Basics of C programming

Course: Introduction to Programming and Data Structure

Laltu Sardar

Institute for Advancing Intelligence (IAI),
TCG Centres for Research and Education in Science and Technology (TCG Crest)

tcg crest
Inventing Harmonious Future

January 31, 2022

The first C program

```c
//FileName: hello.c
//Printing Hello world
#include <stdio.h>
main()
{
  printf("hello, world\n");

}
```

1. Compilation: gcc hello.c /*a.out file will be generated*/
2. Run ./a.out /* default output file*/

Description

- stdio.h: standard input-output library
- printf: a library function
- input string
- \n - newline

Compiling C program

Output your Name

```c
//FileName: namePrint.c
//Prints given name
#include <stdio.h>
main()
{
  char name[] = "Your Name"
  printf("hello, %s\n", name);
}
```

1. Compilation: gcc -g -Wall namePrint.c -o prog2.out
   - gcc → GNU Compiler Collection
   - gcc -g → generates debug info to be used by GDB debugger
   - -Wall → Show all warnings
2. Run: ./prog2.out
3. “.out” – not mandatory

Output your Name

Description

- char: variable type
- name: variable name
- %s : string output format specifier
- Commenting: Not read by the compiler
  - For single line: //
  - For multiple lines: /* your lines */

Basic input/output from/to a file

Input from terminal **during** execution

- printf() : returns total number of Characters Printed, Or negative value if an output error or an encoding error
- scanf() : Reads input of any datatype from (stdin).
    - Stops reading when it encounters **whitespace, newline or EOF**
    - Returns total number of Inputs Scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.
- gets(): Reads string from standard input.
    - Stops stops reading input when it encounters **newline or EOF**.
    - Returns total number of Inputs Scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.

Note: gets() does not stop reading input when it encounters whitespace instead it takes whitespace as a string.

```c
// Program to compute average of two float variables
#include<stdio.h>

float average(float a, float b){
    return ((a+b)/2.0);
}

int main(){
    float a, b, avg;

    scanf("%f %f", &a, &b);  // taking input from terminal
    avg = average(a, b);     //Compauting avarage
    printf("%f",avg); //writing on terminal
    return 0;
}
```

```c
// Program to compute average of two float variables
#include<stdio.h>

float average(float a, float b){
    return ((a+b)/2.0);
}

int main(){
    float a, b, avg;

    scanf("%f %f", &a, &b);  // taking input from terminal
    avg = average(a, b);    //Compauting avarage
    printf("%f",avg); //writing on terminal
    return 0;
}
```

- Sometimes input is large–
- Sometime we have many inputs
- embedding data directly into the source code– **a bad idea and Not practical**
- We require to take input data from files.

Input from file **during** execution

- fprintf() : returns total number of bytes Printed from file, Or negative value if an output error or an encoding error
- fscanf() : Reads input of any datatype from stream.
    - Stops reading when it encounters **whitespace, newline or EOF**
    - Returns total number of Inputs Scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.
- fgets(): Reads string from stream.
    - Stops stops reading input when it encounters **newline or EOF**.
    - Returns total number of Inputs Scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.

Home Work: Assignment 01

Write details (When to use, formats, return values, terminating conditions, etc.) about the following functions:

1. getchar(), fgetc() and getc(), putchar(), putc()
2. getc(), getchar(), getch() and getche(),
3. fgets()/gets()/scanf().
4. fread()/fseek()

fscanf and fprintf

- fscanf and fprintf works same as scanf and printf

```c
// Program to learn basic file operation
#include<stdio.h>

float average(float a, float b){
    return ((a+b)/2.0);
}

int main(){
    float a, b, avg;

    FILE *inp_file_ptr, *out_file_ptr; //File type pointer must be declared

    inp_file_ptr = fopen("input_file.txt","r"); // Opening input file for reading
    fscanf(inp_file_ptr, "%f %f", &a, &b);  // taking input from file
    fclose(inp_file_ptr);   // closing the input file

    avg = average(a, b);    //Compauting avarage

    out_file_ptr = fopen("output_file.txt","w");
    fprintf(out_file_ptr, "%f",avg); //writing on output file
    fclose(out_file_ptr); //closing the output file

    return 0;
}
```

Command Line Arguments: Input from terminal before execution:

Why inputs from command line

- Another form of input
- Useful when you want to control your program from outside.
- To override defaults and have more direct control over the application

