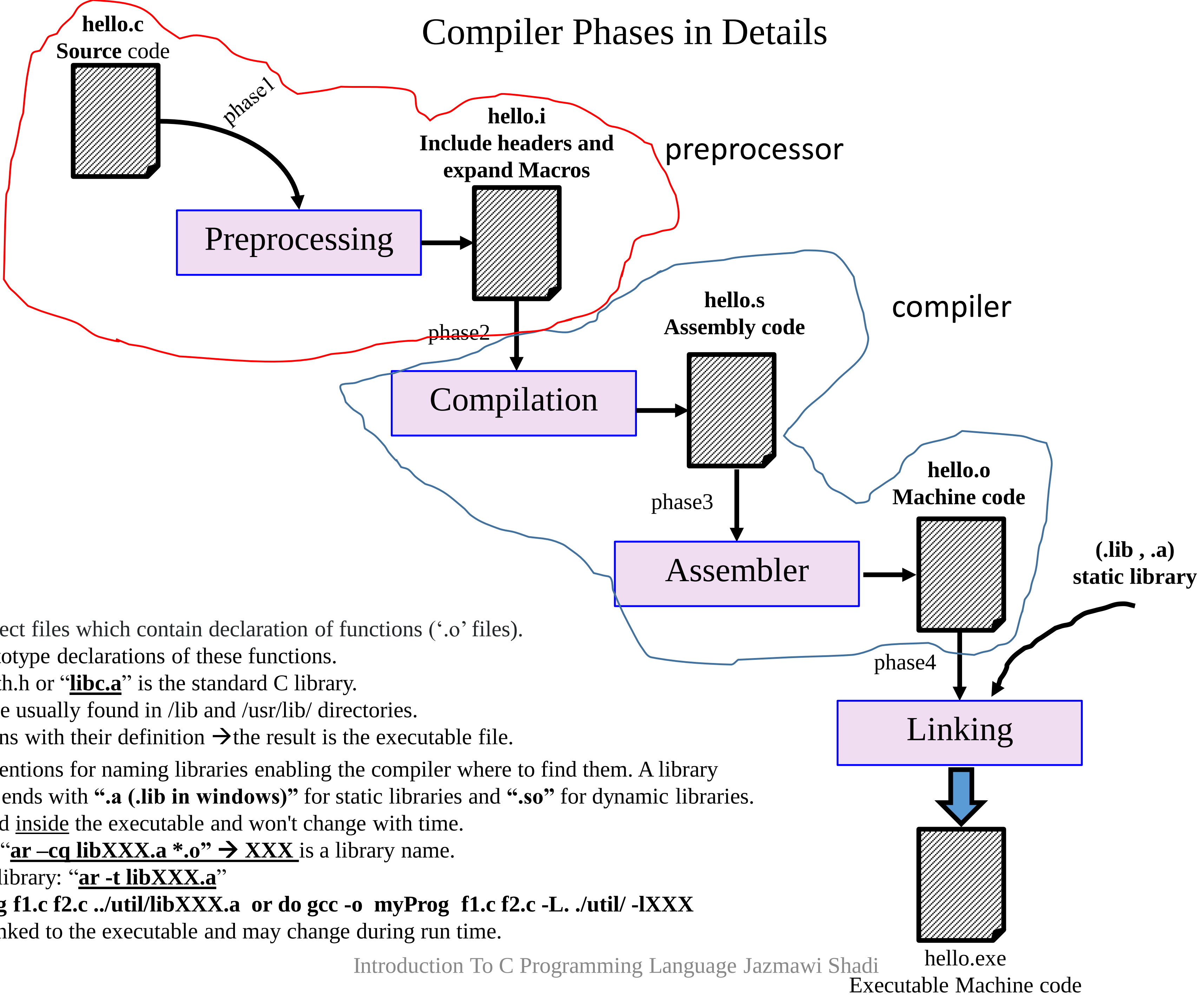
2. Header files. contain preprocessor statements and function declarations (prototypes) which allow source code files to access externally-defined functions. Header files end with ".h".

3. Object files. Binary output files generated by the compiler and consist of function definitions in binary form (not executable). Object files end with ".o" by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in ".obj".

4. Executable files. Binary executable files generated by the **Linker**. Binary executables have no special suffix on Unix operating systems, although they generally end in ".exe" on Windows.

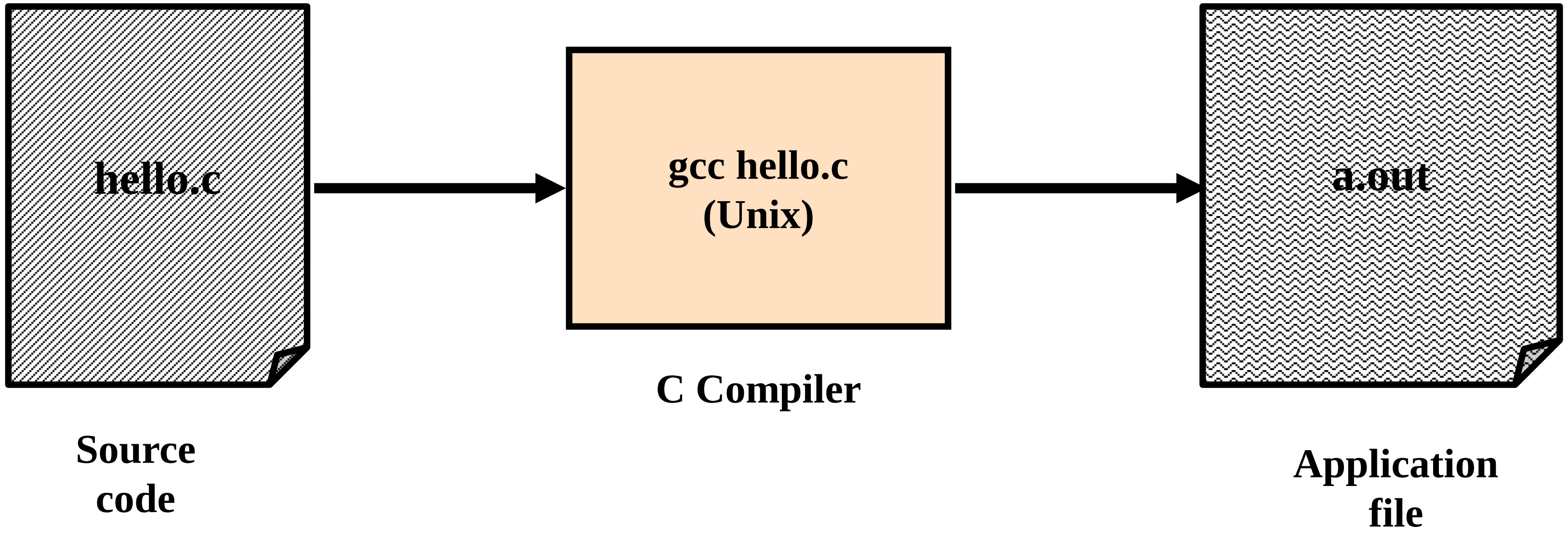
5. Others: There are other kinds of files e.g. libraries which have the suffix ".a" and shared libraries which have the suffux ".so“.

Compiler Phases in Details



- library is a collection of object files which contain declaration of functions (‘.o’ files).
- header files contain the prototype declarations of these functions.
- E.g. “**libm.a**” related to math.h or “**libc.a**” is the standard C library.
- Standard system libraries are usually found in /lib and /usr/lib/ directories.
- linker links those declarations with their definition →the result is the executable file.
- There are a number of conventions for naming libraries enabling the compiler where to find them. A library filename starts with **lib** and ends with **“.a (.lib in windows)”** for static libraries and **“.so”** for dynamic libraries.
- static library: ‘lib’ file linked inside the executable and won't change with time.
  - to create a static library: “**ar –cq libXXX.a *.o” → XXX** is a library name.
  - to show content of static library: “**ar -t libXXX.a**”
  - to link all**: gcc -o myProg f1.c f2.c ../util/libXXX.a or do gcc -o myProg f1.c f2.c -L. ./util/ -lXXX**
- dynamic library: ‘dll’ file linked to the executable and may change during run time.

Compiling C Programs
using gcc compiler



hello.c

gcc hello.c
(Unix)

a.out

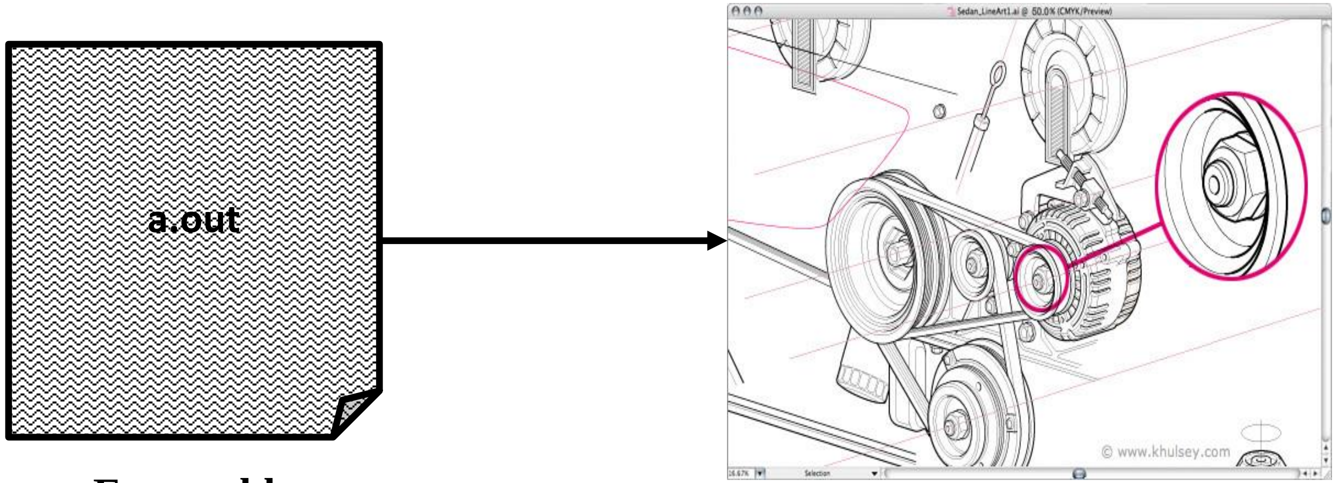
C Compiler

Source
code

Application
file

- hello.c קובץ טקסט אשר מכיל הוראות בשפת C.
- התוכנית hello.c עוברת הידור ע"י gcc Compiler שהינו תומך תקן ANSI-C מעל למערכת Unix.
- תוצאת ההידור נשמרת בקובץ בינארי אשר מקבל שם דיפולטיבי a.out (שניתן לשנות אותו – נראה בהמשך).

