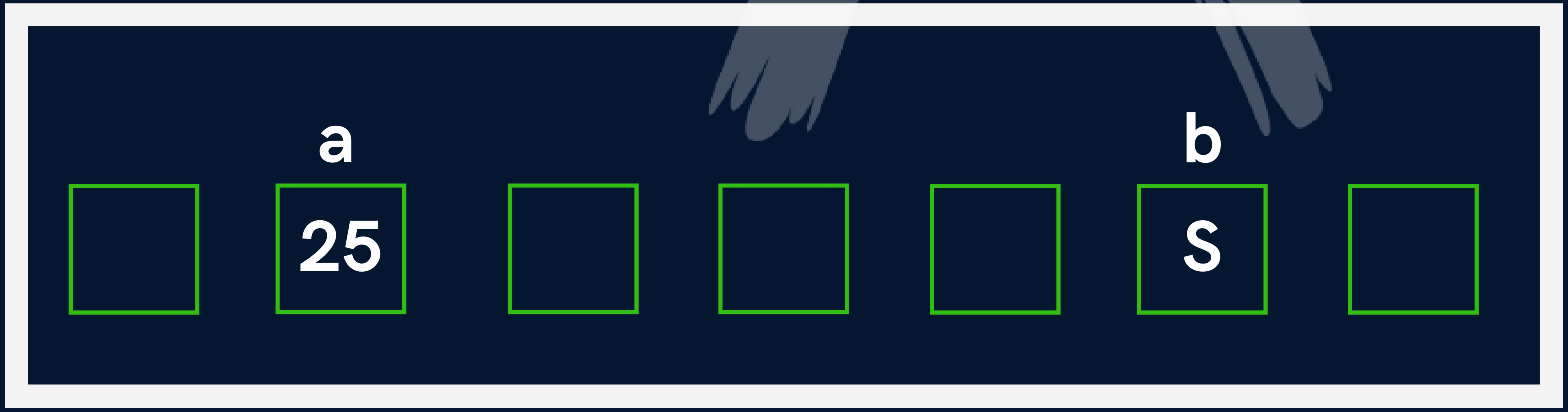
Variables

Variable is the name of a memory location which stores some data.

Memory



Variables

Rules

a. Variables are case sensitive

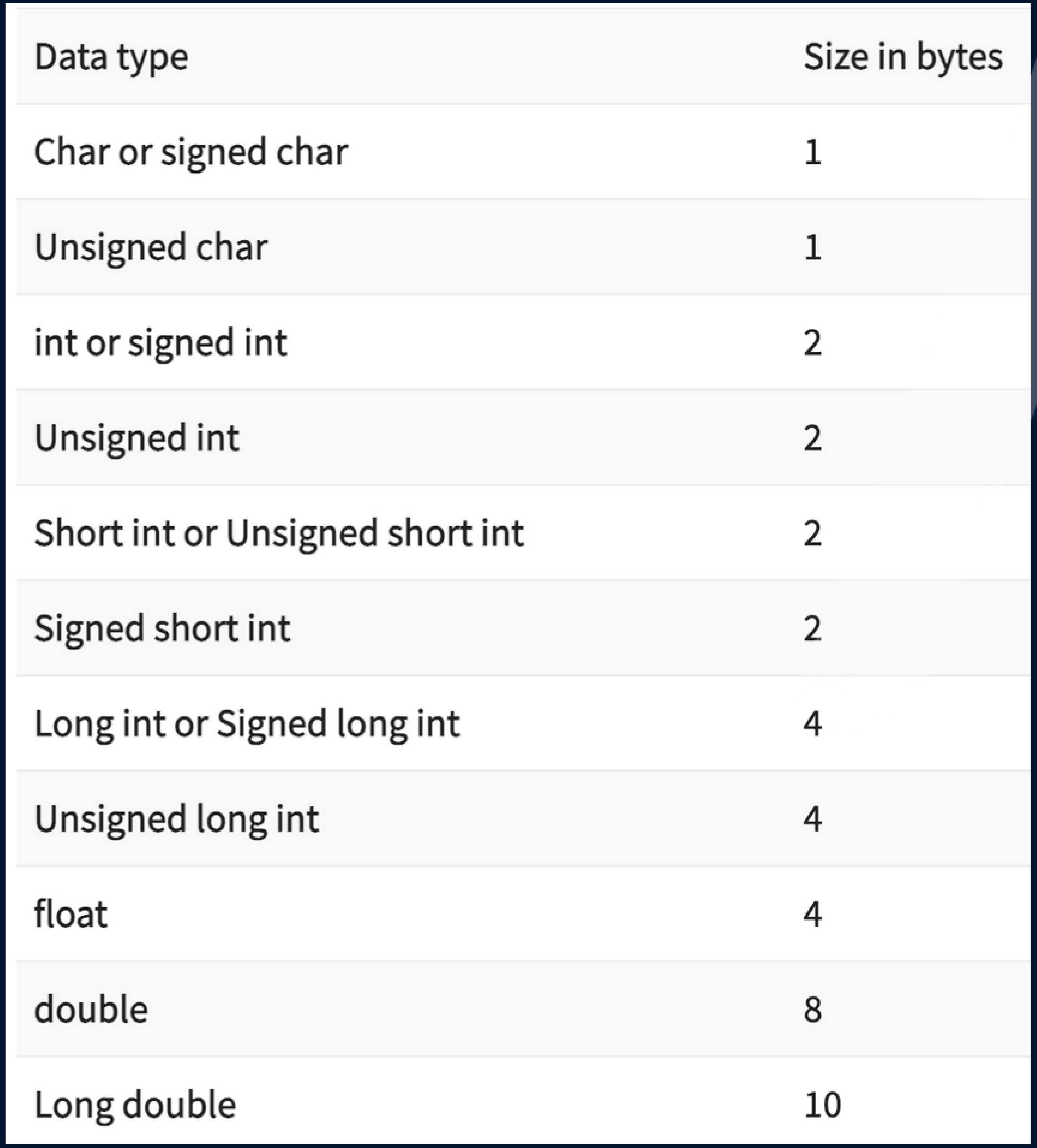
b. 1st character is alphabet or '_'

c. no comma/blank space

d. No other symbol other than '_'

Variables

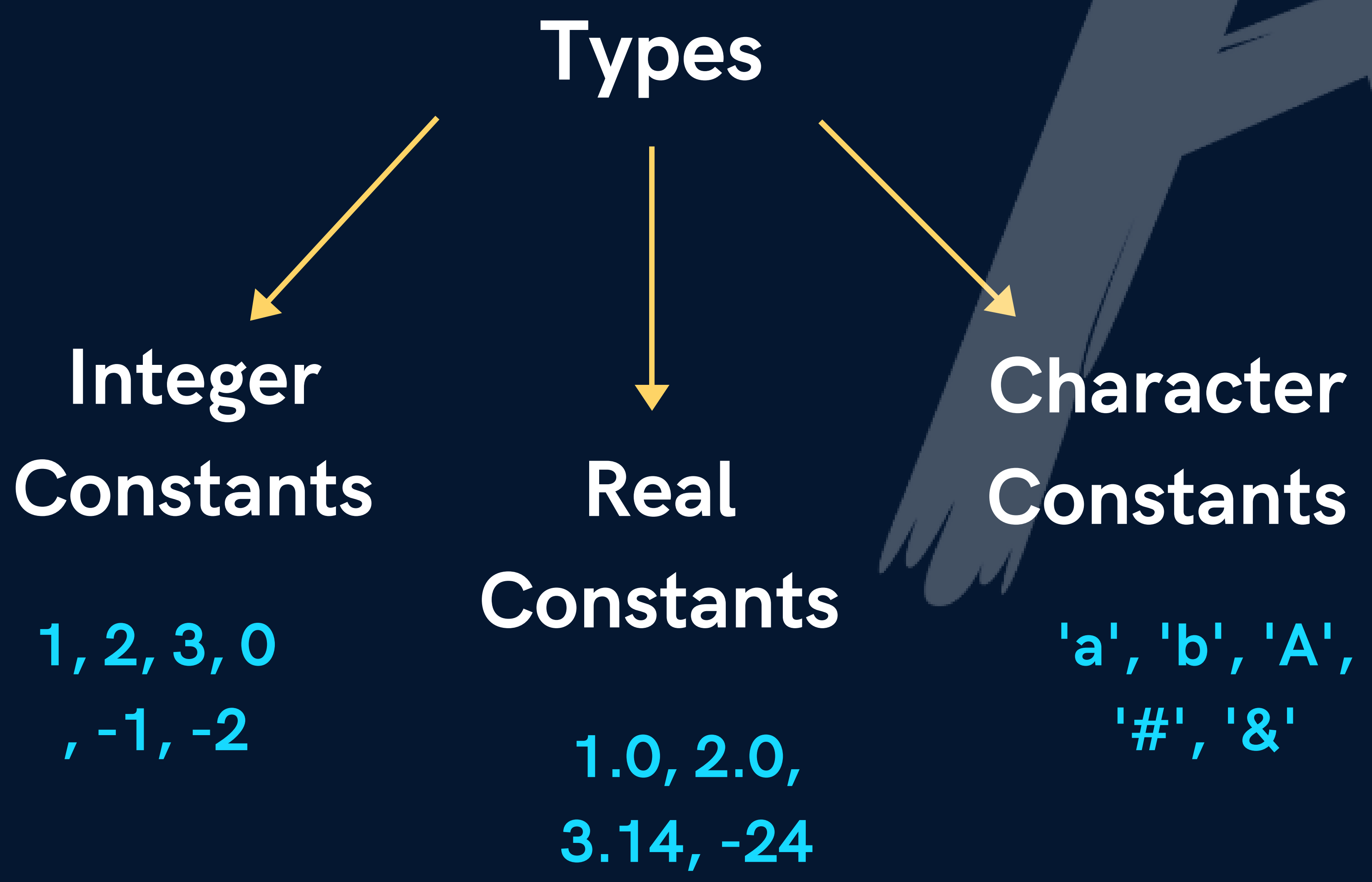
Data Types



Data type	Size in bytes
Char or signed char	1
Unsigned char	1
int or signed int	2
Unsigned int	2
Short int or Unsigned short int	2
Signed short int	2
Long int or Signed long int	4
Unsigned long int	4
float	4
double	8
Long double	10

Constants

Values that don't change(fixed)



Types

Integer Constants

1, 2, 3, 0
, -1, -2

Real Constants

1.0, 2.0,
3.14, -24

Character Constants

'a', 'b', 'A',
'#', '&'

