**Basic Input Output in C**

# Input

* to feed some data into a program
* input can be given in the form of a file or from the command line
* a set of built-in functions to read the given input and feed it to the program as per requirement

# Output

* to display some data on screen, printer, or in any file
* a set of built-in functions to output the data on the computer screen
* as well as to save it in text or binary files

# Standard Files

* C programming treats all the devices as files
* the following three files are automatically opened when
* a program executes to provide access to the keyboard and screen

|  |  |  |
| --- | --- | --- |
| Standard File | File Pointer | Device |
| Standard input | stdin | stderr |
| Standard output | stdout | Screen |
| Standard error | stderr | Your screen |

* file pointers are the means to access the file for reading and writing purpose

# printf() and scanf() and Functions

**int printf(const char \*format, ...);**

* returns the number of characters successfully written on the output
* writes the output to the standard output stream stdout
* produces the output according to the format provided

**int scanf(const char \*format, ...);**

* returns number of items successfully read
* reads the input from the standard input stream stdin
* scans that input according to the format provided
* expects input in the same format as you provided
* stops reading as soon as it encounters a space

#include <stdio.h>

#include <stdlib.h>

int main() {

char a[16];

char b[16];

char c[16];

printf(" %d\n", printf("%s", "geeksforgeeks"));

fflush(stdout);

printf(" %d\n", scanf("%s %s %s", a, b, c));

return 0;

}

Output:

geeksforgeeks 13

learning input output

3

# getchar() and putchar() Functions

**int getchar(void);**

* reads the next available character from the screen and returns it as an integer
* reads only single character at a time
* can use this method in the loop to read multiple characters

**int putchar(int c);**

* puts the passed character on the screen and returns the same character
* puts only single character at a time
* can use this method in the loop to display multiple characters

#include <stdio.h>

int main( ) {

int c;

printf("Enter a value : "); fflush(stdout);

c = getchar( );

printf( "\nYou entered: ");

putchar( c );

return 0;

}

Output:

Enter a value : Learning

You entered: L

# gets() and puts() Functions

**char \* gets(char \*s);**

* reads a line from stdin into the buffer pointed to by s
* until either a terminating newline or EOF (End of File)

**int puts(const char \*s);**

* writes the string 's' and a trailing newline to stdout

#include <stdio.h>

int main( ) {

char str[100];

printf( "Enter a value :"); fflush(stdout);

gets( str );

printf( "\nYou entered: ");

puts( str );

return 0;

}

Output:

Enter a value :Learning cpp

You entered: Learning cpp

# gets() is risky to use

* it does not do any array bound testing, so
* it suffers from buffer overflow
* gets() keep on reading until it sees a newline character
* **Solution:** use fgets(), it makes sure that not more than MAX\_LIMIT characters are read

|  |  |
| --- | --- |
| **gets()** | **fgets()** |
| #include <stdio.h>  int main() {  char str[8];  gets(str);  printf("%s\n", str);    return 0;  }  Output:  Learning CPP  Learning CPP | #include <stdio.h>  #define MAX\_LIMIT 20  int main() {  char str[MAX\_LIMIT];  fgets(str, MAX\_LIMIT, stdin);  printf("%s\n", str);    return 0;  }  Output:  Learning CPP  Learnin |

# What is return type of getchar(), fgetc() and getc()

* return type is int

char ch; /\* May cause problems \*/

while ((ch = getchar()) != EOF) {

putchar(ch);

}

int in;

while ((in = getchar()) != EOF) {

putchar(in);

}

# Scansets in C

* scanset specifiers are represented by %[]
* scanf family functions support scanset specifiers
* inside scanset, we can specify single character or range of characters
* scanf will process only those characters which are part of scanset
* scansets are case-sensitive
* If first character of scanset is ‘^’, then the specifier will stop reading after first occurrence of that character

#include <stdio.h>

int main(void) {

char str[128];

printf("Enter a string: "); fflush(stdout);

scanf("%[A-Z]s", str);

printf("You entered: %s\n", str);

return 0;

}

Output:

Enter a string: LEARNing CPp

You entered: LEARN

#include <stdio.h>

int main(void) {

char str[128];

printf("Enter a string: "); fflush(stdout);

scanf("%[^o]s", str);

printf("You entered: %s\n", str);

return 0;

}

Output:

Enter a string: https:://geeksforgeeks.org

You entered: https:://geeksf

# gets() function by using scan set

* gets() function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF found.

