## printf

#include<stdio.h>

void main**()**

**{**

char ch **=** 'A'**;**

char str**[**20**]** **=** "mayank gupta"**;**

float flt **=** 10.234**;**

int no **=** 150**;**

double dbl **=** 20.123456**;**

printf**(**"Character is %c \n"**,** ch**);**

printf**(**"String is %s \n"**,** str**);**

printf**(**"Float value is %f \n"**,** flt**);**

printf**(**"Integer value is %d \n"**,** no**);**

printf**(**"Double value is %lf \n"**,** dbl**);**

printf**(**"Octal value is %o \n"**,** no**);**

printf**(**"Hex value is %x \n"**,** no**);**

**}**

Character is A  
String is mayank gupta  
Float value is 10.234000  
Integer value is 150  
Double value is 20.123456  
Octal value is 226  
Hex value is 96

Also –

#include <stdio.h>

main**()**

**{**

char s**[**10**]** **=** "mayank g"**;**

printf**(**s**);**

**}**

#include <stdio.h>

int main**()**

**{**

printf**(**"Hello World!"**); // “ is needed rather than ‘ because it is not char but a string.**

**return** 0**;**

**}**

**Note that printf returns integer value corresponding to characters printed.**

## Scanf

# include <stdio.h>

void main**()**

**{**

char ch**;**

char str**[**100**];**

printf**(**"Enter any character \n"**);**

scanf**(**"%c"**,** **&**ch**);**

printf**(**"Entered character is %c \n"**,** ch**);**

printf**(**"Enter any string (upto 100 ch)\n"**);**

scanf**(**"%s"**,** **&**str**);**

printf**(**"Entered string is %s \n"**,** str**);**

**}**

Enter any character

q

Entered character is q

Enter any string (upto 100 ch)

Mayank

Entered string is Mayank

Another program –

#include <stdio.h>

main**() /\*rudimentary calculator from KR page 158\*/**

**{**

double sum**,** v**;**

sum **=** 0**;**

**while** **(**scanf**(**"%lf"**,** **&**v**)** **==** 1**)**

printf**(**"\t%.2f\n"**,** sum **+=** v**);**

**return** 0**;**

**}**

**o/p –**

c:\C>test

2

2.00

3

5.00

4

9.00

Note in above program how scanf has been compared to 1. Basically scanf function returns integer value corresponding to characters successfully scanned.

## getchar()

main**()**

**{**

long nc**;**

nc **=** 0**;**

**while(**getchar**()** **!=** EOF**)**

nc**=**nc**+**1**;**

printf**(**"%ld\n"**,**nc**);**

**}** //keystroke for EOF is CTRL+Z

Alternate way –

#include <stdio.h>

main**()**

**{**

double nc**;**

**for** **(**nc **=**0**;** getchar**()** **!=** EOF**;** **++**nc**)**

**;** // notice that we used ; without any statement.

printf**(**"%.1f\n"**,** nc**);**

}

// .1f means 1 digit after decimal point. To suppress decimal point, use .0f

## putchar():

Notes from K&R –

A character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a character constant, although it is just another way to write a small integer. So, for example, 'A' is a character constant; in the ASCII character set its value is 65, the internal representation of the character A. Of course, 'A' is to be preferred over 65: its meaning is obvious, and it is independent of a particular character set.

The escape sequences used in string constants are also legal in character constants, so '\n' stands for the value of the newline character, which is 10 in ASCII. You should note carefully that '\n' is a single character, and in expressions is just an integer; on the other hand, “\n” is a string constant that happens to contain only one character.

#include <stdio.h>

int main**(){**

char i**;**

**for** **(**i**=**0**;** i **<**10**;** **++**i**){**

printf**(**"%c\n"**,**i**);**

//putchar(i) followed by putchar('\n') can be used above

**};**

printf**(**"for loop ends here\n"**);**

putchar**(**'A'**+**10**);** //o/p is K as ‘A’ has ascii value of 65 and 65+10=75 which is ascii value of ‘K’

putchar**(**'\n'**);**

putchar**(**'A'**+**'1'**);**// o/p=’r’ as ‘A’ has ascii value of 65. 65+49=114 which is ascii value of ‘r’

putchar**(**'\n'**);**

putchar**(**'A'**+**'\n'**);**// o=K as ‘A’ has asci value of 65 and 65+10(‘\n’)=75 which is ascii value of K

**return** 0**;**

**}**

//"" can't be used in putchar('\n').

For i > 128, this program kind of enters into infinite loop. ???

Note – whether char is by default signed or unsigned is implementation dependent. In case of gcc compiler, char is unsigned by default. Although, printable characters are always positive.

## ****gets() and puts()****

#include <stdio.h>

main**()**

**{**

char name**[**50**];**

printf**(**"Enter your name\n"**);**

gets**(**name**);**

printf**(**"Your name is:"**);**

puts**(**name**);**

## }

## ****C – data types:****

There are four data types in C language. They are,

|  |  |
| --- | --- |
| **Types** | **Data Types** |
| Basic data types | int, char, float, double |
| Enumeration data type | Enum |
| Derived data type | pointer, array, structure, union |
| Void data type | Void |

Note –

 The storage size of int data type is 2 or 4 or 8 byte.

 It varies depend upon the processor in the CPU that we use.  If we are using 16 bit processor, 2 byte  (16 bit) of memory will be allocated for int data type. Like wise, 4 byte (32 bit) of memory for 32 bit processor and 8 byte (64 bit) of memory for 64 bit processor is allocated for int datatype.

 int (2 byte) can store values from -32,768 to +32,767

 int (4 byte) can store values from -2,147,483,648 to +2,147,483,647.

 If we use int data type to store decimal values, decimal values will be truncated and we will get only whole number.

