

Computing Laboratory

Review of C – Operators, Expressions, Control Flow, Basic Input/output

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

October, 2021

The C compiler

Source code → **(C compiler)** → **Object code**

Object code → **(C linker)** → **Executable**

Standard conformance of C

| Significant Features | C89 | C99 | C11 |
|--|-----|-----|-----|
| Implicit function declaration | Yes | No | No |
| IEEE 754 floating point support | No | Yes | Yes |
| Inline functions | No | Yes | Yes |
| long long int | No | Yes | Yes |
| complex type, complex.h | No | Yes | Yes |
| variadic macros | No | Yes | Yes |
| gets | Yes | Yes | No |
| Alignment specification | No | No | Yes |
| No return specification, stdnoreturn.h | No | No | Yes |
| Type generic expressions | No | No | Yes |
| Multi-threading, thread.h | No | No | Yes |
| Bounds checking | No | No | Yes |

Note: C99 and C11 are formally known as ISO/IEC 9899:1999 and ISO/IEC 9899:2011, respectively.

The first C program

Source: Welcome2C.c

The first C program

Source: Welcome2C.c

```
#include<stdio.h>
int main(){
    printf("Welcome 2 C");
    return 0;
}
```

The first C program

Source: Welcome2C.c

```
#include<stdio.h>
int main(){
    printf("Welcome 2 C");
    return 0;
}
```

Compilation: gcc Welcome2C.c

The first C program

Source: Welcome2C.c

```
#include<stdio.h>
int main(){
    printf("Welcome 2 C");
    return 0;
}
```

Compilation: gcc Welcome2C.c

“Welcome2C.c” is compiled and the executable “a.out” is created.

The first C program

Source: Welcome2C.c

```
#include<stdio.h>
int main(){
    printf("Welcome 2 C");
    return 0;
}
```

Compilation: gcc Welcome2C.c

“Welcome2C.c” is compiled and the executable “a.out” is created.

Execution: ./a.out

The first C program

Source: Welcome2C.c

```
#include<stdio.h>
int main(){
    printf("Welcome 2 C");
    return 0;
}
```

Compilation: gcc Welcome2C.c

“Welcome2C.c” is compiled and the executable “a.out” is created.

Execution: ./a.out

Welcome 2 C(cursor here!!!)

Dissecting the code

```
#include<stdio.h>
int main(){
    statement;
    function1(argument);
    statement;
    return 0;
}
int function1(int arg){
    function2();
    statement;
}
void function2(){
    statement;
}
```

Dissecting the code

```
#include<stdio.h>
int main(){
    statement;
    function1(argument);
    statement;
    return 0;
}
int function1(int arg){
    function2();
    statement;
}
void function2(){
    statement;
}
```

Note: The program name can be anything.

Basic characteristics

- A C program should have at least the main() function.

Basic characteristics

- A C program should have at least the main() function.
- There can be at most a single main() function in the program.

Basic characteristics

- A C program should have at least the main() function.
- There can be at most a single main() function in the program.
- Every statement in C must end with a semicolon.

Basic characteristics

- A C program should have at least the `main()` function.
- There can be at most a single `main()` function in the program.
- Every statement in C must end with a semicolon.
- C is case sensitive.

Basic characteristics

- A C program should have at least the `main()` function.
- There can be at most a single `main()` function in the program.
- Every statement in C must end with a semicolon.
- C is case sensitive.
- C is a free-form language.

Basic characteristics

- A C program should have at least the `main()` function.
- There can be at most a single `main()` function in the program.
- Every statement in C must end with a semicolon.
- C is case sensitive.
- C is a free-form language.
- C is a loosely typed language.

Basic characteristics

- A C program should have at least the `main()` function.
- There can be at most a single `main()` function in the program.
- Every statement in C must end with a semicolon.
- C is case sensitive.
- C is a free-form language.
- C is a loosely typed language.
- C is a middle level language.

Free-form language

A C statement can be broken into multiple lines.

