Week 6

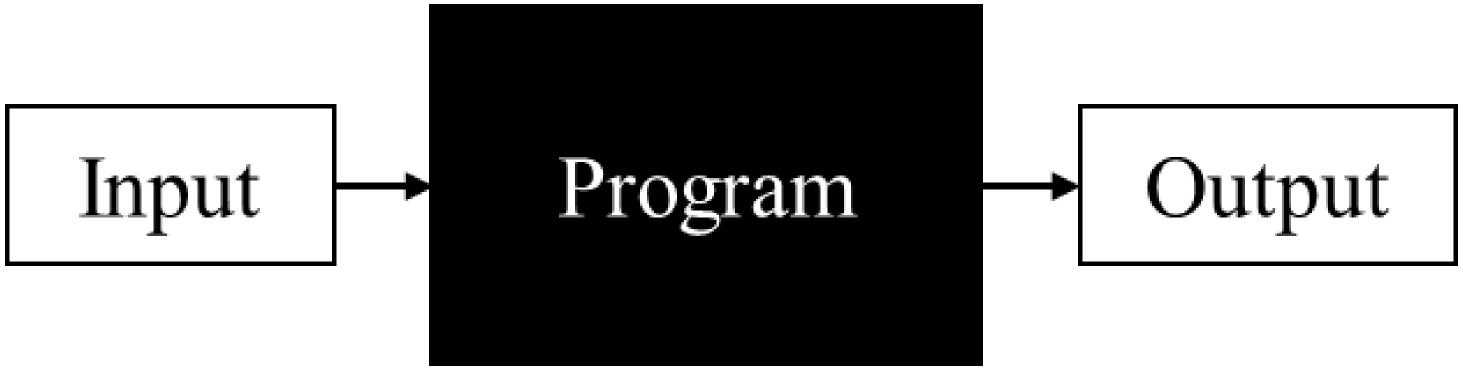
Hari Sethuraman

In today’s lesson

- Recap everything we have learned so far

What is a Program?

- A program is a set of steps that converts an Input to an Output.

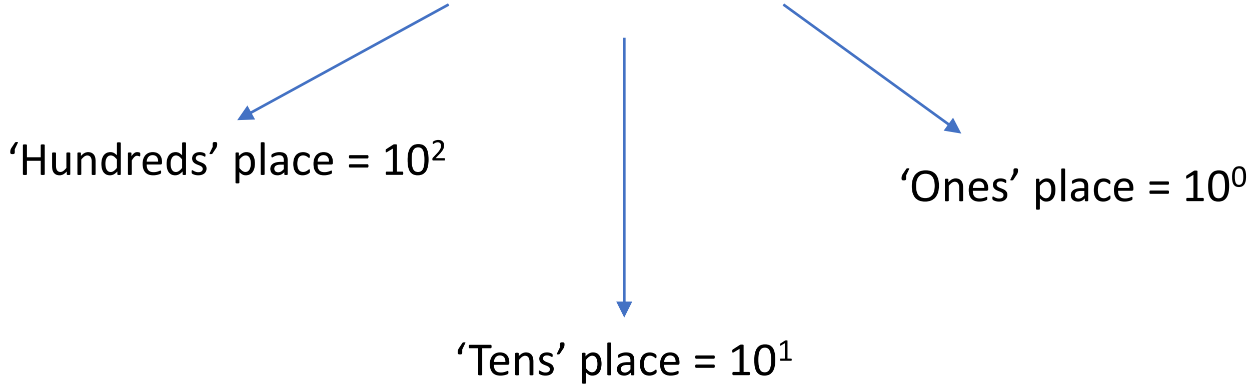


- The input is what the user passes into the program
- The output is what the user wants in return
- The black box is the program we create.

How do we write numbers?

- We write numbers in a form called ‘base ten’

2 1 3



‘Hundreds’ place = 10^2

‘Tens’ place = 10^1

‘Ones’ place = 10^0

Examples

- 5 -> (5 × 10^0)
- 65 -> (6 × 10^1) + (5 × 10^0)
- 125 -> (1 × 10^2) + (2 × 10^1) + (5 × 10^0)

