Fundamentals of Programming

Final Cheat Sheet
First Semester, 2019

Akara Supratak
Jidapa Kraisangka
Pilailuck Panphattarasap

Variables/Data Types/Constants

Integer Data Types: `int`

- `int` data type
- **Declaration**: `int var_name;`
  - Whole numbers and + or – signs
  - Values between -2,147,483,648 to 2,147,483,647

- *Example*
  - Valid integer constant:

    ```
    5    -10     +25    1000
    253  -26351  +36
    ```

  - Invalid integer constant:

    ```
    $255.62  2,523  3.  6,243,892
    1,492.89 +6.0
    ```

Integer Data Types: `char`

- **`char`** data type
- **Declaration**: `char var_name;`
  - Store individual character
  - Printable character: letters, digits, and special symbols

- *Example*
  - Letters: `'L'    'o'    'l'`
  - Digits: `'1'    '0'    '5'`
  - Special symbols: `'$'    '#'    ','`

ASCII and ANSI code

- Character encoding standards
  - ASCII: American Standard Code for Information Interchange (7 bits)
  - ANSI: American National Standards Institute (8 bits)

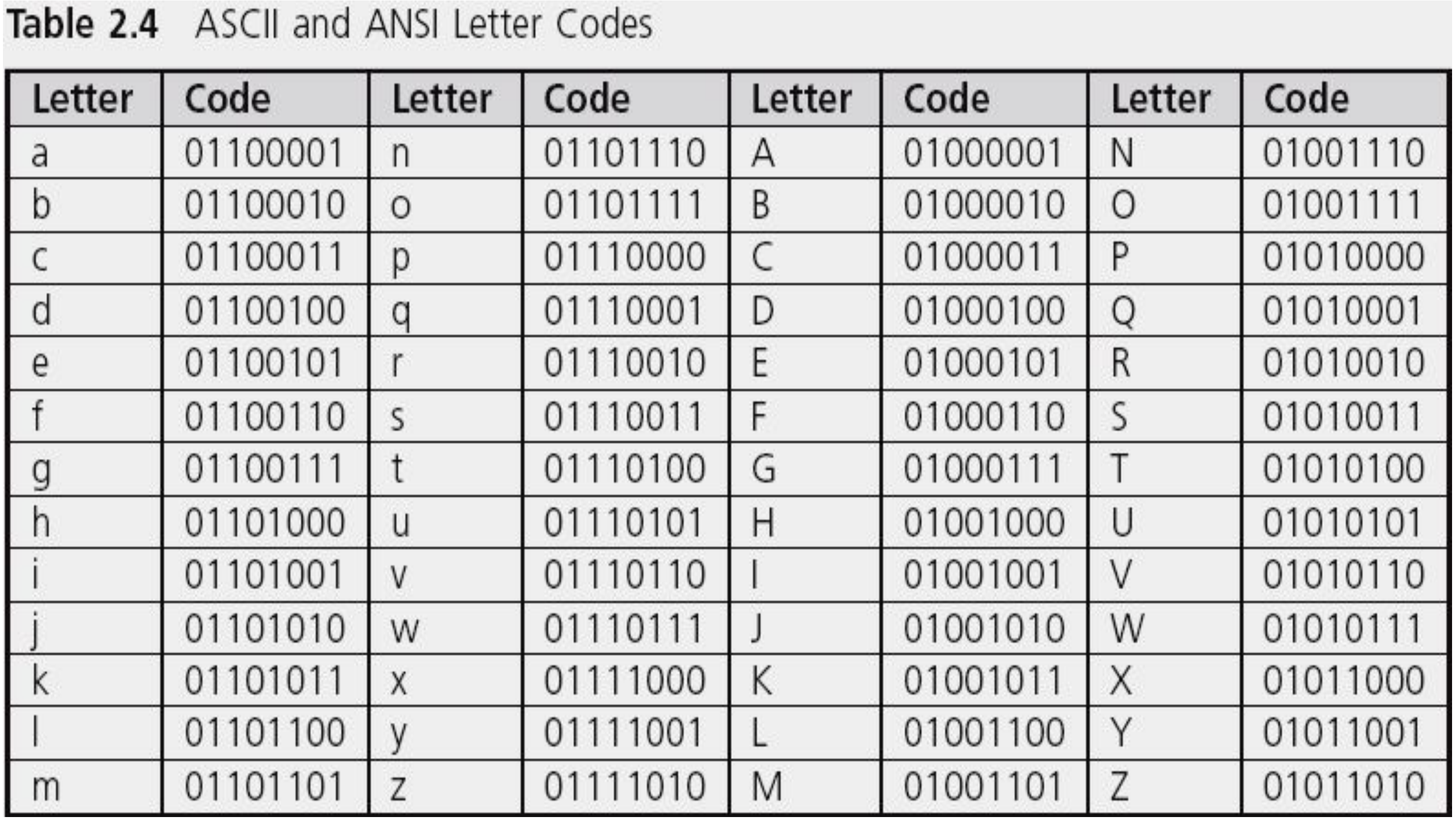
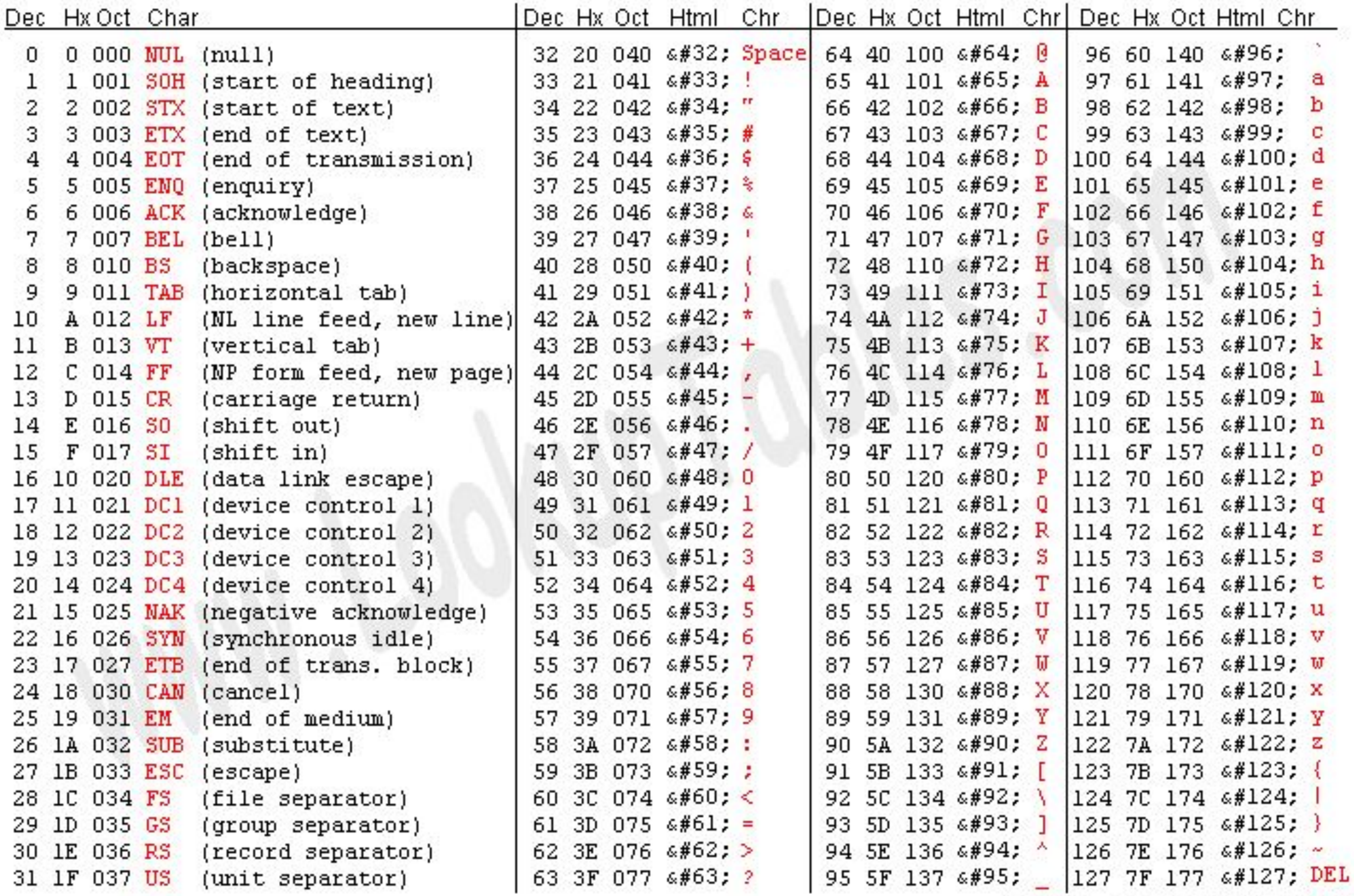


Table 2.4 ASCII and ANSI Letter Codes

Letter	Code	Letter	Code	Letter	Code	Letter	Code
a	01100001	n	01101110	A	01000001	N	01001110
b	01100010	o	01101111	B	01000010	O	01001111
c	01100011	p	01110000	C	01000011	P	01010000
d	01100100	q	01110001	D	01000100	Q	01010001
e	01100101	r	01110010	E	01000101	R	01010010
f	01100110	s	01110011	F	01000110	S	01010011
g	01100111	t	01110100	G	01000111	T	01010100
h	01101000	u	01110101	H	01001000	U	01010101
i	01101001	v	01110110	I	01001001	V	01010110
j	01101010	w	01110111	J	01001010	W	01010111
k	01101011	x	01111000	K	01001011	X	01011000
l	01101100	y	01111001	L	01001100	Y	01011001
m	01101101	z	01111010	M	01001101	Z	01011010