Free-form language

A C statement can be broken into multiple lines. E.g., the statements

```
printf("Welcome 2 C");
```

and

```
printf  
("Welcome 2 C");
```

are both same producing the same output.

Loosely typed language

```
char c;  
int i;  
float f;
```

Loosely typed language

```
char c;  
int i;  
float f;
```

Here, the data types of c, i, f become flexible. They can be used interchangeably.

Loosely typed language

```
char c;  
int i;  
float f;
```

Here, the data types of `c`, `i`, `f` become flexible. They can be used interchangeably. E.g., assigning “`c = 70`” is same as assigning “`c = 'F'`”.

Loosely typed language

```
char c;  
int i;  
float f;
```

Here, the data types of `c`, `i`, `f` become flexible. They can be used interchangeably. E.g., assigning “`c = 70`” is same as assigning “`c = 'F'`”.

Note: Type casting is still there for converting the data types.

Loosely typed language

What will be the output of following?

```
int i = 50, j = 2;
printf("%c, %d\n", i, j);
printf("%c, %d", 2*i, 2*j);
```

Loosely typed language

What will be the output of following?

```
int i = 50, j = 2;
printf("%c, %d\n", i, j);
printf("%c, %d", 2*i, 2*j);
```

Output:

2, 2

d, 4

| | | | | | | | | | | | | | |
|-------------|-----|----|-----|----|-----|----|-----|----|-----|----|-----|-----|-----|
| ASCII value | ... | 48 | ... | 57 | ... | 65 | ... | 90 | ... | 97 | ... | 122 | ... |
| Character | ... | 0 | ... | 9 | ... | A | ... | Z | ... | a | ... | z | ... |

In-built types

Integer data types

| Type | Size* | Minimum | Maximum |
|------------------------|---------|-----------|--------------|
| char | 8 bits | -2^7 | $2^7 - 1$ |
| short int | 16 bits | -2^{15} | $2^{15} - 1$ |
| int | 32 bits | -2^{31} | $2^{31} - 1$ |
| long int | 32 bits | -2^{31} | $2^{31} - 1$ |
| long long int | 64 bits | -2^{63} | $2^{63} - 1$ |
| unsigned char | 8 bits | 0 | $2^8 - 1$ |
| unsigned short int | 16 bits | 0 | $2^{16} - 1$ |
| unsigned int | 32 bits | 0 | $2^{32} - 1$ |
| unsigned long int | 32 bits | 0 | $2^{32} - 1$ |
| unsigned long long int | 64 bits | 0 | $2^{64} - 1$ |

* Dependent on machine configuration.

In-built types

| <i>decimal</i> | <i>binary</i> |
|----------------|------------------|
| 0 | 0000000000000000 |
| 1 | 0000000000000001 |
| 2 | 0000000000000010 |
| 3 | 0000000000000011 |
| ... | ... |
| 32765 | 0111111111111101 |
| 32766 | 0111111111111110 |
| 32767 | 0111111111111111 |
| -32768 | 1000000000000000 |
| -32767 | 1000000000000001 |
| -32766 | 1000000000000010 |
| -32765 | 1000000000000011 |
| ... | ... |
| -3 | 1111111111111101 |
| -2 | 1111111111111110 |
| -1 | 1111111111111111 |

16-bit integers
(two's complement)

In-built types

“Real” (floating point) numbers

| Type | Size |
|-------------|------|
| float | 32 |
| double | 64 |
| long double | 128 |

Examples:

| | |
|----------|------------|
| 1.23456 | 3.45e67 |
| 1. | +3.45e67 |
| .1 | -3.45e-67 |
| -0.12345 | .00345e-32 |
| +.4560 | 1e-15 |

In-built types

“Real” (floating point) numbers

| Type | Size |
|-------------|------|
| float | 32 |
| double | 64 |
| long double | 128 |

