

C Language

• for extern, 'declare' keyword used but optional.

• Constants \rightarrow Integer
Real
Character

• keywords \rightarrow carry special meaning for system compiler
32 keywords there

• Input/output statements
 \rightarrow formatted
 \rightarrow all data types valid
scanf("format-specifier", &arg1, ..., &argn)

printf("format-specifier", arg1, arg2, ...)

\rightarrow unformatted
 \rightarrow works only char data type.

Input

- 1) getch() \rightarrow alphanumeric char
- 2) getchar()
- 3) gets()

Output

- 1) putchar()
- 2) putchar()
- 3) puts()

Expression

\rightarrow a collection of operators and operands that represent a specific value.

- 1) Type Declaration Expression
- 2) Arithmetic Expression
- 3) Infix
- 4) Prefix
- 5) Postfix

```
#include <stdio.h>
int main() {
    printf("Hello World!");
    return 0;
}
```

Comments

- \rightarrow // \rightarrow Single line comment
- /* ... */ \rightarrow Multiline comment

Variables

\rightarrow named memory location to store some data

unsigned numbers

\rightarrow values ≥ 0

signed numbers

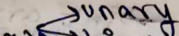
\rightarrow values ≥ 0
and
values < 0

Data types.

- 1) char/signed 1 byte %c
char
- 2) unsigned 1 byte %u
char
- 3) int 2 bytes %d
- 4) unsigned 2 bytes %u
int
- 5) short int 2 bytes
- 6) unsigned 2 bytes
short int
- 7) long int 4 bytes %ld
- 8) unsigned 4 bytes %lu
long int
- 9) float 4 bytes %f
- 10) double 8 bytes %lf
- 11) long double 10 bytes %L

Storage classes.

class	storage	default value	scope	lifetime
1) auto	memory	garbage value	within the block of code where defined	bill starts/remaining within block
2) register	cpu register	zero	local to function	bill end of block of code
3) static	memory	zero	local to code block	value persists between diff. calls
4) extern	memory	zero	global	bill end of pgm execution

- Operators in C
 - ↳ Arithmetic (+, -, *, /, %)
 - ↳ relational (<, <=, >, >=, ==, !=)
 - ↳ All are binary operators
- 
- ```

graph LR
 A(()) --> B[unary]
 A --> C[binary]
 A --> D[ternary]
 A --> E[increment]
 A --> F[decrement]

```

→ logical (AND, OR) → binary  
(0, 1, !0) → ternary

```

graph LR
 A[bitwise] --> B[bitwise AND (&)]
 A --> C[bitwise OR (|)]
 A --> D[exclusive OR (^)]
 A --> E[one's complement (~)]
 A --> F[left shift (<<)]
 A --> G[right shift (>>)]

```

→ ternary →  
variable = (expression) ? true-value :  
false-value

- comma
  - ↳ separates variables
- lowest precedence
- returns value of rightmost operand when multiple comma operators used

```
#include <stdio.h>
int main() {
 int i, j;
 i = (j=10, j+20);
 return 0;
}
```

→ arrow  
↳ used to access structure members when we use pointer variable to access it.

- ⇒ sizeof → returns size of operand in bytes
- can calculate size of datatype and variable
- returns size in integer format

- operator precedence, associativity

- 1)  $()$ ,  $[]$  → L to R
- 2)  $++$ ,  $--$ ,  $-$ ,  $!$ ,  $~$ ,  $&$  → R to L  
size of R to L
- 3)  $*$ ,  $/$ ,  $%$  → L to R
- 4)  $+$ ,  $-$  → L to R
- 5)  $<<$ ,  $>>$  → L to R
- 6)  $<$ ,  $<=$ ,  $>$ ,  $>=$  → L to R
- 7)  $=$ ,  $!=$  → L to R
- 8)  $&$  → L to R
- 9)  $>$  → L to R
- 10)  $!$  → L to R

- Conditional statements
  - if (condition) { }
  - if (condition) { }
  - else { }
  - if (condition) { }
  - else if (condition) { }
  - else { }
  - if (condition) { }
  - if (condition) { }
  - { }

• switch case. int/char

switch (variable/ expression) {

case value-1: ↖ values must be distinct.

break;

case value-2:

break;

⋮

case value-n:

break; → mandatory

default: → optional

}

- goto statement.
  - ↳ goto label;
  - statements;
  - label
  - statements

- loop.
  - ↳ for-loop (controlled)

```
for (init n; cond n; upd n) {
 }
}
```



2) while loop (entry controlled)  
no. of iterations not known.  
syntax:  

```

int i;
while (condition) {
 // statement
 update i;
}

```

3) do-while loop (exit controlled)  
syntax:  

```

do {

```

```

 while (condition);

```

4) break

```

for (i = 1 → 5) {
 if (i == 3) break;
 print("%d", i);
}

```

5) continue

```

for (i = 1 → 5) {
 if (i == 3) continue;
 print("%d", i);
}

```

6) Nested loops

↳ loop in other loop.