#include <stdio.h>

void main**(){**

int d**;**

d **=** 'A' **+** 8**;** //65+8,because 65 is ASCII value for 'A'

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

printf**(**"%d"**,**'A'**);** // ascii value for 'A'

**}**

73

65

// char is integer type. 1 byte size.

// int is integer type. 4 byte size on 32 bits machine.

Also, read below program carefully. Run it and observe what is going on at i= 10 and i = 27.

#include <stdio.h>

void main**(){**

int i**;**

**for(**i**=**0**;** i **<**128**;** **++**i**){**

printf**(**"for i=%d, c is %c\n"**,** i**,**i**);**

**}**

**}**

**for i=0, c is**

**for i=1, c is**

**for i=2, c is**

**for i=3, c is ♥**

**for i=4, c is ♦**

**for i=5, c is ♣**

**for i=6, c is ♠**

**for i=10, c is**

**//blank line is part of output as ‘\n’ represents 10**

**for i=11, c is**

Another example-

#include <stdio.h>

void main**(){**

**if(**'A'**>** 'r' **||** 'A' **>** 70**) // See how ‘OR’ has been implemented here.**

printf**(**"A\n"**);**

**else**

printf**(**"B\n"**);**

**}**

**B //output**

**Notice how characters have been compared. ‘A’ is smaller than ‘r’ because asci value of ‘A’ is 65 which is smaller than the ascii value of ‘r’. Also 65 is less than 70 so both conditions fail and output is ‘B’.**

Note - the compiler "promotes" the smaller type (char) to be the same size as the larger type (int) before combining the values. Promotions are determined at compile time based purely on the **types** of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information. For example, division such as 3.0/2. Here 2 will be promoted to float.

The opposite of promotion, truncation moves a value from a type to a smaller type. In that case, the compiler just drops the extra bits.

Assigning from an integer to a smaller integer (e.g.. long to int, or int to char) drops the most significant bits. Assigning from a

floating point type to an integer drops the fractional part of the number.

#include <stdio.h>

void main**(){**

char ch**;**

int i**;**

i **=** 321**;**

ch **=** i**;**

printf**(**"%d"**,** ch**);**

**}**

65

The assignment will drop the upper bits of the int 321. The lower 8 bits of the number 321 represents the number 65 (321 - 256). So the value of ch will be (char)65 which happens to be 'A'. The assignment of a floating point type to an integer type will drop the fractional part of the number. The following code will set i to the value 3. This happens when assigning a floating point number to an integer or passing a floating point number to a function which takes an integer.

double pi;

int i;

pi = 3.14159;

i = pi; // truncation of a double to fit in an int

// i is now 3

### Modifiers in C language:

* The amount of memory space to be allocated for a variable is derived by modifiers.
* Modifiers are prefixed with basic data types to modify (either increase or decrease) the amount of storage space allocated to a variable.
* For example, storage space for int data type is 4 byte for 32 bit processor. We can increase the range by using long int which is 8 byte. We can decrease the range by using short int which is 2 byte.
* There are 5 modifiers available in C language. They are,
  1. short
  2. long
  3. signed
  4. unsigned
  5. long long

Below table gives the detail about the storage size of each C basic data type in 16 bit processor. Please keep in mind that storage size and range for int and float datatype will vary depend on the CPU processor (8,16, 32 and 64 bit)

|  |  |
| --- | --- |
| **C Data types / storage Size** | **Range** |
| char / 1 | –127 to 127 |
| int / 2 | –32,767 to 32,767 |
| float / 4 | 1E–37 to 1E+37 with six digits of precision |
| double / 8 | 1E–37 to 1E+37 with ten digits of precision |
| long double / 10 | 1E–37 to 1E+37 with ten digits of precision |
| long int / 4 | –2,147,483,647 to 2,147,483,647 |
| short int / 2 | –32,767 to 32,767 |
| unsigned short int / 2 | 0 to 65,535 |
| signed short int / 2 | –32,767 to 32,767 |
| long long int / 8 | –(2power(63) –1) to 2(power)63 –1 |
| signed long int / 4 | –2,147,483,647 to 2,147,483,647 |
| unsigned long int / 4 | 0 to 4,294,967,295 |
| unsigned long long int / 8 | 2(power)64 –1 |

### ****sizeof() operator in C language:****

#include <stdio.h>

#include <limits.h>

void main**()**

{

int a**;** char b**;** float c**;** double d**;**

printf**(**"Storage size for int data type: %d\n"**,** **sizeof(**a**));**

printf**(**"Storage size for char data type: %d\n"**,** **sizeof(**b**));**

printf**(**"Storage size for float data type: %d\n"**,** **sizeof(**c**));**

printf**(**"Storage size for double data type: %d\n"**,** **sizeof(**d**));**

**}**

Storage size for int data type: 4  
Storage size for char data type: 1  
Storage size for float data type: 4  
Storage size for double data type: 8

//above program was run on 32 bit machine.

### Struct

#include <stdio.h>

struct xampl **{**

int x**;**

**};**

int main**()**

**{**

struct xampl structure**;**

struct xampl **\***ptr**;**

structure**.**x **=** 12**;**

ptr **=** **&**structure**;** /\* Yes, you need the & when dealing with structures and using pointers to them\*/

printf**(**"%d\n"**,** ptr**->**x **);**

/\* The -> acts somewhat like the \* when does when it is used with pointers. It says, get whatever is at that memory address Not "get what that memory address is"\*/

printf**(**"%d\n"**,** **\***ptr**);**

getchar**();**

**}**

**12**

**12**

In above example, we used ptr->x. This is kind of shorthand for (\*ptr).x which is same as structure.x. So the rule is this: if you have

a struct, use the dot operator; if you have a *pointer* to a struct, use the arrow operator (**->**).