Decimal ASCII code value



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

The Escape Character

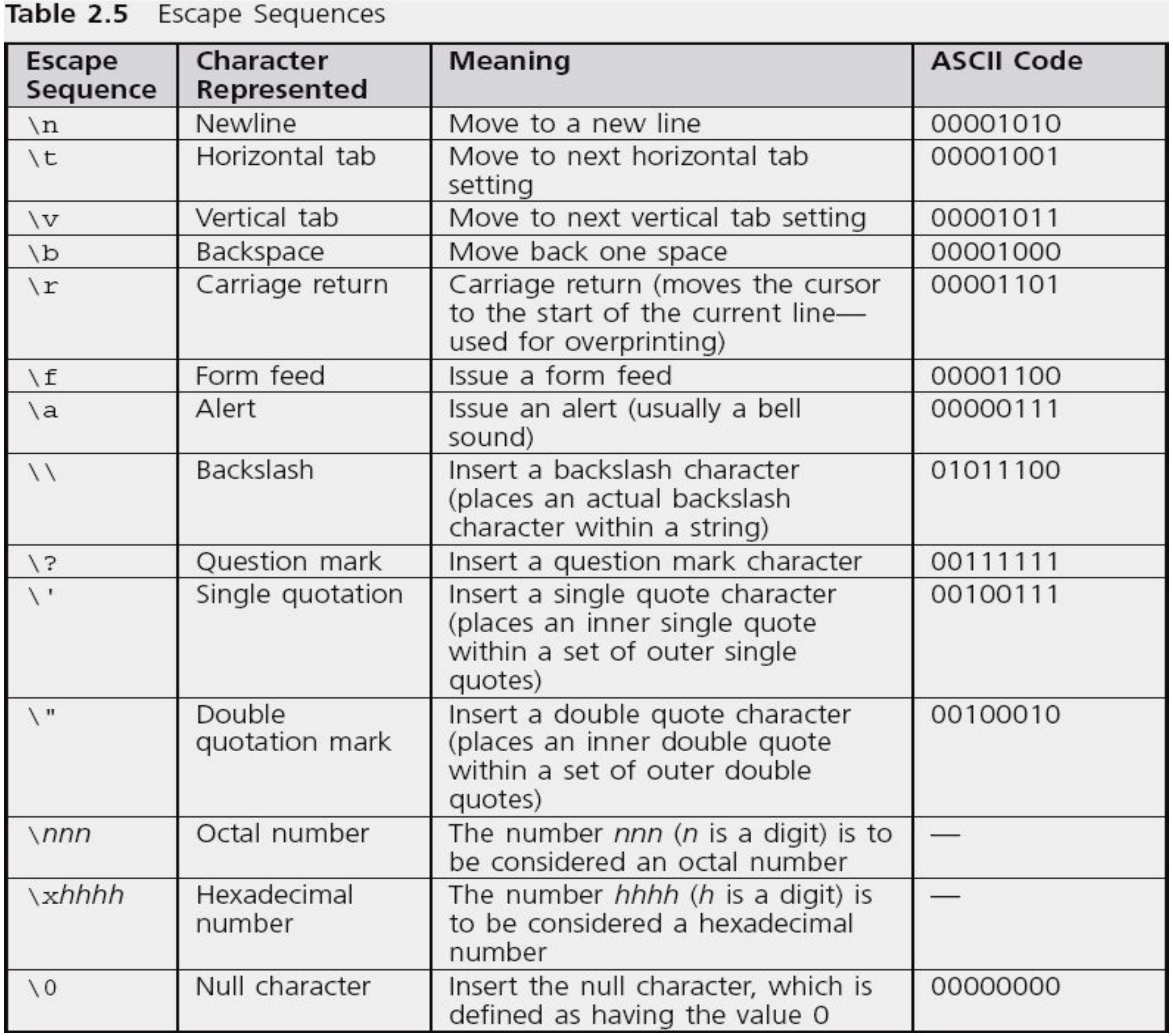


Table 2.5 Escape Sequences

Escape Sequence	Character Represented	Meaning	ASCII Code
\n	Newline	Move to a new line	00001010
\t	Horizontal tab	Move to next horizontal tab setting	00001001
\v	Vertical tab	Move to next vertical tab setting	00001011
\b	Backspace	Move back one space	00001000
\r	Carriage return	Carriage return (moves the cursor to the start of the current line—used for overprinting)	00001101
\f	Form feed	Issue a form feed	00001100
\a	Alert	Issue an alert (usually a bell sound)	00000111
\\	Backslash	Insert a backslash character (places an actual backslash character within a string)	01011100
\?	Question mark	Insert a question mark character	00111111
\'	Single quotation	Insert a single quote character (places an inner single quote within a set of outer single quotes)	00100111
\"	Double quotation mark	Insert a double quote character (places an inner double quote within a set of outer double quotes)	00100010
\nnn	Octal number	The number *nnn* (*n* is a digit) is to be considered an octal number	—
\x*hhhh*	Hexadecimal number	The number *hhhh* (*h* is a digit) is to be considered a hexadecimal number	—
\0	Null character	Insert the null character, which is defined as having the value 0	00000000

Floating-point Data Types

- Also called “real number”
- Can be number zero or any positive or negative number that contains a decimal point

- *Example*
    - Valid floating-point constant:

```
+10.6255   5.        -6.2    3251.92
0.0         0.33     -6.67   +2.
```

    - Invalid floating-point constant:

```
5,326.25    24    123    6,459    $10.29
```

Symbolic Constants

- `#define` can be used to define constant variables, i.e., you cannot change the value
- `#define CNAME value`
- Example

```
#define PI 3.14
#define G 9.81
#define DEBUG 0
```

Operators

1 Arithmetic Operations

Operation	Operator	Type	Operand	Result
Addition	+	Binary	Both are integers	Integer
			One operand is a floating-point number	Floating-point number
Subtraction	-	Binary	Both are integers	Integer
			One operand is a floating-point number	Floating-point number
Multiplication	*	Binary	Both are integers	Integer
			One operand is a floating-point number	Floating-point number
Division	/	Binary	Both are integers	Integer
			One operand is a floating-point number	Floating-point
Modulus	%	Binary	Both are integers	Integer
Negation	-	Unary	Integer or floating-point	Same as operand

Binary: Require two operands
Unary: Require one operand

2 Relational Operators

- Compare two operands to produce a Boolean result
- **True**: non-zero value (i.e. 1)
- **False**: 0

Operator	Meaning	Example

>	Greater than	`3 > 2;`
		`2 > 3;`

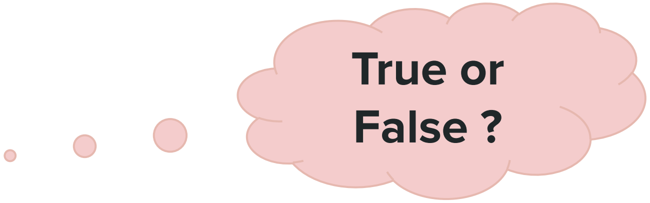
>=	Greater than or equal to	`3 >= 3;`
		`2.9 >= 3;`

<	Less than	`3 < 2;`
		`2 < 3;`

<=	Less than or equal to	`3 <= 3;`
		`3.1 <= 3;`

==	Equal to	`3 == 3;`
		`2 == 3;`

!=	Not equal to	`3 != 3;`
		`2 != 3;`



True or False ?

Note that

- "==" equality operator is different from the "=", **assignment operator**
- the "==" operator on float variables is tricky because of finite precision

3 Logical Operators

True or False ?

Operator	Meaning	Example

`&&`	AND	`(25/5 == 5) && (2+3 == 5);`
		`(3*2 == 6) && (2+3 == 6);`

`||`	OR	`(25/5 == 5) || (2+3 == 5);`
		`(3*2 == 6) || (2+3 == 6);`

`!`	NOT	`!(3*2 == 6);`
		`!(2+3 == 6);`

4 Increment, Decrement Operators

- **Postfix**: increment the value **after** using it.
  - `x++;` means `x = x+1;`
  - `x--;` means `x = x-1;`
  - `y = x++;` means `y = x;`
    `x = x+1;`
  - `y = x--;` means `y = x;`
    `x = x-1;`

```
int i;
i = 6;
printf("%d ", i++);
printf("%d ", i);
```

```
int i;
i = 6;
printf("%d ", i--);
printf("%d ", i);
```

- **Prefix**: increment the value **before** using it.
  - `++x;` means `x = x+1;`
  - `--x;` means `x = x-1;`
  - `y = ++x;` means `x = x+1;`
    `y = x;`
  - `y = --x;` means `x = x-1;`
    `y = x;`

```
int i;
i = 6;
printf("%d ", ++i);
printf("%d ", i);
```

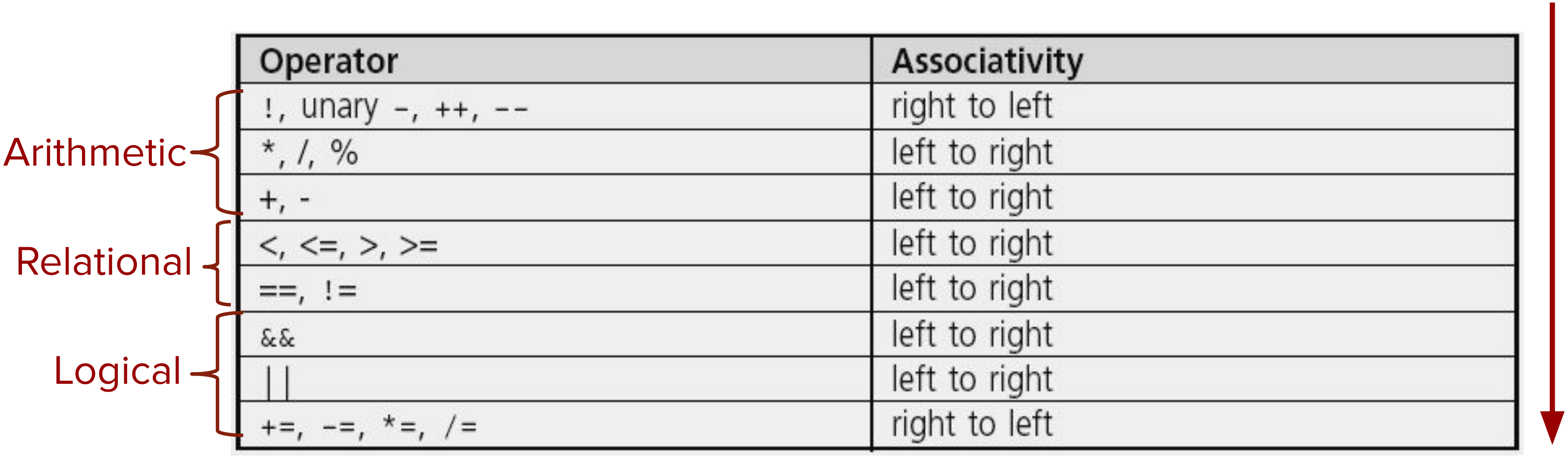
```
int i;
i = 6;
printf("%d ", --i);
printf("%d ", i);
```

5 Assignment Operators

- Assignment operation
  - `x = x+1;`
  - `x = x-1;`
  - `x = x*3;`
  - `x = x/3;`
  - `x = x%3;`
- Compact assignment operation
  - `x+=1;`
  - `x-=1;`
  - `x*=3;`
  - `x/=3;`
  - `x%=3;`

Operator Associativity

- The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.
- C operators listed from highest precedence to lowest precedence



Operator	Associativity
!, unary -, ++, --	right to left
*, /, %	left to right
+, -	left to right
<, <=, >, >=	left to right
==, !=	left to right
&&	left to right
||	left to right
+=, -=, *=, /=	right to left

Arithmetic
Relational
Logical

Input/Output

Interactive Input

- Receive input values from a user
- `scanf()` from `stdio.h`
- `scanf("control-string", list-of-params);`