Keywords

Reserved words that have special meaning to the compiler

32 Keywords in C

Keywords

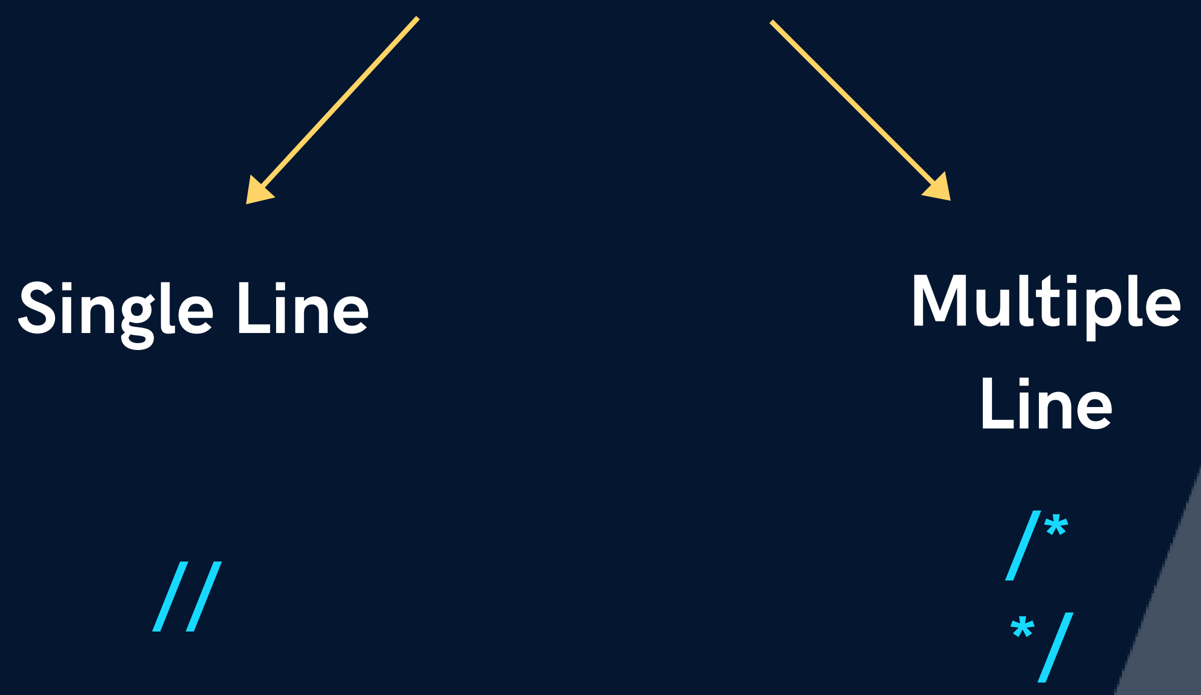
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

Program Structure

```
#include<stdio.h>

int main() {
    printf("Hello World");
    return 0;
}
```

Comments

Lines that are not part of program



Single Line

//

Multiple Line

/*
*/

Output

```
printf(" Hello World ");
```

new line

```
printf(" kuch bhi \n");
```

Output

CASES

1. integers

```
printf(" age is %d ", age);
```

2. real numbers

```
printf(" value of pi is %f ", pi);
```

3. characters

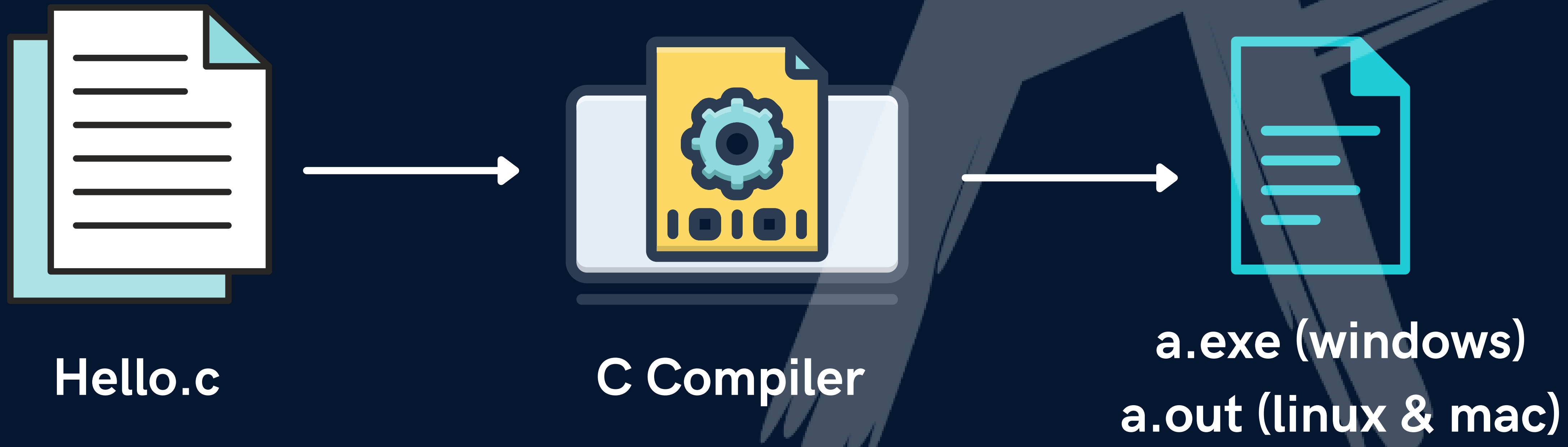
```
printf(" star  looks like this %c ", star);
```

Input

```
scanf(" %d ", &age);
```

Compilation

A computer program that translates C code into machine code



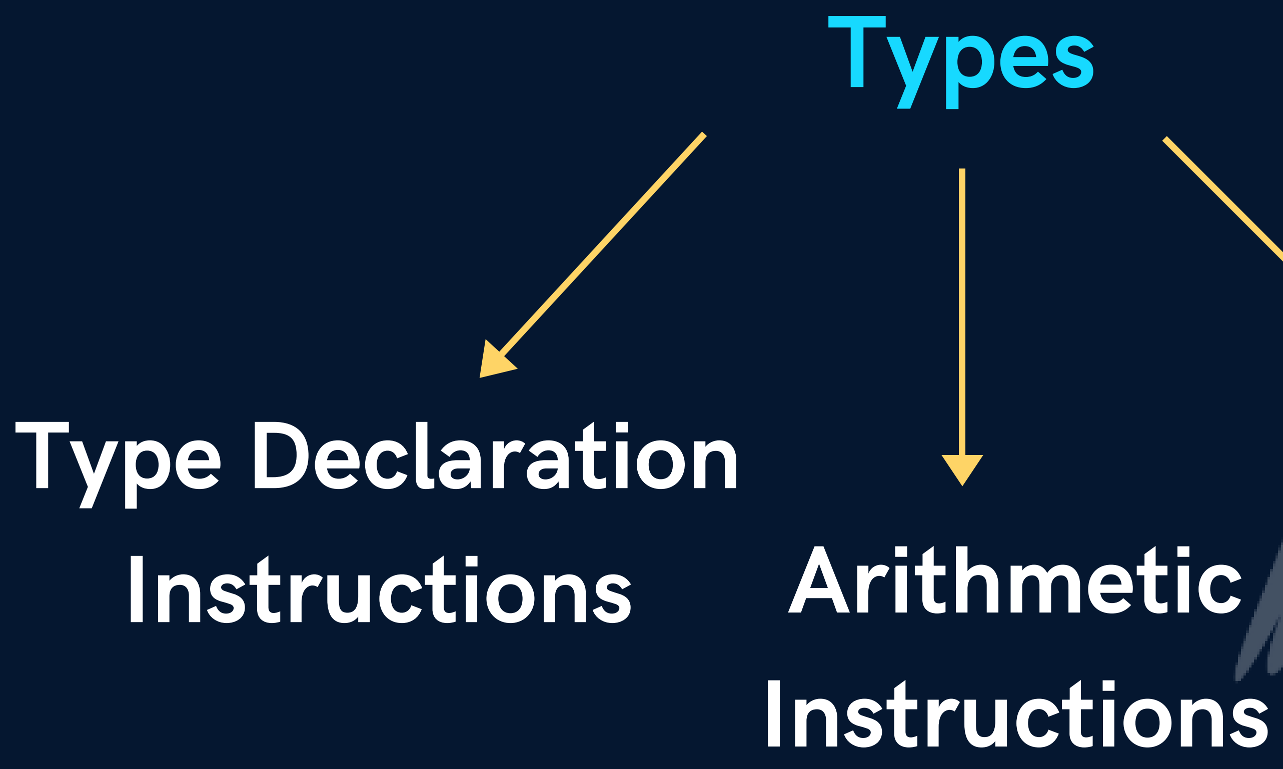
Hello.c

C Compiler

a.exe (windows)
a.out (linux & mac)