**Passing struct pointer to function** –

#include <stdio.h>

struct mutantfrog **{**

int num\_legs**;**

int num\_eyes**;**

**};**

void build\_beejs\_frog**(**struct mutantfrog **\***f**)**

**{**

f**->**num\_legs **=** 10**;**

f**->**num\_eyes **=** 1**;**

**}**

int main**(**void**)**

**{**

struct mutantfrog rudolph**;**

build\_beejs\_frog**(&**rudolph**);** /\* passing a pointer to the struct \*/

printf**(**"leg count: %d\n"**,** rudolph**.**num\_legs**);** /\* prints "10" \*/

printf**(**"eye count: %d\n"**,** rudolph**.**num\_eyes**);}**

#### Typedef

Above, we define a new type ‘struct xampl’ and declare 2 variables ‘structure’ and ‘\*ptr’ of this type. To save typing efforts, we could write this after declaring struct type –

typedef struct xample Foo;

Foo structre, \*ptr;

Now we can use ‘Foo’ as a shorthand for ‘struct xample’. See below for illustration.

Another interesting example -

#include<stdio.h>

struct test**{**

int id**;**

char category**;**

**};**

typedef struct test Test;

void main**()**

**{**

Test test1**,** test2**;**

test1**.**id **=** 1**;**

test1**.**category **=** 'a'**;**

test2**.**category **=** test1**.**category**;**

test2**.**id **=** 2**;**

printf**(**"%s\n"**,** test1**); // this line crashes the program**

printf**(**"%c\n"**,** test2**.**category**);**

**}**

Remember there is no direct way to use **printf()** to print out **struct**. The above program gets compiled but crashes when run because of commented line. If we change %s to %d, the program runs but because we got lucky. The first element of **struct** is an integer and this is what is printed. In essence, this program is wrong because we can’t print struct and this program doesn’t complain & runs silently. C is an unforgiving language.

### struct of arrays

struct fraction {

int numerator;

int denominator; };

struct fraction f1, f2; // declare two fractions

f1.numerator = 22;

f1.denominator = 7;

f2 = f1; // this copies over the whole struct

struct fraction numbers[1000];

numbers[0].numerator = 22; // set the 0th struct fraction

numbers[0].denominator = 7;

Note – ‘struct fraction’ is also a type like ‘int’ or ‘char’, so we can always declare a pointer of this type –

struct fraction \*f1, \*f2;

/\*see carefully how struct type and pointer works together.\*/

#include <stdio.h>

void main**(){**

struct fraction **{**

int numerator**;**

int denominator**;**

**};**

struct fraction **\***f**;**

struct fraction f1**;**

f **=** **&**f1**;**

f1**.**numerator **=** 1**;**

f1**.**denominator **=** 3**;**

printf**(**"f1.numerator : %d\n"**,** f1**.**numerator**);**

printf**(**"address of f1 : %d\n"**,** f**);**

printf**(**"address of f1.num : %d\n"**,** **&**f1**.**numerator**);**

printf**(**"address of f1.den : %d\n"**,** **&**f1**.**denominator**);**

printf**(**"size of f1 : %d\n"**,** **sizeof(**f1**));**

**}**

f1.numerator : 1

address of f1 : 2280532

address of f1.num : 2280532

address of f1.den : 2280536

size of f1 : 8

Read this [SO link](https://stackoverflow.com/questions/346536/difference-between-a-structure-and-a-union) for difference between union and struct.

### enum

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Example 1 -

# include <stdio.h>

void main**()**

**{**

enum MONTH **{**Jan **=** 0**,** Feb**,** Mar**};**

enum MONTH month **=** Mar**;**

**if(**month**==**0**)**

printf**(**"Value of Jan"**);**

**else** **if(**month **==** 1**)**

printf**(**"Month is Feb"**);**

**else(**month **==** 2**);**

printf**(**"Month is Mar"**);**

**}**

Example 2 –

#include<stdio.h>

enum week**{**Mon**,** Tue**,** Wed**,** Thur**,** Fri**,** Sat**,** Sun**};**

int main**()**

**{**

enum week day**;**

day **=** Wed**;**

printf**(**"%d"**,**day**);**

**return** 0**;**

**}**

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Example 3

#include<stdio.h>

enum year**{**Jan**,** Feb**,** Mar**,** Apr**,** May**,** Jun**,** Jul**,**

Aug**,** Sep**,** Oct**,** Nov**,** Dec**};**

int main**()**

**{**

int i**;**

**for** **(**i**=**Jan**;** i**<=**Dec**;** i**++)**

printf**(**"%d "**,** i**);**

**return** 0**;**

**}**

**o/p -** 0 1 2 3 4 5 6 7 8 9 10 11

## if

#include <stdio.h>

main**()**

**{**

int i **=** 10**;**

**if(**1**)**

printf**(**"Hi there\n"**);**

printf**(**"True hence %d\n"**,** i **==** 10**);**

printf**(**"False hence %d\n"**,** i **>** 20**);**

**}**

**Notice how if(1) evaluates to true also notice that first printf prints 1 while second print 0.**

## for loop (infinite loop with ‘break’)

#include <stdio.h>

main**()**

**{**

int i**;**

**for(;;){**

printf**(**"enter some number\n"**);**

scanf**(**"%d"**,** **&**i**);**

**if(**i **==** 3**)**

**break;**

**}**

printf**(**"program terminated"**);**

**}**

**Notice that all three components of ‘for’ loop has been omitted, thereby making this ‘infinite loop’,presumably to be broken by other means, such as ‘break’ or ‘return’.**

**Note – while writing this program, I forgot to include ‘{}’ in for loop, due to this program didn’t compile. If there is single expression, this can be omitted. Otherwise, it must be included.**

## while loop

#include <stdio.h>

void main**(){**

int i**;**

i **=** 0**;**

**while(**i**-**10**){**

printf**(**"%d"**,** i**);**

i**++;**

**if** **(**i **==** 8**)**

**break;**

**}} // o/p -> 1234567**

**If ‘i’ was tested for equality with 12, instead of 8, we would have got 123456789 as ouput. Notice, how ‘break’ works here.**

#include <stdio.h>

//continue use

void main**(){**

int i**;**

i **=** **-** 1**;**

**while(**i**-**10**){**

i**++;**

**if** **(**i **==** 7**)**

**continue;**

printf**(**"%d\n"**,** i**);**

**}}**

**o/p – 01234568910 //each digit in new line. Notice 7 isn’t printed.**

C does not have a distinct boolean type-- int is used instead. The language treats integer 0 as false and all non-zero values as true. So the statement...

i = 0;

while (i - 10) {

...

will execute until the variable i takes on the value 10 at which time the expression (i -10) will become false (i.e. 0) in above example.