Running
C Programs



Executable file

CPU

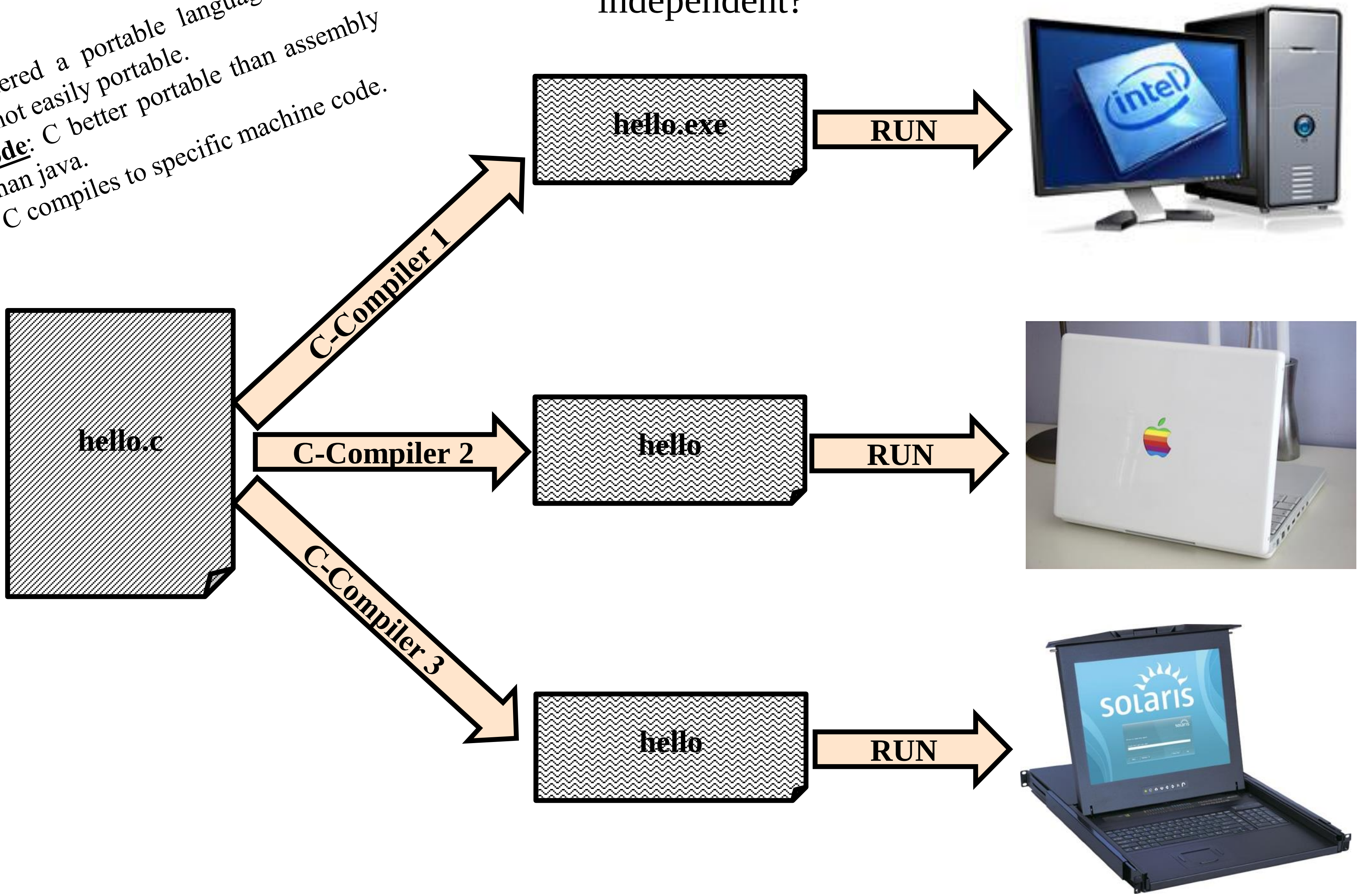
- **a.out הוא הקובץ אשר עבר קומפילציה ע"י gcc compiler מעל למערכת Unix (במערכות של windows הקובץ המתקבל הוא קובץ exe) ומיועד להרצה ישירה על המכונה.**

is C a platform independent?

C is considered a portable language, but in practice is not easily portable.

Source code: C better portable than assembly but less than java.

Binary: C compiles to specific machine code.



ANSI-C Support

- Any program written only in **Standard C** and **without any hardware-dependent assumptions**, will run correctly on any platform with a conforming C implementation, within its resource limits. Without such precautions, programs may compile only on a certain platform or with a particular compiler, due, for example, to the use of non-standard libraries, such as GUI libraries, or to a reliance on compiler- or platform-specific attributes such as the exact size of data types and byte endianness.

- Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware. Unfortunately it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine.

Keywords
Reserved Words

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

C Standard Library

The C Standard Library (**libc**) is a set of function declarations, constants, type and macro definitions used as a reference manual for C programming as specified in the ANSI C standard.

Header files: The application programming interface (API) of the C standard library is declared in a number of header files.

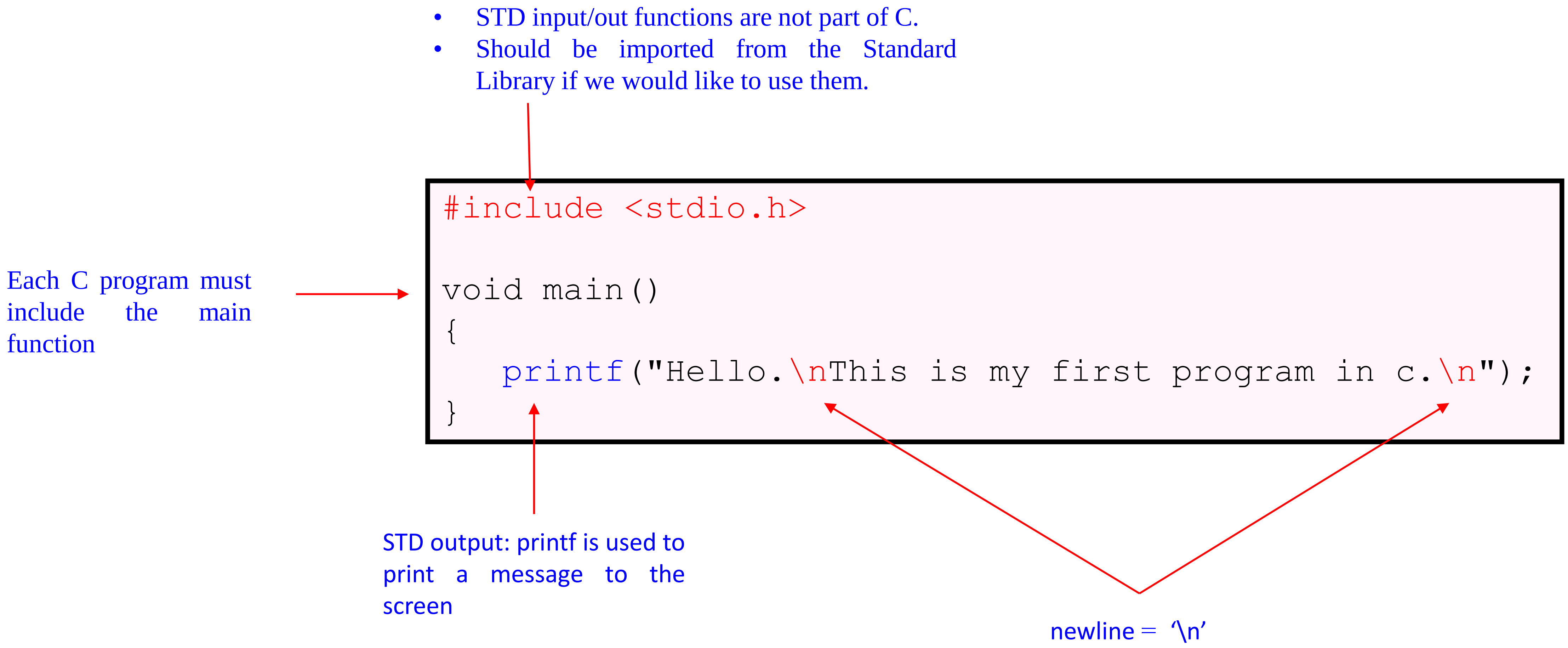
✓ Each header file contains one or more function declarations, data type definitions and macros
✓ shared between several source files

Some commonly used header files:

✓ string.h defines string handling functions.
✓ ctype.h for character manipulation
✓ math.h define mathematical functions such as sin() and cos()
✓ stdlib.h define utility functions such as malloc() and rand()
✓ assert.h contains the assert debugging macro
✓ stdarg.h for accessing a varying number of arguments passed to functions
✓ stdio.h defines core input and output functions
✓ limits.h, float.h constants which define type range values such as INT_MAX

A simple C program

To create a C program :
Step 1: create the source file using any text editor or IDE.
Step 2: the source file must be with the extension “XXX.c”
Step 3: compile the program.



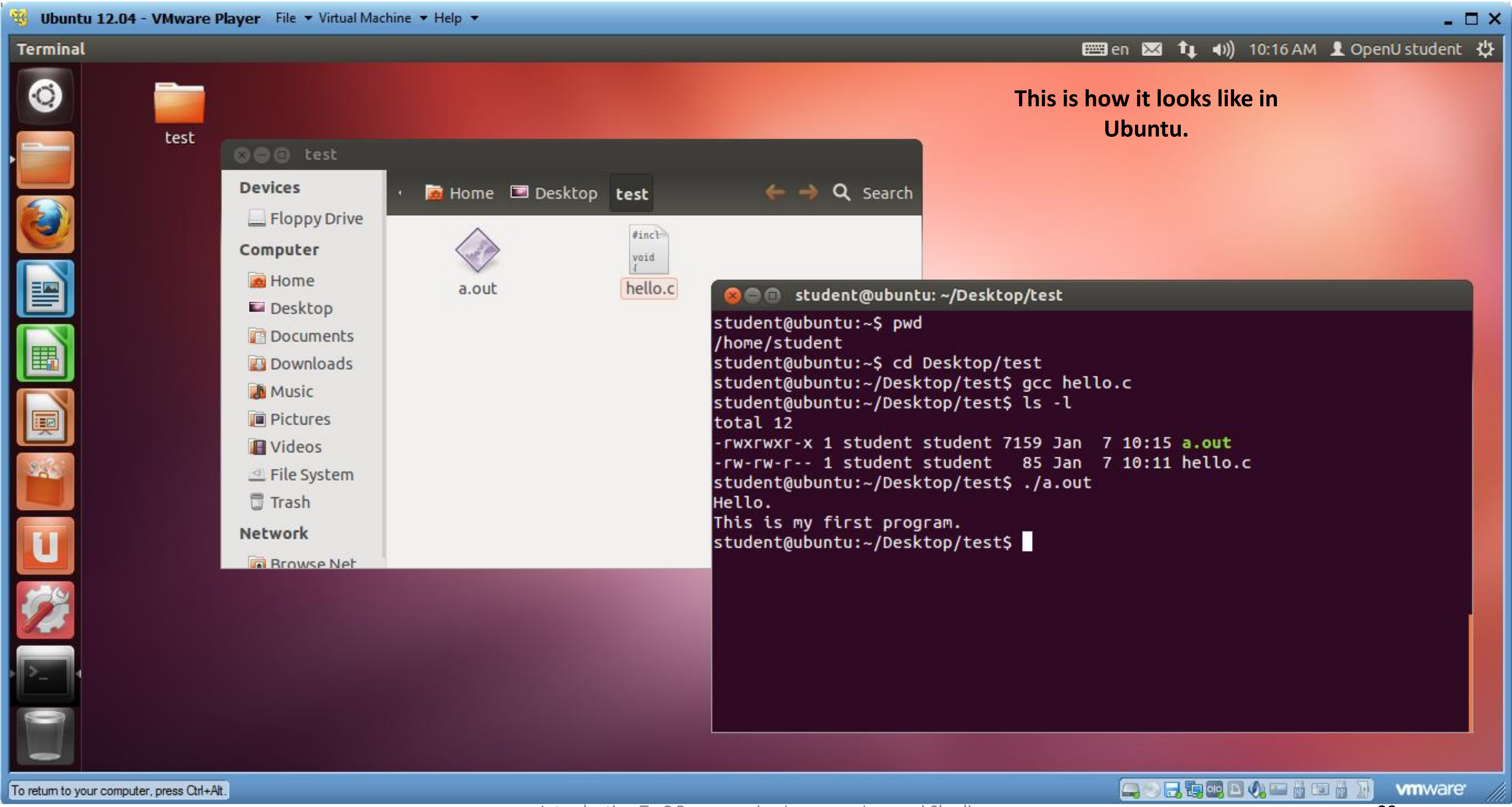
- STD input/out functions are not part of C.
- Should be imported from the Standard Library if we would like to use them.