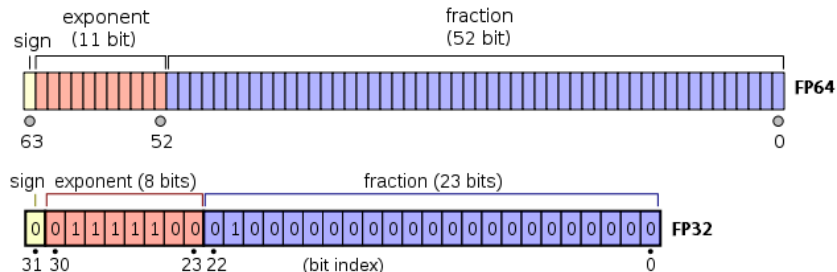
Examples:

| | |
|----------|------------|
| 1.23456 | 3.45e67 |
| 1. | +3.45e67 |
| .1 | -3.45e-67 |
| -0.12345 | .00345e-32 |
| +.4560 | 1e-15 |

- Do not use commas as thousand-separators.
- At times behavior may be counter-intuitive (more about this later).

IEEE 754 floating point support

IEEE standard for floating point arithmetic (IEEE 754) is a technical standard for floating point representation on digital computers.



Boolean values

Any non-zero value is treated as **TRUE** and zero is treated as **FALSE**.

Examples:

| | | | |
|-------------|-------|-------|-------|
| 0 | False | 0e10 | False |
| 1 | True | 'A' | True |
| 6 - 2 * 3 | False | '\0' | False |
| (6 - 2) * 3 | True | x = 0 | False |
| 0.0075 | True | x = 1 | True |

Type casting

```
int a = 5, b = 2;
float d;
d = a/b;
printf("Division result = %f",d);
```

Type casting

```
int a = 5, b = 2;
float d;
d = a/b;
printf("Division result = %f",d);
```

Division result = 2.000000

Type casting

```
int a = 5, b = 2;
float d;
d = (float)a/b;
printf("Division result = %f",d);
```

Type casting

```
int a = 5, b = 2;
float d;
d = (float)a/b;
printf("Division result = %f",d);
```

Division result = 2.500000

Escape sequences

How can we print the following using print statements?

' , " , \

Escape sequences

How can we print the following using print statements?

' , " , \

What will be the output of following?

```
char *title = "Bhattacharyya";  
printf("This is Malay \r%s",title);
```

Escape sequences

How can we print the following using print statements?

' , " , \

What will be the output of following?

```
char *title = "Bhattacharyya";  
printf("This is Malay \r%s",title);
```

Output:

Bhattacharyya

Arithmetic operators

Summation (+)

Subtraction (−)

Multiplication (*)

Division (/)

Modulo division (%)

Arithmetic operators

Summation (+)

Subtraction (−)

Multiplication (*)

Division (/)

Modulo division (%)

Note: Modulo division operator works on the integers (negative too!!!) only and returns the sign of the numerator.

Relational operators

Less than ($<$)

Less than equals to ($<=$)

Greater than ($>$)

Greater than equals to ($>=$)

Equals to ($==$)

Not equals to ($!=$)

Logical operators

Logical and (&&)

Logical or (||)

Logical not (!)

Assignment operators

Assignment (=)

Increment and decrement operators

Increment (++)

Decrement (--)

Increment and decrement operators

```
int i = 5;
i++;
printf("%d", i);
printf("\n%d", i++); // Post-increment operation
printf("\n%d", ++i); // Pre-increment operation
```

Increment and decrement operators

```
int i = 5;
i++;
printf("%d", i);
printf("\n%d", i++); // Post-increment operation
printf("\n%d", ++i); // Pre-increment operation
```

6

6

8

Conditional operators

Ternary (?:)

```
int i = 0;
int j = 1;
int result = (i > j)? i : j;
```


Bitwise operators

Bitwise and (&)

Bitwise or (|)

Bitwise not (~)

Bitwise xor (^)

Left shift (<<)

Right shift (>>)

Bitwise operators

```
int i = 11;
printf("%d", i>>1); // 1 place right shift (i unchanged)
printf("\n%d", i<<2); // 2 places left shift (i unchanged)
```