Instructions

These are statements in a Program



Types

Type Declaration Instructions

Arithmetic Instructions

Control Instructions

Instructions

Type Declaration Instructions → Declare var before using it

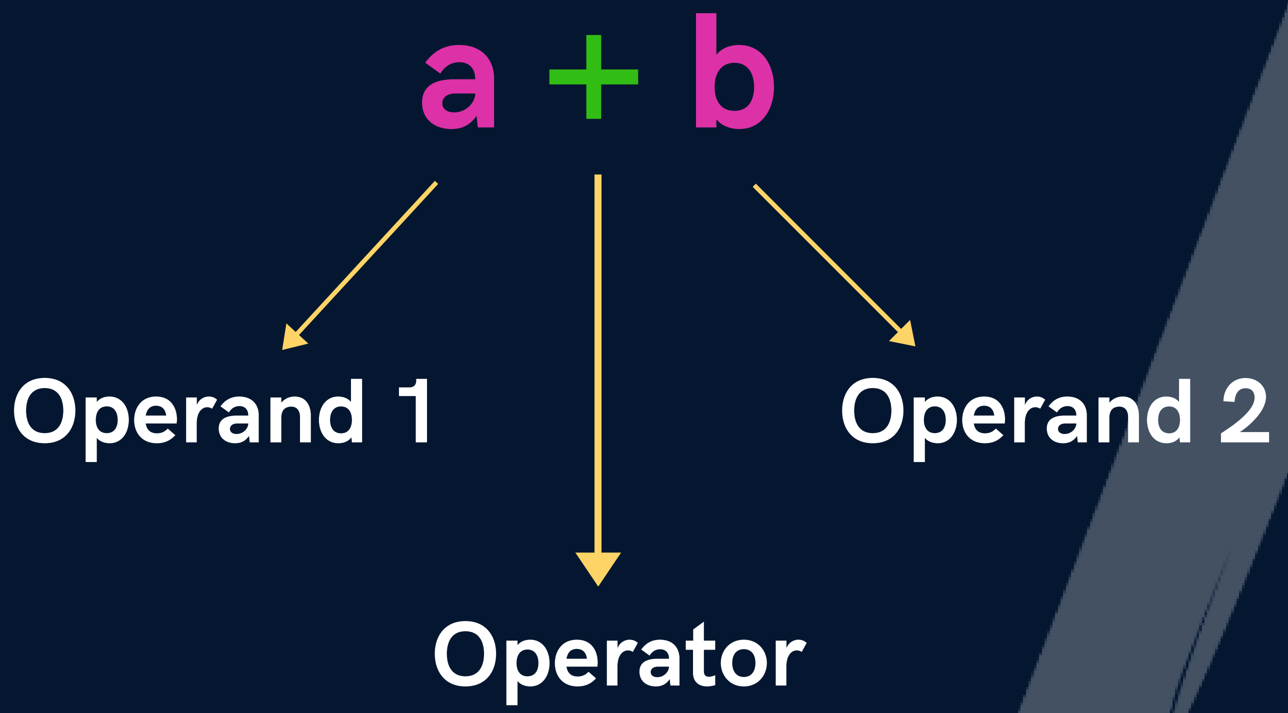
VALID

```
int a = 22;
int b = a;
int c = b + 1;
int d = 1, e;


int a,b,c;
a = b = c = 1;
```

INVALID

```
int a = 22;
int b = a;
int c = b + 2;
int d = 2, e;


int a,b,c = 1;
```

Arithmetic Instructions



a + b

Operand 1

Operator

Operand 2

NOTE - single variable on the LHS

Arithmetic Instructions

VALID	INVALID
a = b + c	b + c = a
a = b * c	a = bc
a = b / c	a = b^c

NOTE - pow(x,y) for x to the power y

Arithmetic Instructions

⭐ Modular Operator %

Returns remainder for int

3 % 2 = 1

-3 % 2 = -1

Arithmetic Instructions

Type Conversion

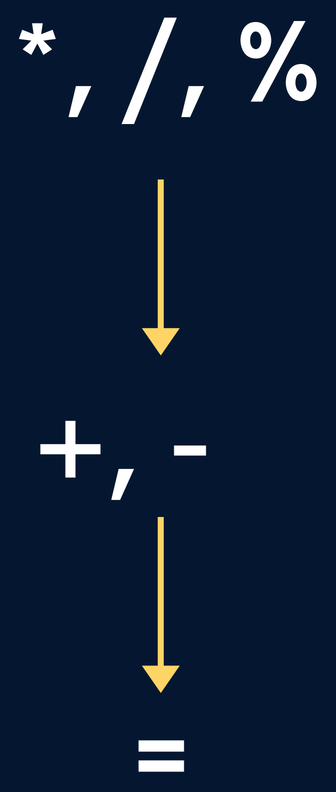
int op int → int

int op float → float

float op float → float

Arithmetic Instructions

Operator Precedence



*, /, %

+, -

=

x = 4 + 9 * 10

x = 4 * 3 / 6 * 2

Arithmetic Instructions

Associativity (for same precedence)

Left to Right

x = 4 * 3 / 6 * 2

Instructions

Control Instructions

Used to determine flow of program

a. Sequence Control

b. Decision Control

c. Loop Control

d. Case Control

Operators

a. Arithmetic Operators

b. Relational Operators

c. Logical Operators

d. Bitwise Operators

e. Assignment Operators

f. Ternary Operator

Operators

Relational Operators

==

>, >=

<, <=

!=

Operators

Logical Operators

&&	AND
||	OR
!	NOT

Operator Precendence

Priority	Operator
1	!
2	*, /, %
3	+, -
4	<, <=, >, >=
5	==, !=
6	&&
7	||
8	=

Operators

Assignment Operators

=

+=

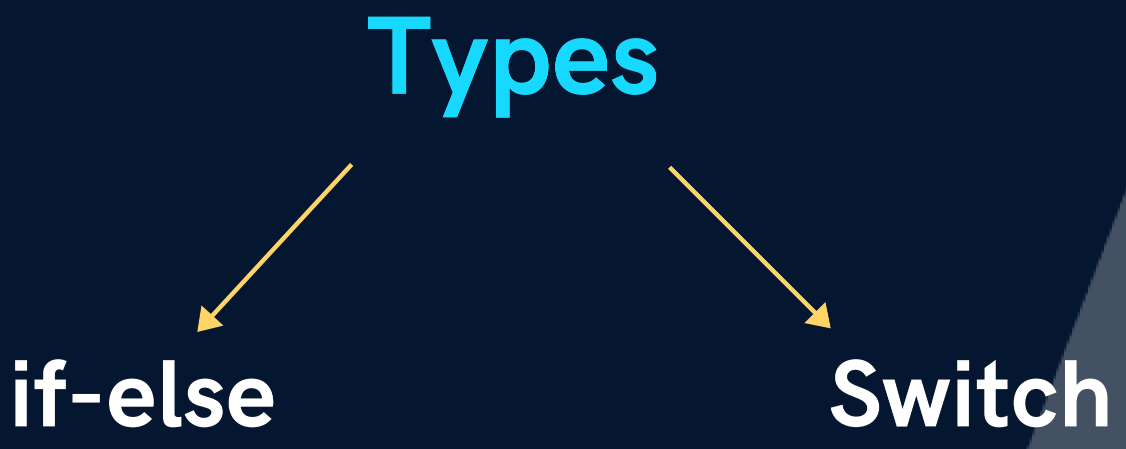
-=

*=

/=

%=

Conditional Statements



Types

if-else

Switch

if-else

```
if(Condition) {
    //do something if TRUE
}
else {
    //do something if FALSE
}
```

Ele is optional block

can also work without {}

else if

```
if(Condition 1) {
    //do something if TRUE
}
else if (Condition 2) {
    //do something if 1st is FALSE & 2nd is TRUE
}
```

Conditional Operators

Ternary

Condition ? doSomething if TRUE : doSomething if FALSE;

give 1 & 0 cases

Conditional Operators

switch

```
switch(number) {
case  C1: //do something
          break;
case C2 : //do something
          break;
default : //do something
}
```

Conditional Operators

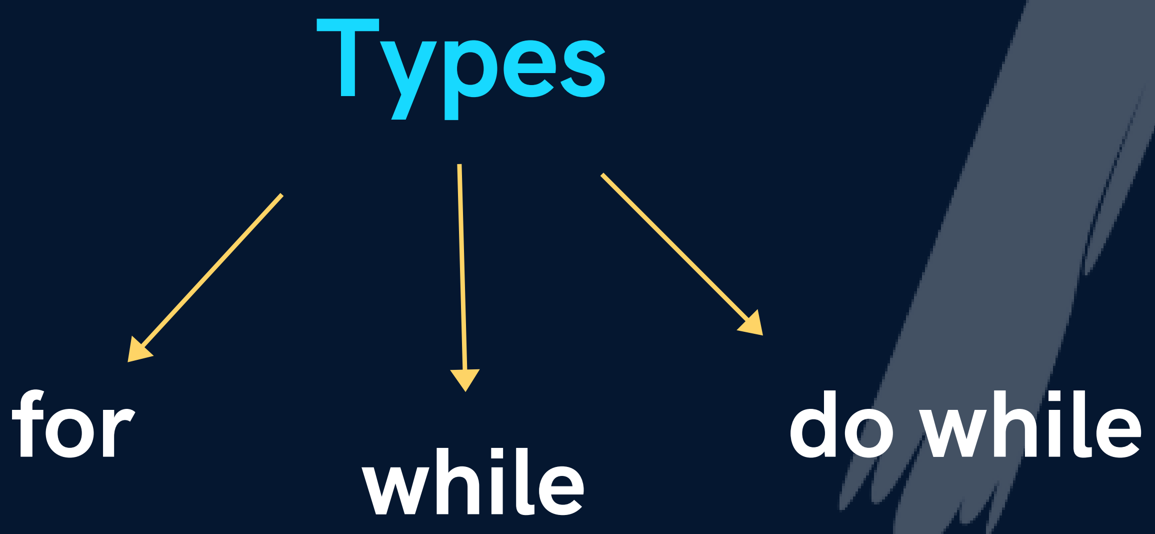
switch Properties

a. Cases can be in any order

b. Nested switch (switch inside switch) are allowed

Loop Control Instructions

To repeat some parts of the program



Types

for

while

do while

for Loop

```
for(initialisation; condition; updation) {

//do something

}
```

Special Things

- Increment Operator

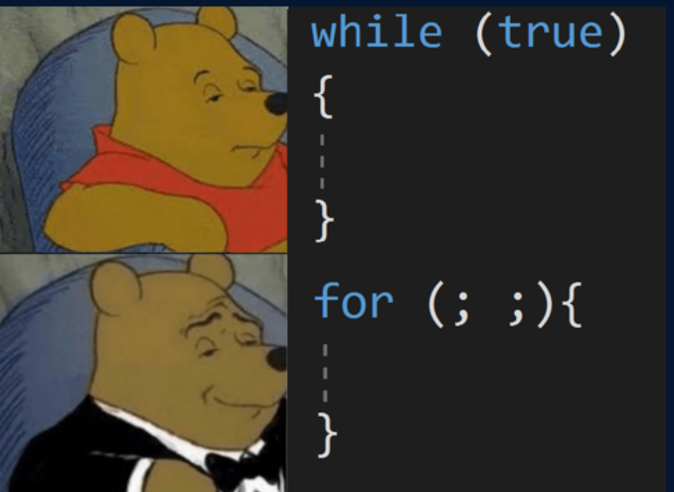
- Decrement Operator

- Loop counter can be float or even character

- Infinite Loop

while Loop

```
while(condition) {

//do something

}
```



```
while (true)
{

}
for (; ;){

}
```

do while Loop

```
do {

//do something

} while(condition);
```

break Statement

↓

exit the loop

continue Statement

↓

skip to next iteration

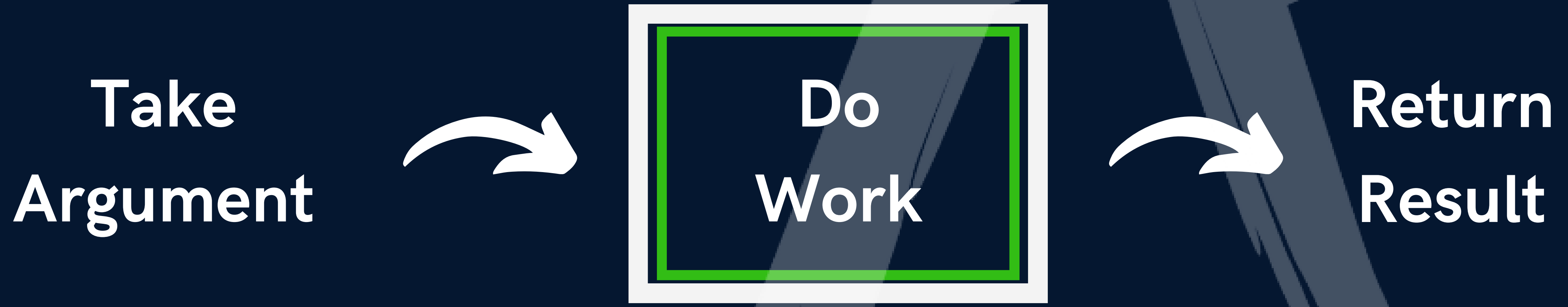
Nested Loops

```
for( .. ) {
    for( .. ) {


    }
}
```