Each C program must include the main function

```
#include <stdio.h>

void main()
{
   printf("Hello.\nThis is my first program in c.\n");
}
```

STD output: printf is used to print a message to the screen

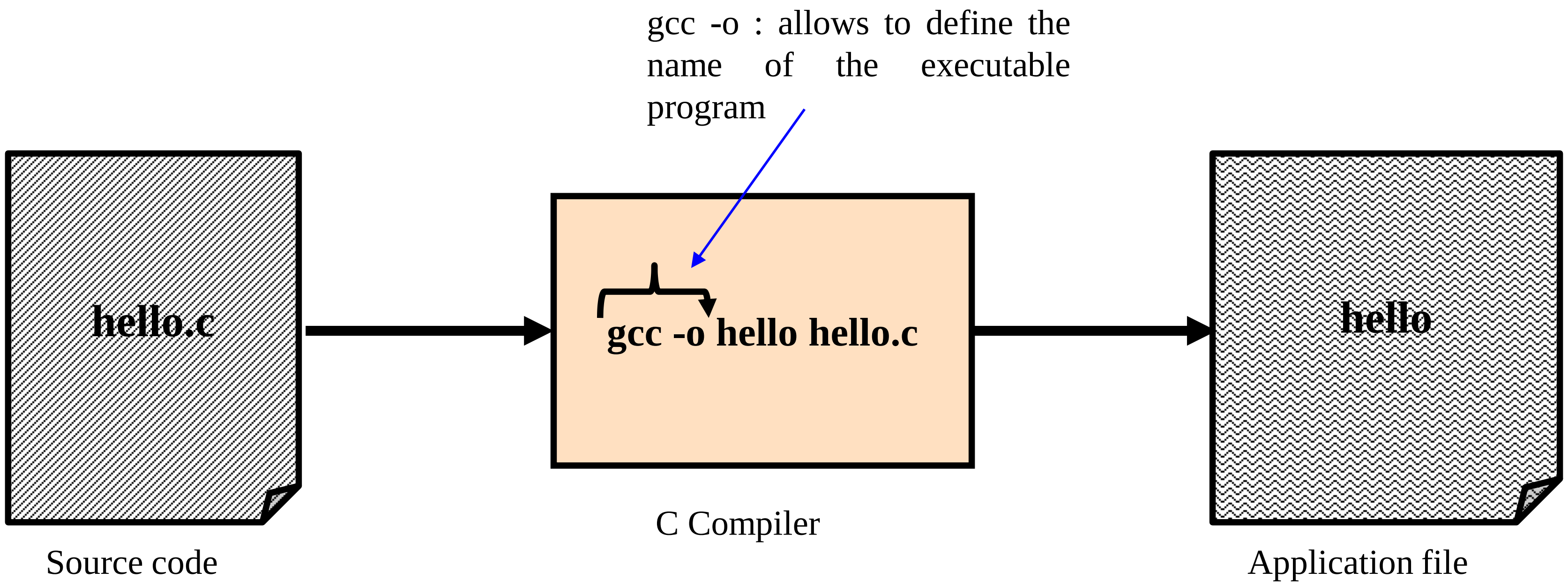
newline = ‘\n’



This is how it looks like in Ubuntu.

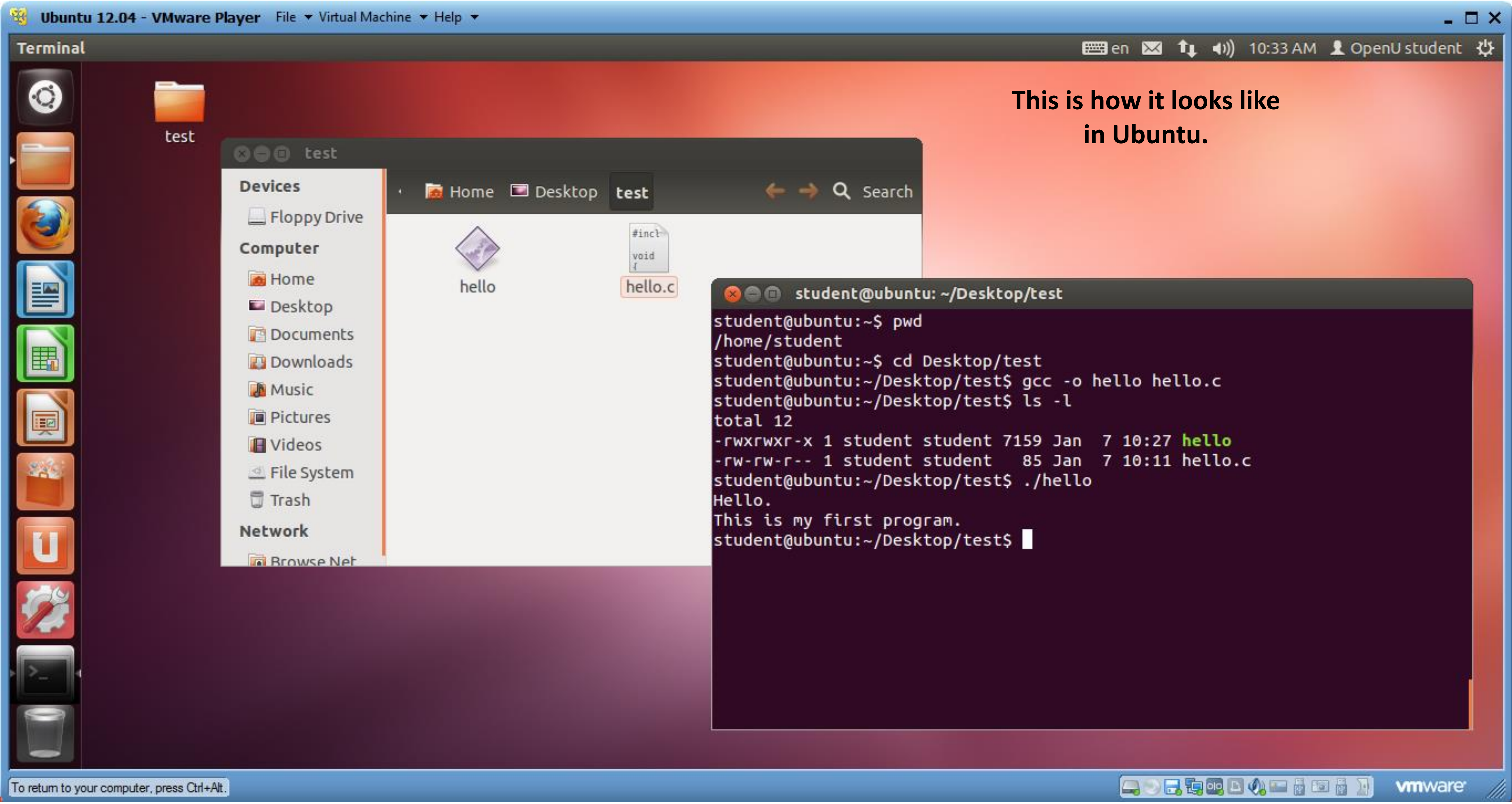
```
student@ubuntu:~$ pwd
/home/student
student@ubuntu:~$ cd Desktop/test
student@ubuntu:~/Desktop/test$ gcc hello.c
student@ubuntu:~/Desktop/test$ ls -l
total 12
-rwxrwxr-x 1 student student 7159 Jan  7 10:15 a.out
-rw-rw-r-- 1 student student   85 Jan  7 10:11 hello.c
student@ubuntu:~/Desktop/test$ ./a.out
Hello.
This is my first program.
student@ubuntu:~/Desktop/test$ 
```

Changing the default name of the executable file using gcc compiler



gcc -o : allows to define the name of the executable program

- hello.c קובץ טקסט אשר מכיל הוראות בשפת C.
- הפקודה gcc –o hello hello.c מאפשרת מתן שם חדש לקובץ ההרצה הנוצר.
- תוצאת ההידור נשמרת בקובץ בינרי אשר מקבל שם hello (לא הדיפולטיבי a.out)



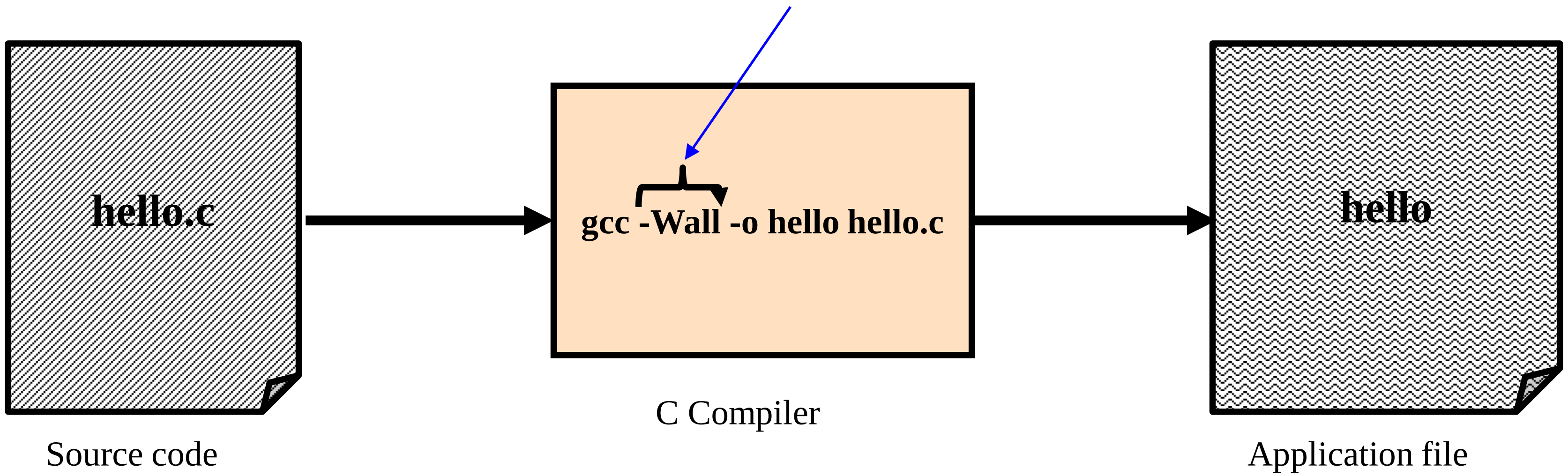
```
student@ubuntu:~$ pwd
/home/student
student@ubuntu:~$ cd Desktop/test
student@ubuntu:~/Desktop/test$ gcc -o hello hello.c
student@ubuntu:~/Desktop/test$ ls -l
total 12
-rwxrwxr-x 1 student student 7159 Jan  7 10:27 hello
-rw-rw-r-- 1 student student   85 Jan  7 10:11 hello.c
student@ubuntu:~/Desktop/test$ ./hello
Hello.
This is my first program.
student@ubuntu:~/Desktop/test$
```

This is how it looks like in Ubuntu.

More gcc Compiler flags

Warning message: a message where the compiler alerts that a goal error may occure.

gcc -Wall: is used to print warning messages



hello.c

gcc -Wall -o hello hello.c

hello

Source code

C Compiler

Application file

- gcc -Wall -o Hello.exe Hello.c
- ✓ -o: specifies the output executable filename.
- ✓ -Wall: prints "all" warning messages.

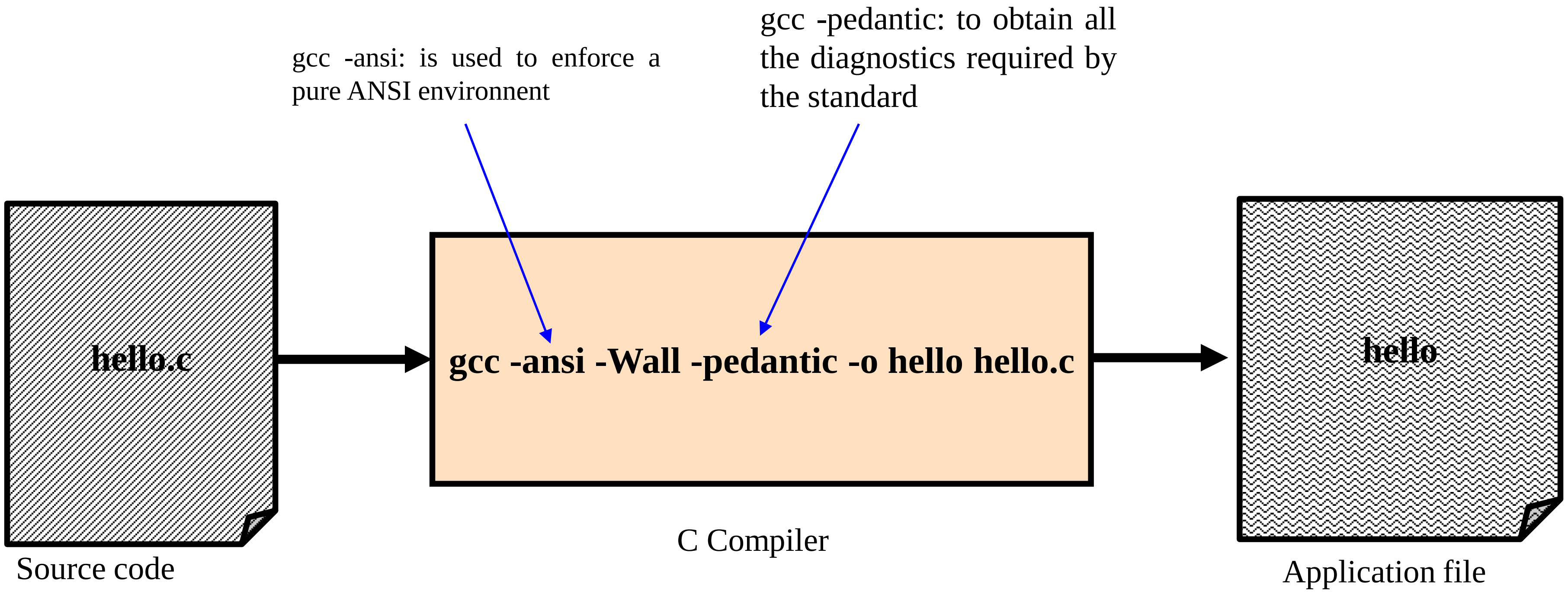
Compiler Messages

Compiler Warnings: indicates that something bad was done. However, the program will pass compilation. These warning should be fixed since they often lead to other problems that will not be so easy to find.