Bitwise operators

```
int i = 11;
printf("%d", i>>1); // 1 place right shift (i unchanged)
printf("\n%d", i<<2); // 2 places left shift (i unchanged)
```

5

44

Bitwise operators

```
int i = 11;
printf("%d", i>>1); // 1 place right shift (i unchanged)
printf("\n%d", i<<2); // 2 places left shift (i unchanged)
```

5

44

| Decimal | Binary | | | | | | | |
|---------|--------|---|---|---|---|---|---|---|
| 11 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 44 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

Operator precedence

What will be the output of the following program?

```
#include<stdio.h>
int main(){
    int n = 10;
    n = 20, 30, 40;
    printf("First n = %d\n",n);
    n = (50, 60, 70);
    printf("Second n = %d\n",n);
    return 0;
}
```

Operator precedence

What will be the output of the following program?

```
#include<stdio.h>
int main(){
    int n = 10;
    n = 20, 30, 40;
    printf("First n = %d\n",n);
    n = (50, 60, 70);
    printf("Second n = %d\n",n);
    return 0;
}
```

Output:

First n = 20

Second n = 70

Operator precedence

| Precedence | Operator | Description | Associativity | |
|------------|--------------------|---|---------------|---------------|
| 1 | ++ -- | Suffix/postfix increment and decrement | Left-to-right | |
| | () | Function call | | |
| | [] | Array subscripting | | |
| | . | Structure and union member access | | |
| | -> | Structure and union member access through pointer | | |
| | (type){list} | Compound literal(c99) | | |
| 2 | ++ -- | Prefix increment and decrement | Right-to-left | |
| | + - | Unary plus and minus | | |
| | ! ~ | Logical NOT and bitwise NOT | | |
| | (type) | Type cast | | |
| | * | Indirection (dereference) | | |
| | & | Address-of | | |
| | sizeof _Alignof | Size-of Alignment requirement(c11) | | |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right | |
| 4 | + - | Addition and subtraction | | |
| 5 | << >> | Bitwise left shift and right shift | | |
| 6 | < <= | For relational operators < and ≤ respectively | | |
| | > >= | For relational operators > and ≥ respectively | | |
| 7 | == != | For relational = and ≠ respectively | | |
| 8 | & | Bitwise AND | | |
| 9 | ^ | Bitwise XOR (exclusive or) | | |
| 10 | | Bitwise OR (inclusive or) | | |
| 11 | && | Logical AND | | |
| 12 | | Logical OR | | |
| 13 | ?: | Ternary conditional | | Right-to-Left |
| 14 | = | Simple assignment | | |
| | += -= | Assignment by sum and difference | | |
| | *= /= %= | Assignment by product, quotient, and remainder | | |
| | <<= >>= | Assignment by bitwise left shift and right shift | | |
| 15 | &= ^= = | Assignment by bitwise AND, XOR, and OR | | |
| | , | Comma | Left-to-right | |

Conditional – if-else

```
if(Condition){
    statement 1;
    statement 2;
}
else{
    statement 3; // Execute if Condition fails
}
```


Conditional – if-else

```
if(Condition){
    statement 1;
    statement 2;
}
else{
    statement 3; // Execute if Condition fails
}
```

Note: A single statement can be included (within if/else) by default without using any brackets.

Conditional – if-else

What will be the output of the following program?

```
if(!printf("Hi!"))
    printf("Hi!");
else
    printf("Bye!");
```

Conditional – if-else

What will be the output of the following program?

```
if(!printf("Hi!"))
    printf("Hi!");
else
    printf("Bye!");
```

Output:

Hi!Bye!