Functions

↓

block of code that performs particular task



Take Argument

Do Work

Return Result

it can be used multiple times

increase code reusability

Syntax 1

Function Prototype

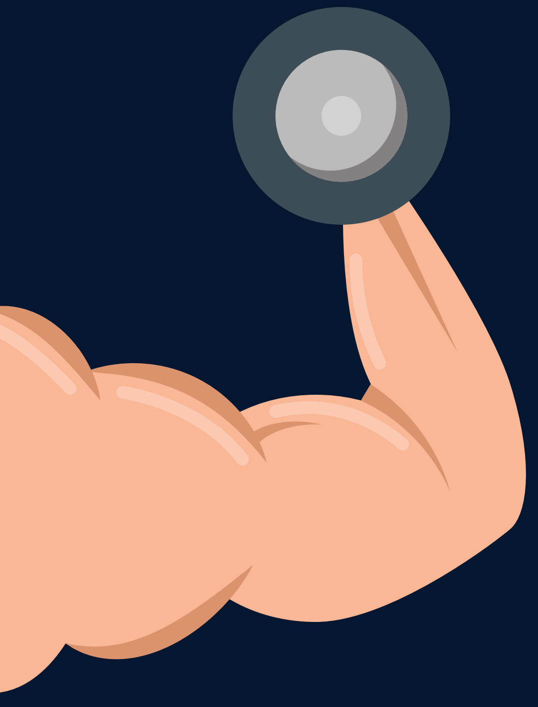
`void printHello( );` ←

> Tell the compiler

Syntax 2

Function Definition

```
void printHello() {
    printf("Hello");
}
```



> Do the Work

Syntax 3

Function Call

```
int main() {
      printHello( );        ←
      return 0;
}
```

> Use the Work

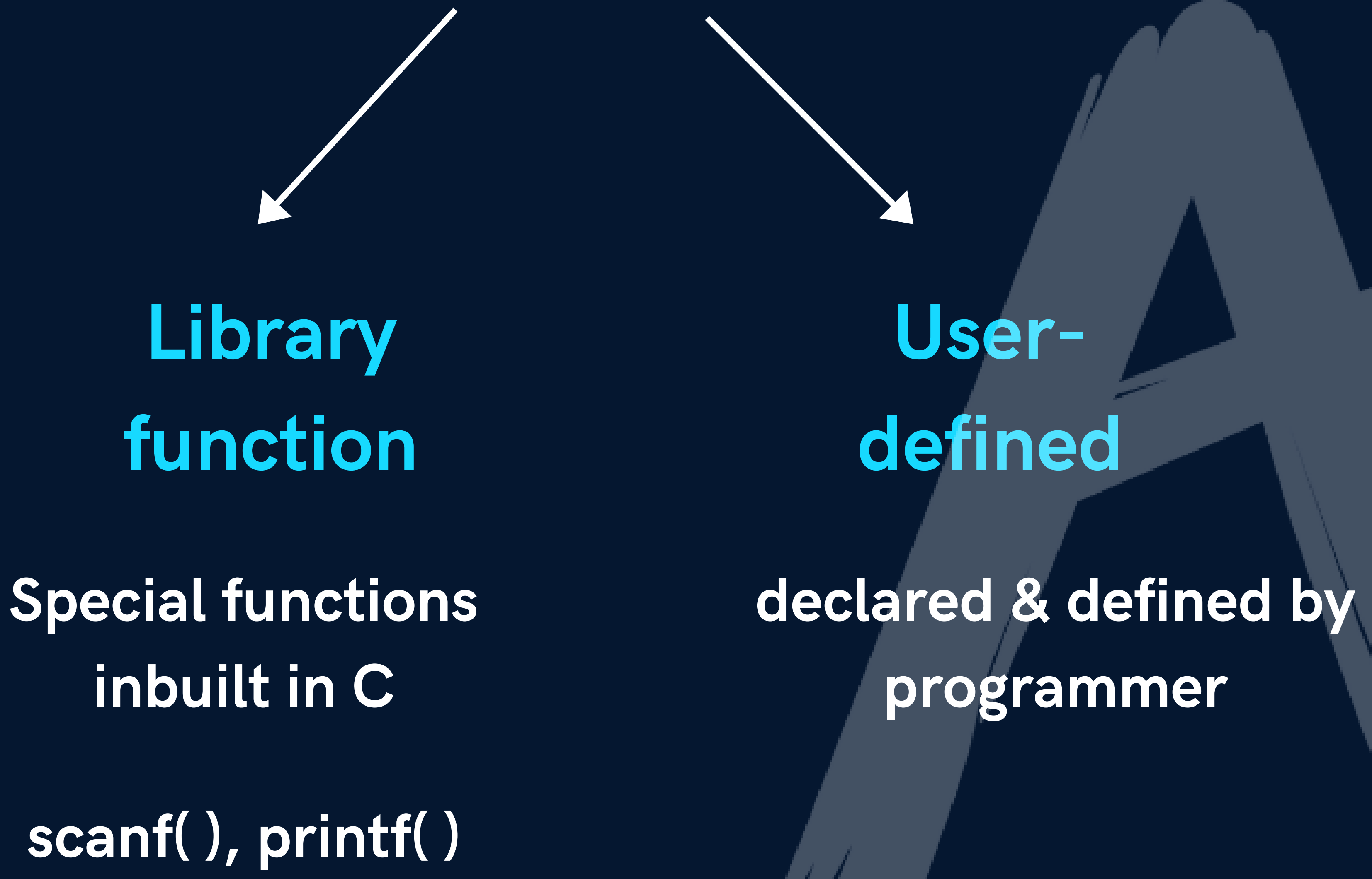
Properties

- Execution always starts from main

- A function gets called directly or indirectly from main

- There can be multiple functions in a program

Function Types



Library function

Special functions inbuilt in C

scanf( ), printf( )

User-defined

declared & defined by programmer

Passing Arguments

functions can take value & give some value

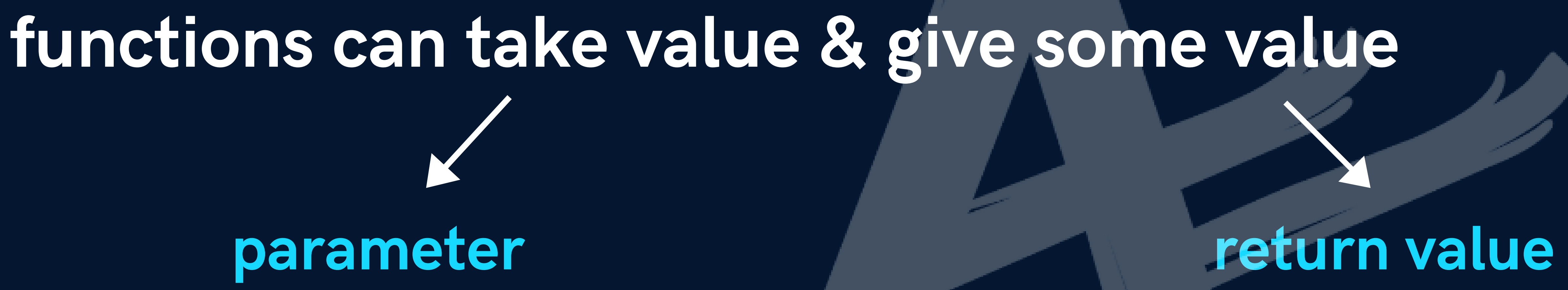
parameter

return value

Passing Arguments

```
void printHello( );

void printTable(int n);

int sum(int a, int b);
```

Passing Arguments



functions can take value & give some value

parameter

return value

Argument v/s Parameter

values that are passed in function call	values in function declaration & definition
used to send value	used to receive value
actual parameter	formal parameters