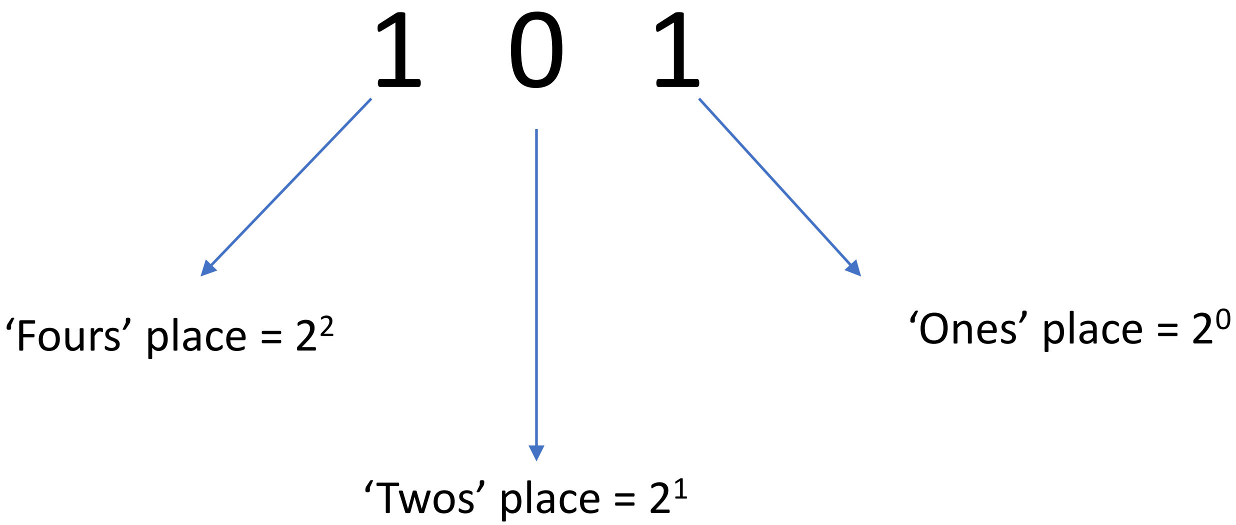
Click to add text

- 213 -> (2 × 10^2) + (1 × 10^1) + (3 × 10^0)

How do computers ‘write’ numbers?

- Computers use something called ‘Base two’
- Instead of each digit being a power of ten
- Computers use Powers of two
    - Each digit can only have two values: 0, 1
    - This is because transistors in computers can only be
        - ‘Off’ = 0
        - Or ‘On’ = 1