• Arrays and Pointers

↳ collection (ordered) of finite homogeneous data.

→ stored in contiguous locations.

→ all elements share common name but diff behavior

size calculation

bytes = size of (datatype) \* size of array

declaration

datatype arrayname [size / subscript]

```

→ int arr[5] = {1, 2, 3, 4, 5};
// int arr[5]
for (i = 0 → 5) {
 scanf("%d", &arr[i]);
}

```

int arr[2][3] = { {1, 2, 3}, {4, 5, 6} };  
OR

```

int arr[2][3];
for (i = 0 → 2) {
 for (j = 0 → 3) {
 scanf("%d", &arr[i][j]);
 }
}

```

OR  
int arr[7][8] = { {1, 2, 3}, {4, 5, 6}, ... }

int m[4][5] = { {1}, {2, 13}, {14, 15, 16}, {173} };  
// {1}, {2, 13}, {14, 15, 16}, {173}

|    |    |    |
|----|----|----|
| 11 | 0  | 0  |
| 12 | 13 | 0  |
| 14 | 15 | 16 |
| 17 | 0  | 0  |

int a[4][5] = {1, 2, 3, 4, 5, 6, 7}

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |

→ 3D array

datatype arrayname [size] [size] [size]

```

int arr[p][q][r]
for (i = 0 → p)
 for (j = 0 → q)
 for (k = 0 → r)
 scanf("%d", &arr[i][j][k]);

```

→ Pointer

↳ int n = 10;

int \* ptr = &n;

\* → value at  
& → address of



→ null ptr → constant with a value of zero defined in several std libraries

→ dangling ptr → change address of location to which ptr was initially pointing

→ generic → pointer which does not have any specific data type

void \* ptr; // date;

```
#include <stdio.h>
int main() {
 int a[2] = {1, 2};
 void * ptr = &a;
 ptr = ptr + sizeof(int);
 printf("%d", *(int*)ptr);
 return 0;
}
```

→ int arr[5] = {0, 1, 2, 3, 4};

int \* ptr = arr; \*ptr = arr[0]

- 1) a[i] → value
- 2) i[a] → value
- 3) a+i → address
- 4) \*(a+i) → value

multidimensional array element

a[i][j] → \*(a+i+j)

&a[i][j] → \*(a+i+j)

→ double ptr

```
int age = 22;
int * ptr = &age;
ptr++;
```

→ strings

↳ array of characters

char string-variable-name[size];

→ ways of initializing

- 1) gets(str)
- 2) fgets(str, 10, stdin) → cannot take a sentence
- 3) scanf("%s", str)
- 4) getchar(ch)
 

```
while(ch != '\0') {
 str[i] = ch;
 i++;
}
```

5) scanf("%s", str) → %c, &c

→ output methods

```
char str[10] = "Hello";
1) printf("%s", str)
2) puts(str)
3) int i;
 while(str[i] != '\0')
 putchar(str[i]);
 i++;
}
```

→ Note

char \* str = "Hello"

→ functions

- 1) atoi() → numeric string to integer  
atof(), atol(), itoa(), ltoa()  
in <stdlib.h>
- 2) int l = strlen("Hello");
- 3) strcat(char \* s1, char \* s2);
- 4) int c = strcmp(str1, str2);
- 5) int age = 23;
 

```
char str[100];
sprintf(str, "My age is %d", age);
puts(str);
```
- 6) char buffer[30] = "Fresh2Refresh 5";
 

```
char name[20];
int age;
scanf(buffer, "%s %d", name, &age);
```
- 7) char \* strstr(const char \* s1, const char \* s2)
- 8) char \* strchr(char \* string)
- 9) char \* strcpy(char \* st1, char \* st2)
- 10) char \* strtok(char \* str, const char \* delimiter)
 

```
if str[] = "Problems-solving-in-C"
char * token = strtok(str, "-")
while(token != NULL) {
 printf("%s\n", token);
 token = strtok(NULL, "-");
}
```



## • Function

return\_type function\_name (parameter list)  
 // body of function  
 {  
 }

if function def is after main fn, we write fn declaration in global declaration section.

if it is written above main fn, no need to write the fn declaration.