NOTE

a. Function can only return one value at a time

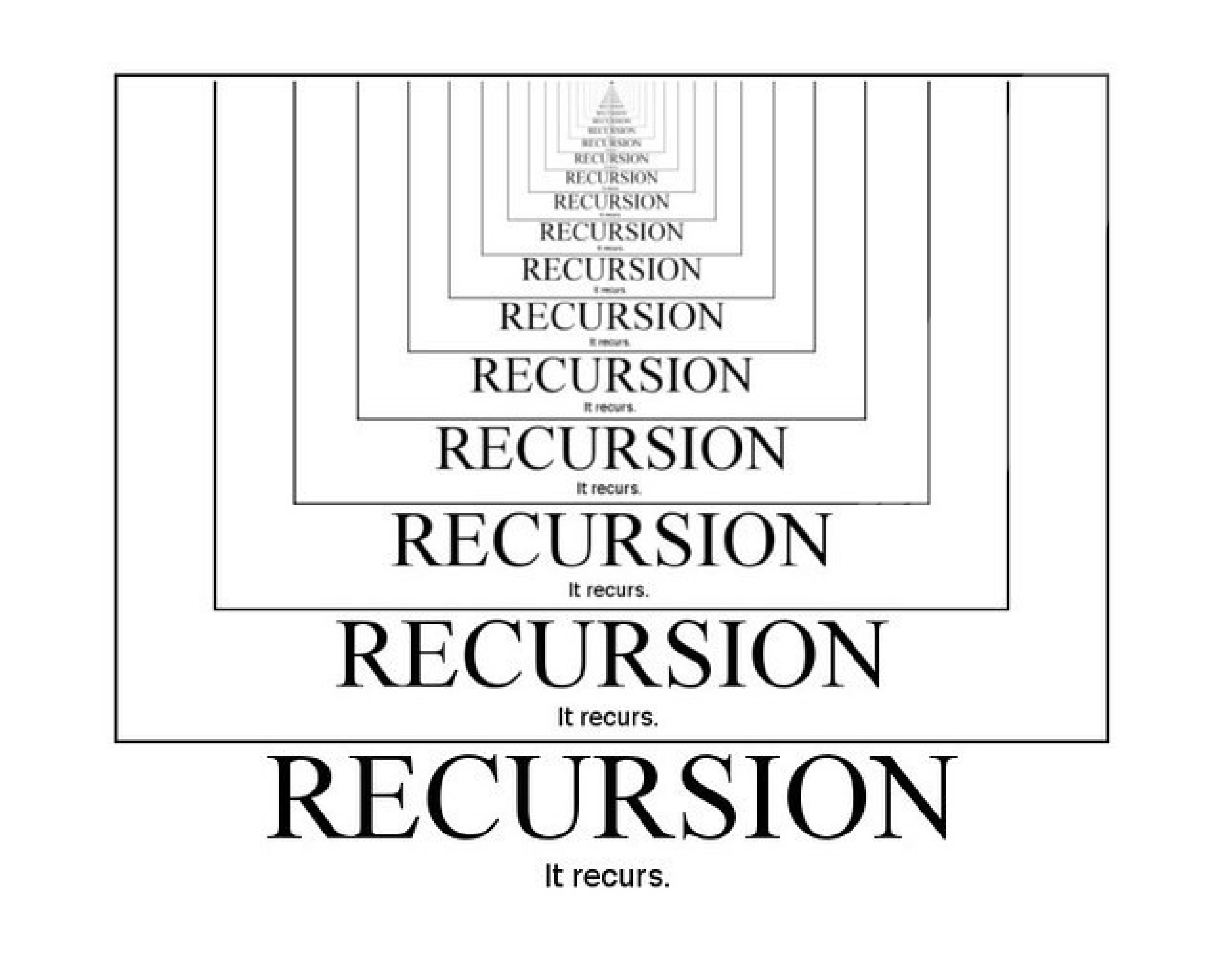
b. Changes to parameters in function don't change the values in calling function.

Because a copy of argument is passed to the function

Recursion

↓

When a **function calls itself**, it's called recursion



Properties of Recursion

a. Anything that can be done with Iteration, can be done with recursion and vice-versa.

b. Recursion can sometimes give the most simple solution.

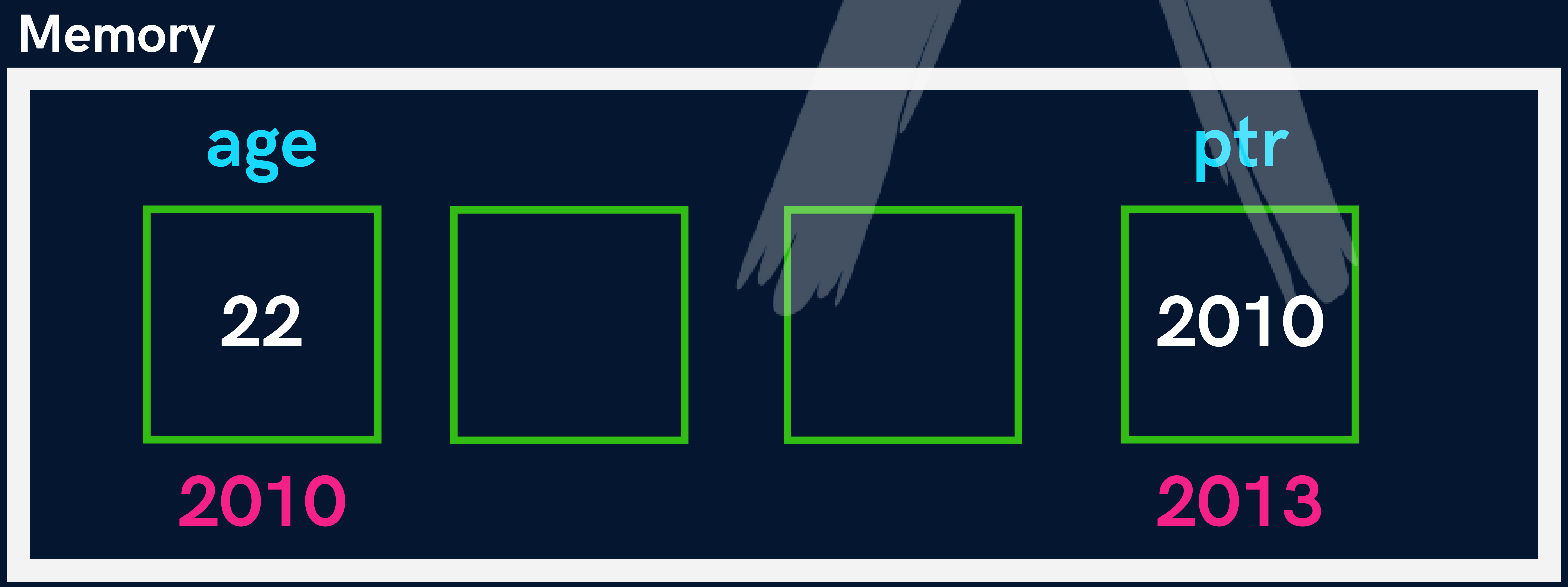
c. Base Case is the condition which stops recursion.

d. Iteration has infinite loop & Recursion has stack overflow

Pointers

↓

A variable that stores the memory address of another variable



Memory

age

22

2010

ptr

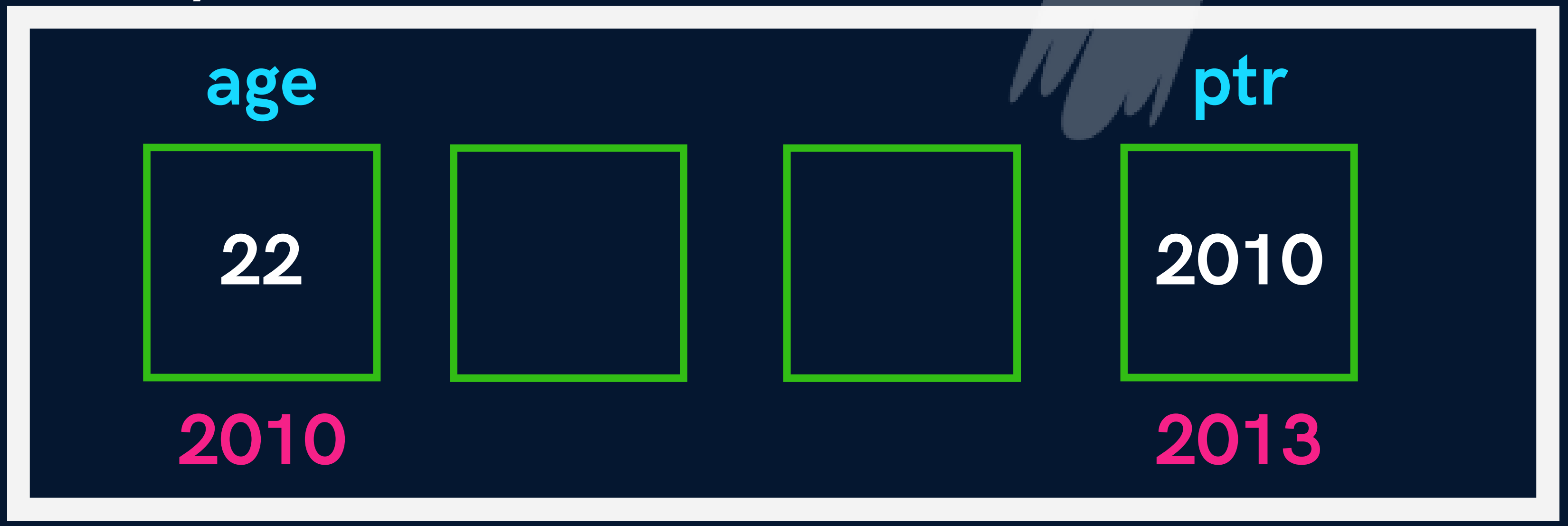
2010

2013

Syntax

```
int age = 22;

int *ptr = &age;

int _age = *ptr;
```

Memory



Declaring Pointers

```
int *ptr;

char *ptr;

float *ptr;
```

Format Specifier

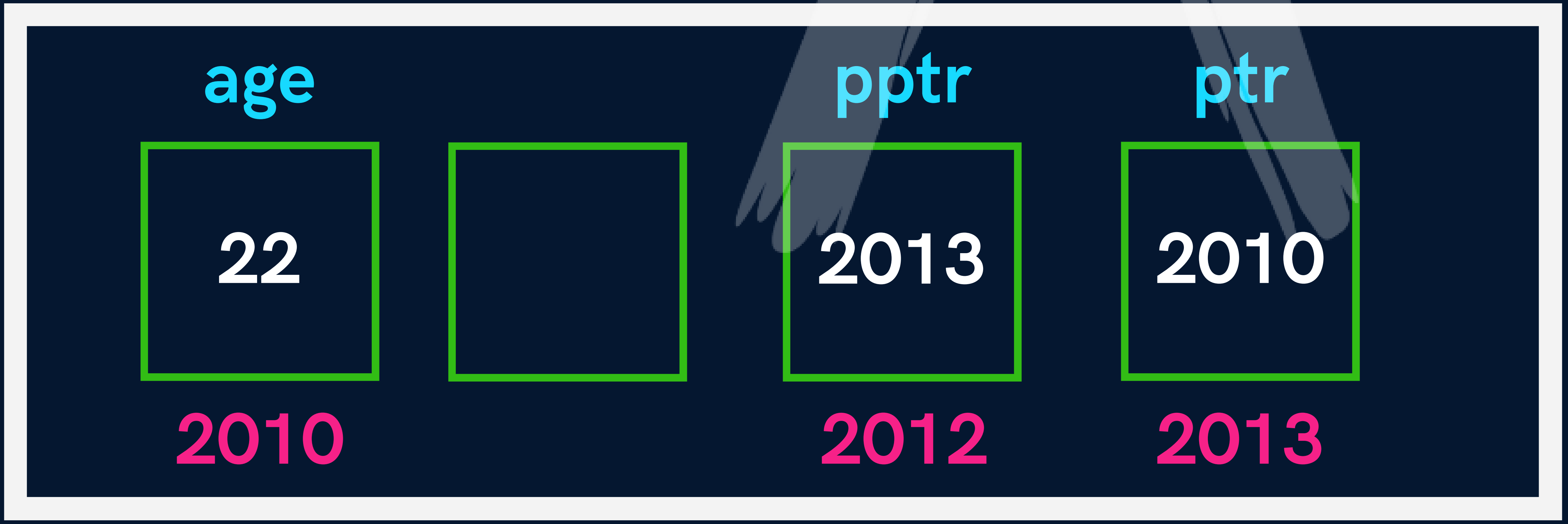
```
printf("%p", &age);

printf("%p", ptr);

printf("%p", &ptr);
```

