Basics of C

Computing Lab

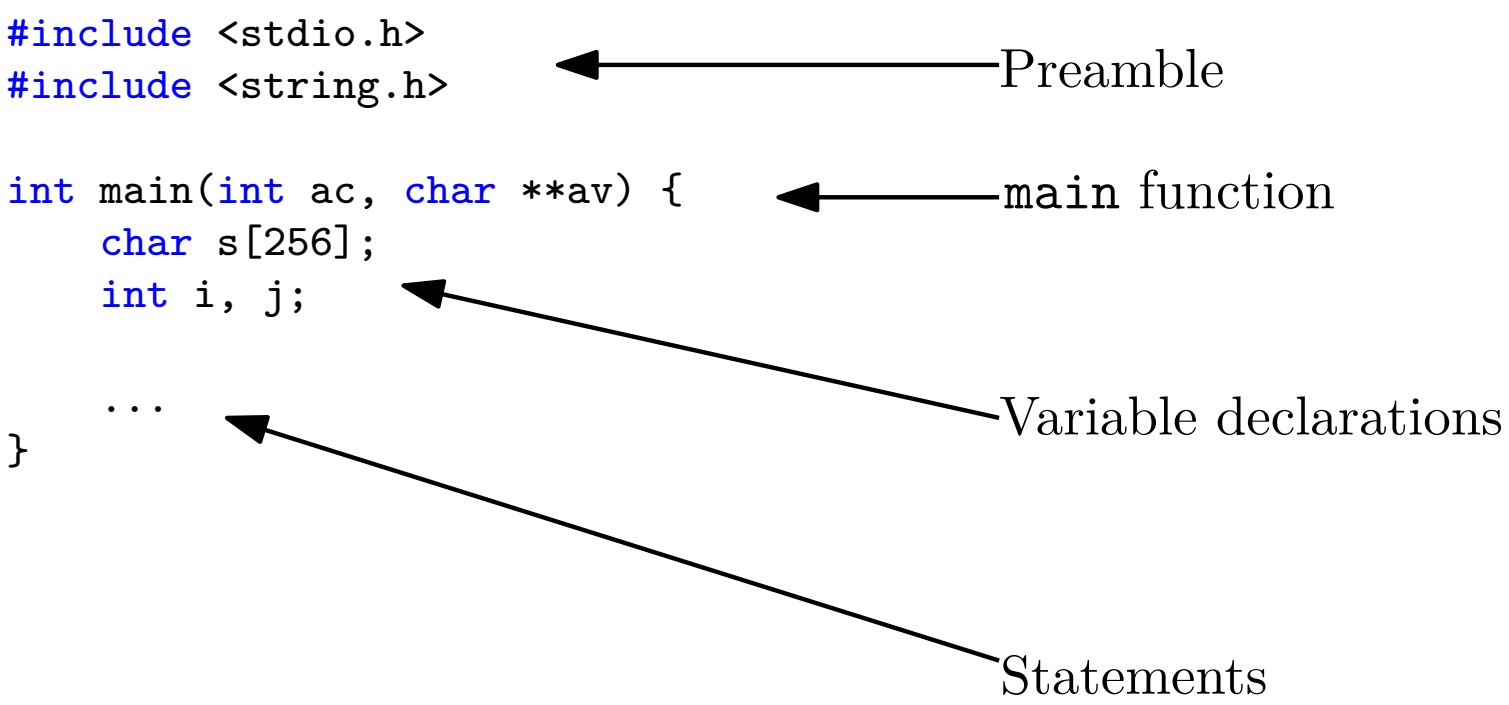
https://www.isical.ac.in/~dfslab

Indian Statistical Institute

Books

- The C Programming Language — Kernighan and Ritchie
- Programming with C — Byron Gottfried (Schaums' Outline series)
- The Practice of Programming — Kernighan and Pike

What does a program look like?



Terminology

- *Memory* = space for calculations, rough work, etc.
- *Variables* = names given to memory locations for convenience
- *Instructions / statements* = each step in the procedure

Example program: blood groups

Algorithm

1. Let a, b, ab, o represent the number of students with blood group A, B, AB and O.
2. Initially, set a, b, ab, o to zero.
3. Consider each student in turn.
4. Let the current student be t. Find out t's blood group.
5. Add 1 to the corresponding count (a, b, ab or o).
6. Report the numbers (a, b, ab and o) when done.