Continuation

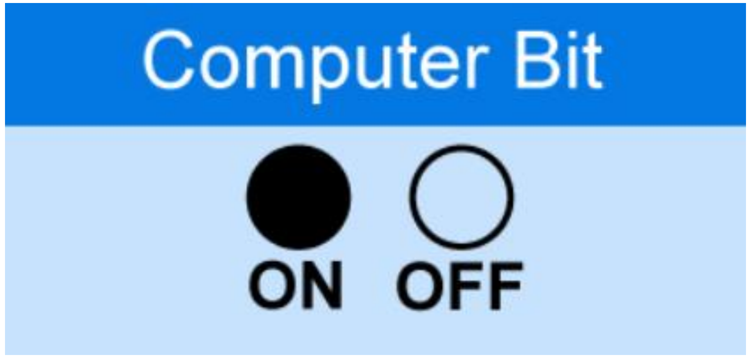


1 0 1

‘Fours’ place = 2^2

‘Twos’ place = 2^1

‘Ones’ place = 2^0

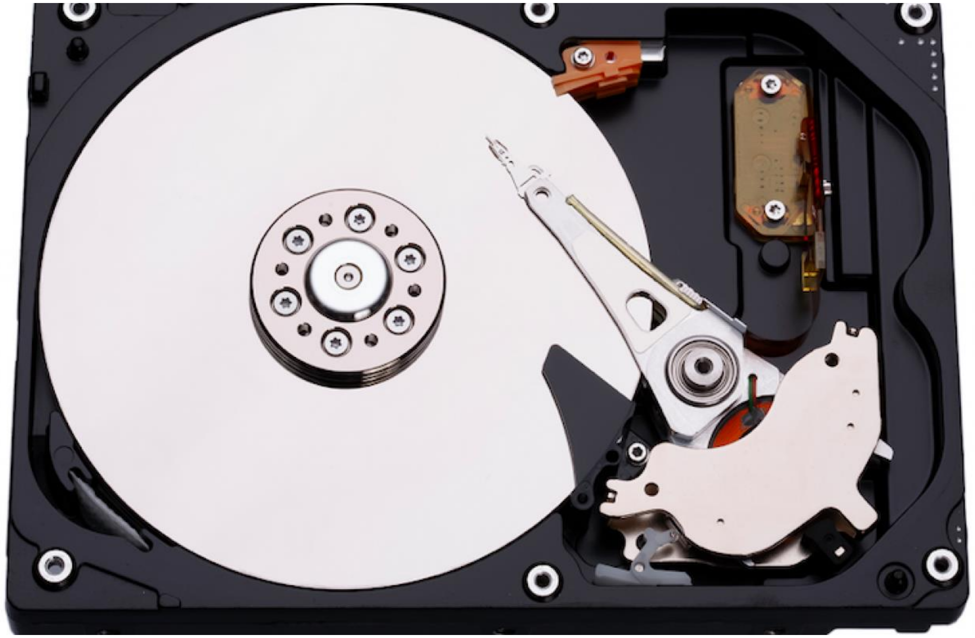


Examples

- 1 -> (1×2^{0})
    - 1
- 1 0 -> $(1 \times 2^{1}) + (0 \times 2^{0})$
    - 2 + 0
        - 2
- 1 1 0 -> $(1 \times 2^{2}) + (1 \times 2^{1}) + (0 \times 2^{0})$
    - 4 + 1 + 0
        - 5
- 1 1 0 1 1 -> $(1 \times 2^{4}) + (1 \times 2^{3}) + (0 \times 2^{2}) + (1 \times 2^{1}) + (1 \times 2^{0})$
    - 16 + 8 + 0 + 2 + 1
        - 27

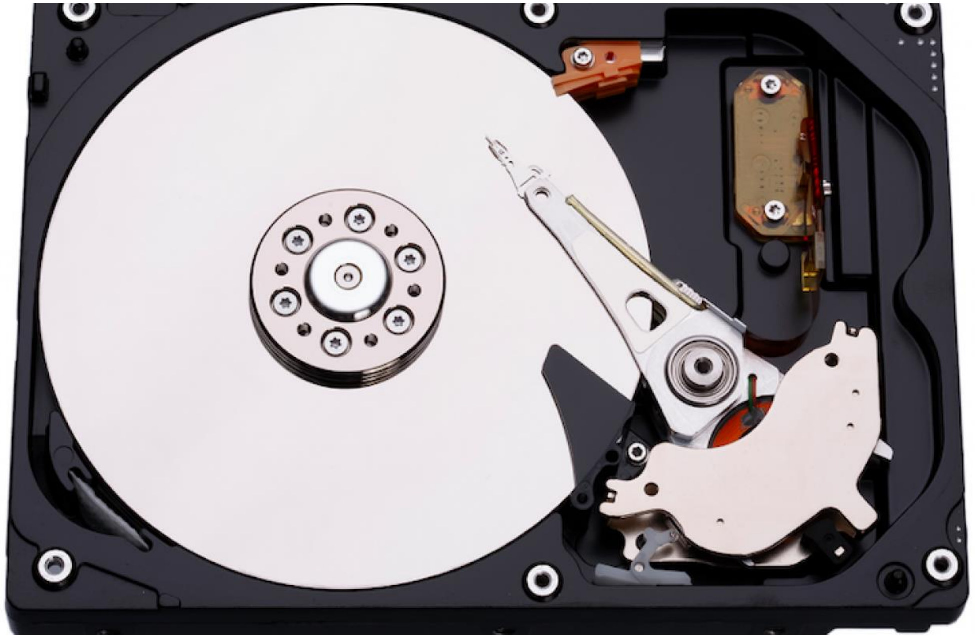
Bits and Bytes

- Each ‘0’ or ‘1’ is equal to **one Bit**
  - 1 1 1 1 -> Four Bits
  - 1 0 1 0 1 -> Five Bits
- Computers group Bits into **Bytes**
  - Eight Bits make **One Byte**
    - 1024 Bytes Make one Kilobyte
    - 1024 Kilobytes make one Megabyte
    - 1024 Megabytes make one Gigabyte
    - …



Bits and Bytes

- Each ‘0’ or ‘1’ is equal to **one Bit**
  - 1 1 1 1 -> Four Bits
  - 1 0 1 0 1 -> Five Bits
- Computers group Bits into **Bytes**
  - Eight Bits make **One Byte**
    - 1024 Bytes Make one Kilobyte
    - 1024 Kilobytes make one Megabyte
    - 1024 Megabytes make one Gigabyte
    - …