Pointer to Pointer

↓

A variable that stores the memory address of another pointer

Memory

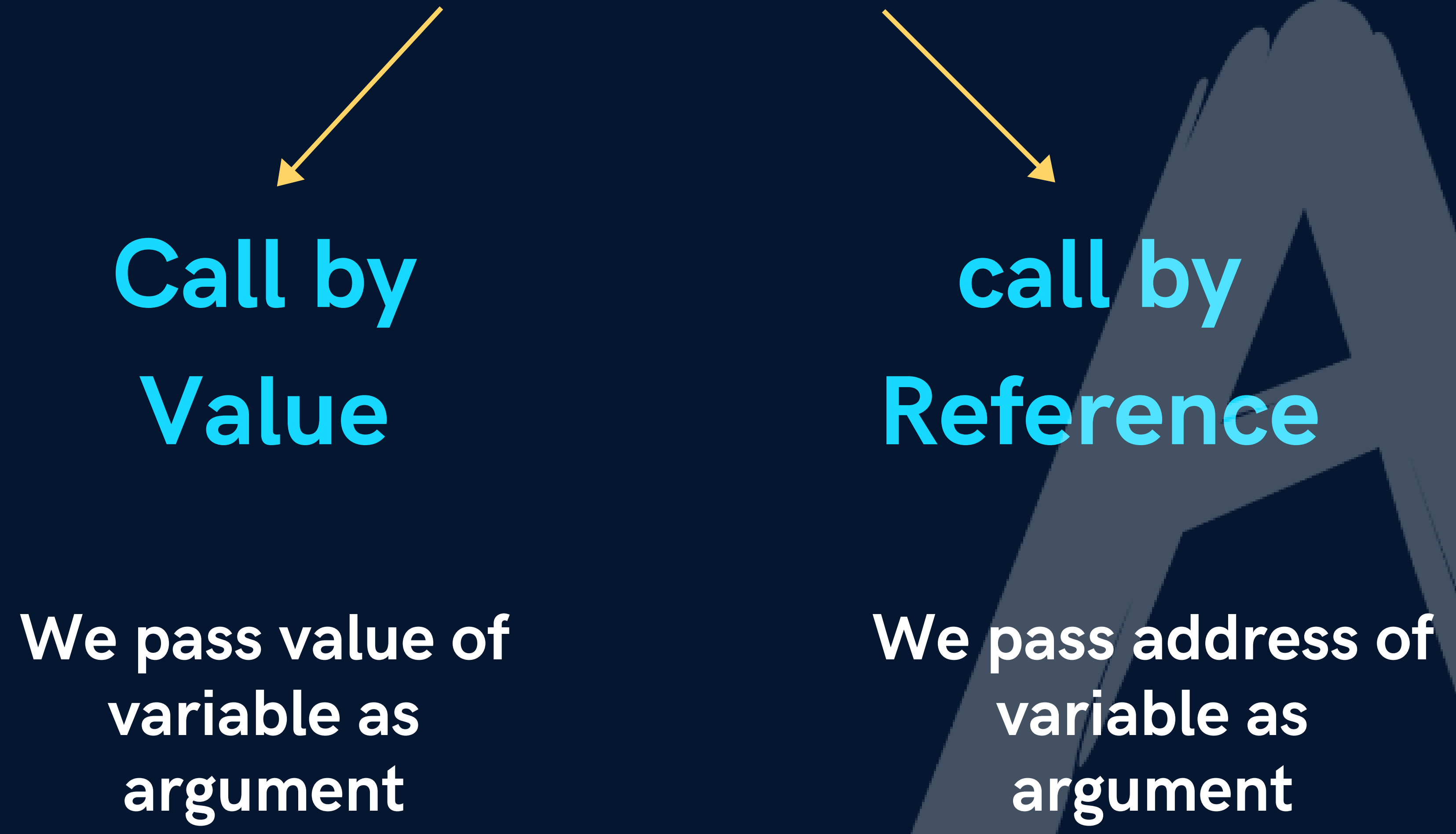


Pointer to Pointer

Syntax

```
int **pptr;

char **pptr;

float **pptr;
```

Pointers in Function Call



Call by Value

We pass value of variable as argument

call by Reference

We pass address of variable as argument

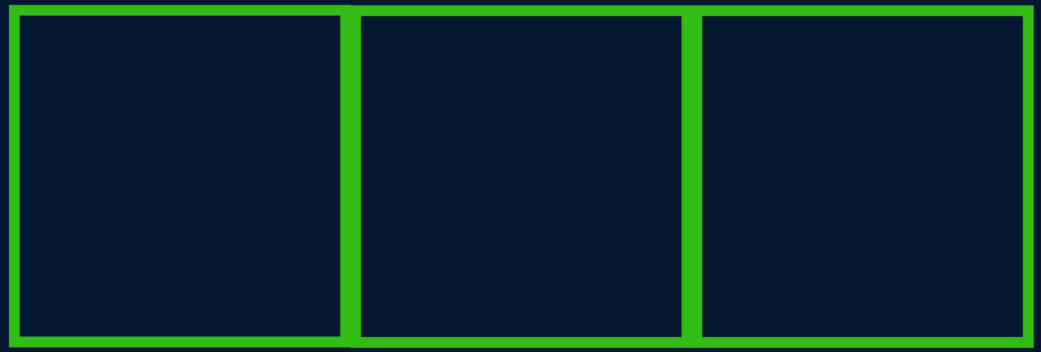
Arrays

↓

Collection of similar data types stored at contiguous memory locations

Syntax

```
int marks[3];

char name[10];

float price[2];
```



Input & Output

```
scanf("%d", &marks[0]);


printf("%d", marks[0]);
```

Inititalization of Array

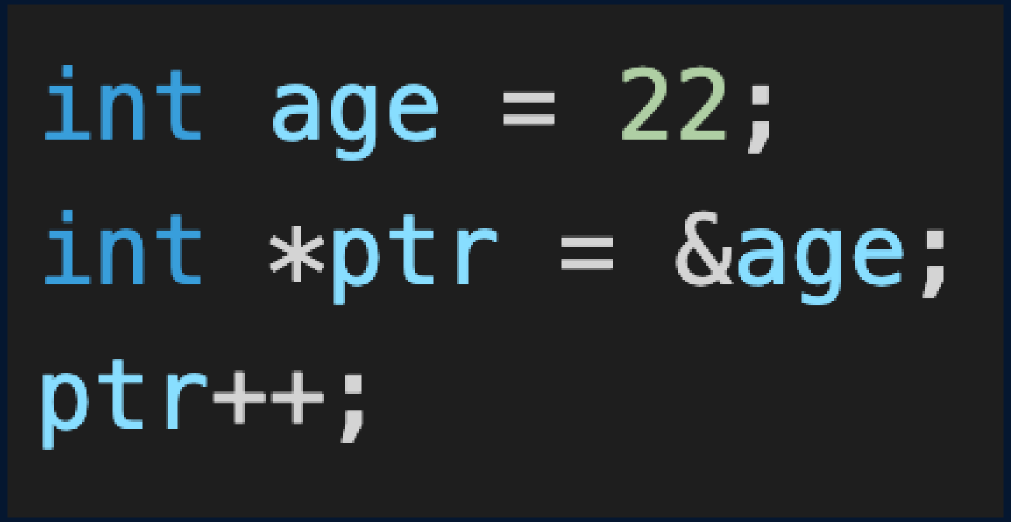
```
int marks[ ] = {97, 98, 89};

int marks[ 3 ] = {97, 98, 89};
```

Memory Reserved :