Compiler Errors: indicates something that **must** be fixed before the code can be compiled

Linker Errors: indicates that a code compiles fine, but a specific function or library is missing.

More important gcc
Compiler flags



gcc -ansi: is used to enforce a pure ANSI environnent

gcc -pedantic: to obtain all the diagnostics required by the standard

- gcc -ansi -Wall -pedantic -o hello hello.c

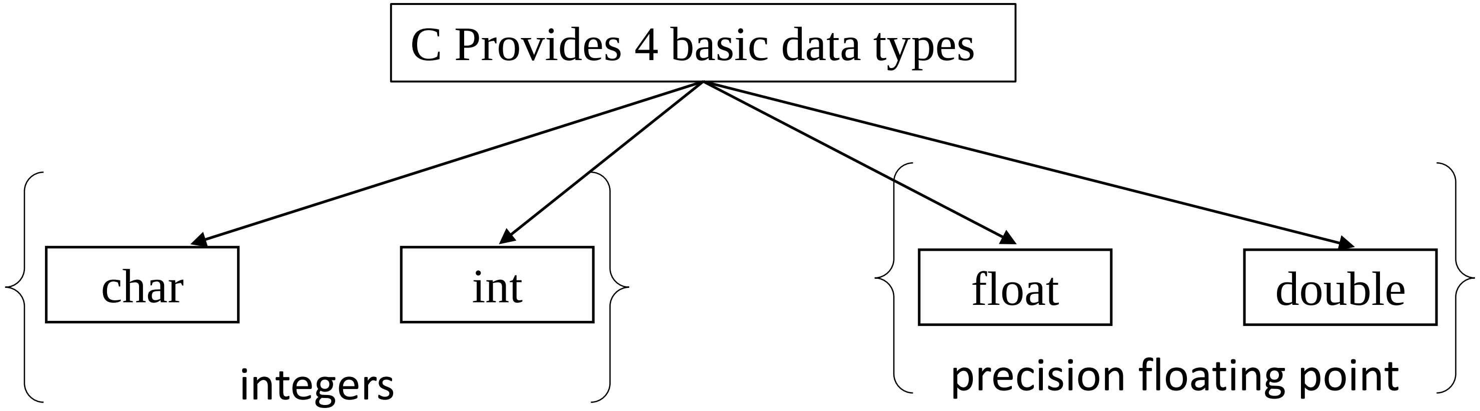
✓ -o: specifies the output executable filename.

✓ -Wall: prints "all" warning messages.

✓ -ansi: Enforces a pure ANSI environment

✓ -pedantic: to obtain all the diagnostics required by the standard, you should also specify ‘-pedantic’ (or ‘-pedantic-errors’ if you want them to be errors rather than warnings)

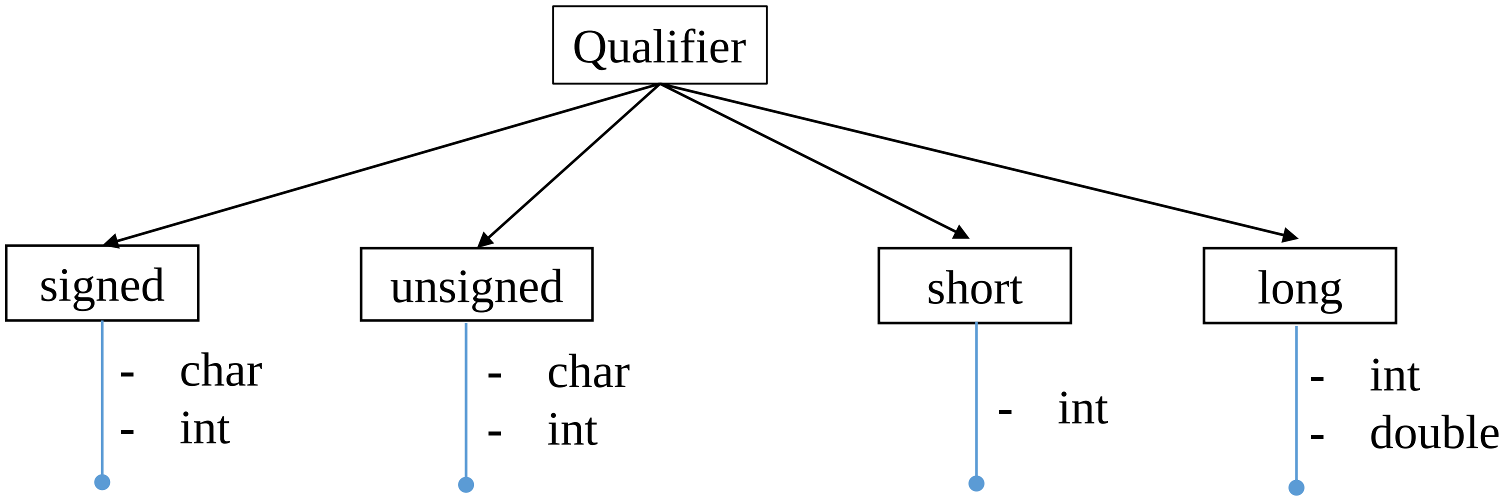
Data Types



- char: a single byte, holds one character.
- int: an integer, reflects the natural size of integers on the host machine
- float: a single-precision floating point
- double: a double-precision floating point

Data type modifiers in C

- keywords used to change the properties of the standard data types.
- Data type modifiers are classified into following types : long, short, unsigned and signed.
- Used to modify (increase/decrease) the amount of storage space allocated to a variable.



Qualifier

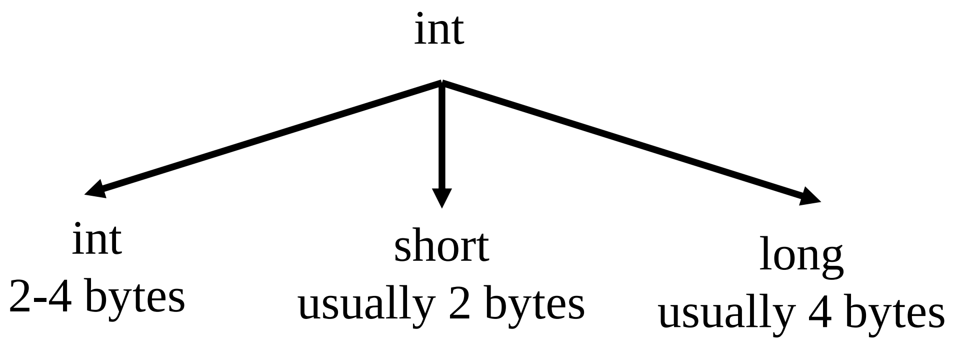
signed
- char
- int

unsigned
- char
- int

short
- int

long
- int
- double

Example

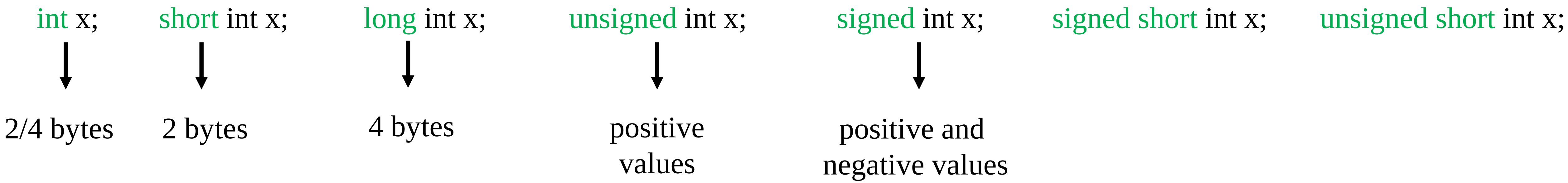


int

int
2-4 bytes

short
usually 2 bytes

long
usually 4 bytes



int x; → 2/4 bytes

short int x; → 2 bytes

long int x; → 4 bytes

unsigned int x; → positive values

signed int x; → positive and negative values

signed short int x;

unsigned short int x;

- Syntax:

  short int x; ←→ short x;
  long int x; ←→ long x;

- All integers (including char) can be defined as signed or unsigned.

Type	Size (bytes)
char	1
int	2/4
short int	2
long int	4
float	4
double	8
long double	10/12
void	-

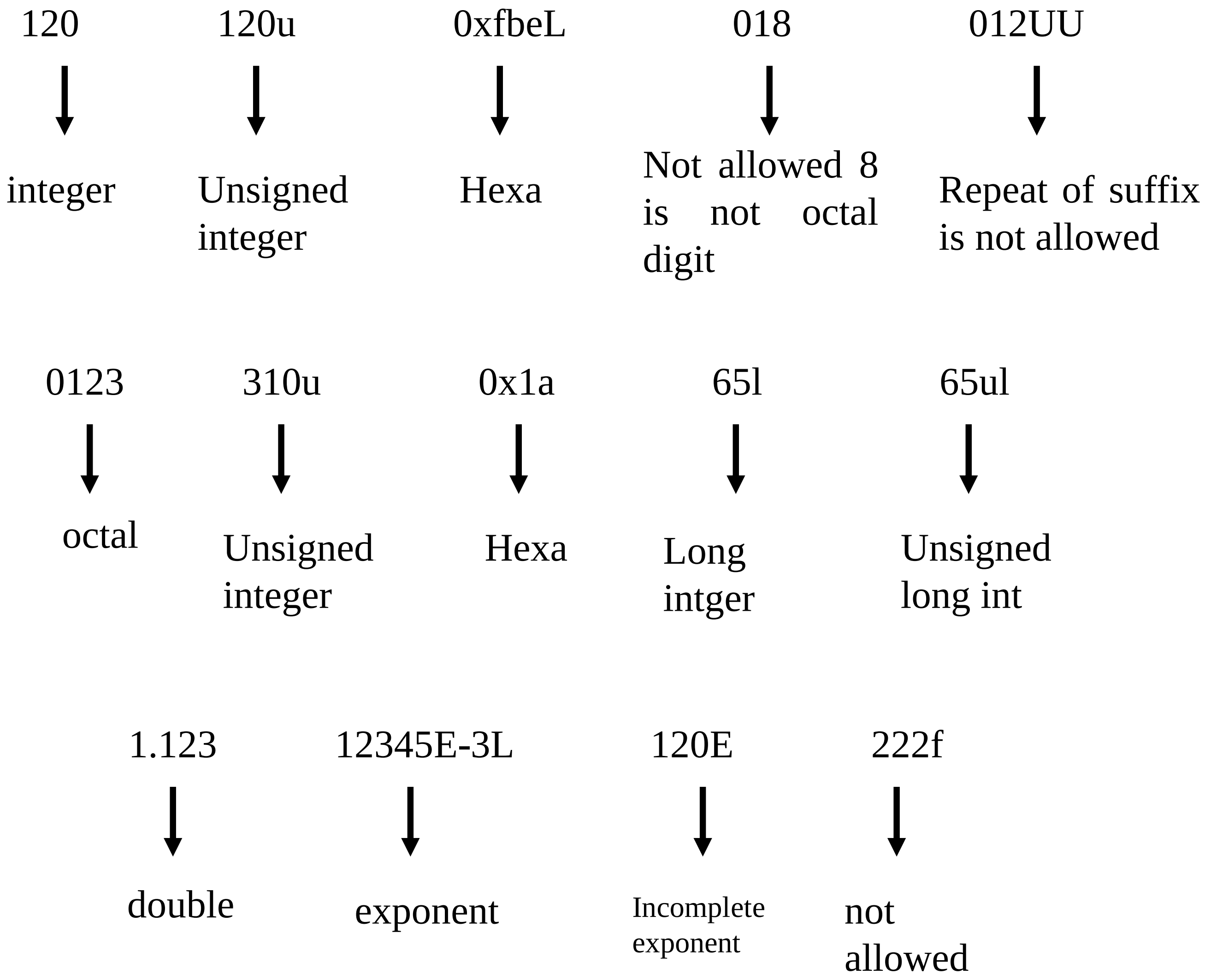
All integers (including char) can be defined as signed or unsigned e,g.