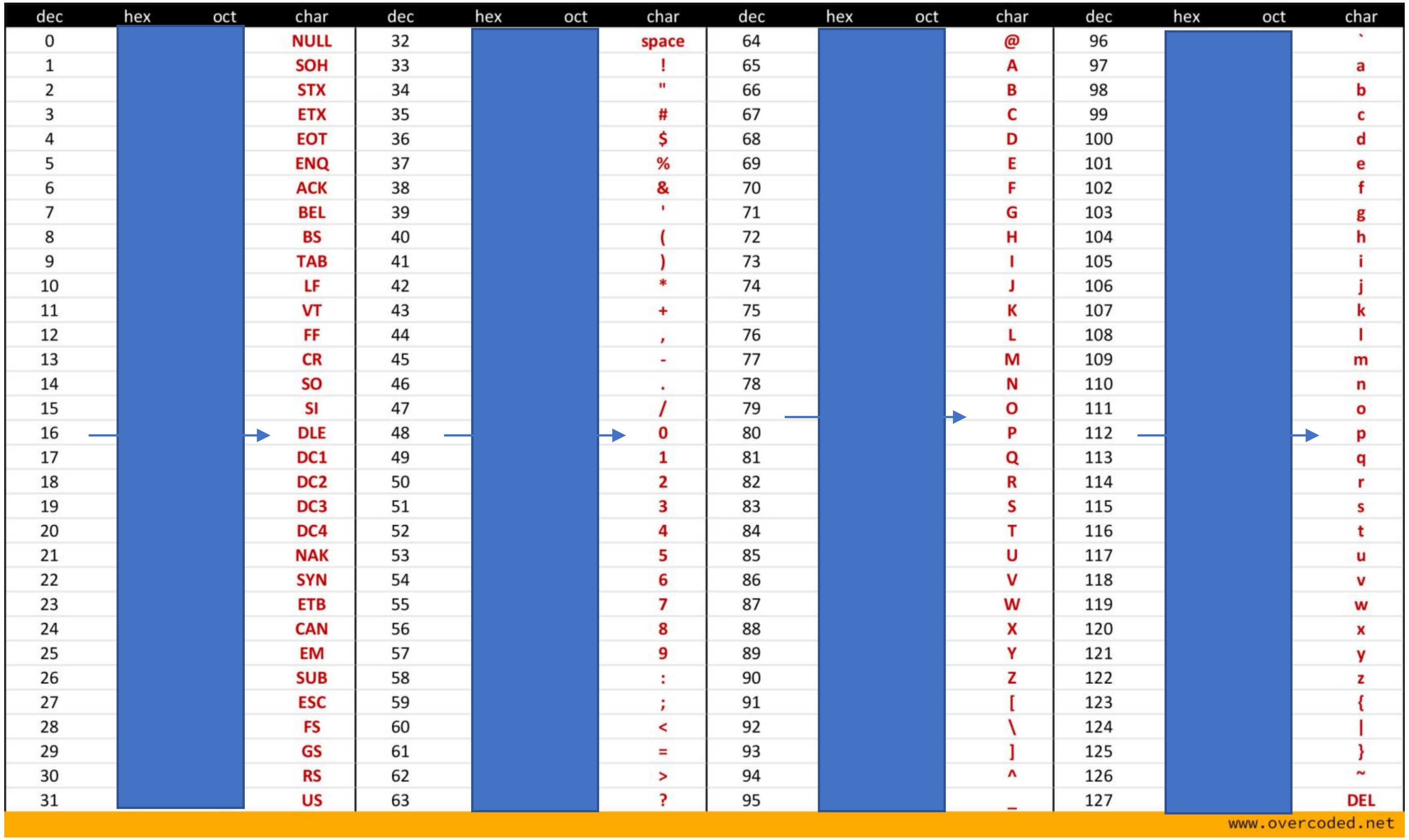


Datatypes

- There are different types of information:
  - Numbers, Strings, Characters
- In order to let the computer know which type we are using, we use data types.

Char

Character: there are 256 different Characters. Represented using ‘ASCII’



dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char
0			NULL	32			space	64			@	96			`
1			SOH	33			!	65			A	97			a
2			STX	34			"	66			B	98			b
3			ETX	35			#	67			C	99			c
4			EOT	36			$	68			D	100			d
5			ENQ	37			%	69			E	101			e
6			ACK	38			&	70			F	102			f
7			BEL	39			'	71			G	103			g
8			BS	40			(	72			H	104			h
9			TAB	41			)	73			I	105			i
10			LF	42			*	74			J	106			j
11			VT	43			+	75			K	107			k
12			FF	44			,	76			L	108			l
13			CR	45			-	77			M	109			m
14			SO	46			.	78			N	110			n
15			SI	47			/	79			O	111			o
16			DLE	48			0	80			P	112			p
17			DC1	49			1	81			Q	113			q
18			DC2	50			2	82			R	114			r
19			DC3	51			3	83			S	115			s
20			DC4	52			4	84			T	116			t
21			NAK	53			5	85			U	117			u
22			SYN	54			6	86			V	118			v
23			ETB	55			7	87			W	119			w
24			CAN	56			8	88			X	120			x
25			EM	57			9	89			Y	121			y
26			SUB	58			:	90			Z	122			z
27			ESC	59			;	91			[	123			{
28			FS	60			<	92			\	124			|
29			GS	61			=	93			]	125			}
30			RS	62			>	94			^	126			~
31			US	63			?	95			_	127			DEL

www.overcoded.net

‘A’ -> 65 -> 0 1 0 0 0 0 0 1

‘!’ -> 33 -> 0 0 1 0 0 0 0 1

‘%’ -> ?

‘=’ ->

Integer

- Typically 2 or 4 bytes in C (represented using *int*)
  - If 2 bytes: value can be between -32,768 and 32,767 (Why?)
    - If we have 2 bytes or 16-bits, then the largest number we can represent is $2^{15} + 2^{14}$ +… + $2^1 + 2^0$ = 65533. (we stack the bytes on top of each other)
    - However, because we want to represent both positive and negative integers, we split this number across the number line
      - Therefore, the minimum is -32768 and maximum is 32767
  - If 4 bytes: value can be between -2,147,483,649 and 2,147,483,649.
- Variations:
  - Unsigned-int: only positive integers (use when you don't need –ve numbers)
  - Long: int but with longer numbers between -9223372036854775808 and 9223372036854775807

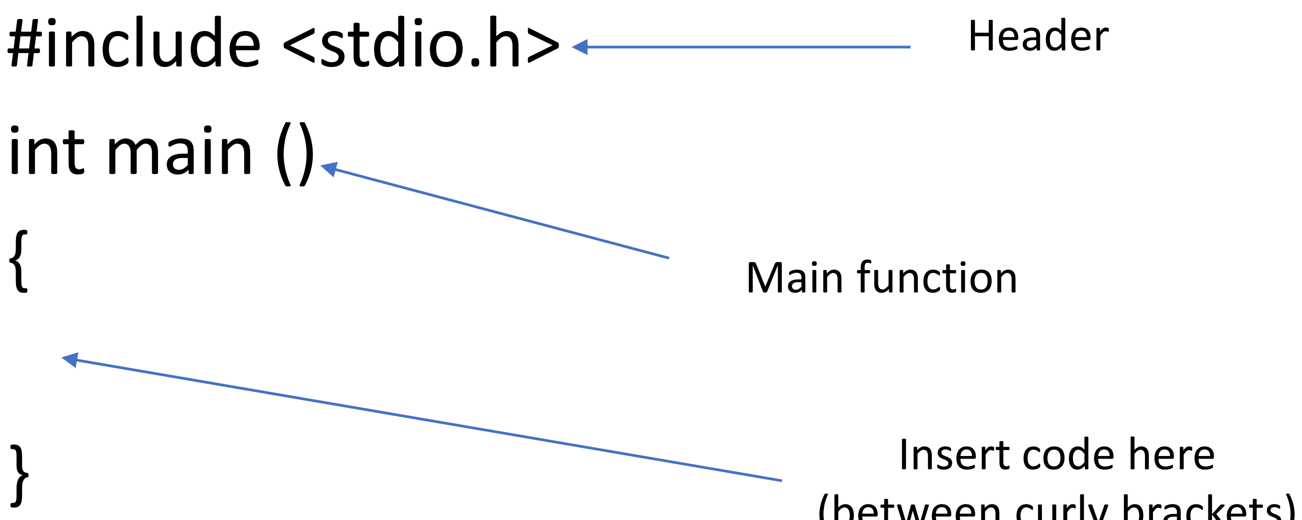
Float

- Used to store decimal values:
  - Usually takes up 4 bytes.
  - Stores up to 6 decimal places
- Variations:
  - Double:
    - Takes up 8 bytes
    - Stores up to 15 decimal places
  - Long Double:
    - Takes up 10 bytes
    - Stores up to 19 decimal places

Boolean

- Typically takes up 1 byte
  - Stores a value of ‘True’ or ‘False’
  - Why not 1-bit? Because computers store info in bytes as a whole