Conditional – switch-case

```
switch (E) {  
    case value1 :  
        statement;  
        break;  
    case value2 :  
        statement;  
        break;  
    ...  
    case valuen :  
        statement;  
        break;  
    default:  
        statement;  
}
```

Iterative – for loop

```
for(Initialization;Condition;Increment/Decrement){  
    statement 1;  
    statement 2;  
}
```

Iterative – for loop

```
for(Initialization;Condition;Increment/Decrement){  
    statement 1;  
    statement 2;  
}
```

Alternatively, we can write

```
for(Initialization;Condition;Increment/Decrement)  
    statement 3;
```

Iterative – for loop

```
for(Initialization;Condition;Increment/Decrement){  
    statement 1;  
    statement 2;  
}
```

Alternatively, we can write

```
for(Initialization;Condition;Increment/Decrement)  
    statement 3;
```

Note: The ‘Initialization’, ‘Condition’, and ‘Increment/Decrement’ are all optional.

Iterative – for loop

What will be the output of the following program?

```
unsigned char i = 0;
for(;i>=0;i++);
    printf("%d\n",i);
```


Iterative – for loop

What will be the output of the following program?

```
unsigned char i = 0;
for(;i>=0;i++);
    printf("%d\n",i);
```

Output:

The program iterates infinitely!!!

Iterative – for loop

What will be the output of the following program?

```
unsigned char i = 0;
for(;i>=0;i++);
    printf("%d\n",i);
```

Output:

The program iterates infinitely!!!

Unsigned characters can never be negative and so $i \geq 0$ never yields FALSE.

Iterative – while loop

```
while(Condition){  
    statement 1;  
    statement 2;  
}
```

Iterative – while loop

```
while(Condition){  
    statement 1;  
    statement 2;  
}
```

Alternatively, we can write

```
while(Condition)  
    statement 3;
```

Iterative – while loop

```
while(Condition){  
    statement 1;  
    statement 2;  
}
```

Alternatively, we can write

```
while(Condition)  
    statement 3;
```

Note: A non-zero value is treated as a TRUE 'Condition', otherwise FALSE.

Iterative – do-while loop

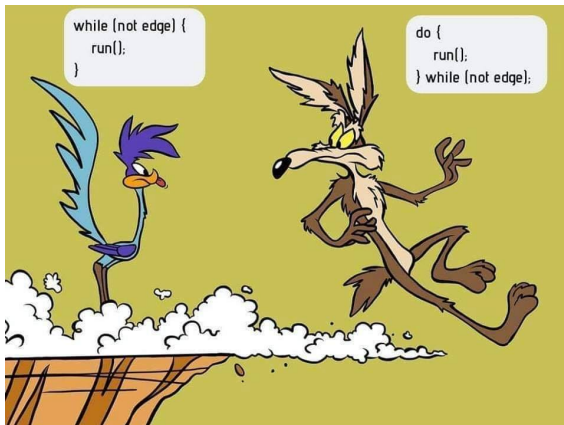
```
do{  
    statement 1;  
    statement 2;  
}while(Condition);
```

Iterative – do-while loop

```
do{  
    statement 1;  
    statement 2;  
}while(Condition);
```

Note: The statements inside the block are executed (by default) at least once irrespective of the 'Condition'.

Iterative – Comparing while and do-while



break and continue

- **break:** immediately jump to the next operation after the loop
- **continue:** do the update operation if applicable, and continue with the next iteration of the loop

break and continue

- **break:** immediately jump to the next operation after the loop
- **continue:** do the update operation if applicable, and continue with the next iteration of the loop

```
for(i=1; i<=100; ++i){  
    printf("%4d",i);  
    if (i%10 != 0)  
        break;  
    printf("\n");  
}
```

break and continue

- **break:** immediately jump to the next operation after the loop
- **continue:** do the update operation if applicable, and continue with the next iteration of the loop

```
for(i=1; i<=100; ++i){
    printf("%4d",i);
    if (i%10 != 0)
        break;
    printf("\n");
}
```

```
i = 0;
while(i < 100){
    ++i;
    printf("%4d",i);
    if (i%10 != 0)
        continue;
    printf("\n");
}
```