Symbol	Format
`%d`	integer
`%f`	float
`%lf`	double
`%c`	character
`%s`	text
`%u`	unsigned integer

Formatted Output

- Print out values to the terminal
- `printf()` from `stdio.h`
- `printf("<String>");`
- `printf("<String> and/or <control-str>", list-of-params);`

Symbol	Format
`%d`	integer
`%f`	float
`%lf`	double
`%c`	character
`%s`	text
`%u`	unsigned integer

Formatted Output

- You can specify two thing in control string

  `"%<v1>.<v2>f"`

  - `<v1>` is the total display width (including the decimal point)
  - `<v2>` is the number of digits after the decimal points (precision)
- Example
  - Right-justified

    ```
    printf("%10.3fYo!\n", 25.67);
    ```

    ```
    ____25.670Yo!
    ```

  - Left-justified

    ```
    printf("%-10.3fYo!\n", 25.67);
    ```

    ```
    25.670____Yo!
    ```

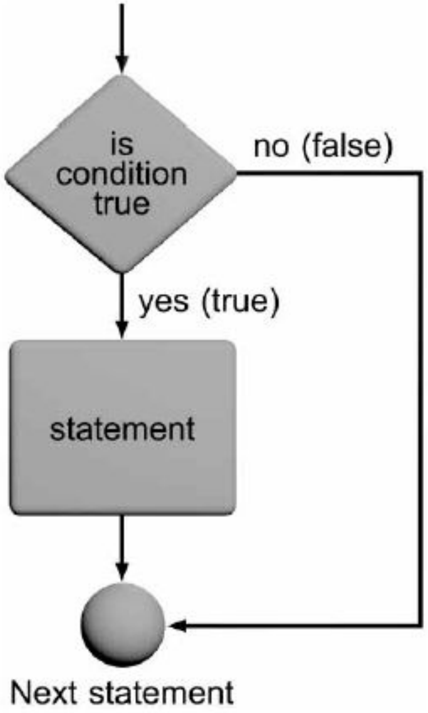
`if-else, switch` Statements

`if` statements

- The `statement` is only executed if the `expression` has a non-0 value (i.e., TRUE)

```
if (expression) {
     statement1;
     statement2;
     statement3;
}

if (expression)
     statement1;
```

```
int age = 78;
int discount = 0;
if (age > 60)
     discount = 20;
```

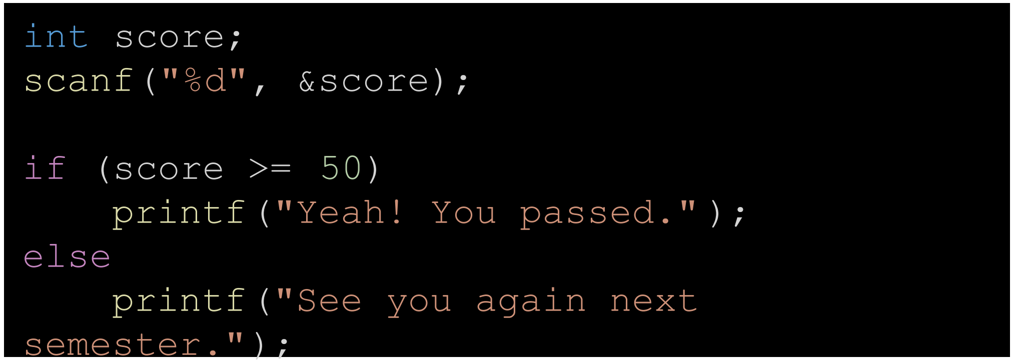


`if-else` statements

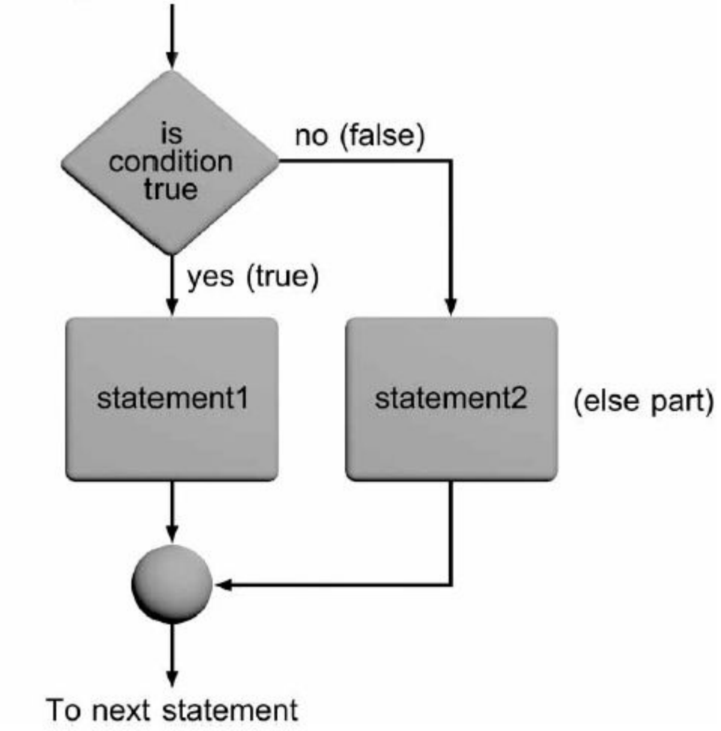
- The `statement` in `else` will be executed if the `expression` has a 0 value (i.e., FALSE)

```
if (expression) {
     statement1;
     statement2;
}
else {
     statement1;
     statement2;
}
```

```
if (expression)
     statement1;
else
     statement1;
```

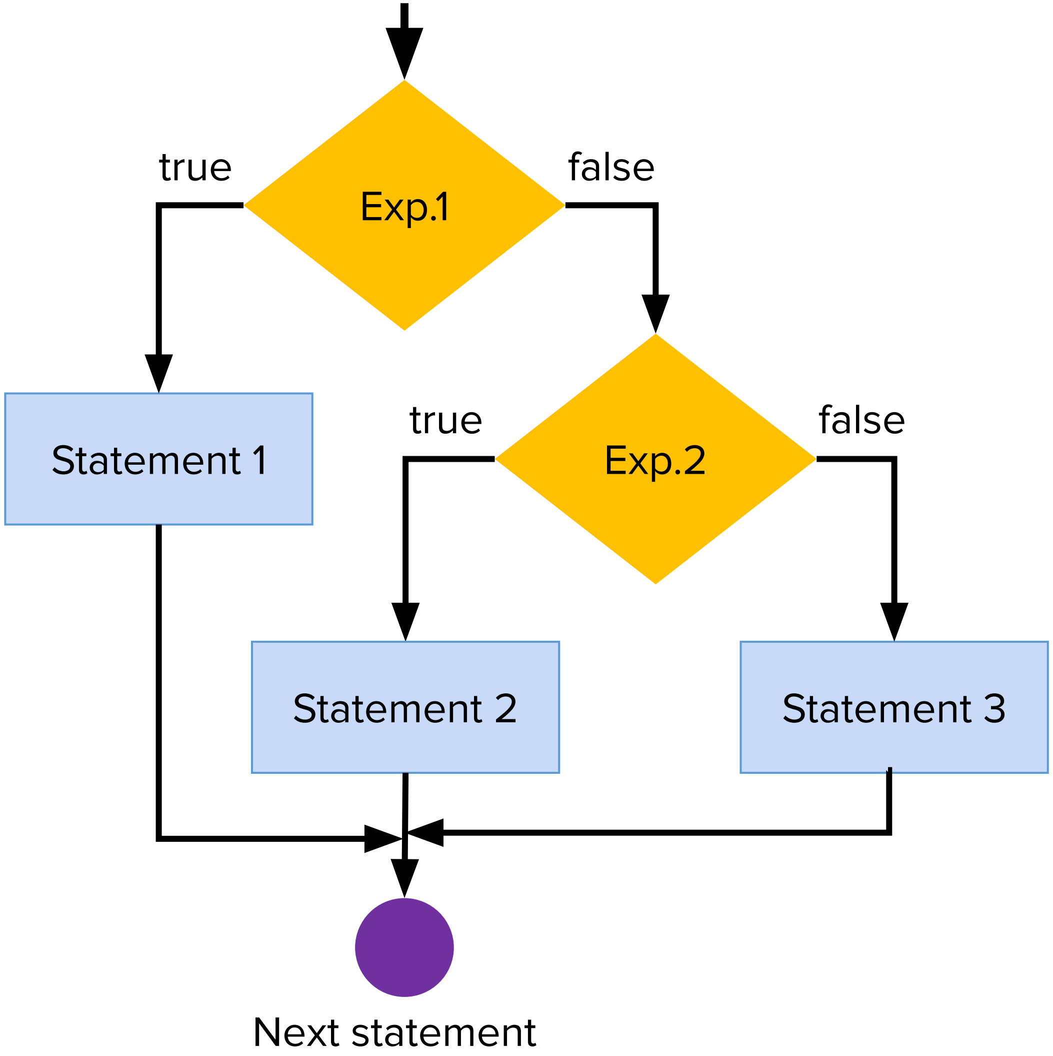


```
int score;
scanf("%d", &score);

if (score >= 50)
    printf("Yeah! You passed.");
else
    printf("See you again next
semester.");
```



if-else chain (nested)

```
if (expression1)
   statement1;
else if (expression2)
   statement2;
else
   statement3;

next_statement;
```



Ternary Operator

- A shorter version of the `if-else` statement
- Typically used to assign a value to a variable with condition

```
if (expression)
    true_statement1;
else
    false_statement1;
```

```
expression ? true_statement1 : false_statement1;
```