How a C-program looks like



```
#include <stdio.h>
int main ()
{

}
```

Header

Main function

Insert code here
(between curly brackets)

Printf

- printf is a function, that prints out (or displays) whatever is passed to it:
  - Example:
    - printf(“Hi”);
      - Will print Hi
  - Including a ‘\n’ at the end of the string we want to print skips a line

Printf + placeholder

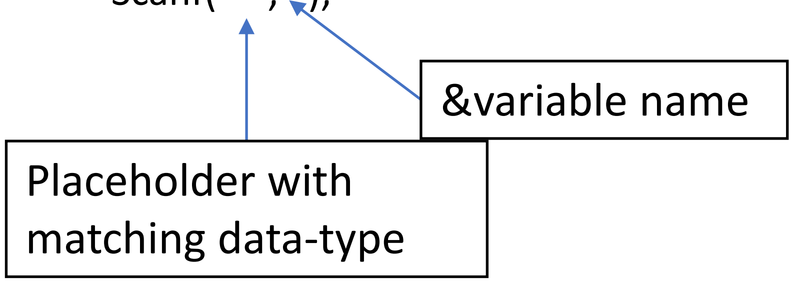
- If we want to print the value of a variable using a printf statement, we need to use placeholders:
  - *printf(“hello, %d”, i);*

    placeholder

    i is the variable
  - *%d* for integer variables
  - *%c* for chars
  - *%f* for floats
  - …
- Multiple place holders:
  - *printf(“hello, %d, %c”, i, c);*
  - List the variables in order of the placeholder
  - Make sure the datatypes match or else the program will crash!

scanf

- To get the input from the user and store it in a variable, we use *scanf*
    - Scanf consists of two parts
        - Scanf(“ ”,);



&variable name

Placeholder with matching data-type

-
    - Example: get an integer input and store it in the variable *I*
        - *int i = 0;* **initializing the variable value**
        - *scanf(“%d”, &i);* **get input from the user**
    - The computer stores the input once the user pressers the enter button
    - *Why the ‘&’? We use this symbol in order to tell the computer the location in memory of *i*. if we just used *i*, we would pass the value of *i*, which is currently *0*.
    - This is not very useful, therefore, by using the ‘&’ before the variable name, we pass the location of the variable in memory rather than the value, so that the computer can directly write the input value there

Operators

- For any kind of logic or operation, we use ‘Operators’. There are 3 main types:
  - Arithmetic operators
  - Relational operators
  - Logical operators

Arithmetic operators

- Used to computer arithmetic operations like addition, subtraction…
  - + used to add numbers
    - Example
      - Int x = 3 + 5 + 7; (x = 15)
  - - used to subtract numbers
  - * used to multiply numbers
  - / for dividing
  - % is the ‘modulo operator’. Returns the remainder of division:
    - Int x = 5;
    - Int y = 2;
    - Int z = x % y; Remainder of dividing first number by second
      - Z = 1

Relational operators

```
Int a = 1;
Int b = 0;
```

- Used to compare variables or numbers (returns a true or false value)
  - == used to check if two numbers or variables are equivalent
    - Not assignment! (=)
    - (a == b) -> returns false
  - != used to check if two numbers or variables are not equal
    - (a != b) -> returns true
  - > and < used to check if one variable or number is greater or lesser than the other
    - (a > b) -> returns true
  - <= and >= used for ‘lesser than or equal to’ and ‘greater than or equal to’

bool a = true;
bool b = false;

To use bool, import <bool.h>

Logical operators

- Compares two Boolean values and returns a Boolean value
  - &&: means 'and'
    - Both values must be true to return true
    - a && b -> returns false because b is false
  - ||: means 'or'
    - At least one operation must be true to return true
    - a || b -> returns true because a is true
    - True || true -> true
    - True || false -> true
    - False || true -> true
    - False || false -> false

If conditions

- If conditions run the code inside them when a certain condition is met:

```
if (        )
{

}
```