beac during lexical analysis, starts from top to bottom and from left to right

types of function calling

```
void swap(int x, int y) {
 int t;
 t = x;
 x = y;
 y = t;
}
```

```
void swap(int *x, int *y) {
 int t;
 t = *x;
 *x = *y;
 *y = t;
}
```

swap(hx, ly); // pass by value

function types (parameter list)

- no argument, no return
- argument return
- no arg, return
- arg, no return

## • Pass arrays to function

```
#include <stdio.h>
void fun(int arr[]) {
 int i;
 for (i = 0; i < 5; i++)
 arr[i] += 10;
}

void main() {
 int arr[5];
 for (i = 0; i < 5; i++)
 scanf("%d", &arr[i]);
 fun(arr);
}
```

• function ptr  
 return\_type fun\_ptr (arg) =  
 { function  
 #include <stdio.h>  
 void fun(int a) {  
 printf(a);  
 }

```
void main() {
 void (*fun_ptr)(int) = fun;
 (*fun_ptr)(15);
}
```

a fn ptr points to code not data, a fn ptr stores start of executable code.

```
#include <stdio.h>
void add(int a, int b) {
 printf("%d", a+b);
}
```

```
void sub(int a, int b) {
 printf("%d", a-b);
}
```

```
void mul(int a, int b) {
 printf("%d", a*b);
}
```

```
void main() {
 void (*fun_ptr)(int, int) =
 { add, sub, mul };
 int ch, a, b;
```

```
 scanf("%d %d %d", &a, &b, &ch);
 if (ch == 1)
```

```
 (*fun_ptr)(a, b);
 else
```

## • Structures

```
struct student {
 char name[100];
 int roll;
 float cgpa;
}
```

```
→ struct student {
 char name[100];
 int roll;
 float cgpa;
}
```

```
int main() {
 struct student s1, s2, s3;
 s1 = {"Harshit", 52, 9.16};
 s2 = {0};
 s3 = {0};
}
```

→ Array of structure.

```
struct student {
 char name[100];
 int roll;
 float cgpa;
};

int main() {
 struct student cse[100];
 for (i = 0 → 100) {
 scanf(
 "%s %d %f", &cse[i].name,
 &cse[i].roll,
 &cse[i].cgpa);
 }
}
```

→ Ptr to structures.

```
struct student s1;
struct student *ptr;
ptr = &s1;
```

→ structure containing ptr.

```
struct character {
 char c;
 char *cp;
};

void main() {
 struct character x;
 char ch = 'A';
 x.c = ch;
 x.cp = &ch;
}
```

→ structure within structure

```
struct date {
 int day, month, year;
};

typedef struct student {
 int rollno;
 char name[80];
 float percent;
 struct date da;
};

int main() {
 struct student x;
 scanf("%d %s %f %d %d %d", &x.rollno,
 &x.name, &x.percent,
 &x.da.day, &x.da.month,
 &x.da.year);
}
```

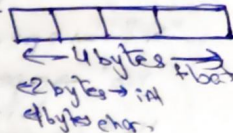
• Unions.

In union, members share memory with each other.

```
union unionName {
 datatype member1;
 datatype member2;
 ...
 datatype memberN;
};
```

eg union demo {

```
 char ch;
 int i;
 float f;
};
```



union demo x;

only one data member stored at a time.

```
#include <stdio.h>
```

```
union point {
```

```
 int x;
 int y;
};
```

```
void main() {
```

```
 union point p;
```

```
 p.x = 10;
```

```
 p.y = 20;
```

```
 printf("%d %d", p.x, p.y);
}
```

to get 10 20, 20 20  
print separately

• Enums.

```
enum enum_name {
```

```
 value1, value2, ..., valueN;
```

```
}
```

↳ by default

these are assigned value

• Dynamic memory allocation

1) malloc() →

```
ptr = (*data-type) malloc(
 locations * sizeof(data-type))
```

2) calloc() →

```
ptr = (*data-type) calloc(locations,
 sizeof(data-type))
// memory location initialized with 0.
```

3) free(ptr)

4) realloc

```
ptr = realloc(ptr, new_size)
```



## • File handling

1) FILE \* fptr;

2) FILE \* fptr;

fptr = fopen("filename", mode);

3) fclose(fptr)

4) "r" → open to read

"rb" → open to read in binary

"w" → open to write

"wb" → open to write in binary

"a" → open to append

5) char ch;

fscanf(fptr, "%c", &ch);

6) char ch = 'A';

fprintf(fptr, "%c", ch);

7) fgetc(fptr)

fputc('A', fptr)

8) EOF → fgetc returns EOF to show file has ended