1) signed char → -128 – +127)
2) unsigned char → 0 – +255
3) signed short
4) unsigned short
5) unsigned int
6) unsigned long

* char = signed char
int = signed int
short = signed short
:

The standard headers <limits.h> and <float.h> contain symbolic constants for all of these sizes, along with other properties of the machine and compiler

Literals



120 → integer

120u → Unsigned integer

0xfbeL → Hexa

018 → Not allowed 8 is not octal digit

012UU → Repeat of suffix is not allowed

0123 → octal

310u → Unsigned integer

0x1a → Hexa

65l → Long intger

65ul → Unsigned long int

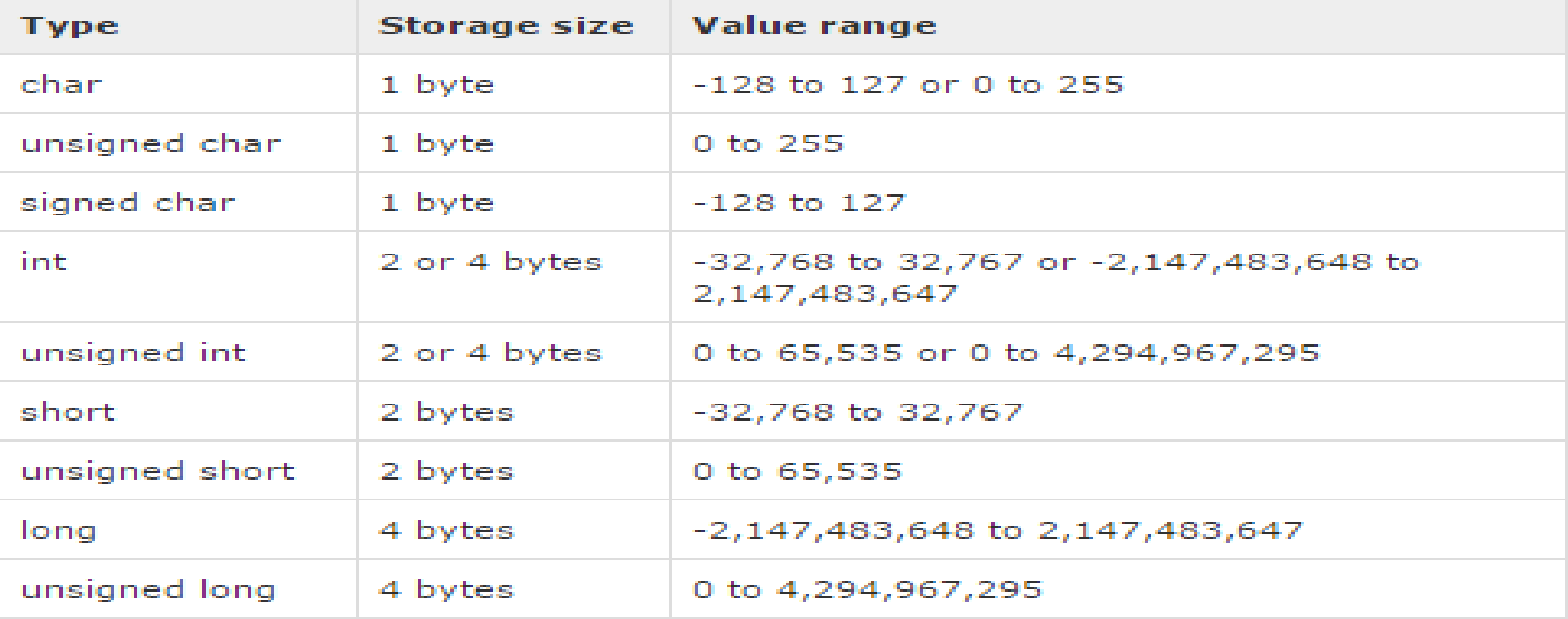
1.123 → double

12345E-3L → exponent

120E → Incomplete exponent

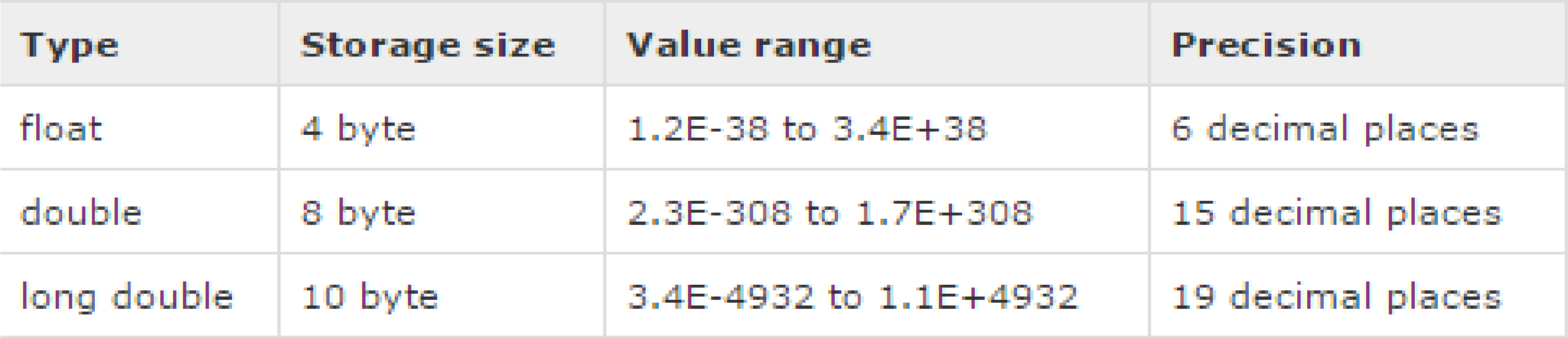
222f → not allowed

Integer Types Range



Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Floating-Point Types Range



Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

A program to check the storage size of a data type

```c
#include <stdio.h>

int main() {

   printf("Bytes in char %d\n", sizeof(char));
   printf("Bytes in int %d\n", sizeof(int));
   printf("Bytes in short int %d\n", sizeof(short int));
   printf("Bytes in long int %d\n", sizeof(long int));
   printf("Bytes in float %d\n", sizeof(float));
   printf("Bytes in double %d\n", sizeof(double));
   printf("Bytes in long double %d\n", sizeof(long double));

   return 0;
}
```

R **sizeof**: is used to get
E the exact size of a
M type or a variable on
E a particular
M platform. The result
B is the number of
E **bytes** of the specific
R type.

Output:

```
Bytes in char 1
Bytes in int 4
Bytes in short int 2
Bytes in long int 4
Bytes in float 4
Bytes in double 8
Bytes in long double 12
```

A program to Check the value range of C-Types

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>
int main()  {
   printf("The number of bits in a byte %d\n", CHAR_BIT);
   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 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 UNSIGNED INT = %u\n", UINT_MAX);
   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);
   printf("The maximum value of UNSIGNED LONG = %lu\n", ULONG_MAX);
   printf("The minimum value of FLOAT = %e\n", FLT_MIN);
   printf("The maximum value of FLOAT = %e\n", FLT_MAX);
   printf("The minimum value of DOUBLE = %e\n", DBL_MIN);
   printf("The maximum value of DOUBLE = %e\n", DBL_MAX);
   return 0;
}
```

Output:

```
The number of bits in a byte 8
The minimum value of CHAR = -128
The maximum value of CHAR = 127
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 UNSIGNED INT = 4294967295
The minimum value of LONG = -2147483648
The maximum value of LONG = 2147483647
The maximum value of UNSIGNED LONG = 4294967295
The minimum value of FLOAT = 1.175494e-38
The maximum value of FLOAT = 3.402823e+38
The minimum value of DOUBLE = 2.225074e-308
The maximum value of DOUBLE = 1.797693e+308
```

Input/Output in C
Using printf and scanf

STDO – printf
STDO default is the screen

```
int printf(char *format, arg1, arg2, ...);
```

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

Example1

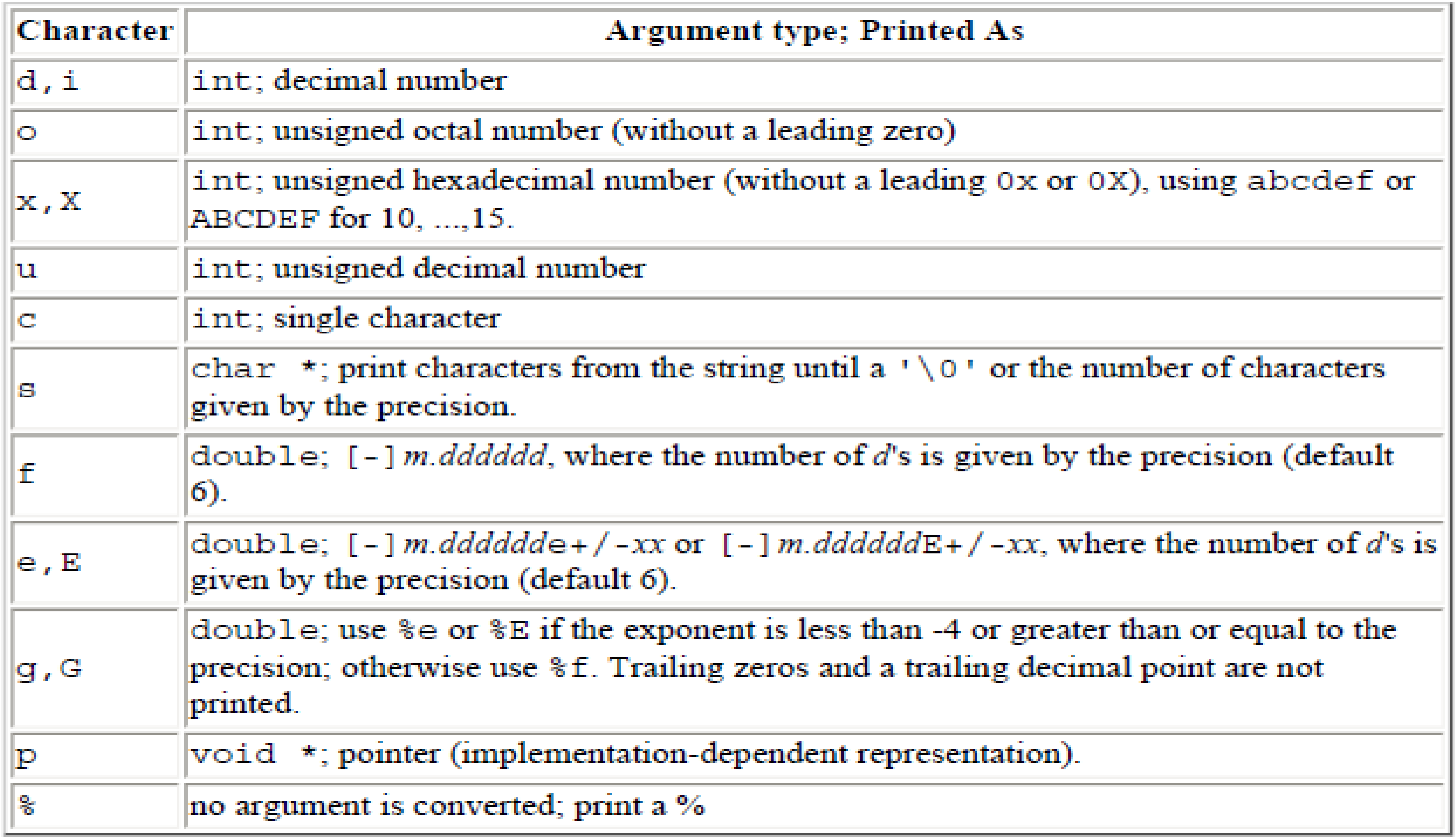
```
#include <stdio.h>
int main() {

  printf("\nX+Y=%d", 2+3);
  printf("\nX+Y=%d", (2+3));
  printf("\nX*Y=%d\n", 2*3);

  return 0;
}
```

Output:

```
X+Y=5
X+Y=5
X*Y=6
```

printf is used for printing to the standard output

Basic Printf Conversions



Character	Argument type; Printed As
d, i	int; decimal number
o	int; unsigned octal number (without a leading zero)
x, X	int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ...,15.
u	int; unsigned decimal number
c	int; single character
s	char *; print characters from the string until a '\0' or the number of characters given by the precision.
f	double; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
e, E	double; [-]*m.dddddd*e+/-*xx* or [-]*m.dddddd*E+/-*xx*, where the number of *d*'s is given by the precision (default 6).
g, G	double; use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed.
p	void *; pointer (implementation-dependent representation).
%	no argument is converted; print a %

The_C_Programming_Language_Kernighan_And_Ritchie_2nd

String format : %[flags][width][.precision][length]specifier

http://www.cplusplus.com/reference/cstdio/printf/

flags	description
-	Left-justify within the given field width; Right justification is the default (see *width* sub specifier).
+	add plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a **-** sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier).

.precision	Description
.number	For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is countered. If the period is specified without an explicit value for *precision*, 0 is assumed.
.*	*precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	*width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

Example

```c
#include <stdio.h>
int main() {
  printf("\nInteger->%d<-", 123);
  printf("\nInteger->%6d<-", 123);
  printf("\nInteger->%06d<-", 123);
  printf("\nInteger:->%6d<-", 123456789);
  printf("\nInteger:->%i<-", 123);
  printf("\nFloat:->%6.3f<-", 1.5);
  printf("\nFloat:->%6.3f<-", 1.456789);
  printf("\nFloat:->%6.3f<-", 1234.456789);
  printf("\nHexa:->%x<-", 255);
  printf("\nOctal:->%o<-", 255);
  printf("\nUnsigned value->%u<-", 255);
  printf("\nChar:->%c<-", 'a');
  printf("\n->%s<-", "Hello, world!");
  printf("\n->%15s<-", "Hello, world!");
  printf("\n->%.10s<-", "Hello, world!");
  printf("\n->%-10s<-", "Hello, world!");
  printf("\n->%-15s<-", "Hello, world!");
  printf("\n->%.15s<-", "Hello, world!");
  printf("\n->%15.10s<-", "Hello, world!");
  printf("\n->%-15.10s<-\n", "Hello, world!");
  return 0;
}
```

```
Output:
Integer->123<-
Integer->   123<-
Integer->000123<-
Integer:->123456789<-
Integer:->123<-
Float:-> 1.500<-
Float:-> 1.457<-
Float:->1234.457<-
Hexa:->ff<-
Octal:->377<-
Unsigned value->255<-
Char:->a<-
->Hello, world!<-
->  Hello, world!<-
->Hello, wor<-
->Hello, world!<-
->Hello, world!  <-
->Hello, world!<-
->     Hello, wor<-
->Hello, wor     <-
```

Say s is a string type then :

- Don’t use printf(s);
- Better use printf("%s", s);

```
int main()
{
   char* s = "Hello %d World.\n";
   printf(s);
   return 0;
}
```

✗

Output:

```
warning: format not a string literal and no format arguments.
Hello -1219067482 World.
```

```
int main()
{
   char* s = "Hello %d World.\n";
   printf("%s", s);
   return 0;
}
```

✓

Output:

```
Hello %d World.
```

Example

```
#include <stdio.h>
int main()  {
  printf("\n%u",-1);
  printf("\n%d",4294967295u);
  printf(“\n%d   %d   %x", 32767,0x7fff,32767); //0x7fff=32767
  printf("\n%d   %u   %x", 0x8000,0x8000,32768);
  printf("\n%ld   %lx", 0x7fffffff,2147483647);
  printf("\n%lu   %lx", 0xffffffff,4294967295);
  printf("\n%lu   %lx", 0xfffffffflu,4294967295lu);
  printf("\n%d  %d  %f  %f", (int)1.,1.,(float)1,1);
  return 0;
}
```

```
Output:
→4294967295
→-1
→32767   32767   7fff
→32768   32768   8000
→2147483647   7fffffff → warning: %ld’ expects type ‘long int’, but argument 2 has type ‘int’.
warning: ‘%lx’ expects type ‘long unsigned int’, but argument 3 has type ‘int’
→4294967295   ffffffff →warning: constant is unsigned only in ISO C90 . warning: ‘%lu’ expects
argument of type ‘long unsigned int’, but argument 2 has type ‘unsigned int’
→4294967295   ffffffff
→1 0  0.000000  0.000000 →warning:%d’ expects type ‘int’, but argument 3 has type ‘double’
                          warning:‘%f’ expects type ‘double’, but argument 5 has type ‘int’
```

STDI Scanf command

```
int scanf(char *format, ...)
```

scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments. It reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments

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

  int a, b, c;
  printf("Insert 2 numbers:");
  scanf("%d %d", &a , &b);
  c=a+b;
  printf("\n%d + %d = %d\n", a, b, c);

  return 0;
}
```

Scanf is used for getting input from the keyboard

Strange behavior of scanf!!!

```c
#include <stdio.h>
int main()  {
  char c;
  scanf("%c",&c);
  printf("Result:%c\n",c);

  scanf("%c",&c);
  printf("Result:%c\n",c);
  return 0;
}
```

Output:

```
student@ubuntu:~/Desktop/test$ ./a.out
ab
Result:a
Result:b
```

scanf reads from a **temporary buffer** found on memory!!!

How to avoid this?

Reading a String from STDIN
STDIN default is the Keyboard

version1

```c
int main()  {
  char name [100];
  printf("Please insert your name:");
  scanf("%s", name);
  printf("\nHello %s\n", name);
  return 0;
}
```

A String is an array of characters. This is dangerous if the input length exceed 100. Why?

version2

```c
int main()  {
  char name [100];
  printf("Please insert your name:");
  scanf("%99s", name);
  printf("\nHello %s\n", name);
  return 0;
}
```

It is safe now as maximum characters to read will not exceed 99. Why not adding 100 instead 99? It is because scanf will add ‘\0’ which indicates end of string

version3

```c
int main()  {
  char name [100];
  int i=0;
  printf("Please insert your name:");
  while(i<sizeof(name) && (scanf("%c",&name[i])) && name[i++]!='\n');
  name[i-1]='\0';
  printf("\nHello %s\n", name);
  return 0;
}
```

מה יודפס?

```
int main()  {
  char c;
  scanf("%c",&c);
  printf("The num is:%d\n",c);
  return 0;
}
```

Output:

```
2
The num is:50
```

```
int main()  {
  int c;
  scanf("%c",&c);
  printf("The num is:%d\n",c);
  return 0;
}
```

Output:

```
warning: ‘%c’ expects ‘char *’, but type is‘int *’
2
The num is:-1216598222
```

מה ההבדל

```
int main()  {
  int c=0;
  scanf("%c",&c);
  printf("The num is:%d\n",c);
  return 0;
}
```

Output:

```
warning: %c’ expects ‘char *’, but type is‘int *’
2
The num is:50
```

התוכנית קולטת רדיוס
ומחשבת את היקף המעגל $2\pi r$

Symbolic Constant

```
#include <stdio.h>

#define PI 3.141593

int main()   {

  float radios;
  printf("Insert a radios:");
  scanf("%f",&radios);
  printf("\nCircumference is:%f\n", 2*radios*PI);

  return 0;
}
```

Character Input and Output Using **getchar** and **putchar**

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

  char c;
  c=getchar();
  putchar(c);

  return 0;
}
```

getchar: reads the next input character from a text stream and returns its value.
putchar: prints a character each time it is called.

Character Counting
Using getchar and scanf

version1

```c
#include <stdio.h>
int main()  {
   char c;
   short s=0;

  while((c=getchar()) !='\n' )
  {
    putchar(c);
    putchar('\n');
    s++;
  }
  printf("%d\n",s);

  return 0;
}
```

Input: a bcd e
Output:

```
a

b
c
d

e
7
```

version2

```c
#include <stdio.h>
int main()  {
   char c;
   short s=0;

  while((scanf("%c",&c)) && c!='\n')
  {
    printf("%c\n",c);
    s++;
  }
  printf("%d\n",s);

  return 0;
}
```

Character/Word/Line Counting
Using getchar

```c
#include <stdio.h>
int main()   {
  int c;
  short nc,nw,nl;
  nc=nw=nl=0;
  /*read a file*/
  while((c=getchar()) != EOF )
  {
    nc++;
    if(c==' ' || c=='\n' || c=='\t')
       nw++;
    if(c=='\n')
       nl++;
  }
  printf("Characters:%d Words:%d Lines:%d\n",nc,nw,nl);
  return 0;
}
```

Input:

```
a b c
aa bb
```

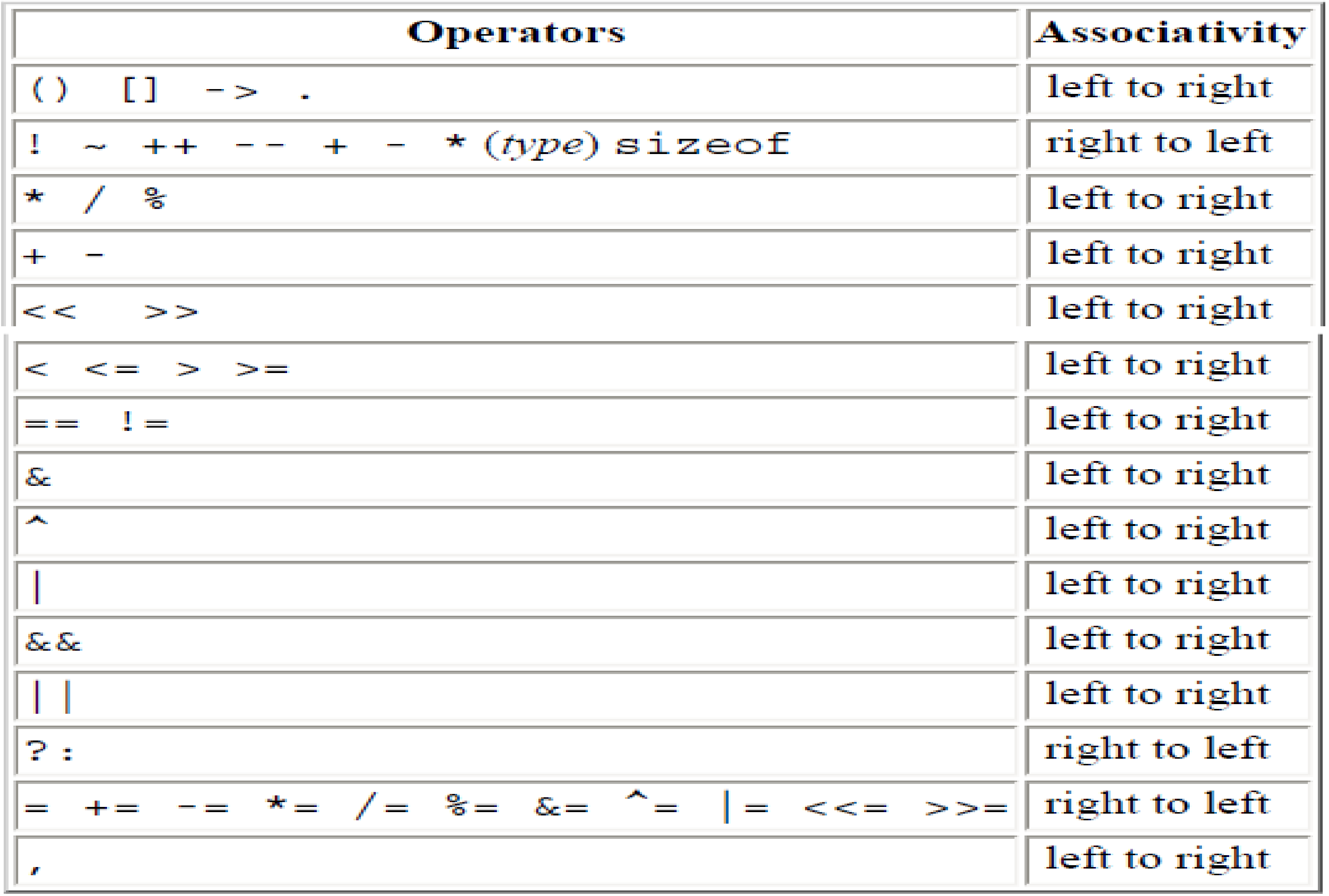
Output:

```
Characters:12 Words:5
Lines:2
```

Including ‘\n’

It is recommended to define EOF as int and not char why?

Precedence and Order of Evaluation



Operators	Associativity
( ) [ ] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Unary & +, -, and * have higher precedence than the binary forms.

Remeber: It is considered a bad programming practice to write code that depends on order of evaluation.

Note that the precedence of the bitwise operators &, ^, and | falls below == and !=. This implies that bit-testing expressions like
if ((x & MASK) == 0) ...
must be fully parenthesized to give proper results.

The_C_Programming_Language_Kernighan_And_Ritchie_2nd

```
int a = 2;
int c = a++ + a++;
printf("%d  %d\n" , a, c);
```

output:

```
4 4
```

```
#include <stdio.h>
void print(int a , int b)
{
  printf("%d %d\n", a, b);
}
int main() {
int a;
 a=2; print(a,a++);
 a=2; print(a++,a);
 a=2; print(a++,a++);
 a=2; print(a,++a);
 a=2; print(++a,a);
 a=2; print(++a,++a);
 a=2; print(a++,++a);
 a=2; print(++a,a++);
 return 1;
}
```

output:

```
3 2
2 3
3 2
3 3
3 3
4 4
3 4
4 2
```

- **C does not specify the order in which the operands of an operator are evaluated.**

✓exceptions are (&&, ||, ?:, and `,'.)

✓E.g. :

```
x = f() + g();
```

f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence.

- Similarly, the order in which function arguments are evaluated is not specified, so the statement:

```
printf("%d %d\n", ++n, power(2, n)); /* WRONG */
```

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution, of course, is to write :

```
++n;
printf("%d %d\n", n, power(2, n));
```

- Function calls, nested assignment statements, and increment and decrement operators cause “side effects” - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement

```
a[i] = i++;
```

The question is whether the subscript is the old value of i or the new. Compilers can interpret this in different ways, and generate different answers depending on their interpretation

The_C_Programming_Language_Kernighan_And_Ritchie_2nd

Increment And Decrement
(++ , --)

Example1

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

  int count=1;
  count=count+1;
  count+=1;
  count--;
  count++;
  ++count;
  --count;
  printf("%d", count);
  printf("\n%d",count++);
  printf("\n%d",++count);
  printf("\n%d",(2*count++)+5);
  printf("\n%d\n", count);

  return 1;
}
```

Output:

```
3
3
5
15
6
```

Example2

```
int main()  {
  int a,b,c;
  a=1;b=2; c=a+++b;
  printf("%d %d %d", a, b, c) ;//a=2 b=2 c=3
  a=1;b=2; c=a+++++b;//error but c=a++ + ++b is ok
  a=1;b=2; c=a+++b++;
  printf("%d %d %d", a, b, c) ;//a=2 b=3 c=3
  a=1;b=2; c=a++ + b++;
  printf("%d %d %d", a, b, c) ;//a=2 b=3 c=3
  a=1;b=2; c=++a + ++b;
  printf("\n%d %d %d", a, b, c); //a=2 b=3 c=5
  a=1;b=2; c=a++ + ++b;
  printf("\n%d %d %d", a, b, c); //a=2 b=3 c=4
  a=1;b=2; c=++a+b++;
  printf("\n%d %d %d", a, b, c); //a=2 b=3 c=4
  // Error-->a=1;b=2;c=++a++;c=++(a++);c=(++a)++;
  a=1;b=2; c = (a++ + b++) * a; //warning: operation on ‘a’ may be undefined
  printf("\n%d %d %d", a, b, c); //a=2 b=3 c=3
  a=1;b=2; c = a++ + b++ * a; //warning: operation on ‘a’ may be undefined
  printf("\n%d %d %d", a, b, c); //a=2 b=3 c=3
  a=1;b=2; c = a++ + b++ + a++ + b++; //warning: operation on a and b may be undefined
  printf("\n%d %d %d", a, b, c); //a=3 b=4 c=6
  return 0;
}
```

Integer Promotion

- Integer promotion is the process by which values of integer type, smaller than int or unsigned int, are converted either to int or unsigned int.
- Some data types like *char* , *short int* take less number of bytes than *int*, these data types are automatically promoted to *int* or *unsigned int* when an operation is performed on them. This is called integer promotion. For example no arithmetic calculation happens on smaller types like *char*, *short* and *enum*. They are first converted to *int* or *unsigned int*, and then arithmetic is done on them. If an *int* can represent all values of the original type, the value is converted to an *int* . Otherwise, it is converted to an *unsigned int*.

Example 1

```
#include <stdio.h>
int main()   {
    int  x = 1;
    char y = 'a'; // 'a' = 97
    int z;

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

Output:
98

Integer promotion was done by the compiler so converting ‘a’ to its ASCII value before performing the actual “+” operation.

Example 2

```c
#include <stdio.h>
int main() {
    char a = -1;
    unsigned char b = 0xff;

    printf("a=%c b=%c\na=%x b=%x\n", a, b, a, b);

    (a == b) ? printf("true\n") : printf("false\n");

   return 0;
}
```

Output:

```
a=� b=�
a=ffffffff b=ff
false
```

‘a’ and ‘b’ have same char binary representation. But once comparing them ‘a’ and ‘b’ have deferent vlues. Thus because they are first converted to int. ‘a’ is a signed *char*, when it is converted to *int*, its value becomes -1 (int signed value of -1). ‘b’ is *unsigned char*, when it is converted to *int*, its value becomes 255. The values -1 and 255 have different representations as *int*, so the output is false.

Example 3

```
#include <stdio.h>
int main()
{
    char a = 30, b = 40, c = 10;
    char d = (a * b) / c;

    printf ("%d ", d);

    return 0;
}
```

Output:
120

At first look, the expression (a*b)/c seems to cause arithmetic overflow because signed characters can have values only from -128 to 127 (in most of the C compilers), and the value of subexpression ‘(a*b)’ is 1200 which is greater than 128. But integer promotion happens here in arithmetic done on char types and we get the appropriate result without any overflow.

C type Casting

- Type casting is a way to convert a variable from one data type to another e.g. converting an int value into a char value. A way to achieve this we can chose to explicitly convert a type to another by using cast operator.

- Type conversions can be achieved by:

✓ implicitly - performed by the compiler automatically.
✓ explicitly – by using the **cast operator**.

- It is considered good programming practice to use the cast operator whenever a type conversion is necessary.
- It is best practice to convert lower data type to higher data type to avoid data loss.
- Data will be truncated when higher data type is converted to lower.

```
#include <stdio.h>
int main() {
   int a = 1;
   double c;

   c=a; // → 1.000000
   return 0;
}
```

implicit conversion: the value of a has been promoted from int to double.

```
#include <stdio.h>
int main() {
   int a = 1, b = 2;
   double c;

   c=a/b; // → 0.000000
   c = (double)a/b; //→ 0.500000
  return 0;
}
```

explicit conversion

Usual Arithmetic Conversion

- The usual arithmetic conversions are implicitly performed to cast a value to a common type when the **operands** have deferent types. First *integer promotion is performed* and if the operands still have different types, then they are converted to a higher type.

- C, for any expression except assignments, is implicitly converting a type from a lower size type to a higher size type as shown in bellow diagram.

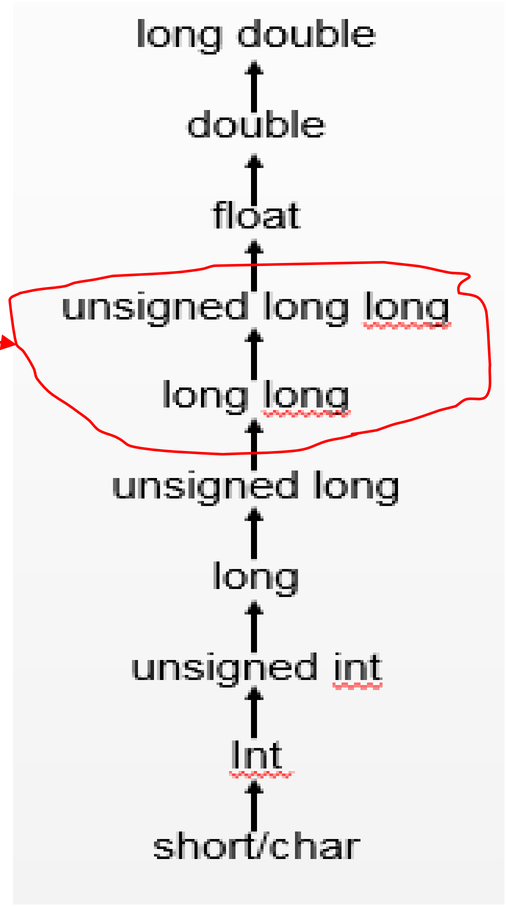
- The usual arithmetic conversions are not performed for:

✓ The assignment operators.

✓ The logical operators && and || …

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

   int  x = 1;
   char y = 'a'; → 'a' = 97
   float z;

   z = x + y; →98.000000

   return 0;
}
```

Here first y promoted to integer, but as the final value is float, usual arithmetic conversion applies and the compiler converts x and y into float.



Not supported by ansi-c

Usual Arithmetic Conversion
Common rules

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

- First, if either operand is long double, the other is converted to long double.
- Otherwise, if either operand is double, the other is converted to double.
- Otherwise, if either operand is float, the other is converted to float.
- Otherwise, the integral promotions are performed on both operands; then, if either operand is unsigned long int, the other is converted to unsigned long int.
- Otherwise, if one operand is long int and the other is unsigned int, the effect depends on whether a long int can represent all values of an unsigned int; if so, the unsigned int operand is converted to long int; if not, both are converted to unsigned long int.
- Otherwise, if one operand is long int, the other is converted to long int.
- Otherwise, if either operand is unsigned int, the other is converted to unsigned int.
- Otherwise, both operands have type int.

The_C_Programming_Language_Kernighan_And_Ritchie_2nd

Example 1

```c
#include <stdio.h>
int main() {
      char a=97,b='a';
      int c=97,d='a';
      float e=12.0F, f=12.34567F,g=12;

      printf("%d %d %d %d\n",(int)a,(int)b,c,d);//→97 97 97 97
      printf("%c %c %c %c\n",a,b,(char)c,(char)d);/→a a a a
      printf("%d\n",e); //Warnning →0
      printf("%d\n",g); //Warnning →0
      printf("%d\n",f); //Warnning →-1610612736
      printf("%d\n",(int)f);//12
      printf("%d\n",(char)255);//-1
      printf("%d\n",(char)256);//0
      printf("%d\n",(char)128);//-128
      printf("%f %f\n",(float)c ,(float)b);//97.000000 97.000000
  return 0;
}
```

Example 2

```c
#include <stdio.h>
int main() {
  char c='a';
  int a = c;
  short b = c;
  float f = c;
  double d = c;
  printf("%c %d %d %f %f\n",c,a,b,f,d); //output: a 97 97 97.000000 97.000000

  d=97;  c=d;  a=d;  b=d;  f=d;
  printf("%c %d %d %f %f\n",c,a,b,f,d); //output: a 97 97 97.000000 97.000000

  d=-1;  c=d;  a=d;  b=d;  f=d;
  printf("%c %d %d %f %f\n",c,a,b,f,d); //output: � -1 -1 -1.000000 -1.000000
  {
     int x=0xffff;
     short y=x;
     unsigned short z= x;
     printf("%d %d %d\n",x,y,z); //output: 65535 -1 65535

     printf("%d %d %d\n",0xffffffff,(short)0xffffffff,(char)0xffffffff); //output: -1 -1 -1
     printf("%d %u %u\n",0xffffffff,(short)0xffffffff,(char)0xffffffff);//output: -1 4294967295 4294967295
  }
  return 0;
}
```

Example 3

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

    int a=3, b=2, c;
    float d=3, e=2, f;

    c=a/b;  printf("%d %f\n", c,c); //Warrning: 1 0.000000
    c=d/e;  printf("%d %f\n", c,c); //Warrning: 1 0.000000
    c=a/b;  printf("%d %f\n", c,(float)c); //1 1.000000
    c=d/e;  printf("%d %f\n", c,(float)c); //1 1.000000
    f=a/b;  printf("%f\n",f);  //1.000000
    f=d/e;  printf("%f\n",f);  //1.500000
    f=a/e;  printf("%f\n",f);  //1.500000
    f=(float)a/b;  printf("%f\n",f);  //1.500000
    f=a/(float)b;  printf("%f\n",f);  //1.500000
    f=(int)d/e;  printf("%f\n",f);  //1.500000
  return 0;
}
```

Example 4

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

      printf("%c\n",0);
      printf("%c\n",'0');
      printf("%c\n",'\0');
      printf("%c\n",(char)0);
      printf("%c\n",(char)'0');
      printf("%d\n",(int)'0');

  return 0;
}
```