Condition goes between these brackets

Enter code between curly braces

If the condition inside brackets is true, then run code between curly braces

Else

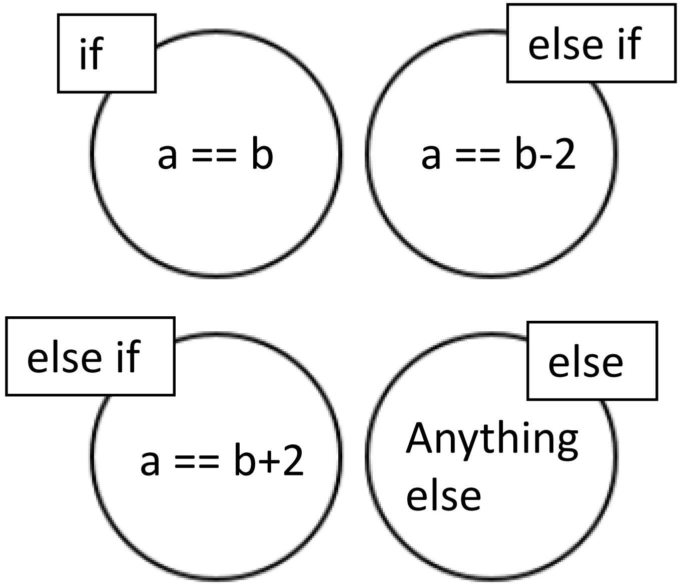
- We use ‘else’ after an if condition:
  - If __ is true, then execute __. Else, execute __
  - Does not require a condition

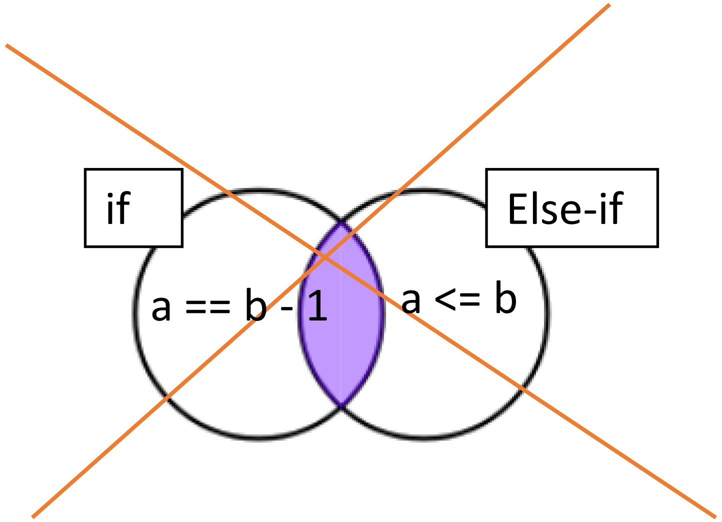
```
else
{


}
```

Enter code between curly braces

Else if

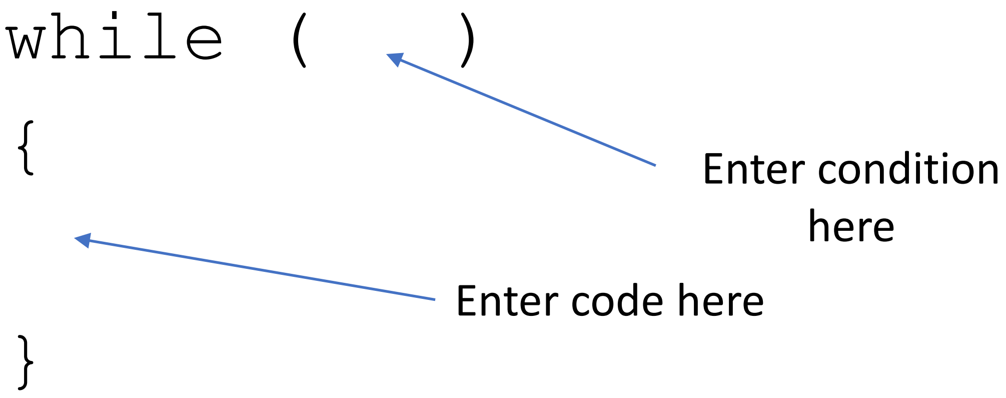
- Use else if statements if you want to stack multiple ifs, and only want one of them to execute
  - Always comes after an if-condition
  - IMPORTANT: all ifs, else ifs, and else's must have Mutually Exclusive conditions and code





Loops

- Loops are essentially an if condition that repeats itself:
  - while (a condition is true), repeat {the code in the curly braces}
  - One cycle of a loop is called an ‘Iteration’



```
while (    )
{

}
```

Enter condition here

Enter code here

Program

- Task: Print all integers from 1 through 20 each in a new line

Method 1

Manually print all numbers

```
printf(“1\n”);
printf(“2\n”);
…
printf(“19\n”);
printf(“20\n”);
```

Bad code

- Always avoid repeating lines in programming as it makes the code less aesthetic, and it makes debugging MUCH harder