Example 1: blood groups (contd.)

1. Let a, b, ab, o represent the number of students with blood group A, B, AB and O.
2. Initially, set a, b, ab, o to zero.
3. Consider each student in turn.
4. Let the current student be t. Find out t's blood group.
5. Add 1 to the corresponding count (a, b, ab or o).
6. Report the numbers (a, b, ab and o) when done.

Example 1: blood groups (contd.)

```c
#include <stdio.h>

int main(void)
{
    /* declarations */
    int c;
    unsigned int a, b, ab, o;
    unsigned int n, i;

    a = 0; b = 0; ab = 0; o = 0;
    printf("Enter the number of students: ");
    scanf("%d", &n); getchar();
```

Example 1: blood groups (contd.)

```
for (i = 1; i <= n; i++) {
    printf("Enter the blood group of mtc15%02d: ", i);
    c = getchar(); getchar();
    if (c == 'A')
        a = a + 1;
    else if (c == 'B')
        b = b + 1;
    else if (c == 'a')
        ab = ab + 1;
    else if (c == 'O')
        o = o + 1;
    else
        printf("Invalid blood group %c, skipping\n", c);
}
```

Example 1: blood groups (contd.)

```
    printf("Number of students with blood group A: %d\n", a);
    printf("Number of students with blood group B: %d\n", b);
    printf("Number of students with blood group AB: %d\n", ab);
    printf("Number of students with blood group O: %d\n", o);

    return 0;
}
```

More examples (from the Screening Test) I

1. Consider a reservoir, fitted with a set of taps and drains. Write a program that takes as input details about the taps and drains attached to the reservoir, and prints the correct output from among the following options.
2. Consider a point that starts from the origin, and moves East, West, North or South for M steps. Write a program that takes the directions of the M steps, and computes the final distance of the point from the origin. (Assume that in each step, the point travels unit distance.)

   Input format: A sequence of letters from the set $\{E, W, N, S\}$ denoting the direction of movement of the point at each step. The length of the sequence will **NOT** be given to you in advance. The first character not belonging to the set will mark the end of the input.

   Output format: Your program should print the distance from the final position of the point to the origin, as a floating point number.

More examples (from the Screening Test) II

3. A sequence $\{s_1, s_2, \ldots, s_m\}$ of $m > 1$ strings is said to satisfy property $\mathcal{P}$ if
   - all the strings $s_1, s_2, \ldots, s_m$ are of the same length; and
   - for any i $(1 \leq i \leq m-1)$, the strings s_i and s_{i+1} differ in *at most* two positions.

   Write a program that takes a sequence of strings as input, and determines whether the strings satisfy property $\mathcal{P}$ or not.

Agenda

- Variables, built-in types and operators
- Conditional statements (`if`): executing statements based on whether some condition holds or does not hold
- Loops (`for`, `while`, `do`): executing statements repeatedly
- Input and output

Memory, variables

- Unit of storage = 1 byte (8 bits)
- Storage units are consecutively numbered
- Number assigned to storage = *location* / *address*
- Variable $\equiv$ name assigned to a storage location
- Variables must be **defined** and **initialised** before use

Examples:

```
char c;
int count_a, count_b;
```

LATER: difference between definition vs. **declaration**

Variable names

- REQUIRED
  - must start with a letter or underscore (_)
  - can contain only letters, underscores, digits
  - cannot match *reserved* words (`main`, `for`, `while`, ...)
  - case-sensitive
- RECOMMENDED
  - use "meaningful" names (i.e., not just `a`, `a1`, `a2`, `b`, `c`, `aaaa`, ...)
  - use `under_scores` or `CamelCase` for long names

Types

- **ALL** data stored in memory as a sequence of 0s and 1s
- Variable's type determines how a sequence of 0s and 1s is interpreted