Example:

```c
int main(int argc, char *argv[]) {
  /* ... */
}
```

or

```c
int main(int argc, char **argv) {
  /* ... */
}
```

```c
// Program to compute average of two float variables
#include<stdio.h>
#include<stdlib.h> //that contains atof

float average(float a, float b){
     return ((a+b)/2.0);
}
int main(int argc, char *argv[]){
     float a, b, avg;
     if (argc==3){
          a = atof(argv[1]); //converting string to float
          b = atof(argv[2]);
     }else{
          scanf("%f %f", &a, &b);  // taking input from terminal
     }
     avg = average(a, b);     //Compauting avarage
     printf("%.2f",avg); //writing on terminal
     return 0;
}
```

```c
// Program to compute average of two float variables
#include<stdio.h>
#include<stdlib.h> //that contains atof

float average(float a, float b){
    return ((a+b)/2.0);
}
int main(int argc, char *argv[]){
    float a, b, avg;
    if (argc==3){
        a = atof(argv[1]); //converting string to float
        b = atof(argv[2]);
    }else{
        scanf("%f %f", &a, &b);  // taking input from terminal
    }
    avg = average(a, b);    //Compauting avarage
    printf("%.2f",avg); //writing on terminal
    return 0;
}
```

- argc (ARGument Counter): is The number of command-line arguments passed. It includes the name of the program
- argv (ARGument Vector): An array of strings pointers listing all the arguments.
- argv[0] is the name of the program , After that till argv[argc-1] every element is command-line arguments.
- Only strings can be taken from command line.

Basics of C programming:

Variables and Arithmetic Expressions

```c
/* filenme: FahToCel.c
  print Fahrenheit-Celsius table
  for fahr = 0, 20, ..., 300
*/
#include <stdio.h>
main()
{
  int fahr, celsius;     //variable Declaration
  int lower, upper, step;
  lower = 0; /* lower limit of temperature scale */ // variable assignment
  upper = 300; /* upper limit */
  step = 20; /* step size */
  fahr = lower;
  while (fahr <= upper) {    //while loop
    celsius = 5 * (fahr-32) / 9;
    printf("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
}
```

Description

- Variable declaration
- Assign value to a variable
- Each variable must have a format specifier in printf

Building block of a Programming Language

1. Memory = space for calculations, rough work, etc.
2. Variables = names given to memory locations for convenience
3. Instructions = each step in the procedure

Naming rules of variables

Naming Rule of variables

1. Span: letters and digits
2. 1st character must be a letter
3. set of letters = { a, b, . . ., z, A, B, . . ., Z, _ }
4. The underscore "_" is count as letter
5. names are case sensitive.

Traditional C practice

- use lower case for variable names
- use all upper case for symbolic constants.

Variable-Name Examples

- abc_123 → valid
- _abc123 → valid
- _123 → valid
- _123abc → valid
- 123_abc → invalid

Tips

Variable name should be given in such a way that usage of the variable can be guessed easily from its name.

Should not be unnecessary long

Output format specifiers

Format Specifiers

- Format specifiers define the type of data to be printed on standard output.
- You need to use format specifiers whether you're printing formatted output with printf() or accepting input with scanf().

Some frequently used format specifiers

1. %d – decimal integer
2. %6d – decimal integer, at least 6 characters wide
3. %f – floating point
4. %6f – floating point, at least 6 characters wide
5. %.2f – floating point, 2 characters after decimal point
6. %6.2f – floating point, at least 6 wide and 2 after decimal point

Symbolic Constants

```c
#include <stdio.h>
#define LOWER 0 /* lower limit of table */
#define UPPER 300 /* upper limit */
#define STEP 20 /* step size */
/* print Fahrenheit-Celsius table */
main()
{
  int fahr;
  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
  printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

```c
#define name replacement list
```

- symbolic constants are *string of characters*:
- They are not variables
- they do not appear in declarations
- In compiled files, they do not exists
- Conventionally written in upper case only

If

```
if (condition) {
  // block of code to be executed
  //if the condition is true
}
```

Example:

```
int a = 10;
int b = 2;
if (a > b) {
  printf("a is greater than b");
}
```

If-Else

```
if (condition) {
  // block of code to be executed
  //if the condition is True
}else{
  // block of code to be executed
  //if the condition is False
}
```

```
int a = 10;
int b = 2;
if (a > b) {
  printf("a is greater than b");
}else{
  printf("a is less than b");
}
```

If-Else in a single line:

```
  condition ? expression-true : expression-false
```

```
  int a = 10, b = 2;
  (a > b)? printf("a is greater than b"): printf("a is less than b");
```

Else-If

```
if (test expression1) {
  // statement(s)
}
else if(test expression2) {
  // statement(s)
}
else if (test expression3) {
  // statement(s)
}
.
.
else {
  // statement(s)
}
```

```
if (marks > 85) {
  printf("First Class with Distinction");
}
else if(marks > 60) {
  printf("First Class");
}
else if (marks>40) {
  print("Passed");
}
else {
  print("Failed");
}
```

Switch: Psudocode

```
switch (expression)
{
  case constant1:
  // statements
  break;

  case constant2:
  // statements
  break;
  .
  .
  .
  default:
  // default statements
}
```

Switch: Example