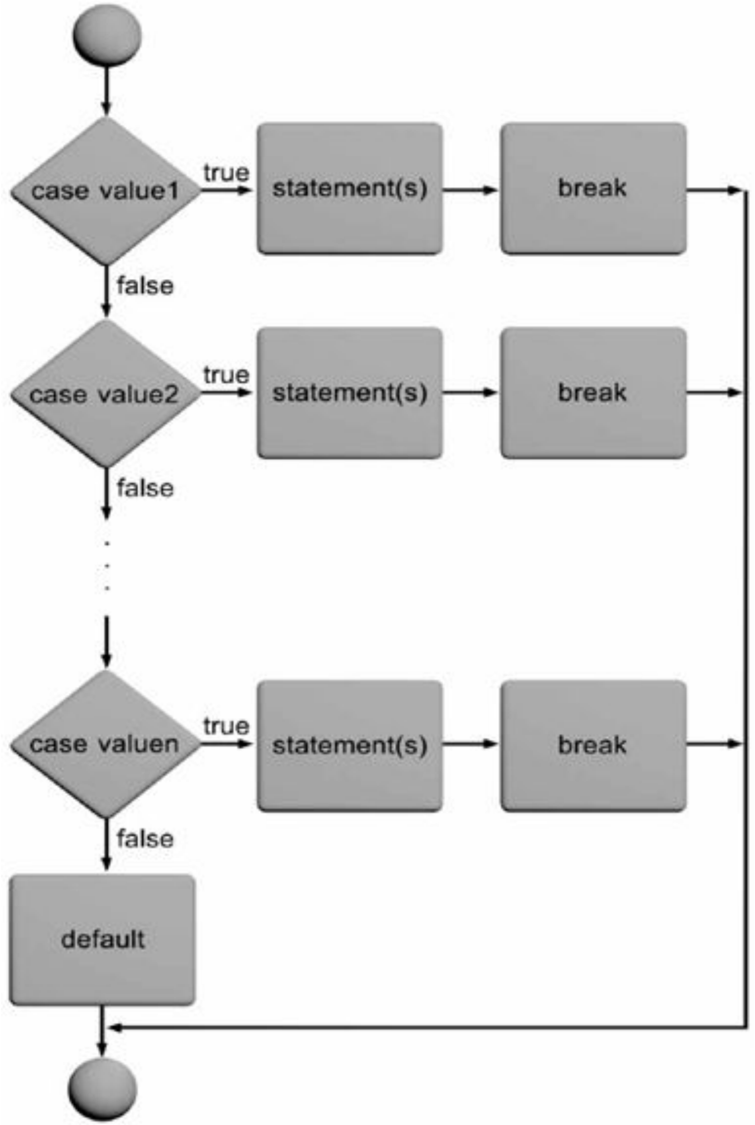
switch statement

- A specialized **selection** statement that can be used in place of an if-else statement.
- Evaluate an int_exp (**int** or **char**) : do the matched case
- If break statement is not used, all cases after the correct case is executed.

```
switch (int_exp){
    case value1:
        // code to be executed if
        // int_expr == value1
        statement1;
        statement2;
        ...
        break;
    case value2:
        // code to be executed if
        // int_expr == value2
        statement3;
        statement4;
        ...
        break;
    ...
    default:
        // code to be executed if
        // int_expr doesn't match any
        // no break; required
}
```

`switch` statement (cont.)

```
switch (int_exp){
     case value1:
          statement11;
          statement12;
          ...
          break;
     case value2:
          statement21;
          statement22;
          ...
          break;
     ...
     case valueN:
          statementN1;
          statementN2;
          ...
          break;
     default:
           statementD1;
           statementD2;
          ...
}
```

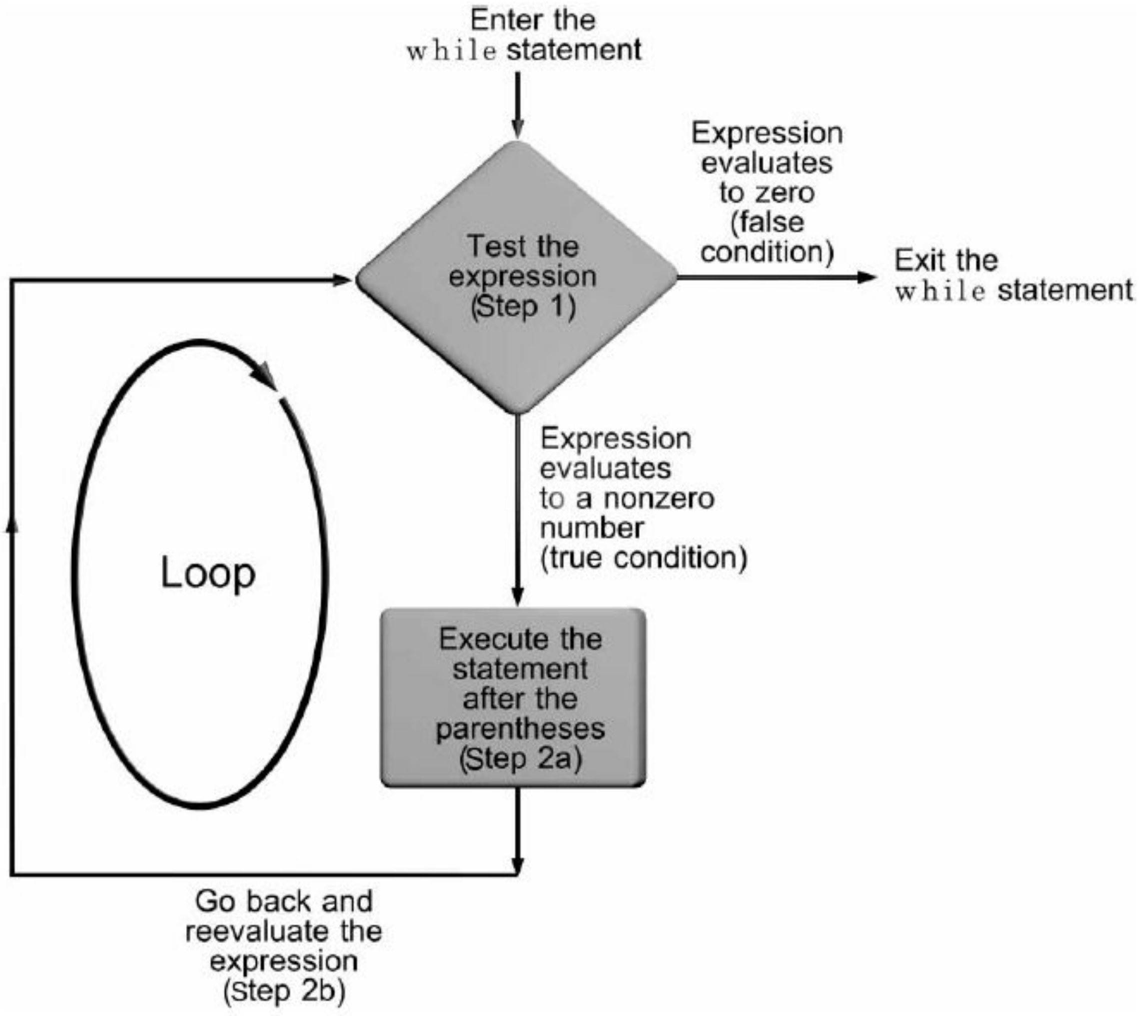


Loops

Loops/Repetitions

- Pre-test loops
  - `for`: know at compile time how many times this loop will execute.
  - `while`: you don't know how many times a loop will actually execute at runtime.
- Post-test loops
  - `do-while`: your loop should execute at least one time.
    - E.g., input validation need to receive an input before validation

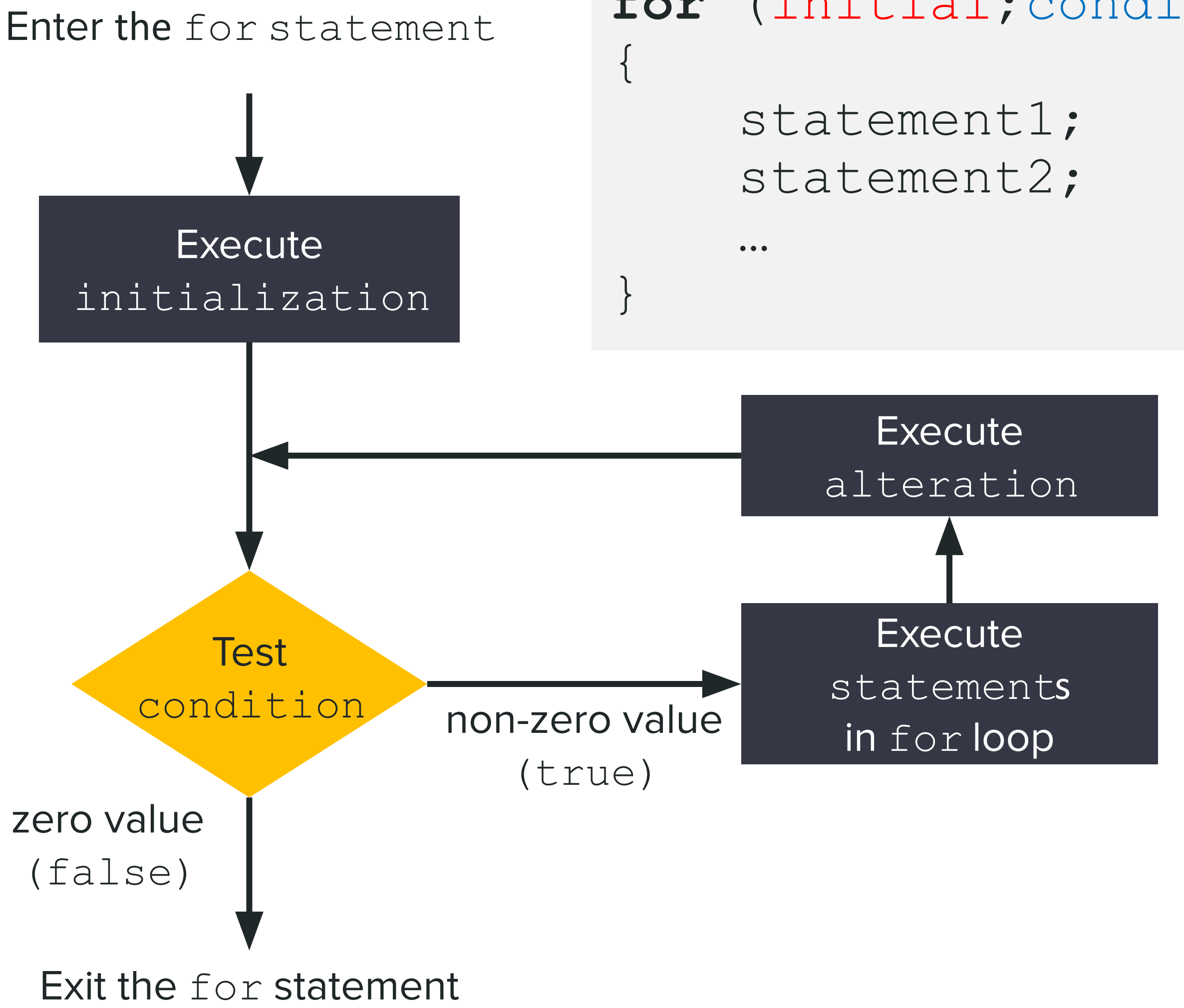
while statement



```
while (expression)
{
        statement1;
        statement2;
        …
}

statement;
…
```

The for Statement

```
for (initial;condition;alteration)
{
      statement1;
      statement2;
      …
}
```