/\* implementation of gets() function using scanset \*/

#include <stdio.h>

int main(void) {

char str[128];

printf("Enter a string with spaces: "); fflush(stdout);

scanf("%[^\n]s", str);

printf("You entered: %s\n", str);

return 0;

}

Output:

Enter a string with spaces: Learning CPP

You entered: Learning CPP

# How to print % using printf()

1. printf("%%");
2. printf("%c", '%');
3. printf("%s", "%");

**int printf(const char \*format, ...);**

* format string is composed of zero or more directive
* ordinary characters (not %), which are copied unchanged to the output stream
* conversion specifications, each of argument (and it is an error if insufficiently many arguments are given)
* character % is followed by one of the following characters.
  + The flag character
  + The field width
  + The precision
  + The length modifier
  + The conversion specifier
* A `%' is written. No argument is converted.
* The complete conversion specification is`%%'

# use of %n in printf()

* %n is a special format specifier
* causes printf() to load the variable pointed by the corresponding argument
* with a value equal to the number of characters that have been printed by printf() before the occurrence of %n

#include <stdio.h>

int main() {

int c;

printf("geeks for %ngeeks ", &c);

printf("%d", c);

return 0;

}

Output:

geeks for geeks 10

# puts() vs printf()

* less expensive (implementation of puts() is generally simpler than printf())
* if the string has formatting characters like ‘%’, then printf() would give unexpected results
* puts() moves the cursor to next line
* **fputs(str, stdout);** If you do not want the cursor to be moved to next line
* if str is a user input string, then use of printf() might cause security issues
* The problem is that the user can:
  + crash the program: printf ("%s%s%s%s%s%s%s%s%s%s%s%s")
  + view the stack: printf ("%08x %08x %08x %08x %08x\n");
  + view memory on any location, or
  + even write an integer to nearly any location in the process memory.
* This leads to an attacker being able to:
  + Overwrite important program flags that control access privileges
  + Overwrite return addresses on the stack, function pointers, etc

check this

<http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf>

i)

puts("Geeksfor");

puts("Geeks");

Geeksfor

Geeks

ii)

fputs("Geeksfor", stdout);

fputs("Geeks", stdout);

GeeksforGeeks

iii)

printf("Geek%sforGeek%s");

warning: format '%s' expects a matching 'char \*' argument [-Wformat=]

Geekª­lÿforGeek

iv)

puts("Geek%sforGeek%s");

Geek%sforGeek%s

# Difference between getc(), getchar(), getch() and getche()

## getc()

**int getc(FILE \*stream);**

* reads a single character from any given input stream
* **on success:** returns the corresponding integer value (typically ASCII value of read character)
* **on failure:** returns EOF

#include <stdio.h>

int main() {

printf("%c", getc(stdin));

return(0);

}

Output:

Input: g (press enter key)

Output: g

## getchar()

**int getchar(void);**

* getchar() reads from standard input
* getchar() ≈ getc(stdin)

#include <stdio.h>

int main() {

printf("%c", getchar());

return 0;

}

Input: g(press enter key)

Output: g

## getch()

**int getch();**

* reads also a single character from keyboard
* it does not use any buffer, so the entered character is immediately returned without waiting for the enter key.
* *nonstandard function and is present in conio.h*
* mostly used by MS-DOS compilers like Turbo C
* not part of the C standard library or ISO C, nor is it defined by POSIX

#include <stdio.h>

#include <conio.h>

int main() {

printf("%c", getch());

return 0;

}

Input: g (Without enter key)

Output: Program terminates immediately.

But when you use DOS shell in Turbo C,

it shows a single g, i.e., 'g'

## getche()

**int getche(void);**

* non-standard function present in conio.h
* reads a single character from the keyboard and

displays immediately on output screen without waiting for enter key

#include <stdio.h>

#include <conio.h>

int main() {

printf("%c", getche());

return 0;

}

Input: g(without enter key as it is not buffered)

Output: Program terminates immediately.

But when you use DOS shell in Turbo C,

double g, i.e., 'gg'

# How to change the output of printf() in main()

* Use macro arguments

#include <stdio.h>

void fun() {

// add statement to print 10 in main

//#define printf(X, Y) printf(X, 10) // added macro, case - 1

// no macro - case - 2

}

int main() {

int i = 10;

fun();

i = 20;

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

return(0);

}

Output:

10 // case - 1

20 // case – 2