## Do – while loop

#include <stdio.h>

int main**()**

**{**

int x **=** 1**;**

**do** **{**

/\* "Hello, world!" is printed at least once even though the condition is false \*/

printf**(** "Hello, world!\n" **);**

x**++;**

printf**(**"%d\n"**,** x**);**

**}** **while** **(** x **<**0 **);**

**}**

**o/p-**

**Hello World!**

**2 // ‘do-while’executes statements at least once.**

## Pointer

* Pointers in C language is a variable that stores/points the address of another variable.
* A Pointer in C is used to allocate memory dynamically i.e. at run time.
* The pointer variable might be belonging to any of the data type such as int, float, char, double, short etc.
* Pointer Syntax : data\_type \*var\_name; Example : int \*p;  char \*p;
* Where, \* is used to denote that “p” is pointer variable and not a normal variable.
* Normal variable stores the value whereas pointer variable stores the address of the variable.
* The content of the C pointer always be a whole number i.e. address.
* Always C pointer is initialized to null, i.e. int \*p = null.
* The value of null pointer is 0.
* & symbol is used to get the address of the variable.
* \* symbol is used to get the value of the variable that the pointer is pointing to.
* If a pointer in C is assigned to NULL, it means it is pointing to nothing.
* Two pointers can be subtracted to know how many elements are available between these two pointers.
* But, Pointer addition, multiplication, division are not allowed.
* The size of any pointer is 2 byte (for 16 bit compiler).

#include <stdio.h>

int main**()**

**{**

int **\***ptr**,** q**;**

q **=** 50**;**

/\*address of q is assigned to ptr\*/

ptr **=** **&**q**;**

/\*below we get the value of q using ptr variable by way of dereferencing\*/

printf**(**"%d\n"**,** **\***ptr**);**

/\*below we get the value of address hold by ptr\*/

printf**(**"%d\n"**,** ptr**);**

**return** 0**;**

**}**

**o/p-**

**50**

**1244984**

**Another example –**

#include <stdio.h>

int main**()**

**{**

int x**, \***p**;**

/\* A pointer to an integer ("\*p" is an integer, so p must be a pointer to an integer) \*/

p **=** **&**x**;** /\* Read it, "assign the address of x to p" \*/

scanf**(**"%d"**,** **&**x**);** /\* Put a value at address of x, we could also use p here \*/

printf**(**"%d\n"**,** **\***p**);** /\* Note the use of the \* to get the value \*/

printf**(**"%d\n"**,** p**);** /\*value of p, that is address\*/

getchar**();**

**}**

Input value – 45

Output –

45

1244984

In above example, after running above above program, we input some value, say, 45. This is then assigned to ‘x’. By way of expression ‘p = &x’, we get the address of ‘x’ and assign the same to ‘p’. Later on, by way of ‘\*p’, we get the value of x through ‘dereferencing’. Note that ‘p’ stores the value of address of ‘x’. The second ‘printf’ prints this value.

### Pointer and array

From KR –

int x **=**1**,** y **=**2**,** z**[**10**];**

int **\***ip**;** /\* ip is a pointer to int\*/

ip **=** **&**x**;** /\* ip now points to int\*/

y **=** **\***ip**;** /\* y is now 1\*/

**\***ip **=** 0**;** /\* x is now 0\*/

ip **=** **&**z**[**0**];** /\* now points to z[0]\*/

#include <stdio.h>

int main**()**

**{**

int b**[**10**]** **={**0**};**

int **\***p**,\***q,\*r**;**

p **=** **&**b**[**0**];**

q **=** **&**b**[**1**];**

q **=** **&**b**[**2**];**

b**[**0**]** **=** 1**;**

b**[**1**]** **=** 3**;**

printf**(**"%d\n"**,** p**);**

printf**(**"%d\n"**,** **\***p**);**

printf**(**"%d\n"**,** q**);**

printf**(**"%d\n"**,** **\***q**);**

printf**(**"%d\n"**,++**q**);**

printf**(**"%d\n"**,** **\***r**);**

printf**(**"%d\n"**,** r**);**

**return** 0**;**

**}**

2280496

1

2280500

3

2280504

0

2280504

Notice how pointer variable values got incremented when array is of ‘int’ type. If variables were of ‘char’ type then o/p would’ve been –

2280526

a

2280527

b

2280528

-snipped output--

Notice that in both cases, how values of pointer variables got incremented.

Pointer initialized with NULL

#include <stdio.h>

#include <string.h>

main**()**

**{**

char **\***p**;**

char **\***q **=** **NULL;**

printf**(**"address of p = %u\n"**,** p**);**

strcpy**(**p**,**"hello"**);**

printf**(**"%s\n"**,**p**);**

printf**(**"Copying \"Goodbye\" to q\n"**);**

strcpy**(**q**,** "Goodbye"**);**

printf**(**"String Copied\n"**);**

printf**(**"%s\n"**,**q**);**

**}**

o/p ->

address of p = 1629905056

hello

Copying "Goodbye" to q

2 [main] test 3296 cygwin\_exception::open\_stackdumpfile: Dumping stack trace to test.exe.stackdump

See how above program crashed when we initialized q to NULL.