Tips – Handling Control Variables

- Increasing order

Start from 1: i=1
End at 100: i<=100
Increased by 1: i++

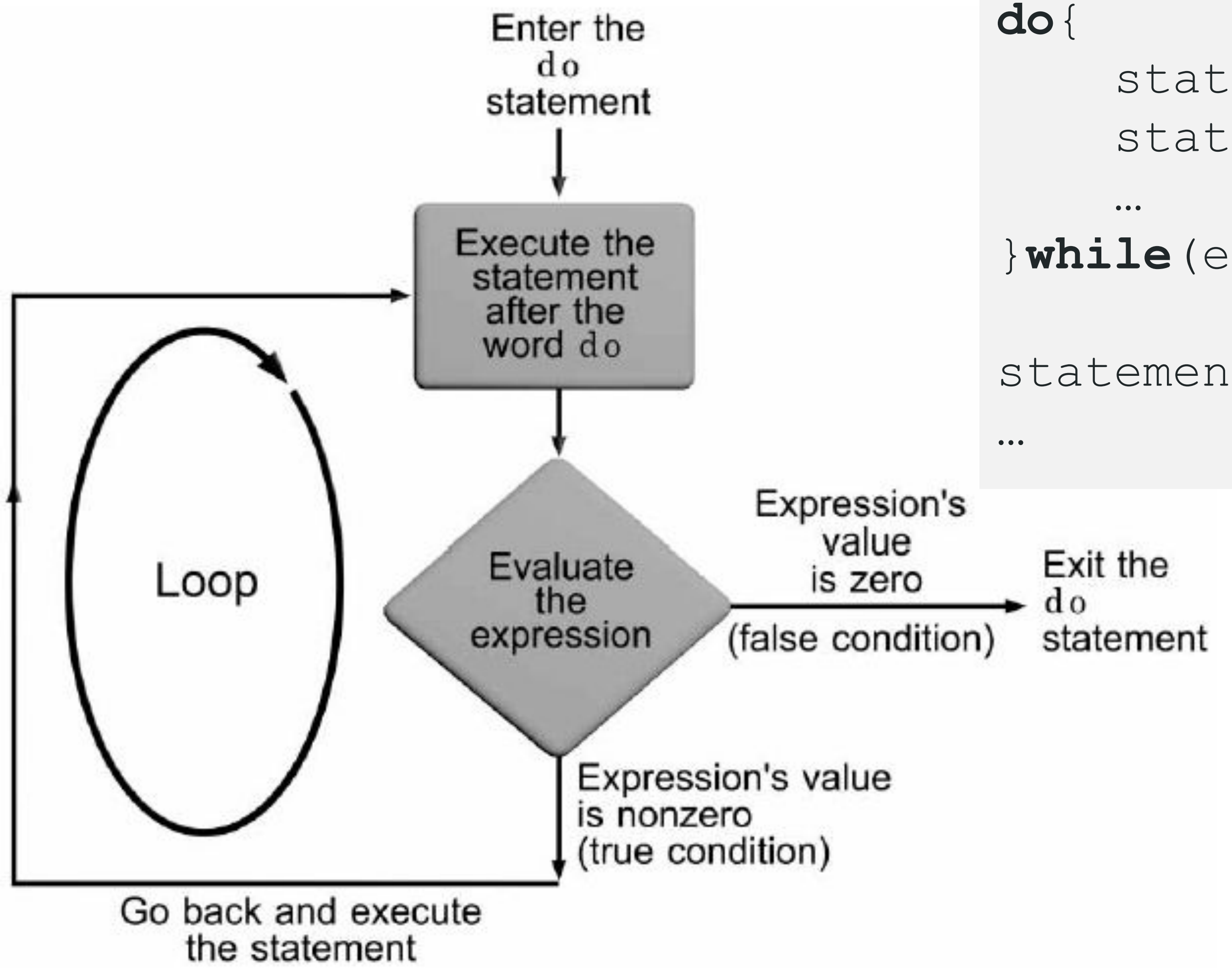
```
int i=1;
while (i <= 100) {
    printf("%d ", i);
    i=i+1;
}
printf("\n");
```

- Decreasing order

Start from 20: i=20
End at 0: i>=0
Decreased by 5:
i=i-5

```
int i=20;
while (i >= 0) {
    printf("%d ", i);
    i=i-5;
}
printf("\n");
```

do-while statement


Enter the do statement
Execute the statement after the word do
Loop
Evaluate the expression
Expression's value is zero
(false condition)
Exit the do statement
Expression's value is nonzero (true condition)
Go back and execute the statement

```
do{
    statement1;
    statement2;
    …
}while(expression);

statement;
…
```

while/for / do-while

```
init;
while(expr)
{
    statement1;
    statement2;
    …
    alt;
}
```

```
for(init;expr;alt)
{
    statement1;
    statement2;
    …
}
```

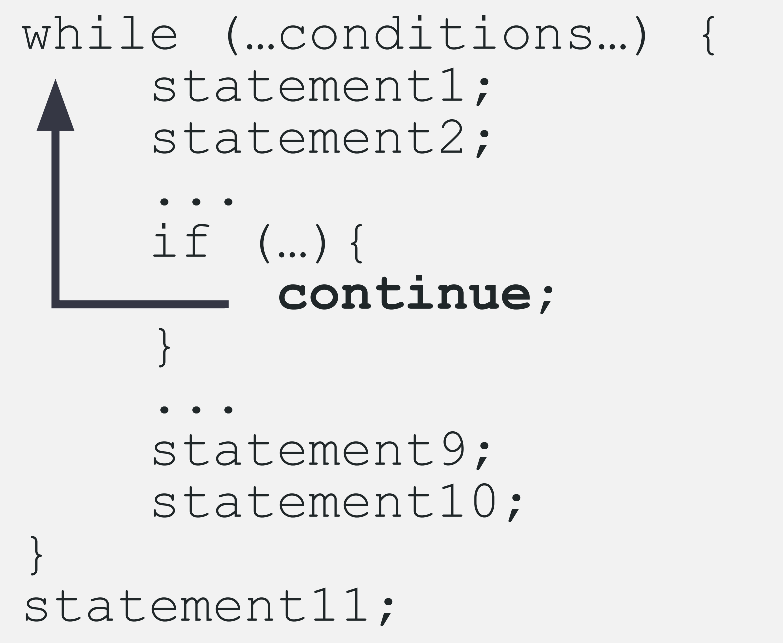
```
init;
do
{
    statement1;
    statement2;
    …
    alt;
}
while(expr);
```

Don’t know/Know
number of loops

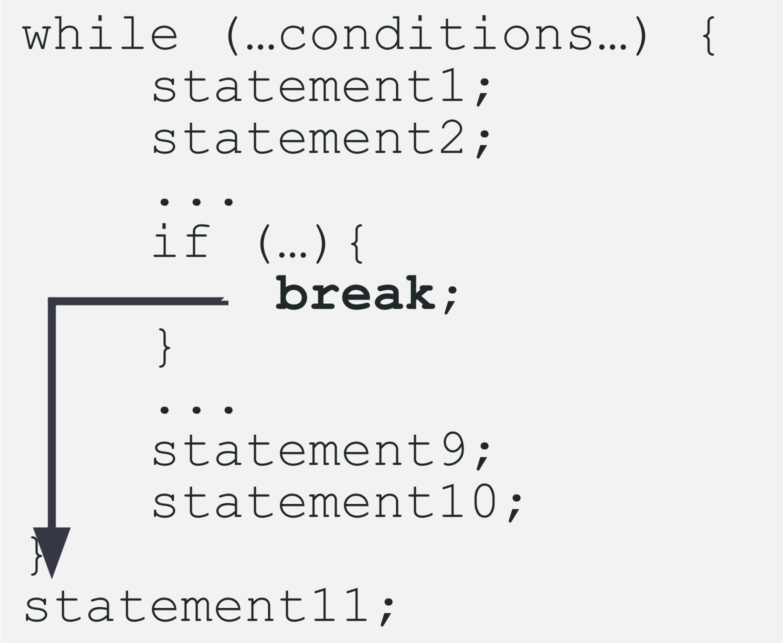
Know
number of loops

Don’t know/Know
number of loops
(*executed at least once*)

continue; vs break; statement



```
while (…conditions…) {
    statement1;
    statement2;
    . . .
    if (…){
        continue;
    }
    . . .
    statement9;
    statement10;
}
statement11;
```

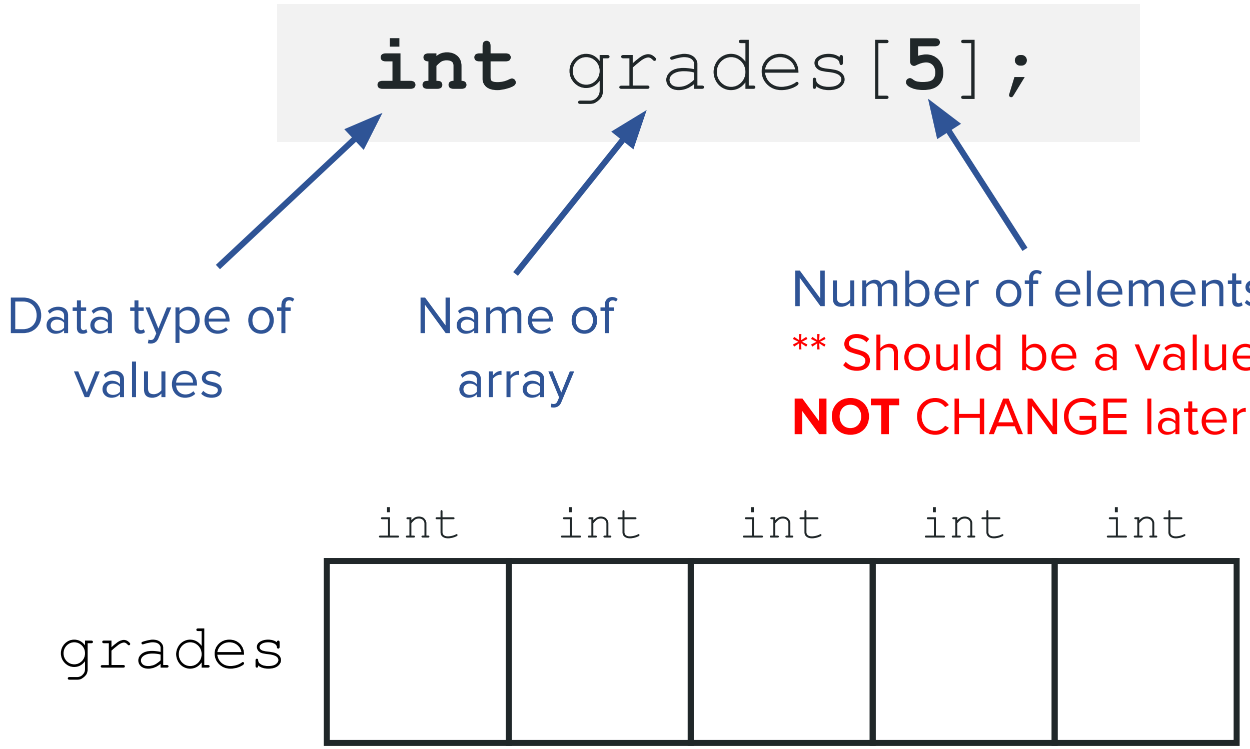


```
while (…conditions…) {
    statement1;
    statement2;
    . . .
    if (…){
        break;
    }
    . . .
    statement9;
    statement10;
}
statement11;
```