Output:

```

0


0
48
```

Example 5

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

      int x=0;
      int y='0';
      printf("%d\n", x);
      printf("%d\n", y);
      printf("%c\n", (char)x);
      printf("%c\n", (char)y);
      printf("%c\n", x);
      printf("%c\n", y);

  return 0;
}
```

Output:

```
0
48

0

0
```

Example 6

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

       int a=5 , b=2, x;
       double y;

       x=a/b;
       printf("%d\n", x);

       y=a/b;
       printf("%f\n", y);

       y=(double)a/b;
       printf("%f\n", y);

       x=a%b;
       printf("%d\n", x);

  return 0;
}
```

Output:

```
2
2.000000
2.500000
1
```

Example 7

```
#include <stdio.h>
int main() {
  int i;
  char c;
  float f;
  i=128;c=i;printf("%d\n",c);//-128
  i=255;c=i;printf("%d\n",c);//-1
  i=256;c=i;printf("%d\n",c);//0
  i=257;c=i;printf("%d\n",c);//1

  f=-128;c=f;printf("%d\n",c);//-128
  f=255;c=f;printf("%d\n",c);//-1
  f=256;c=f;printf("%d\n",c);//0
  f=257;c=f;printf("%d\n",c);//1

  f=-128.5678;c=f;printf("%d\n",c);//-128
  f=255.5678;c=f;printf("%d\n",c);//-1
  f=256.5678;c=f;printf("%d\n",c);//0
  f=257.5578;c=f;printf("%d\n",c);//1
  f=1.5678;c=f;printf("%d\n",c);//1
  f=0.999;c=f;printf("%d\n",c);//0
 return 0;
}
```

Example 8

```c
#include <stdio.h>
int main()   {
  int x = -1;
  unsigned int y =-1;
  unsigned int z;
  z=x+y;
  printf("%d %u %X\n",z,z,z);
  return 0;
}
```

Output:

```
-2 4294967294  FFFFFFE
```

```c
#include <stdio.h>
int main()   {
  char x = -1;
  unsigned char y =-1;
  unsigned int z;
  z=x+y;
  printf("%d %u %X\n",z,z,z);
  return 0;
}
```

Output:

```
254  254  FE
```

```c
#include <stdio.h>
int main()   {
  char x = -1;
  char y =-1;
  unsigned int z;
  z=x+y;
  printf("%d %u %X\n",z,z,z);
  return 0;
}
```

Output:

```
-2 4294967294  FFFFFFE
```

Example 9

```c
#include <stdio.h>
int main()  {
  int x = -2;
  unsigned int y =1;
  unsigned int z;
  z=x+y;
  printf("%d %u %x\n",z,z,z);
  return 0;
}
```

Output:

```
-1 4294967295 ffffffff
```

```c
#include <stdio.h>
int main()  {
  int x = -2;
  unsigned int y =1;
  long int z;
  z=x+y;
  printf("%ld %lu %lx\n",z,z,z);
  return 0;
}
```

Output:

```
-1 4294967295 ffffffff
```

Mapping grades to British scale

```c
#include <stdio.h>
int main() {
        double grade;
        printf("Insert grade: ");
        scanf("%lf",&grade);
        if (grade == 100)
            printf ("Excellent\n");
        else if (grade >= 90 && grade < 100)
            printf ("Very good\n");
        else if (grade >= 80 && grade < 90)
            printf ("Good\n");
        else if (grade >= 60 && grade < 80)
            printf ("Passed!!!\n");
        else if (grade >=0 && grade < 60)
            printf ("Failed\n");
        else  printf ("Wrong grade\n");

  return 0;
}
```

Excellent	grade=100
Very good	90<=grade<100
Good	80<=grade<90
Passed	60<=grade<80
Failed	otherwise

הערך המודפס

הערך שנקלט

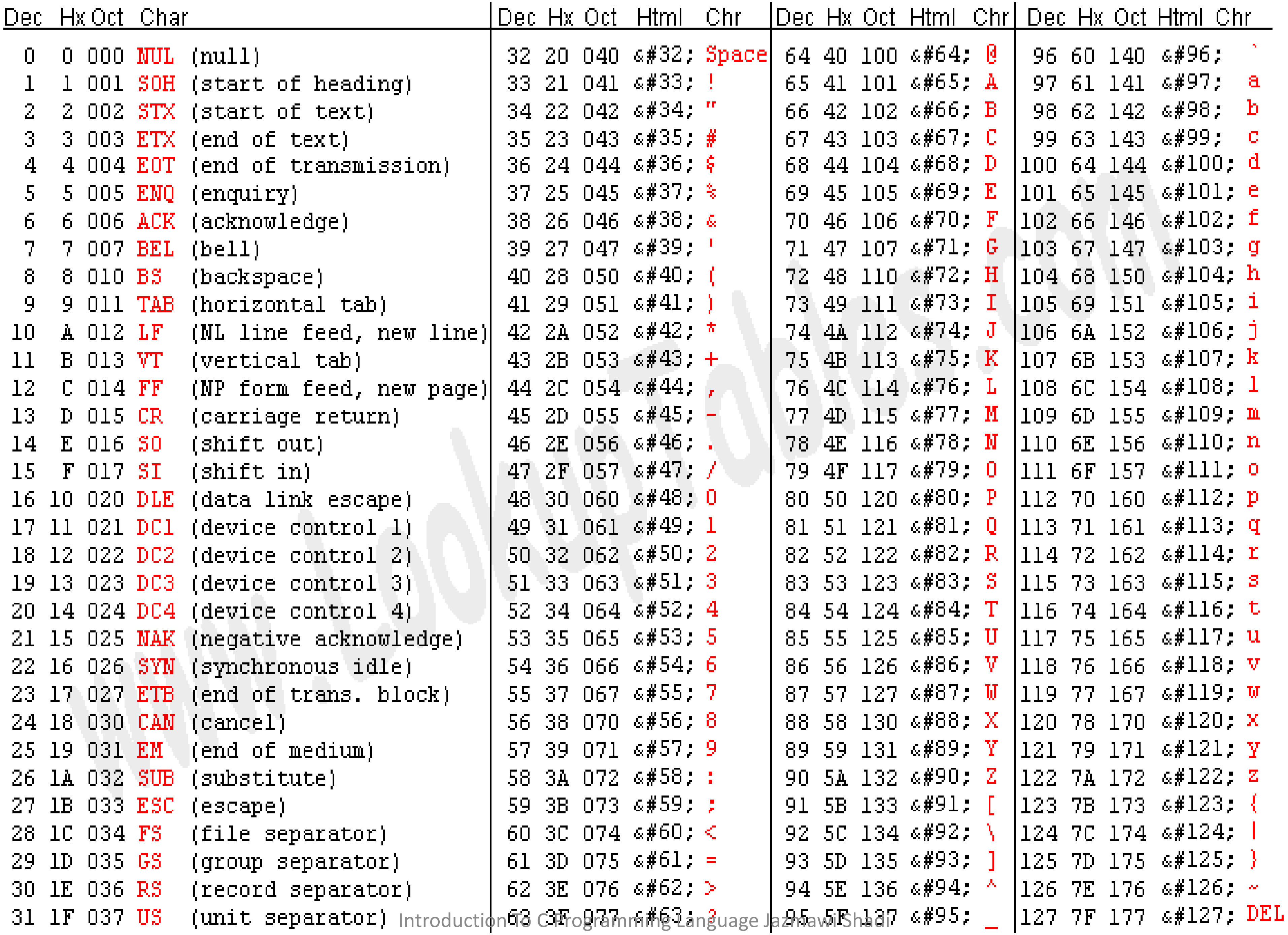
version 1
Using control flow

```
if (<expression>) {
  <statement>
}
else if(<expression>) {
  <statement>
}
else {
  <statement>
}
```

```
int main() {
      double grade;
      printf("Insert grade:");
      scanf("%lf",&grade);
      if (grade > 100 || grade < 0)
                printf ("Wrong grade\n");
      else {
        switch( (int)grade/10 ) {
          case 0: case 1: case 2: case 3: case 4: case 5:
                   printf ("Failed\n");
                   break;
          case 6: case 7:
                   printf("Passed!!!\n");
                   break;
          case 8:  printf("Good\n");
                   break;
          case 9:  printf("Very good\n");
                   break;
          case 10: printf("Excellent\n");
                   break;
        }
      }
  return 0;
}
```

version 2
Using switch

```
switch (<expression>) {
 case <const-expression-1>:
        <statement>
        break;
 case <const-expression-2>:
        <statement>
        break;
 case <const-expression-3>:
        <statement>
        break;
 case <const-expression-4>:
        <statement>
        break;
 default: // optional
         <statement>
}
```

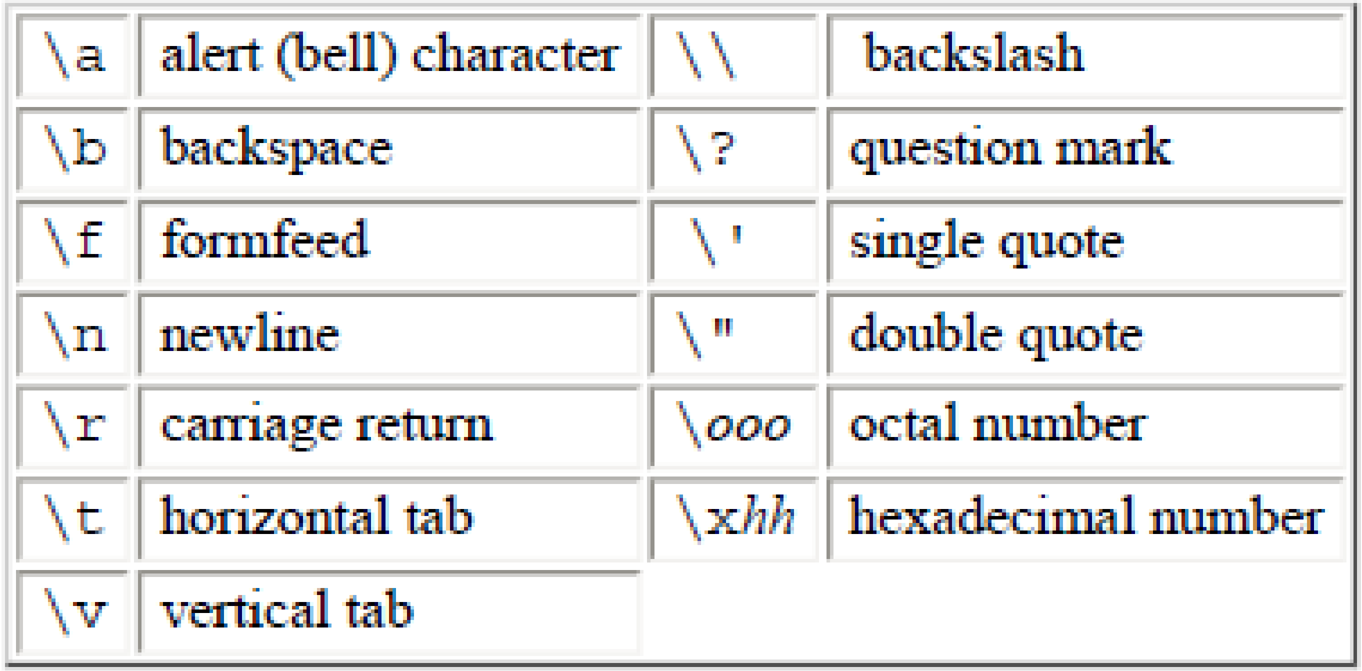


Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Ascii Table

Source: www.LookupTables.com

The complete set of escape sequences is



\a	alert (bell) character	\\	backslash
\b	backspace	\?	question mark
\f	formfeed	\'	single quote
\n	newline	\"	double quote
\r	carriage return	\ooo	octal number
\t	horizontal tab	\xhh	hexadecimal number
\v	vertical tab		

The character constant '\0' represents the character with value zero, the null character. '\0‘ is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0.

The_C_Programming_Language_Kernighan_And_Ritchie_2nd

END