Another example from KR -

#include <stdio.h>

int strlen**(**char **\***s**)**

**{**

int n**;**

**for** **(**n **=** 0**;** **\***s **!=** '\0'**;** s**++)**

**{**

n**++;**

printf**(**"%d\n"**,**s**);** /\*this is not needed. Just to see what gets printed for s\*/

**}**

**return** n**;**

**}**

main**()**

**{**

char str**[**26**];**

printf**(**"Enter some string\n"**);**

scanf**(**"%s"**,** **&**str**);**

strlen**(**str**);**

printf**(**"length of string is %d"**,** strlen**(**str**));**

**}**

/\*for input like "a c", this program returns weird output. inverted comma is part of string\*/

/\*From KR - as formal parameters in a function definition,

char s[];

and

char \*s;

are equivalent.\*/

In for loop above, we used the condition \*s != ‘\0’. In reference to this, a note from KR (page 38) –

The character constant ‘\0’ represents the character with zero value, 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.

Also,from page 39 –

Be careful to distinguish between a character constant and a string that contains a single character: ‘x’ is not the same as “x”. The former is an integer, used to produce the numeric the value of the letter x in the machine’s character set. The latter is an array of characters that contains one character (the letter x) and a ‘\0’.

### Printing Pointer

#include <stdio.h>

#include <stdlib.h>

int G **=** 0**;** /\*global var\*/

int main**(**int argc**,** char **\*\***argv**)**

**{**

static int s**;**

int a**;**

int **\***p**;**

p **=** malloc**(sizeof(**int**));**

printf**(**"&G = %p\n"**,** **(**void **\*)** **&**G**);**

printf**(**"&s = %p\n"**,** **(**void **\*)** **&**s**);**

printf**(**"&a = %p\n"**,** **(**void **\*)** **&**a**);**

printf**(**"&p = %p\n"**,** **(**void **\*)** **&**p**);**

printf**(**"p = %p\n"**,** **(**void **\*)** p**); /\*notice ‘p’ format specifier\*/**

printf**(**"p = %d\n"**,** **(**void **\*)** p**); /\*notice ‘d’format specifier\*/**

printf**(**"main = %p\n"**,** **(**void **\*)** main**);**

free**(**p**);**

**}**

&G = 0x406018

&s = 0x40601c

&a = 0x22cc5c

&p = 0x22cc58

p = 0x20048ec0 /\*hex representation of address\*/

p = 537169600 /\*decimal representation of address\*/

main = 0x4011d0

Also note that, ‘void \*’ is not necessary above. It is machine dependent.

## Memory segment

This part is from Think OS book.

The data of a running process is organized into 5 segments:

• The code segment contains the program text; that is, the machine language instructions that make up the program.

• The static segment contains immutable values, like string literals. For example, if your program contains the string "Hello, World", those character will be stored in the static segment.

• The globals segment contains global variables and local variables that are declared static.

• The heap segment contains chunks of memory allocated at run time, usually by calling the C library function malloc.

• The stack segment contains the run-time stack, which is made up of stack frames. Each time a function is called, a stack frame is allocated to contain the parameters and local variables of the function.

The arrangement of these segments is determined partly by the compiler and partly by the operating system. The details vary from one system to another, but in the most common arrangement:

• The text segment is near the “bottom” of memory, that is, at addresses near 0.

• The static segment is often just above the text segment, that is, at higher addresses.

• The global segment is often just above the static segment.

• The heap is often above the static segment. As it expands, it grows up toward larger addresses.

• The stack is near the top of memory; that is, near the highest addresses in the virtual address space. As the stack expands, it grows down toward smaller addresses.

To determine the layout of these segments on your system, try running this program -

#include <stdio.h>

#include <stdlib.h>

int global;

int main ()

{

int local = 5;

void \*p = malloc(128);

char \*s = "Hello, World";

printf ("Address of main is %p\n", main);

printf ("Address of global is %p\n", &global);

printf ("Address of local is %p\n", &local);

printf ("Address of p is %p\n", p);

printf ("Address of s is %p\n", s);

}

main is the name of a function; when it is used as a variable, it refers to the address of the first machine language instruction in main, which we expect to be in the text segment. global is a global variable, so we expect it to be in the global segment. Local is a local variable, so we expect it to be on the stacks refers to a “string literal", which is a string that appears as part of the program (as opposed to a string that is read from a file, input by a user, etc.). We expect the location of the string to be in the static segment (as opposed to the pointer, s, which is a local variable). p contains an address returned by malloc, which allocates space in the heap. “malloc” stands for “memory allocate.” The format sequence %p tells printf to format each address as a “pointer”, so it displays the results in hexadecimal. When I run this program, the output looks like this (I added spaces to make it easier to read):

Address of main is 0x 40057d

Address of global is 0x 60104c

Address of local is 0x7ffe6085443c

p points to 0x 16c3010

s points to 0x 4006a4

As expected, the address of main is the lowest, followed by the location of the string literal. The location of global is next, then the address p points to. The address of local is much bigger. The largest address has 12 hexadecimal digits. Each hex digit corresponds to 4 bits, so it is a 48-bit address. That suggests that the usable part of the virtual address space is 248 bytes.