1D-Array

Array Declaration

- To create an array, we use a declaration statement



```
int grades[5];
```

Data type of values

Name of array

Number of elements in the array

** Should be a value that WILL **NOT** CHANGE later **

Array Declaration

- To create an array, we use a declaration statement

```
#define N 5
int grades[N];
```

Use #define to create a constant for an array size (at compile-time)

```
int n=5;
int grades[n];
```

Use a variable to specify the size of an array (at compile-time)

```
int n;
scanf("%d", &n);
int grades[n];
```

Use a variable to specify the size of an array (at run-time)

Array Declaration

- If the size is fixed, declare the SIZE of an array as a **constant**, e.g., **`#define N 5`**

```
#define N 5
...
int  array_num[N];
```

- If the size is specified by a user, receive an integer from a users and declare the array

```
int n;
scanf("%d", &n);
int array_num[n];
```

Examples

```
int grades[5] = {98, 87, 92, 79, 85};
char codes[4] = {'x', 'a', 'm', 'n'};



int grades[] = {98, 87, 92, 79, 85};
char codes[] = {'x', 'a', 'm', 'n'};
```

Array Initialization

- If you partially initialize an array, the compiler sets the remaining elements to zero

```
float length[7] = {8.8, 6.4, 4.9, 11.2};
char codes[6] = {'x', 'a', 'm', 'n'};
```

- Thus, it’s easy to initialize all the elements of an array to zero as follows:

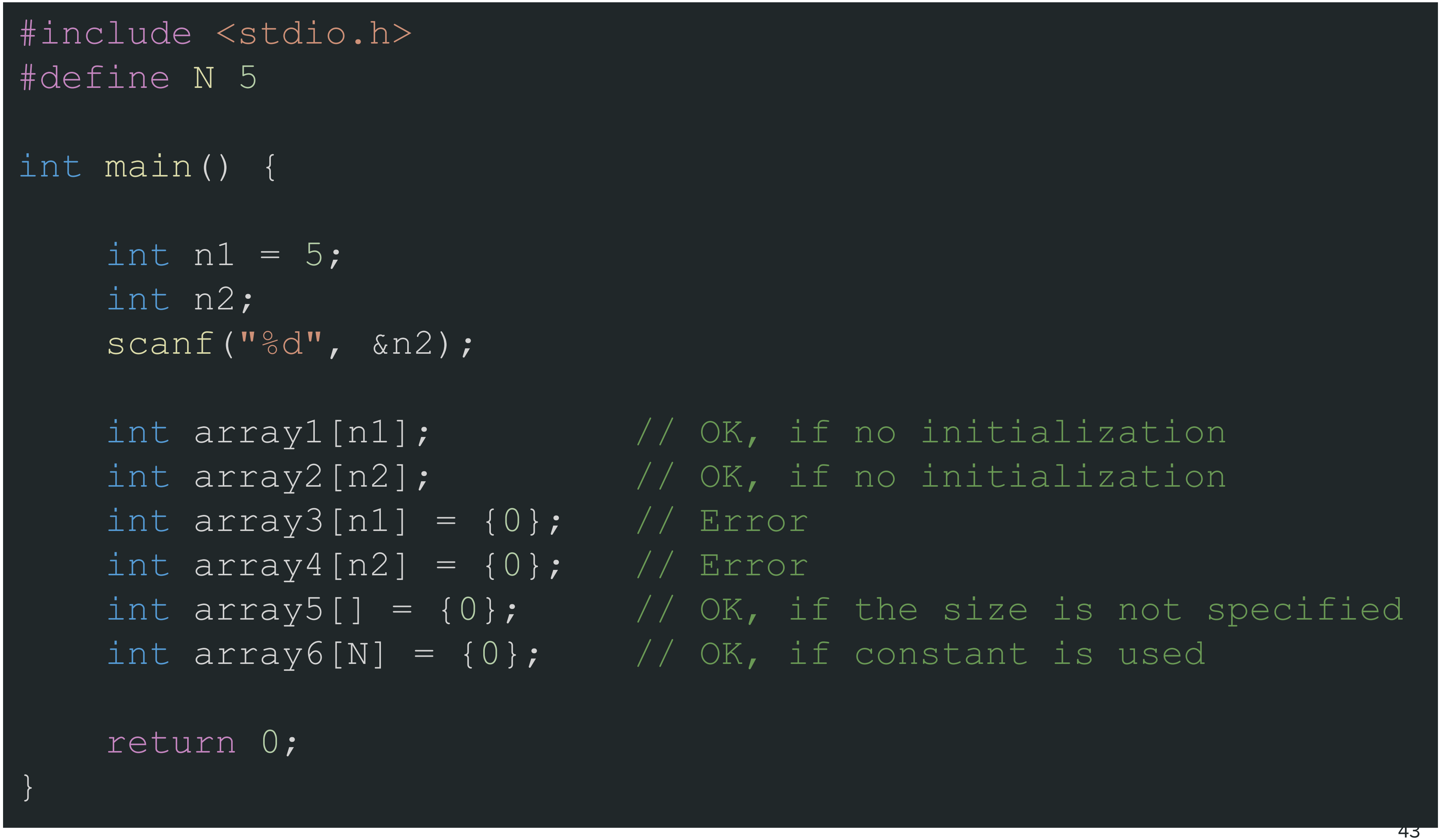
```
float length[7] = {0};
```

Array Initialization

- If all values in the array are known, you are allowed to initialized as assignment

```
#define N 3
...
int array_num[N]={1,2,3};
int array_num2[]={1,2,3};
```

Array Declaration vs. Initialization



```c
#include <stdio.h>
#define N 5

int main() {

    int n1 = 5;
    int n2;
    scanf("%d", &n2);

    int array1[n1];             // OK, if no initialization
    int array2[n2];             // OK, if no initialization
    int array3[n1] = {0};       // Error
    int array4[n2] = {0};       // Error
    int array5[] = {0};         // OK, if the size is not specified
    int array6[N] = {0};        // OK, if constant is used

    return 0;
}
```

Using loops for manipulating arrays

- We can use any expression of type `int` as an array index, e.g. `a[i]`, `a[i+1]`, etc.
- We can run the same code block for each element of an array.

```
int zeros[10];
zeros[0] = 0;
zeros[1] = 0;
zeros[2] = 0;
zeros[3] = 0;
zeros[4] = 0;
...
zeros[9] = 0;
```

⟷

```
int zeros[10];
for (int i=0; i<10; i++) {
    zeros[i] = 0;
}
```

Multidimensional Array

Multidimensional Array

- Multidimensional arrays are arrays with **two or more** dimensions.
- All elements are of the same type.

```
// Define an n-D array
datatype arrayName[size_1][size_2]...[size_n];
```

Functions

Define a function: Function header

- Identifies the data **type** of the **return value**
- Provides the function with a **name**
- Specifies a list of **parameters/arguments** in order, and type of values expected by the function

Function Header

```
return_dtype func_name(dtype1 param1, dtype2 param2)
{
    statement1;  // may define new params
    statement2;  // may use params
    …
    return return_value;
}
```

Function Prototype

- Declaration statement for a function (similar to define a variable)
- Specify function name, parameters and return type as the same we define a function header

```
return_dtype func_name(dtype1 param1,...);
```

Pointer and Address

Pointers

Pointer Declaration	`int *p_num;`
Point to a variable	`p_num = &x;`
Dereferencing	`a = 15 + *p_num;`

	Variable (int x)	Pointer (int *ptr)
Value	x	*ptr
Address	&x	ptr

Function Call

Pass by value: Passing copies of values of variables to a function

Pass by reference: Passing copies of addresses of variables to a function

Pass by Ref. Pass by Val. Pass by Ref.

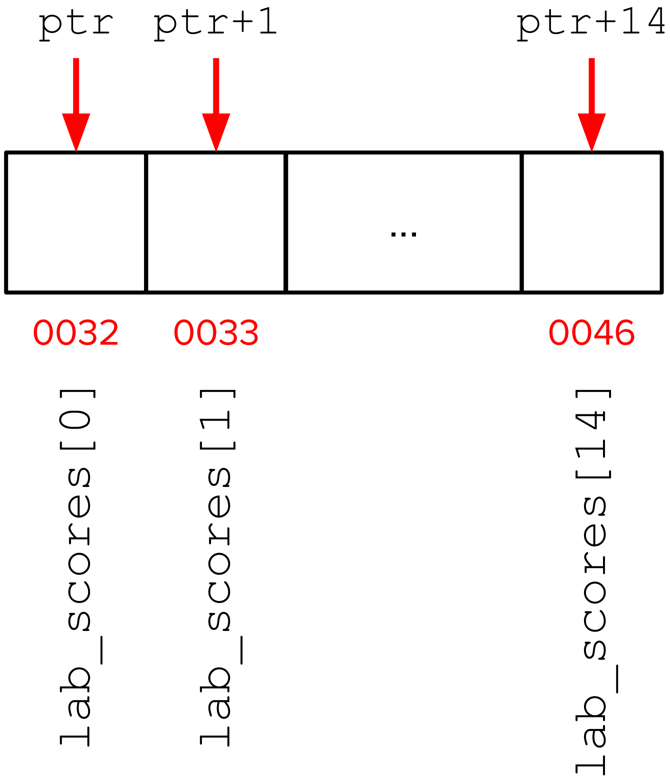
```
return_type function_name(type1 *name1, type2 name2, type3 *name3)
{
    statement1;  // may define new params
    statement2;  // may use arguments
    …
    return expression;
}
```

Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **"pointer + offset"** to access each element in the array.

```
int lab_scores[15] = {...};

// Point to the 1st element
int *ptr;
ptr = &lab_scores[0];

// Access array elements
int i;
for (i=0 ; i<15 ; i++) {
    printf("%d ", *(ptr+i));
}
```

offset

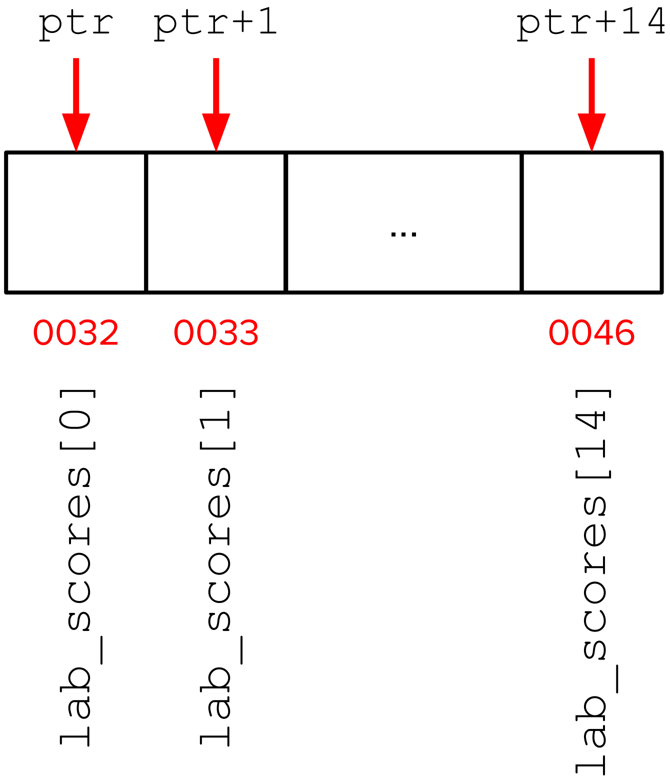


Access Array Elements with Pointer

By making a pointer points to the first element of the array, we can use **“pointer + offset”** to access each element in the array.

```
int lab_scores[15] = {...};

// Point to the 1st element
int *ptr;
ptr = &lab_scores[0];

// Access array elements
int i;
for (i=0 ; i<15 ; i++) {
    printf("%d ", ptr[i]));
}
```

offset



String

String Input and Output

There are several built-in C functions that we can use:

Input	Output
`fgets()`	`puts()`
`scanf()`	`printf()`
`getchar()`	`putchar()`

String Input and Output

fgets()

```
char *fgets(char *str, int n, FILE *stream)
```

fgets() reads a line from a terminal and stores it into the string pointed to by str.

- str: the pointer to an array of chars to store the input string
- n: the maximum number of chars to be read (including ‘\0’)
- stream: the pointer to a FILE object that identifies the stream where chars are read from

It stops when whichever below comes first:

- (n-1) characters are read
- Newline (\n) character is read
- End-of-file (EOF) is reached

String Input and Output

`scanf("%s", str)`

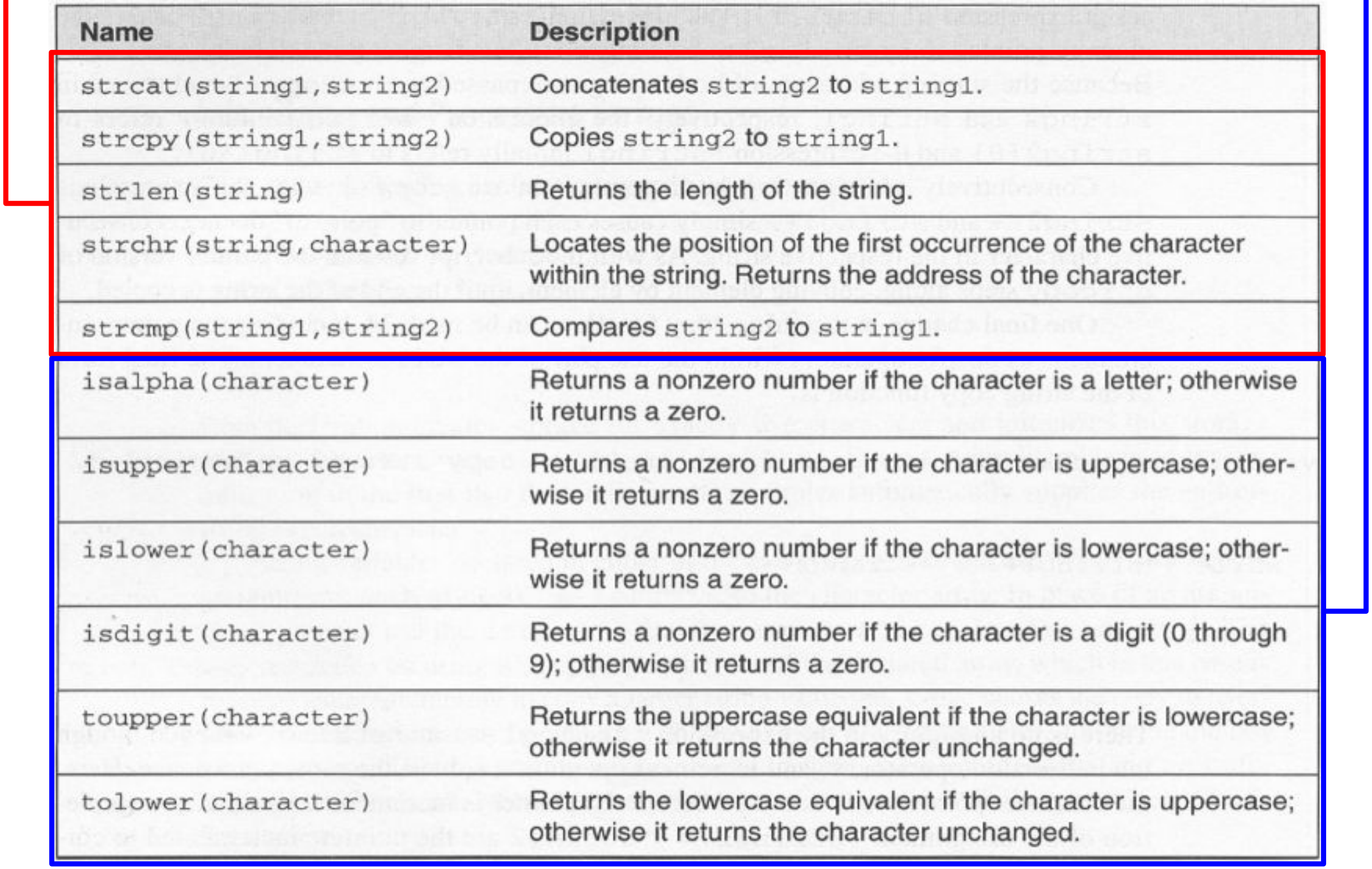
```
#include <stdio.h>
#define MAX_LEN 15

int main()
{
    char input_str[MAX_LEN];
    scanf("%s", input_str);
    printf("%s", input_str);
    return 0;
}
```

- Read characters until the first whitespace or the newline.
- A terminating null-character ('`\0`') is automatically added at the end of the string.

String Library Functions

`#include <string.h>`

`#include <ctype.h>`



Name	Description
`strcat(string1,string2)`	Concatenates `string2` to `string1`.
`strcpy(string1,string2)`	Copies `string2` to `string1`.
`strlen(string)`	Returns the length of the string.
`strchr(string,character)`	Locates the position of the first occurrence of the character within the string. Returns the address of the character.
`strcmp(string1,string2)`	Compares `string2` to `string1`.
`isalpha(character)`	Returns a nonzero number if the character is a letter; otherwise it returns a zero.
`isupper(character)`	Returns a nonzero number if the character is uppercase; otherwise it returns a zero.
`islower(character)`	Returns a nonzero number if the character is lowercase; otherwise it returns a zero.
`isdigit(character)`	Returns a nonzero number if the character is a digit (0 through 9); otherwise it returns a zero.
`toupper(character)`	Returns the uppercase equivalent if the character is lowercase; otherwise it returns the character unchanged.
`tolower(character)`	Returns the lowercase equivalent if the character is uppercase; otherwise it returns the character unchanged.

Reference: [3,4]

String Library Functions

`strcpy()` vs. `strncpy()`

- `strcpy(strto,strfrom)`: copy `strfrom` to `strto`
- `strncpy(strto,strfrom,n)`: copy `n` chars from `strfrom` to `strto`

`strcmp()` vs. `strncmp()`

- `strcmp(str1,str2)`: compare `str1` and `str2`
- `strncmp(str1,str2,n)`: compare first `n` chars of `str1` and `str2`

`strcat()` vs. `strncat()`

- `strcat(strto,strfrom)`: append `strfrom` to `strto`
- `strncat(strto,strfrom,n)`: append `n` chars from `strfrom` to `strto`

String Library Functions

`strchr()` vs. `strrchr()`

- `strchr(str,c)`: find char `c` in `str` and return pointer to first occurrence
- `strrchr(str,c)`: find char `c` in `str` and return pointer to last occurrence

And more ...

Struct

Structure Declaration

Below is how to create a variable of the structure:

```
struct struct_name
{
   datatype var_name1;
   datatype *var_name2;
   datatype var_name3[size];
   ...
};
```

```
struct struct_name var_name;
```

```
struct Date
{
   int month;
   int day;
   int year;
};
struct Date birth_day;
```

```
struct Point
{
   int x;
   int y;
};
struct Point p1, p2;
```

Array of Structure

As we can use `struct` to define a new data type, we can also create an array of such new type.

```
struct struct_name
{
  datatype var_name1;
  datatype *var_name2;
  datatype var_name3[size];
  ...
};
```

```
struct struct_name var_name[size];
```

typedef

`typedef` can be used to give a type a new name.

We can then use `typedef` to shorten the code we use to create a structure variable.

```
struct struct_name
{
  datatype var_name1;
  datatype *var_name2;
  datatype var_name3[size];
  ...
};
```

```
typedef struct struct_name short_name;
```

```
short_name var_name1, var_name2;
// struct struct_name var_name1, var_name2;
```

File Input/Output

Read Data from File

1. **Create a file stream**: We use `FILE` structure declared in `stdio.h`

```
FILE *stream_name;
```

2. **Open a file stream**: We use `fopen()` declared in `stdio.h`

```
stream_name = fopen("filename","r");
```

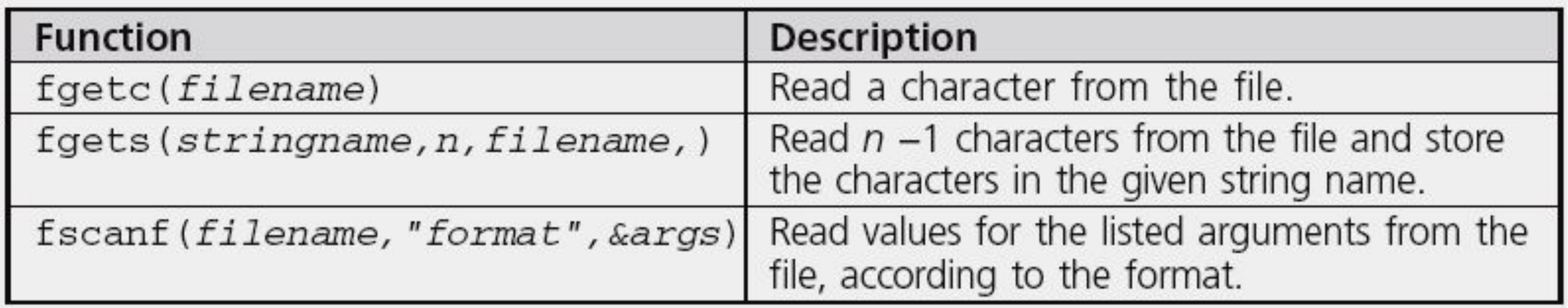
   ○ `"r" (read-only)`

   ○ If a file opened for reading does not exist, `fopen()` returns the NULL address value.