```c
char operation;
double n1, n2;
printf("Enter an operator (+, -, *, /): ");
scanf("%c", &operation);
printf("Enter two operands: ");
scanf("%lf %lf",&n1, &n2);

switch(operation)
{
  case '+':
  printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
  break;

  case '-':
  printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
  break;

  case '*':
  printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
  break;

  case '/':
  printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
  break;

  // operator doesn't match any case constant +, -, *, /
  default:
    printf("Error! operator is not correct");
}
```

For and While

```c
for ( init; condition; increment ) {
  statement(s);
}
```

```c
int i;

/* for loop execution */
for( i = 1; i < 10; i = i + 1 ){
  printf("value of i: %d\n", i);
}
```

```c
while(condition) {
  statement(s);
}
```

```c
int i = 1;

/* while loop execution */
while( i < 10 ) {
  printf("value of i: %d\n", i);
  i++;
}
```

for and while loop

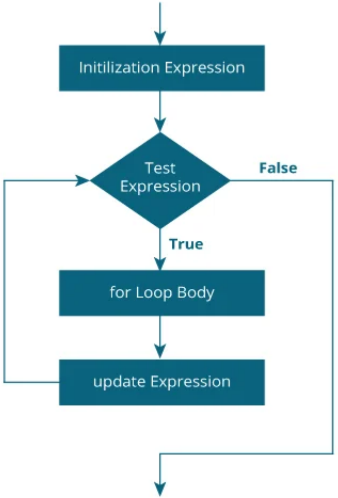
Some frequently used format specifiers

1. The for statement is a loop– a generalization of the while.
2. Three parts— separated by semicolons.
3. The first part– the initialization
4. The second part– Loop controller/ loop terminator
5. The third part– condition re-evaluation

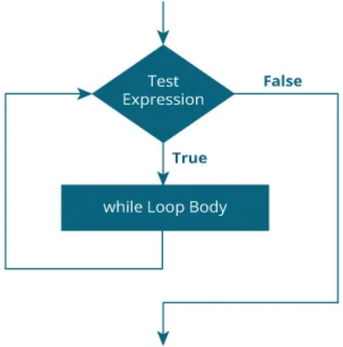
‘For’ or ‘while’: which to use?

- whatever you want
- ‘for’ is more compact. It keeps the loop control statements together in one place

Flowchart of for

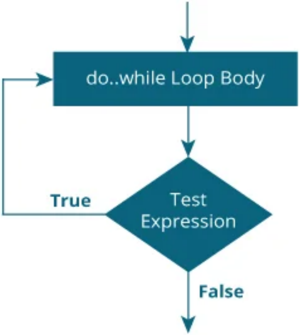


Flowchart of For loop



Flowchart of For while loop

Do-While



Flowchart of Do-while loop

Pseudocode:

```
do {
  // the body of the loop
}
while (testExpression);
```

Example:

```
    int i;

    /* for loop execution */
    i = 1;
    do{
      printf("value of i: %d\n", i);
      i = i + 1 ;
    } while(i < 10);
```

Question: What to use For or While or Do-While?

Break and Continue

`break` statement **terminates** a loop

```
for (int i = 1; i <= 40; i++) {
  printf("value of i: %d\n", i);
  if (i == 10) {
    break;  // terminates the loop

  }
}
```

`continue` **skips** a current iteration of a loop.

```
for (int i = 1; i <= 10; i++{
  printf("value of i: %d\n", i);
  if (i == 3) {
    continue;
  }
}
```

Functions

- So far, we have used printf, open, etc
- We know, we have to pass some parameter, it returns some values
- We don't have to know *how* it is defined
- We only have to know what is defined, and whats are its outputs

Why?

When we do same things with different values, we keep it in functions

Functions

```
return-type function-name(parameter declarations, if any)
{
  declarations
  statements
}
```

Functions

```
return-type function-name(parameter declarations, if any)
{
  declarations
  statements
}
```

```
#include <stdio.h>
int power(int m, int n); //declaration needed
/* test power function */
main()
{
  int i;
  for (i = 0; i < 10; ++i)
  printf("%d %d %d\n", i, power(2,i), power(-3,i));
  return 0;
}
/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
  int i, p;
  p = 1;
  for (i = 1; i <= n; ++i)
  p = p * base;
  return p;
}
```

TOP Secret to be an Expert in programming

Only Secret: Practice!

- Practice code/program writing
- Practice to solve daily eligible problems with coding
- Practice to take new coding challenges

topics tp be covered in some next class

- Character Arrays
- External Variables and Scope

Character array or String

```
char str[]     = "TCG_Crest";
char str[50]   = "TCG_Crest";
char str[]     = {'T','C','G','_','C','r','e','s','t','\0'};
char str[14]   = {'T','C','G','_','C','r','e','s','t','\0'};
```

str =	T	C	G	_	C	r	e	s	t	\0
	0x12345	0x12346	0x12347	0x12348	0x12349	0x12350	0x12351	0x12352	0x12353	0x12356

- Accessing characters: str[0] = T, str[2] = G, etc
- '\0' is null character, terminating character