Pointer Arithmetic

Pointer can be incremented & decremented

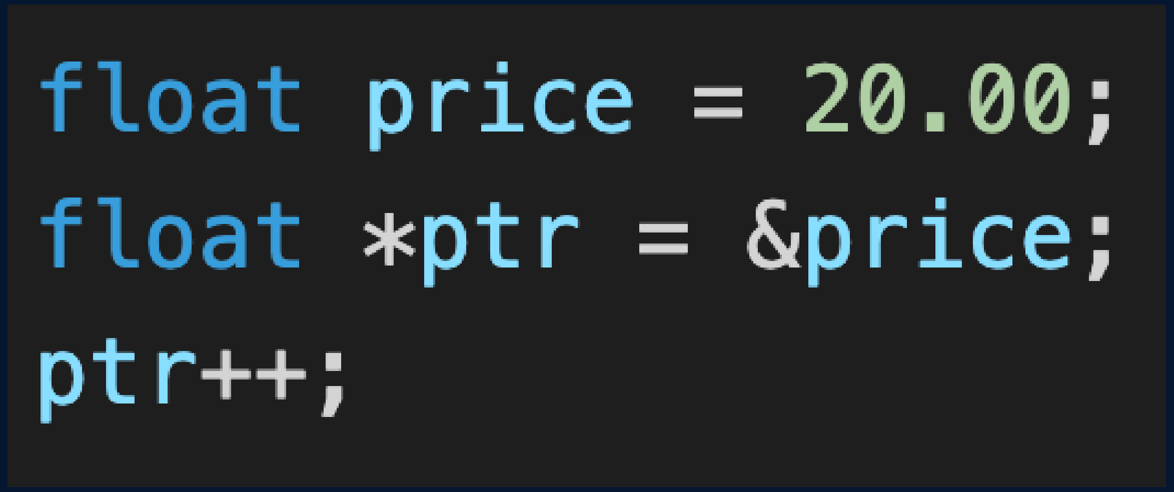
CASE 1



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

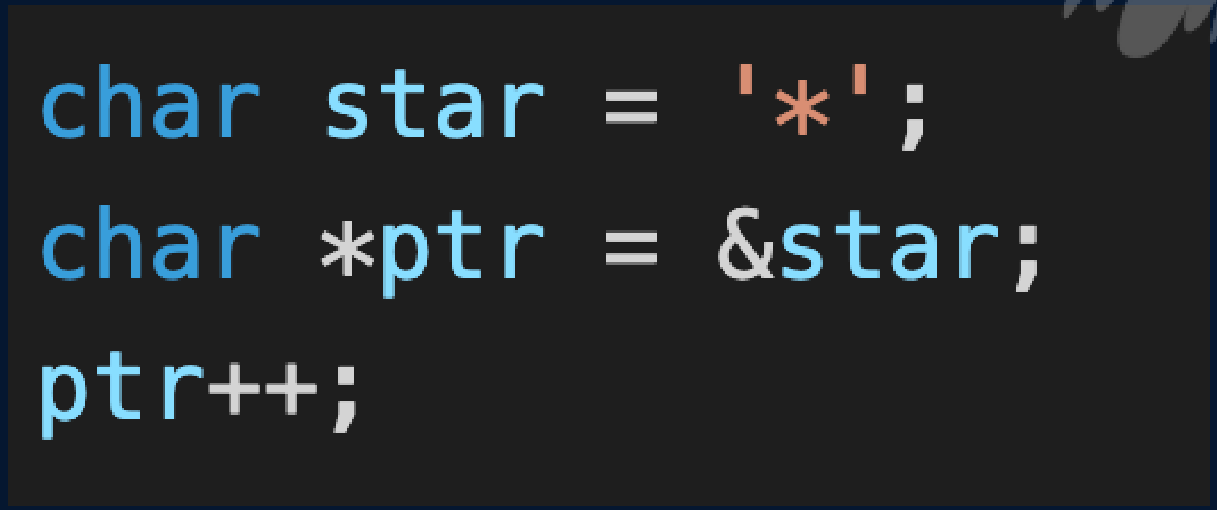
Pointer Arithmetic

CASE 2



```
float price = 20.00;
float *ptr = &price;
ptr++;
```

CASE 3



```
char star = '*';
char *ptr = &star;
ptr++;
```

Pointer Arithmetic

- We can also subtract one pointer from another

- We can also compare 2 pointers

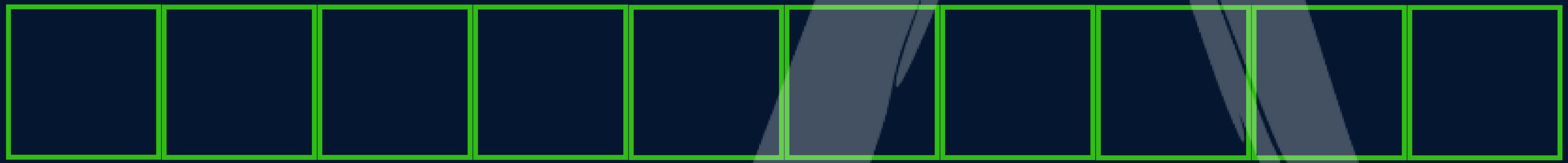
Array is a Pointer

```
int *ptr = &arr[0];
```

```
int *ptr = arr;
```

Traverse an Array

```
int aadhar[10];

int *ptr = &aadhar[0];
```



Arrays as Function Argument

```
//Function Declaration
void printNumbers (int arr[ ], int n)
                    OR
void printNumbers (int *arr, int n)


//Function Call
printNumbers(arr, n);
```

Multidimensional Arrays

2 D Arrays

```
int arr[ ][ ] = { {1, 2}, {3, 4} };    //Declare


//Access

arr[0][0]

arr[0][1]

arr[1][0]

arr[1][1]
```

Strings

↓

A character array terminated by a '\0' (null character)

null character denotes string termination

EXAMPLE

```
char name[ ] = {'S', 'H', 'R', 'A', 'D', 'H', 'A', '\0'};

char class[ ] = {'A', 'P', 'N', 'A', ' ', 'C', 'O', 'L', 'L', 'E', 'G', 'E', '\0'};
```

Initialising Strings

```
char name[ ] = {'S', 'H', 'R', 'A', 'D', 'H', 'A','\0'};

char name[ ] = "SHRADHA";



char class[ ] = {'A', 'P', 'N', 'A', ' ', 'C', 'O', 'L', 'L', 'E', 'G', 'E', '\0'};

char class[ ] = "APNA COLLEGE";
```

What Happens in Memory?

```
char name[ ] = {'S', 'H', 'R', 'A', 'D', 'H', 'A','\0'};

char name[ ] = "SHRADHA";
```

name

S	H	R	A	D	H	A	\0
2000	2001	2002	2003	2004	2005	2006	2007

String Format Specifier

↓

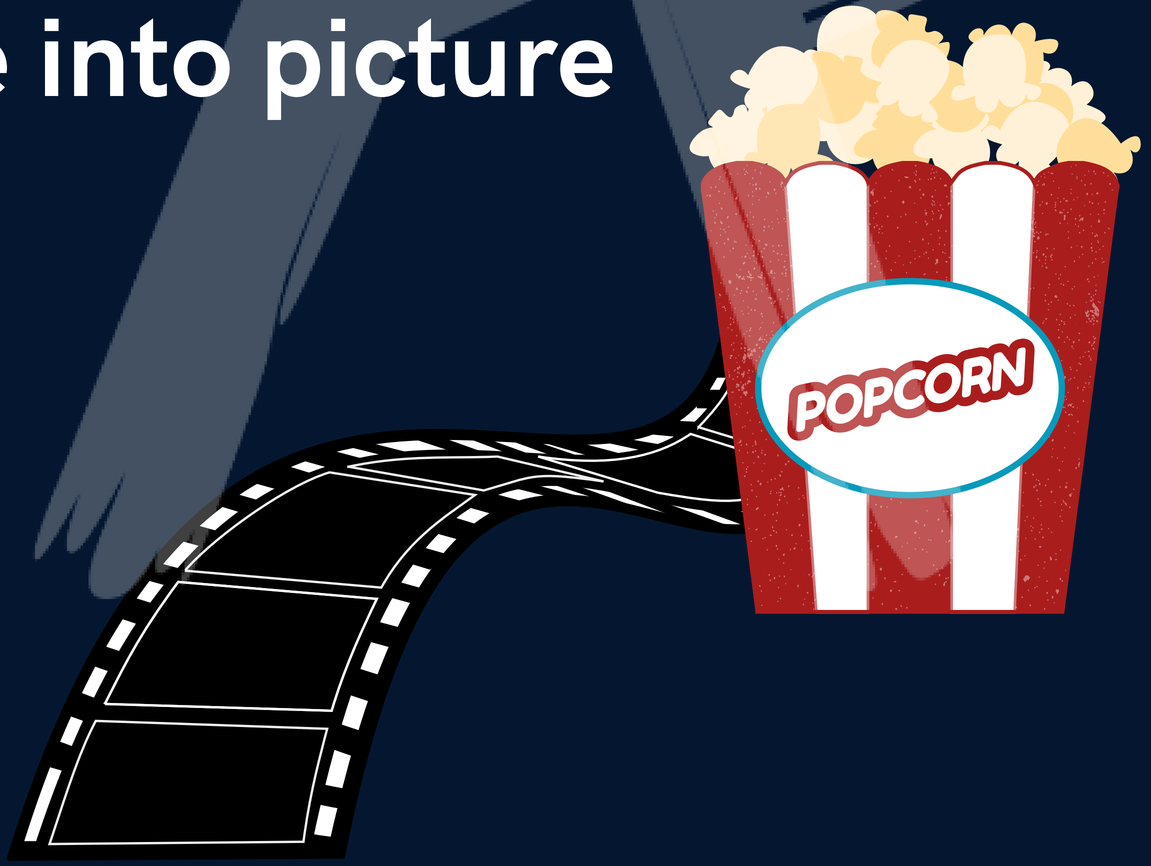
"%s"

```
char name[ ] = "Shradha";

printf("%s", name);
```

IMPORTANT

scanf( ) cannot input multi-word strings with spaces

Here,
gets( ) & puts( ) come into picture



String Functions

gets(str) → Dangerous & Outdated

input a string
(even multiword)

puts(str)

output a string

fgets(str, n, file)

stops when n-1
chars input or new
line is entered

String using Pointers

```
char *str = "Hello World";
```

Store string in memory & the assigned address is stored in the char pointer 'str'

```
char *str = "Hello World";    //can be reinitialized
```

```
char str[ ] = "Hello World";
//cannot be reinitialized
```

Standard Library Functions

↓

<string.h>

1 strlen(str)

count number of characters excluding '\0'

Standard Library Functions

↓

<string.h>

2 strcpy(newStr, oldStr)

copies value of old string to new string

Standard Library Functions

↓

<string.h>

3 strcat(firstStr, secStr)

concatenates first string with second string

firstStr should be large enough

Standard Library Functions

↓

<string.h>

4 strcpm(firstStr, secStr)

Compares 2 strings & returns a value

0 -> string equal

positive -> first > second (ASCII)

negative -> first < second (ASCII)

Structures

↓

a collection of values of different data types

EXAMPLE

For a student store the following :

name (String)

roll no (Integer)

cgpa (Float)

Syntax

```
struct student {

    char name[100];

    int roll;

    float cgpa;

};
```