Read Data from File

3. **Read data from the opened file stream**

- Use `fgetc()`, `fgets()` and `fscanf()` in `stdio.h`



Function	Description
`fgetc(`*`filename`*`)`	Read a character from the file.
`fgets(`*`stringname,n,filename,`*`)`	Read *n* –1 characters from the file and store the characters in the given string name.
`fscanf(`*`filename,"format",&args`*`)`	Read values for the listed arguments from the file, according to the format.

  - `fgetc()` and `fscanf()` return `EOF` when the end-of-file marker is detected
  - `fgets()` returns a `NULL` instead

- Examples

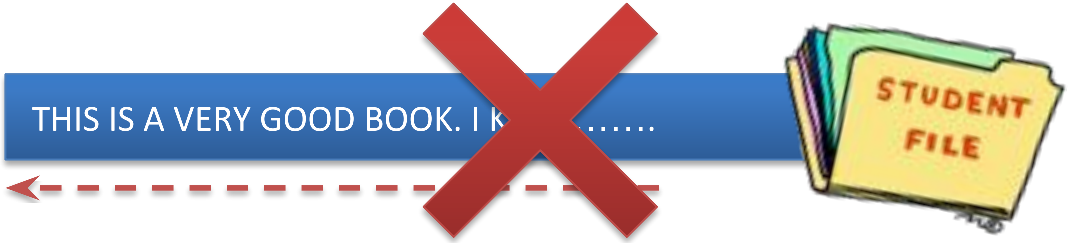
```
fgetc(in_file);
fgets(message, 10, in_file);
fscanf(in_file, "%lf", &price);
```

Read Data from File

4. **Close the opened file stream**
   - Use `fclose()` in `stdio.h`

```
fclose(stream_name);
```

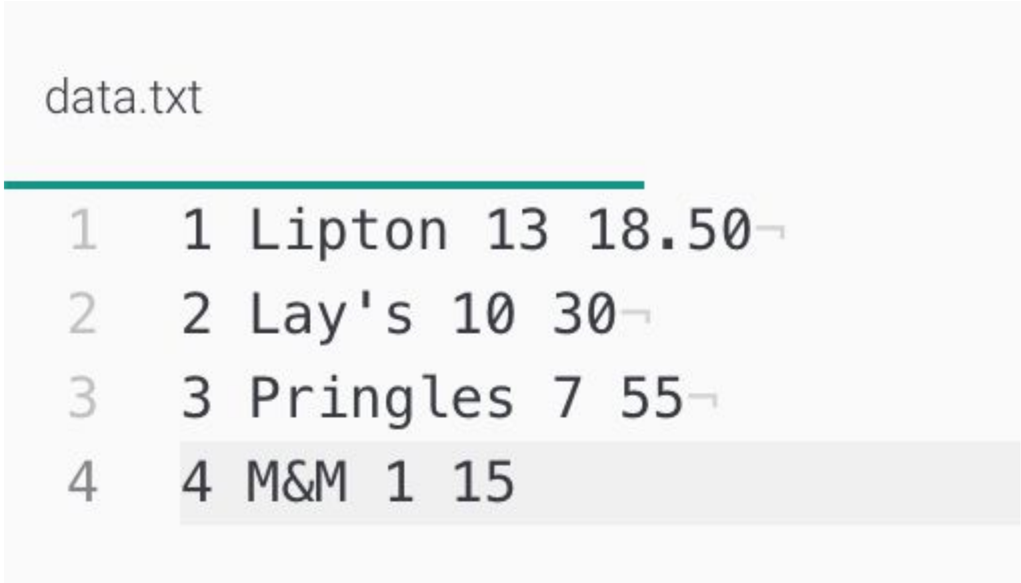
   - Because all computers have a limit on the maximum number of files that can be open at one time
   - Closing files that are no longer needed is a good practice



Example

Suppose we have created a file, named “`data.txt`”, that contains the following data.

For simplicity, please make sure that the datafile is in the same directory as the C program file.



```
1 Lipton 13 18.50
2 Lay's 10 30
3 Pringles 7 55
4 M&M 1 15
```

Example fscanf()

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

int main() {
    // Create a file stream
    FILE *in_file;
    // Link the file stream to a file, named "data.txt"
    in_file = fopen("data.txt", "r");
    // Check whether the file has been opened successfully
    if (in_file == NULL) {
        printf("The file was not successfully opened.\n");
        printf("Please check that the file currently exists.\n");
        exit(1);
    }
    // Read data from the stream until the end of the opened file
    int id, qty;
    char name[16];
    float price;
    while (fscanf(in_file, "%d %s %d %f", &id, name, &qty, &price) != EOF) {
         printf("ID: %d, Name: %s, Qty: %d, Price: %.2f\n", id, name, qty, price);
    }
    fclose(in_file); // Close a file stream
    return 0;
}
```

Example fgets()

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

int main() {
    // Create a file stream
    FILE *in_file;
    // Link the file stream to a file, named "data.txt"
    in_file = fopen("data.txt", "r");
    // Check whether the file has been opened successfully
    if (in_file == NULL) {
        printf("The file was not successfully opened.\n");
        printf("Please check that the file currently exists.\n");
        exit(1);
    }
    // Read data from the stream until the end of the opened file
    int max_length = 51;
    char lines[max_length];
    while (fgets(lines, max_length, in_file) != NULL) {
        printf("%s", lines);
    }
    fclose(in_file); // Close a file stream
    return 0;
}
```

Write Data to File

1. **Create a file stream**: We use `FILE` structure declared in `stdio.h`

```
FILE *stream_name;
```

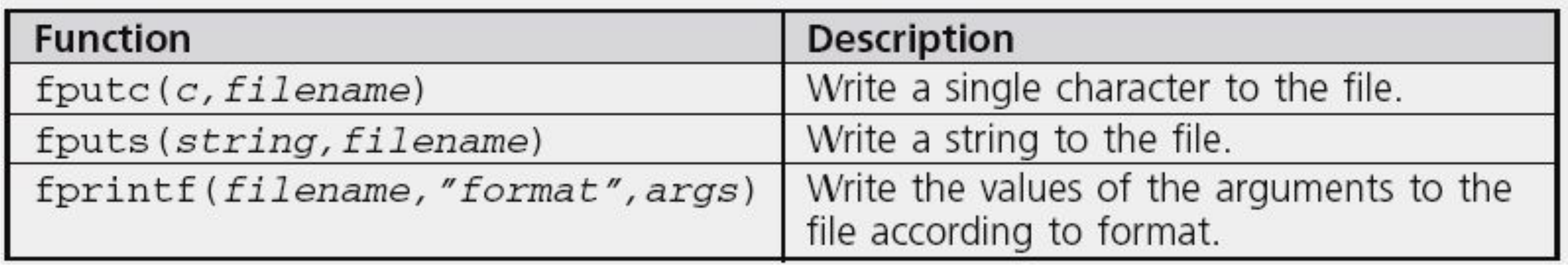
2. **Open a file stream**: We use `fopen()` declared in `stdio.h`

```
stream_name = fopen("filename","w");
stream_name = fopen("filename","a");
```

- `"w"` (write-only): Create a file for writing; overwrite the existing file (if any).
- `"a"` (append): Open a file for writing, where the data are written at the end of the existing file; if there is no existing file, a new file is created for writing.

Write Data to File

3. **Write data to the opened file stream**
    - Use `fputc()`, `fputs()` and `fprintf()` in `stdio.h`



Function	Description
fputc(*c,filename*)	Write a single character to the file.
fputs(*string,filename*)	Write a string to the file.
fprintf(*filename,"format",args*)	Write the values of the arguments to the file according to format.

    - Example

```
fputc('a',out_file);
fputs("Hello world!",out_file);
fprintf(out_file,"%s %n",descrip,price);
```

Write Data to File

4. **Close the opened file stream**
    - Use `fclose()` in `stdio.h`

    ```
    fclose(stream_name);
    ```

    - Similar to save file
    - Closing files that are no longer needed is a good practice

Example

Suppose we want to create a file, named "`output.txt`", that contains the following data.

```
struct Item list_items[3] = {
    {5, "Coca Cola", 50, 14},
    {6, "Matcha Latte", 10, 120},
    {7, "Doritos", 12, 34.5},
};
```

Example fprintf()

```
struct Item
{
    int id;
    char name[16];
    int qty;
    float price;
};
```

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

int main() {
    struct Item list_items[3] = {
         {5, "Coca Cola", 50, 14},
         {6, "Macha Latte", 10, 120},
         {7, "Doritos", 12, 34.5},
    };
    // Create a file stream
    FILE *out_file;
    // Link the file stream to a file, named "output.txt" for writing
    out_file = fopen("output.txt", "w");
    // out_file = fopen("output.txt", "a");  // For appending

    // Check whether the file has been opened successfully
    if (out_file == NULL) {
         printf("Failed to open the file.\n");
         exit(1);
    }
    // Write data into file (similar to printf())
    for (int i=0 ; i<3 ; i++) {
         fprintf(out_file, "ID: %d, Name: %s, Qty: %d, Price: %.2f\n",
              list_items[i].id, list_items[i].name, list_items[i].qty, list_items[i].price
         );
    }
    fclose(out_file); // Close a file stream
    return 0;
}
```

Example

fprintf()

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

int main() {
   struct Item list_items[3] = {
       {5, "Coca Cola", 50, 14},
       {6, "Macha Latte", 10, 120},
       {7, "Doritos", 12, 34.5},
   };
   // Create a file stream
   FILE *out_file;
   // Link the file stream to a file, named "output.txt" for writing
   out_file = fopen("output.txt", "w");
   // out_file = fopen("output.txt", "a");  // For appending

   // Check whethe
   if (out_file ==
       printf("Fai
       exit(1);
   }
   // Write data into file (similar to printf())
   for (int i=0 ; i<3 ; i++) {
       fprintf(out_file, "ID: %d, Name: %s, Qty: %d, Price: %.2f\n",
           list_items[i].id, list_items[i].name, list_items[i].qty, list_items[i].price
       );
   }
   fclose(out_file); // Close a file stream
   return 0;
}
```

```
struct Item
{
   int id;
   char name[16];
   int qty;
   float price;
};
```

output.txt

```
ID: 5, Name: Coca Cola, Qty: 50, Price: 14.00
ID: 6, Name: Matcha Latte, Qty: 10, Price: 120.00
ID: 7, Name: Doritos, Qty: 12, Price: 34.50
```