Input/Output

I/O from the terminal

- printf, scanf
- getchar, putchar

Input/Output

I/O from the terminal

- printf, scanf
- getchar, putchar

I/O from files

- File pointers: FILE *
- fopen, fclose
- fprintf, fscanf
- fgetc, fputc
- fgets, fputs

Input/Output

I/O from the terminal

- printf, scanf
- getchar, putchar

I/O from files

- File pointers: FILE *
- fopen, fclose
- fprintf, fscanf
- fgetc, fputc
- fgets, fputs

- Practice reading man pages.
e.g. \$ man fopen
- **Do not use gets()!**

Format specifiers

Specifying the format of a variable in the formatted Input/Output (`printf()` / `scanf()`) statement.

| Format specifier | Purpose |
|--------------------------|--|
| <code>%c</code> | character I/O |
| <code>%s</code> | string/series of characters I/O |
| <code>%d</code> | signed integer I/O (assumes base 10) |
| <code>%i</code> | signed integer I/O (auto detects base) |
| <code>%u</code> | unsigned integer I/O |
| <code>%o</code> | unsigned octal integer I/O |
| <code>%x/%X</code> | unsigned hexadecimal integer I/O |
| <code>%e/%E/%f/%g</code> | double/floating point I/O |
| <code>%lf</code> | long double I/O |

More on scanf ()

We can take input strings with delimiters (e.g., spaces or commas) in between as follows.

```
char str[20];  
scanf("%[^\\t\\n]s", str);
```


Command line arguments

The `main()` function can receive arguments directly from the command line as follows.

```
#include<stdio.h>
int main(int argument_count, char *argument_value[]){
    int i;
    for(i=0; i<argument_count; i++){
        printf("%s\n",argument_value[i]);
    }
    return 0;
}
```

Note: `argument_count` keeps the number of arguments passed from the command line and `argument_value[]` stores the arguments as array of strings. Both these variable names are user-defined.

Command line arguments

Execution:

```
./a.out DFS is fun
```

Output:

```
./a.out  
DFS  
is  
fun
```

Problems – Day 2

- The abundancy of a natural number n is defined as the rational number $\frac{\sigma(n)}{n}$, the ratio between the sum of divisors of the number and the number itself. A number n is defined as friendly if it shares abundancy with one or more other numbers. This means there might exist a pair of numbers i and j such that $i \neq j$ but $\frac{\sigma(i)}{i} = \frac{\sigma(j)}{j}$. For example, 6 and 28 are friendly with each other because $\frac{\sigma(6)}{6} = \frac{\sigma(28)}{28} = 2$. Write a program to verify whether a pair of integers given as user input are friendly or not.
- Given a positive integer n as user input, find out the number of trailing zeros in $n!$.

Note: This can be done with $\log_5 n$ number of divisions.

Problems – Day 2

- 3 Suppose m and n are (signed) integers and x and y are floating variables. Write logical conditions that evaluate to TRUE if and only if:
- $x + y$ is an integer.
 - m lies strictly between x and y .
 - m equals the integer part of x .
 - x is positive with integer part at least 3 and with fractional part less than 0.3.
 - m and n have the same parity (i.e., are both odd or both even).
 - m is a perfect square.
- 4 Write a program to print the following pattern using generic controls over print. Let the line number be user input.

```
*
* *
*  *
*   *
```

Problems – Day 2

- 5 Consider an n -digit number. Square it and add the right n digits to the left n or $n - 1$ digits. If the resultant sum is same as the original number, then it is called a Kaprekar number. E.g., 45 is a Kaprekar number. Write a program to verify whether a given number is Kaprekar or not.
- 6 Let m and n be 32-bit unsigned integers. Use bitwise operations to assign to m the following functions of n :
- 1 if n is odd, 0 if n is even.
 - 1 if n is divisible by 8, 0 otherwise.
 - 2^n (Assume that $n \leq 31$).
 - n rotated by k positions to the left for some integer $k \geq 0$.
 - n rotated by k positions to the right for some integer $k \geq 0$.