## #include

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

#include "foo.h" // refers to a "user" foo.h file --

// in the originating directory for the compile

#include <foo.h> // refers to a "system" foo.h file --

// in the compiler's directory somewhere

## #define

#include <stdio.h>

#define PI 3.1415

#define circleArea(r) (PI\*r\*r)

int main**()**

**{**

int radius**;**

float area**;**

printf**(**"Enter the radius: "**);**

scanf**(**"%d"**,** **&**radius**);**

area **=** circleArea**(**radius**);**

printf**(**"Area = %.2f"**,** area**);**

**return** 0**;**

**}**

## C program to display its own source code using \_\_FILE\_\_

#include <stdio.h>

int main**()** **{**

FILE **\***fp**;**

char c**;**

fp **=** fopen**(**\_\_FILE\_\_**,**"r"**);**

**do** **{**

c **=** getc**(**fp**);**

putchar**(**c**);**

**}**

**while(**c **!=** EOF**);**

fclose**(**fp**);**

**return** 0**;**

**}**

Another example – (predefined macros)

#include <stdio.h>

int main**()**

**{**

printf**(**"Current time: %s\n"**,**\_\_TIME\_\_**);** //calculate the current time

printf**(**"Current date: %s\n"**,**\_\_DATE\_\_**);**

printf**(**"Current file: %s\n"**,**\_\_FILE\_\_**);**

printf**(**"Current time: %s\n"**,**\_\_LINE\_\_**);** // at this point program crashed

// this not working -> printf("Current time: %s",\_\_STDC\_\_);

**}**

## Unary Increment Operators: ++ --

The unary ++ and -- operators increment or decrement the value in a variable. There are "pre" and "post" variants for both operators which do slightly different things (explained below)

*var*++ increment "post" variant

++*var* increment "pre" variant

*var*-- decrement "post" variant

--*var* decrement "pre" variant

int i = 42;

i++; // increment on i

// i is now 43

i--; // decrement on i

// i is now 42

**Pre and Post Variations**

The Pre/Post variation has to do with nesting a variable with the increment or decrement operator inside an expression -- should the entire expression represent the value of the variable before or after the change? I **never** use the operators in this way (see below), but an example looks like...

int i = 42;

int j;

j = (i++ + 10);

// i is now 43

// j is now 52 (NOT 53)

j = (++i + 10)

// i is now 44

// j is now 54

## Conditional Expression -or- The Ternary Operator

The conditional expression can be used as a shorthand for some if-else statements. The general syntax of the conditional operator is:

<expression1> ? <expression2> : <expression3> This is an expression, not a statement, so it represents a value. The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2. Otherwise, it evaluates and returns expression3.

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of..

.

if (x < y) {

min = x;

}

else {

min = y;

}

12

You just say - min = (x < y) ? x : y;

## Function

#include <stdio.h>

Static int add**(**int a**,** int b**)** //function declaration

/\*static means function can only be called in the same file where it is defined.\*/

**{**

int c**;** //function definition

c **=** a **+** b**;**

**return** c**;**

**};**

int main**()**

**{**

int x**;** int y**;** int res**;**

/\*forgot to assign values to x and y. program ran anyway and gave weird result\*/

scanf**(**"%d"**,** **&**x**);**

scanf**(**"%d"**,** **&**y**);**

res **=** add**(**x**,**y**);** //function call

printf**(**"result is %d\n"**,** res**);**

**return** 0**;**

**}**

When you pass a value to a function, *a copy of that value* gets made in this magical mystery world known as *the stack*.

**Function Prototypes**

you have to define the function before you used it, otherwise the compiler wouldn't know about it ahead of time, and would bomb out with an error. This isn't quite strictly true. You can notify the compiler in advance that you'll be using a function of a certain type that has a certain parameter list and that way the function can be defined anywhere at all, as long as the *function prototype* has been declared first. Fortunately, the function prototype is really quite easy. It's merely a copy of the first line of the function definition with a semicolon tacked on the end for good measure. For example, this code calls a function that is defined later, because a prototype has been declared first:

#include <stdio.h>

Static int add**(**int a**,** int b**)**; //function prototype

int main**()**

**{**

int x**;** int y**;** int res**;**

/\*forgot to assign values to x and y. program ran anyway and gave weird result\*/

scanf**(**"%d"**,** **&**x**);**

scanf**(**"%d"**,** **&**y**);**

res **=** add**(**x**,**y**);** //function call

printf**(**"result is %d\n"**,** res**);**

**return** 0**;**

**}**

Static int add**(**int a**,** int b**)**; //function declaration

**{**

int c**;** //function definition

c **=** a **+** b**;**

**return** c**;**

**}**

#### ****How to call C functions in a program?****

There are two ways that a C function can be called from a program. They are,

1. Call by value
2. Call by reference

#### ****1. Call by value:****

* In call by value method, the value of the variable is passed to the function as parameter.
* The value of the actual parameter can not be modified by formal parameter.
* Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

* Actual parameter – This is the argument which is used in function call.
* Formal parameter – This is the argument which is used in function definition

**Example:**

#include<stdio.h>

void swap**(**int a**,** int b**)**

**{**

int tmp**;**

tmp **=** a**;**

a **=** b**;**

b **=** tmp**;**

**};**

int main**()**

**{**

int m **=** 22**,** n **=** 44**;**

printf**(**"values before swap m = %d\n and n = %d\n"**,**m**,**n**);**

swap**(**m**,**n**);**

printf**(**"values after swap m = %d\n and n = %d"**,**m**,**n**);**

**}**

values before swap m = 22

and n = 44

values after swap m = 22

and n = 44

* In this program, the values of the variables “m” and “n” are passed to the function “swap”.
* These values are copied to formal parameters “a” and “b” in swap function and used.
* In reality, no swapping happened because values of ‘m’ and ‘n’ were copied to ‘a’ and ‘b’ and swapping happened to local copies of ‘a’ and ‘b’.