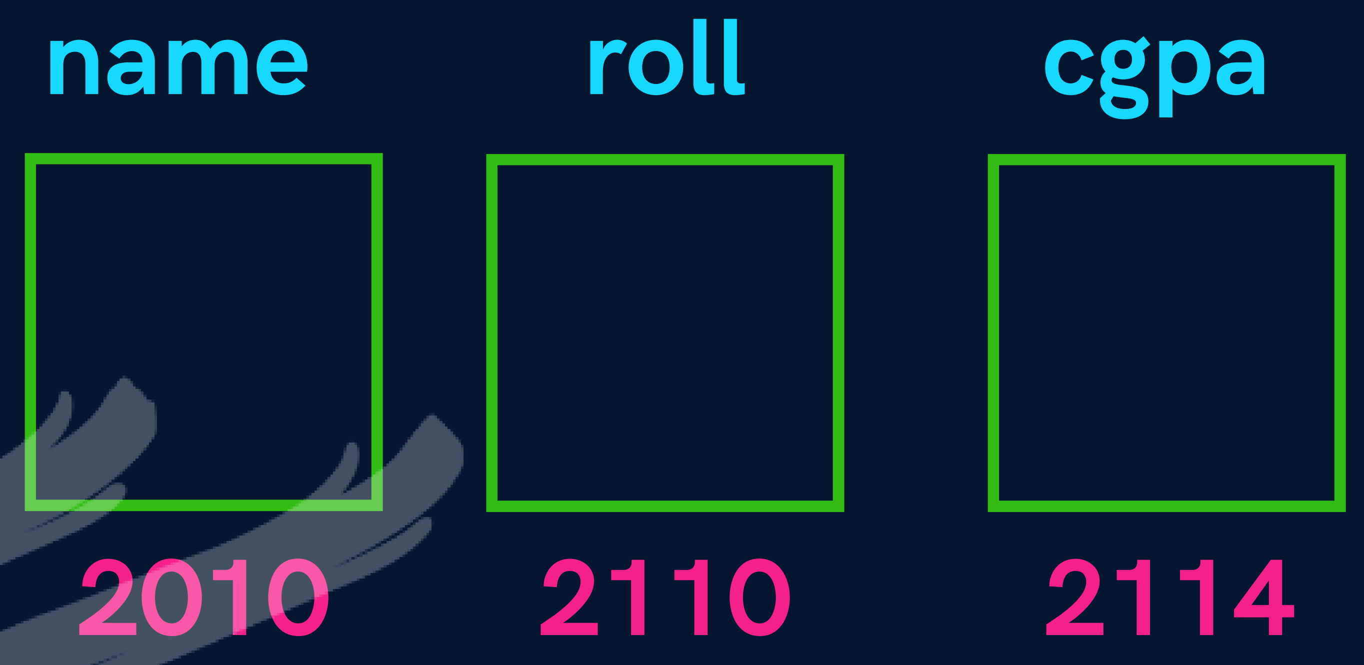
```
struct student s1;

s1.cgpa = 7.5;
```

Syntax

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

Structures in Memory

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



name roll cgpa

2010 2110 2114

structures are stored in contiguous memory locations

Benefits of using Structures

- Saves us from creating too many variables

- Good data management/organization

Array of Structures

```
struct student ECE[100];

struct student COE[100];

struct student IT[100];
```

ACCESS

```
IT[0].roll = 200;
IT[0].cgpa = 7.6;
```

Initializing Structures

```
struct student s1 = { "shradha", 1664, 7.9};

struct student s2 = { "rajat", 1552, 8.3};

struct student s3 = { 0 };
```

Pointers to Structures

```
struct student s1;

struct student *ptr;

ptr =&s1;
```

Arrow Operator

(*ptr).code ⟷ ptr->code

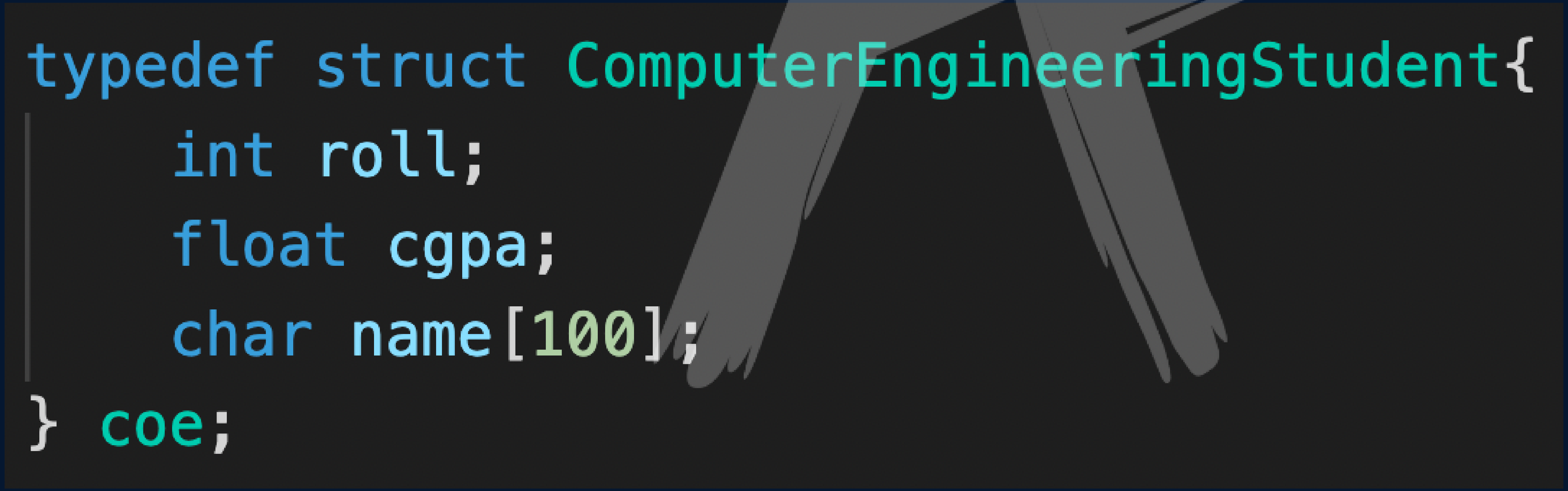
Passing structure to function

```
//Function Prototype
void printInfo(struct student s1);
```

typedef Keyword

↓

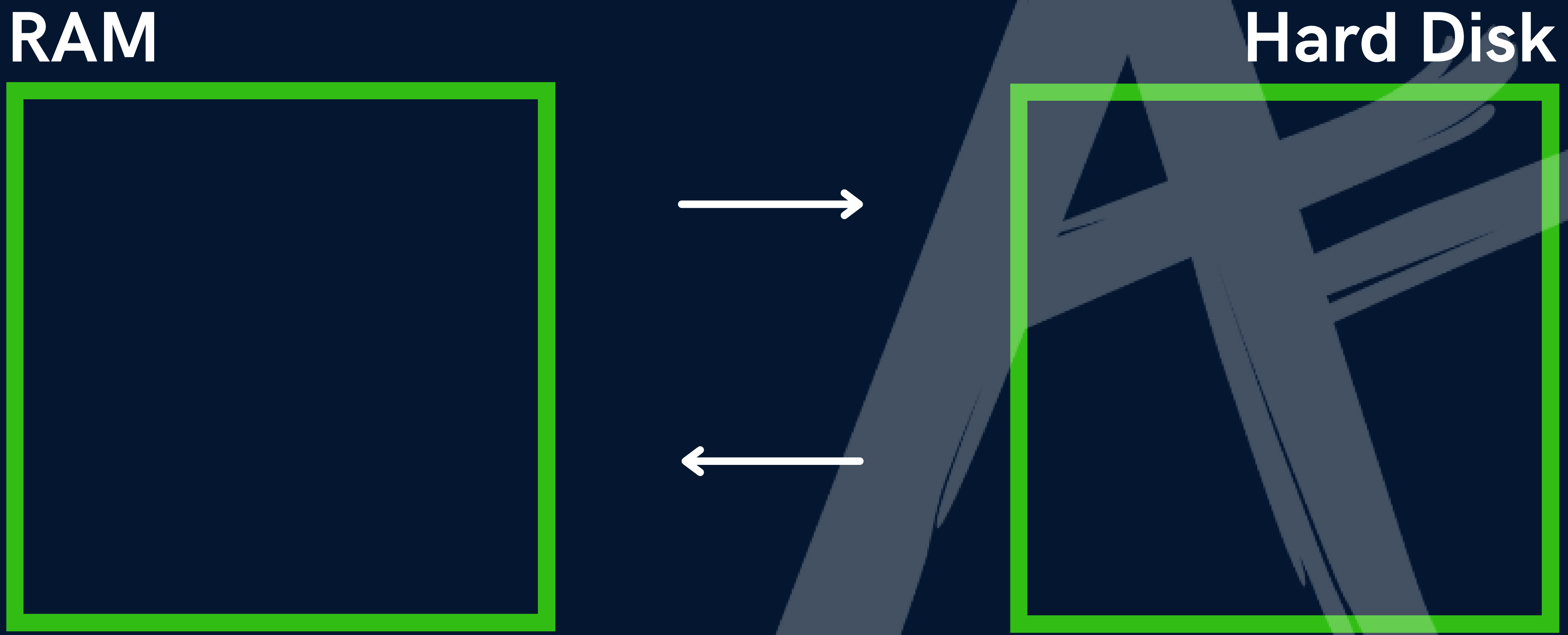
used to create alias for data types



```c
typedef struct ComputerEngineeringStudent{
    int roll;
    float cgpa;
    char name[100];
} coe;
```

coe student1;

File IO



File IO

FILE - container in a storage device to store data

- RAM is volatile
- Contents are lost when program terminates
- Files are used to persist the data

Operation on Files

Create a File

Open a File

Close a File

Read from a File

Write in a File

Types of Files

Text Files	Binary Files
textual data	binary data
.txt, .c	.exe, .mp3, .jpg

File Pointer

FILE is a (hidden)structure that needs to be created for opening a file

A FILE ptr that points to this structure & is used to access the file.

```
FILE *fptr;
```

Opening a File

```
FILE *fptr;

fptr = fopen("filename", mode);
```

Closing a File

```
fclose(fptr);
```

File Opening Modes

"r"	open to read
"rb"	open to read in binary
"w"	open to write
"wb"	open to write in binary
"a"	open to append

BEST Practice

Check if a file exists before reading from it.

Reading from a file

```c
char ch;

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

Writing to a file

```
char ch = 'A';

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

Read & Write a char

```
fgetc(fptr)

fputc( 'A', fptr)
```

EOF (End Of File)

fgetc returns EOF to show that the file has ended

Dynamic Memory Allocation

↓

It is a way to allocate memory to a data structure during the runtime.

We need some functions to allocate
& free memory dynamically.

Functions for DMA

a. malloc( )

b. calloc( )

c. free( )

d. realloc( )

malloc( )

memory allocation

takes number of bytes to be allocated

& returns a pointer of type void

```
ptr = (*int) malloc(5 * sizeof(int));
```

calloc( )

continuous allocation

initializes with 0

```
ptr = (*int) calloc(5, sizeof(int));
```

free( )

We use it to free memory that is allocated using malloc & calloc

```
free(ptr);
```

realloc( )

reallocate (increase or decrease) memory using the same pointer & size.

```
ptr = realloc(ptr, newSize);
```