Example:

byte

0	1	0	0	0	0	0	1

- **Integer value:** 65
- **Character representation: 'A'**
- For arithmetic operations: interpreted as integer
  `x = x + 65` and `x = x + 'A'` mean the same thing
  `x = x - 48` and `x = x - '0'` mean the same thing
- For printing:
  - as integer (`printf("%d\n", x)`): `65` is printed
  - as character (`printf("%c\n", x)` or `putchar(x)`): `A` is printed

Built-in types: integer data types

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

** in bits (typical)

- Use `sizeof` if you need to know the actual size, e.g., `sizeof(a)`

Signed and unsigned

- **Unsigned types:** if a variable of unsigned type occupies k bits, its value can be between 0 and $2^k - 1$.
- **Signed types:**
  - Bit sequences stored are the same as for unsigned types (i.e., $B = b_{k-1}b_{k-2}\ldots b_1 b_0$).
  - **BUT** they are interpreted differently.
    - if $b_{k-1} = 0$, B is interpreted as for unsigned types;
    - if $b_{k-1} = 1$, B is interpreted as a *negative* number in **two's complement representation**.
  - Range of values: -2^{k-1} to $+(2^{k-1} - 1)$

Two's complement representation

x is a variable of integer type, stored in k bits.

- If $0 \leq x \leq 2^{k-1} - 1$, x is represented as usual in binary.
- If $x < 0$, it is represented in (k-bit) two's complement form by the number $2^k - |x|$ (in binary).
  Examples:
  - `char x = -1;` `x` is represented by $2^8 - 1 = 255 = 1111\ 1111$.
  - `char x = -128;` `x` is represented by $2^8 - 2^7 = 2^7 = 1000\ 0000$.
- Thumbrule to compute the k-bit two's complement representation of $x < 0$:
  1. Let B denote the k bit representation of $|x|$.
  2. Flip each bit of B to get B'.
  3. Add 1 to B'.

This is called the one's complement of B.

(Why does this work?)

Built-in types: “real” (floating point) numbers

Type	Size**
float	32
double	64
long double	128

- Use apostrophe ’ as thousand-separator, if required.
- See `float.h` for limits and other gory details.
  (use `locate` if required)
- At times behaviour may be counter-intuitive (see `counterintuitive-floats.c`).

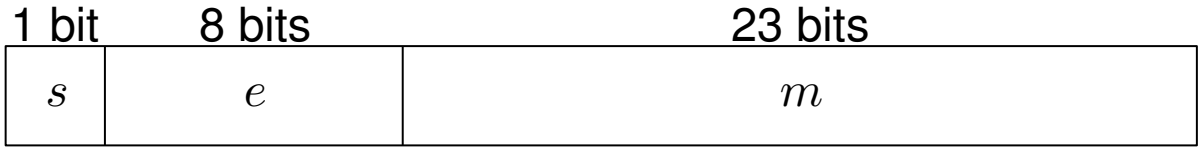
Examples:

Decimal notation	Exponential / scientific notation
`1.23456`	`3.45e67`
`1.`	`+3.45e67`
`.1`	`-3.45e-67`
`-0.12345`	`.00345e-32`
`+.4560`	`1e-15`

`e` means ‘10 to the power’

Built-in types: “real” (floating point) numbers

IEEE 754 representation



$$\equiv (-1)^s \; 1.m \times 2^{127+e}$$

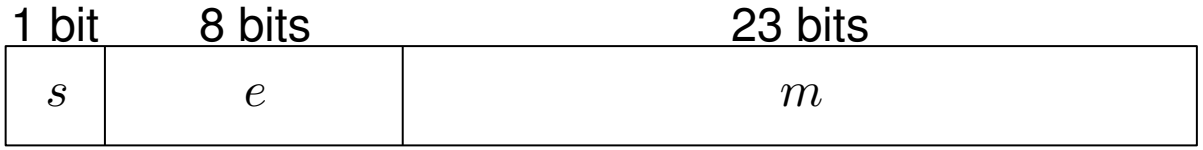
- 11 bits for double-precision (64 bit) floats
- $e \in \{-127, \ldots .128\}$

- 52 bits for double-precision (64 bit) floats
- implicit leading one is never stored

NOTE: The original exponent e plus a constant bias (127 for 32-bit rep.) is actually stored in the *exponent* field.