#### ****2. Call by reference:****

* In call by reference method, the address of the variable is passed to the function as parameter.
* The value of the actual parameter can be modified by formal parameter.
* Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example: -

#include<stdio.h>

void swap**(**int **\***a**,** int **\***b**)**

**{**

int tmp**;**

tmp **=** **\***a**;**

**\***a **=** **\***b**;**

**\***b **=** tmp**;**

**};**

int main**()**

**{**

int m **=** 22**,** n **=** 44**;**

printf**(**"values before swap m = %d\n and n = %d\n"**,**m**,**n**);**

swap**(&**m**,&**n**);**

printf**(**"values after swap m = %d\n and n = %d" **,m,** **n);**

**}**

values before swap m = 22

and n = 44

values after swap m = 44

and n = 22

* In this program, the address of the variables “m” and “n” are passed to the function “swap”.
* These values are not copied to formal parameters “a” and “b” in swap function.
* Because, they are just holding the address of those variables.
* This address is used to access and change the values of the variables.

#### Scope and Lifetime of a variable

Every variable in C programming has two properties: type and storage class. Type refers to the data type of a variable. And, storage class determines the scope and lifetime of a variable. There are 4 types of storage class:

1. automatic (local)
2. external (global)
3. static
4. register

Local variables are defined inside the functions and can’t be accessed from outside. Global variables are defined outside of functions and can be accessed by any function. Example of static variable given below –

#include <stdio.h>

void display**()**

**{**

static int c **=** 0**;**

printf**(**"%d "**,**c**);**

c **+=** 5**;**

**};**

int main**()**

**{**

display**();**

display**();**

**};**

**o/p ->**

**0 5**

**If we had removed ‘static’, we would get –**

**0 0**

Another illustration of local variables

int frotz(int a)

{

int b;

b = 10; /\* in scope (from the local definition) \*/

a = 20; /\* in scope (from the parameter list) \*/

c = 30; /\* ERROR, out of scope (declared in another block, in main()) \*/

}

int main(void)

{

int c;

c = 20; /\* in scope \*/

b = 30; /\* ERROR, out of scope (declared above in frotz()) \*/

return 0;

}

#### Parameters to ‘main’ function

#include <stdio.h>

#include <stdlib.h>

int SumRange**(**int start**,** int end**)**

**{**

int i**;**

int sum**;**

sum **=** 0**;**

**for(**i **=** start**;** i **<**end**;** i**++){**

sum **+=** i**;**

**}**

**return** sum**;**

**}**

int main**(**int argc**,** char **\*\***argv**)**

**{**

int start**;**

int end**;**

**if(**argc **!=**3**){**

fprintf**(**stderr**,** "Usage: %s\n start end"**,** argv**[**0**]);**

**return** 1**;**

**}**

start **=** atoi**(**argv**[**1**]);**

end **=** atoi**(**argv**[**2**]);**

printf**(**"SumRange(%d, %d) = %d\n"**,** start**,** end**,** SumRange**(**start**,**end**));**

**return** 0**;**

**}**

In above program notice that how ‘main’ function got 2 parameters, namely, ‘argc’ and argv. Also notice the use of function ‘atoi’ from ‘stdlib’ library.

Suppose there is a program ‘echo’ which echoes its commandline arguments on a single line, separated by blanks. That is the command

echo hello, world

prints the output

hello, world

For this, we pass two parameters to ‘main’ function, namely ‘argc’ and ‘\*argv[]’. Per standard, argv[argc]has to be a null pointer. See illustration below.

world\0

hello,\0

echo\0

0

00

argv: 

‘argv’ is a pointer to an array of character strings that contain the arguments. An example from KR –

clarg.c

#include <stdio.h>

main **(**int argc**,** char **\***argv**[])**

**{**

int i**;**

**for** **(**i **=**1**;** i **<** argc**;** i**++)**

printf**(**"%s%s"**,** argv**[**i**],** **(**i **<** argc**-**1**)** **?** " " **:** ""**); /\*see this carefully\*/**

printf**(**"\n"**);**

**return** 0**;**

**}**

c:\C\KR>clarg 1 2

1 2

Since argv is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. See the another version of above program – (Read page 114-115 from KR to better understand above and below example)

#include <stdio.h>

main **(**int argc**,** char **\***argv**[])**

**{**

**while(--**argc **>** 0**)**

printf**(**"%s%s"**,** **\*++**argv**,** **(**argc **>** 1**)** **?** " " **:** ""**);**

printf**(**"\n"**);**

**return** 0**;**

**}**

## FILE I/0

Declare a pointer of type file -

FILE \*fptr;

Connecting to file –

ptr = fopen("fileopen","mode")

For Example: -

fopen("E:\\cprogram\\newprogram.txt","w");

fopen("E:\\cprogram\\oldprogram.bin","rb");

Closing a file –

fclose(fptr); //fptr is the file pointer associated with file to be closed.

| **File Mode** | **Meaning of Mode** | **During Inexistence of file** |
| --- | --- | --- |
| R | Open for reading. | If the file does not exist, fopen() returns NULL. |
| Rb | Open for reading in binary mode. | If the file does not exist, fopen() returns NULL. |
| W | Open for writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| Wb | Open for writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| A | Open for append. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| Ab | Open for append in binary mode. i.e, Data is added to end of file. | If the file does not exists, it will be created. |
| r+ | Open for both reading and writing. | If the file does not exist, fopen() returns NULL. |
| rb+ | Open for both reading and writing in binary mode. | If the file does not exist, fopen() returns NULL. |
| w+ | Open for both reading and writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb+ | Open for both reading and writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a+ | Open for both reading and appending. | If the file does not exists, it will be created. |
| ab+ | Open for both reading and appending in binary mode. | If the file does not exists, it will be created. |

Example – writing to and reading from text file using fprintf and fscanf

#include <stdio.h>

int main**()**

**{**

int num**;**

FILE **\***fptr**;**

fptr **=** fopen**(**"C:\\c\\program.txt"**,**"w"**);**

**if(**fptr **==** **NULL)**

**{**

printf**(**"Error!"**);**

exit**(**1**);**

**}**

printf**(**"Enter num: "**);**

scanf**(**"%d"**,&**num**);**

fprintf**(**fptr**,**"%d"**,**num**);**

fscanf**(**fptr**,**"%d"**,** **&**num**);**

printf**(**"Value of n=%d"**,** num**);**

fclose**(**fptr**);**

**return** 0**;**

**}**

### Reading and writing to a binary file (using fread() and fwrite())

Use –

fwrite(address\_data,size\_data,numbers\_data,pointer\_to\_file);

fread(address\_data,size\_data,numbers\_data,pointer\_to\_file);