Method 2

Use a loop to print all numbers

```
int i = 1;
while (i <= 20)
{
    printf(“%d\n”, i);
    i = i + 1;
}
```

The iterator variable: a variable that changes its value after each iteration

Incrementing *i* after each iteration

Incrementing shortcuts

- i = i + 1;
- i += 1;
- i++;

All mean the same thing
(increase i’s value by 1)

Arrays

- Arrays are a data structure that can store multiple variables of the same data type
  - An array of characters, integers, etc…
- They are an ordered list of elements
- To access the elements of an array, we need to provide the element’s ‘Index’
  - The Index starts counting from 0
    - To access the first element, you would need to pass in *0* as the index
    - To access the *n*th element, you would need to pass in (n-1) as the index
  - Use the name of the array followed by square brackets
    - *a[0]*

Arrays (continuation)

- Declaring an array is like declaring a regular variable
  - `char a[] = {'a', 'b', 'c', ... , 'x', 'y', 'z'};`

    Datatype name The elements in the array
  - If you don’t know the elements, but know the length:
    - `char a[26];`
- When declaring an array, you need to know either the length or the elements in it.
- Suppose we had an array *a* which stores all the alphabets (in lower case)
  - `char a[] = {'a', 'b', 'c', ... , 'x', 'y', 'z’};`
  - `Printf(“%c\n”, a[5]);` what would this print?

Getting elements into an array

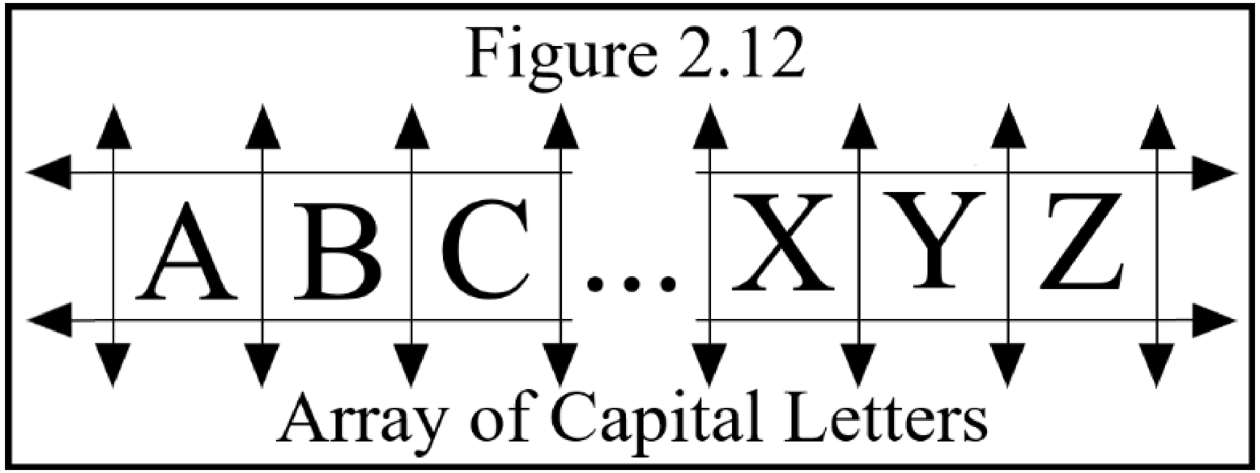
- To get elements into an array created by the user, you need to know the length *l* of the array.
    - Ask this first through a scanf
- Then, create an array length *l*
- Using a loop, ask the individual elements through a scanf

Program

```
int l = 0;
scanf("%d", &l);
int nums [l];

int i = 0;
while (i < l)
{
    scanf("%d", &nums[i]);
    i++;
}

i = 0;
while (i < l)
{
    printf("%d, ", nums[i]);
    i++;
}
```

How are arrays represented in memory

- If you remember, the memory is like a large grid of bytes
- An array stores its elements consecutively in memory
- When you declare an array, the program allocates (or reserves) a certain amount of memory for the array
  - How much? The number of elements * the size of an element



Homework - Fibonacci

- You might know the Fibonacci sequence:
  - 0, 1, 1, 2, 3, 5, 8 …
- The *n*th number in the sequence is the sum of the *(n-1)*th number and the *(n-2)*th number
- You are given a single integer *n* as an input (1 ≤ n ≤ 100).
  - If *n* is lesser than 1 or greater than 100, print "Enter a number between 1 and 100" and quit the program (you can do this by simply writing *return 0;* in the if condition)
  - Print the *n*th number in the Fibonacci sequence.