Built-in types: “real” (floating point) numbers

IEEE 754 representation



$$\equiv (-1)^s \; 1.m \times 2^{127+e}$$

- 11 bits for double-precision (64 bit) floats
- $e \in \{-127, \ldots .128\}$

- 52 bits for double-precision (64 bit) floats
- implicit leading one is never stored

NOTE: The original exponent e plus a constant bias (127 for 32-bit rep.) is actually stored in the *exponent* field.

Special numbers (SRC: HARRIS AND HARRIS, 2ND ED.)

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
$-\infty$	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

Operators

Arithmetic operators: `+ - * / %`

Assignment operators: `= += -= *= /= %=`

Increment decrement operator: `++ --`

Relational operators: `== != < <= >= >`

Boolean operators: `&& || !`

See

https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B#Operator_precedence

for a complete list, along with precedence and associativity.

Boolean values and expressions

- Any non-zero value is **TRUE**; zero is **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

- **Lazy evaluation:** Boolean expressions are evaluated from left to right; evaluation stops as soon as the truth value of the expression is determined.

  Examples:

  - When `i == N`, `A[i]` (i.e., `A[N]`) is not checked.

Strings

- Recall: `char` $\equiv$ 1 byte $\equiv$ single character **OR** small integer
- *String* $\equiv$ sequence of single characters in successive locations
- Must be terminated by *null character* $\equiv$ `0` **OR** `'\0'`

Exercise:

- difference between `2`, `'2'` and `"2"`
- difference between `a`, `'a'` and `"a"`

Conditionals: if, if-else

```
if (condition) {
   statements
}
```

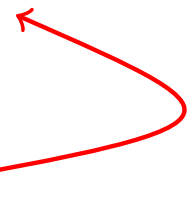
```
if (condition) {
   statements
}
else {
   statements
}
```

```
if (condition) {
   statements
}
else if (condition) {
   statements
}
...
else {
   statements
}
```

No braces { } required for single statement, but remember the semi-colon;

- Each `else` is paired with the closest preceding `else`-less `if`

```
if (condition1)
   if (condition2) {
      ...
   }
 else { ... }
```



Conditionals: switch

```
switch (E) {
    case value1 :
       statement;
       break;
    case val2 :
       statement;
       break;
    ...
    case valn :
       statement;
       break;
    default:
       statement;
 }
```

Loops

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

```
do {
     statement;
} while (condition);
```

```
for ( initialisation ; condition ; update operation ) {
    statement;
}
```

Loops

- **break:** immediately jump to the next statement after the loop.
- **continue:** for `for` loops, do the update operation; continue with the next iteration of the loop.

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

`i` is incremented after `continue`.

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

`i` is **NOT** incremented after `continue`.

Some useful library functions

- **Mathematical functions:** `#include <math.h>`
- **Character types:** `#include <ctype.h>`
- **String functions:** `#include <string.h>`
- **Miscellaneous functions:** `#include <stdlib.h>`

Look up the man pages!

Practice problems – I

1. Write a program to determine the roots of a quadratic equation $ax^2 + bx + c = 0$. Your program should ask for the values of a, b and c, and print the roots (real or complex).
2. Read a sequence of positive integers $a_0, a_1, a_2, \ldots$ (the length of the sequence will not be known a priori) and determine $\max_i \sum_{j=0}^{4} a_{i+j}$.

   Note that you do not need to store the complete sequence in order to compute the required quantity.
3. Problem 1 from

   `https://www.isical.ac.in/~mtcs-mentor-comm/exams/ptest2022.pdf`.

Supplementary reading

1. Review lectures 1–4 from `https://cse.iitkgp.ac.in/~pallab/course/2022/spring%202022/pds%20theory%202022/index.html`
2. Review the list of reserved words in C (Kernighan & Ritchie, Appendix A.2.4).
3. Review the list of escape sequences in C (`'\n'`, `'\\'`, ...) (Kernighan & Ritchie, Section 2.3).

Acknowledgements

- `http://cse.iitkgp.ac.in/~pds/notes/`
  (please see the above page for many more practice problems)