#include <stdio.h>

struct threeNum

**{**

int n1**,** n2**,** n3**;**

**};**

int main**()**

**{**

int n**;**

struct threeNum num**;**

FILE **\***fptr**;**

**if** **((**fptr **=** fopen**(**"C:\\C\\program.bin"**,**"wb"**))** **==** **NULL){**

printf**(**"Error! opening file"**);**

// Program exits if the file pointer returns NULL.

exit**(**1**);**

**}**

**for(**n **=** 1**;** n **<** 5**;** **++**n**)**

**{**

num**.**n1 **=** n**;**

num**.**n2 **=** 5**\***n**;**

num**.**n3 **=** 5**\***n **+** 1**;**

fwrite**(&**num**,** **sizeof(**struct threeNum**),** 1**,** fptr**);**

**}**

fclose**(**fptr**);**

**return** 0**;**

**}**

Program for reading from binary file –

#include <stdio.h>

struct threeNum

**{**

int n1**,** n2**,** n3**;**

**};**

int main**()**

**{**

int n**;**

struct threeNum num**;**

FILE **\***fptr**;**

**if** **((**fptr **=** fopen**(**"C:\\C\\program.bin"**,**"rb"**))** **==** **NULL){**

printf**(**"Error! opening file"**);**

// Program exits if the file pointer returns NULL.

exit**(**1**);**

**}**

**for(**n **=** 1**;** n **<** 5**;** **++**n**)**

**{**

fread**(&**num**,** **sizeof(**struct threeNum**),** 1**,** fptr**);**

printf**(**"n1: %d\tn2: %d\tn3: %d\t"**,** num**.**n1**,** num**.**n2**,** num**.**n3**);**

**}**

fclose**(**fptr**);**

**return** 0**;**

**}**

### Getting data using fseek() –

fseek(FILE \* stream, long int offset, int whence)

Note - didn’t understand the output of following program.

#include <stdio.h>

struct threeNum

**{**

int n1**,** n2**,** n3**;**

**};**

int main**()**

**{**

int n**;**

struct threeNum num**;**

FILE **\***fptr**;**

**if** **((**fptr **=** fopen**(**"C:\\c\\program.bin"**,**"rb"**))** **==** **NULL){**

printf**(**"Error! opening file"**);**

// Program exits if the file pointer returns NULL.

exit**(**1**);**

**}**

// Moves the cursor to the end of the file

fseek**(**fptr**,** **sizeof(**struct threeNum**),** SEEK\_END**);**

**for(**n **=** 1**;** n **<** 5**;** **++**n**)**

**{**

fread**(&**num**,** **sizeof(**struct threeNum**),** 1**,** fptr**);**

printf**(**"n1: %d\tn2: %d\tn3: %d\t"**,** num**.**n1**,** num**.**n2**,** num**.**n3**);**

**}**

fclose**(**fptr**);**

**return** 0**;**

**}**

## ‘system’ function

#include <stdio.h>

main**()**

**{**

system**(**"calc"**);**

system**(**"date"**);** /\*displays result only when CTRL+C pressed\*/

**}**

## String

This part is from Think OS book.

First, remember that C strings are null-terminated. When you allocate space for a string, don’t forget the extra byte at the end.

Also, the letters and numbers in C strings are encoded in ASCII. The ASCII codes for the digits “0” through “9” are 48 through 57, not 0 through 9. The ASCII code 0 is the NUL character that marks the end of a string. And the ASCII codes 1 through 9 are special characters used in some communication protocols. ASCII code 7 is a bell; on some terminals, printing it makes a sound.

The ASCII code for the letter “A” is 65; the code for “a” is 97. Here are those codes in binary:

65 = b0100 0001

97 = b0110 0001

A careful observer will notice that they differ by a single bit. And this pattern holds for the rest of the letters; the sixth bit (counting from the right)

acts as a “case bit”, 0 for upper-case letters and 1 for lower case letters. As an exercise, write a function that takes a string and converts from lowercase to upper-case by flipping the sixth bit. As a challenge, you can make a faster version by reading the string 32 or 64 bits at a time, rather than one character at a time. This optimization is made easier if the length of the string is a multiple of 4 or 8 bytes. If you read past the end of a string, you are likely to see strange characters. Conversely, if you write a string and then accidentally read it as an int or float, the results will be hard to interpret. For example, if you run:

char array[] = "allen";

float \*p = array;

printf("%f\n", \*p);

You will find that the ASCII representation of the first 8 characters of my name, interpreted as a double-precision floating point number, is

69779713878800585457664.

## Pitfalls

#include <stdio.h>

int main**(**void**)**

**{**

unsigned int a **=** 1000**;**

signed int b **=** **-**1**;**

**if(**a**>**b**)** puts**(**"a is more than b"**);**

**else** puts**(**"a is less than or equal to b"**);**

**return** 0**;**

**}**

o/p->

a is less than or equal to b

Since 1000 is more than -1 you would expect the output to be a is more than b, however that will not be the case. Arithmetic operations between different integral types are performed within a common type defined by the so called usual arithmetic conversions (see the language specification, 6.3.1.8). In this case the "common type" is unsigned int, Because, as stated in Usual arithmetic conversions,

*714- Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.*

This means that int operand b will get converted to unsigned int before the comparison. When -1 is converted to an unsigned int the result is the maximal possible unsigned int value, which is greater than 1000, meaning that a > b is false.

Confusion - in above illustration, b still prints to be -1. Why?