Embedded C

Understanding Embedded aspects of C programming

Team Emertxe

Goals

Goals & Objectives

- To cover the depth of the C language from its core.
- To cover all the functionalities of the language, more important from the industry perspective.
- By the end, students are expected to have made their concepts very clear and become comfortable using the language to any level.
- Mastering the basics, Playing with all kinds of Pointers, Var Args, Recursions, Functions, Files, Preprocessor Directives, and many other industry relevant topics.
- As an overall experience, it will take you through all the nitty-gritties of C through a well organized set of examples and exercises

ToC

- Introduction
- Brief History
- Programming languages
- The C standard
- Important Characteristics
- Keywords

Introduction

Language

- Simply put, a language is a stylized communication technique.
- The more varied vocabulary it has, the more expressive it becomes. Then, there is grammar to make meaningful communication statements out of it.
- Similarly, being more specific, a programming language is a stylized communication technique intended to be used for controlling the behavior of a machine (often a computer),by expressing ourselves to the machine.
- Like the natural languages, programming languages too, have syntactic rules (to form words) and semantic rules (to form sentences), used to define the meaning.

Programming Languages: Types

- Procedural
- Object Oriented
- Functional
- Logical
- And many more

Brief History

- Prior to C, most of the computer languages (such as Algol) were academic oriented, unrealistic and were generally defined by committees.

- Since such languages were designed having application domain in mind, they could not take the advantages of the underlying hardware and if done, were not portable or efficient under other systems.

- It was thought that a high-level language could never achieve the efficiency of assembly language.

Portable, efficient and easy to use language was a dream.

Brief History ...

- It was a revolutionary language and shook the computer world with its might. With just 32 keywords, C established itself in a very wide base of applications.

- It has lineage starting from CPL, (Combined Programming Language) a never implemented language.

- Martin Richards implemented BCPL as a modified version of CPL. Ken Thompson further refined BCPL to a language named as B.

- Later Dennis M. Ritchie added types to B and created a language, what we have as C, for rewriting the UNIX operating system.

Programming languages

- There are various types of programming languages, compared on various parameters
- From Embedded system engineer's view it should be seen how close or how much away from the hardware the language is
- Based on that view programming languages can be categorized into three areas:

  Assembly language (ex: 8051)

  Middle level language (ex: C)

  High level / Scripting language (ex: Shell)

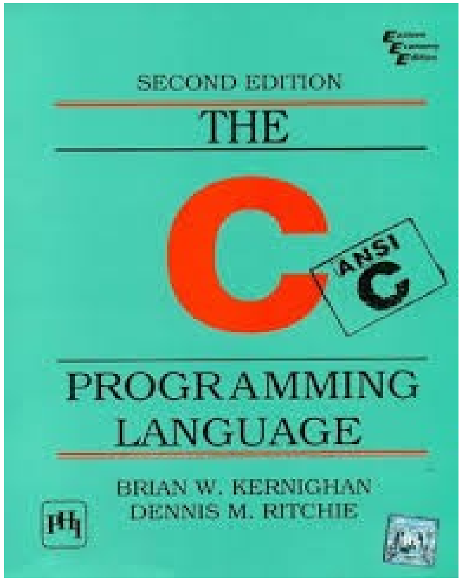
- Each programming language offers some benefits with some shortcomings
- Depending on the need of the situation appropriate language needs to be chosen
- This make language selection is a key criteria when it comes to building real time products!

A comparison

Language parameter	Assembly	C	Shell
Speed	High	Medium	Medium
Portability	Low	Medium	High
Maintainability	Low	Medium	High
Size	High	Medium	Low
Easy to learn	Low	Medium	High

Shell or any scripting language is also called as ‘interpreted’ language as it doesn’t go thru compilation phase. This is to keep the language simple as the purpose is different than other languages.

The C Standard



- “The C programming language” book served as a primary reference for C programmers and implementers alike for nearly a decade.
- However it didn’t define C perfectly and there were many ambiguous parts in the language.
- As far as the library was concerned, only the C implementation in UNIX was close to the ’standard’.
- So many dialects existed for C and it was the time the language has to be standardized and it was done in 1989 with ANSI C standard.
- Nearly after a decade another standard, C9X, for C is available that provides many significant improvements over the previous 1989 ANSI C standard.

Important Characteristics

- C is considered as a middle level language.
- C can be considered as a pragmatic language.
- It is indented to be used by advanced programmers, for serious use, and not for novices and thus qualify less as an academic language for learning.
- Gives importance to curt code.
- It is widely available in various platforms from mainframes to palmtops and is known for its wide availability.

Important Characteristics...

- It is a general-purpose language, even though it is applied and used effectively in various specific domains.
- It is a free-formatted language (and not a strongly-typed language)
- Efficiency and portability are the important considerations.
- Library facilities play an important role

Keyword

- In programming, a keyword is a word that is reserved by a program because the word has a special meaning
- Keywords can be commands or parameters
- Every programming language has a set of keywords that cannot be used as variable names
- Keywords are sometimes called reserved names

Keywords ...

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

Basics

ToC

- Data representation
- Basic data types
- Data type, variables & Qualifiers – I
- Statements
- Declarations & Definitions
- Conditional constructs
- Type conversions
- Operators
- Program memory layout
- Storage classes
- Quiz

Data representation

Data representation

- Why bits?
  - Representing information as bits
  - Binary/Hexadecimal
  - Byte representations
- ANSI data storage
- Embedded specific data storage
- Floating point representation

Base 2

- Base 2 Number Representation

- Electronic Implementation
  - Easy to store
  - Reliably transmitted on noisy and inaccurate wires
  - Straightforward implementation of arithmetic functions

Base 2

Integer Representation

- Positive numbers representation:

Example below shows how $(13)_{10} = (1101)_2$ is represented in 32 bit machine.

	MSB																									LSB							
	3rd Byte								2nd Byte								1st Byte								0th Byte							
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bits	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1

Base 2

Integer Representation

- Negative numbers representation:

Negative numbers are represented by 2's complement .

Below example shows how $(-13)_{10}$ are converted to binary form and stored in the 32 – bit machine.

STEPS:

Step1: Convert $(13)_{10}$ to binary equivalent.

$$(13)_{10} = (0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101)_2$$

Step2: Take 1's complement of binary equivalent.(i.e converting all 1's to 0's & all 0's to 1's)

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0010)_2$$

Base 2

Integer Representation

Step3: Adding 1 we get,

1111 1111 1111 1111 1111 1111 1111 0010

\+ 1

\---

1111 1111 1111 1111 1111 1111 1111 0011 $= (-13)_{10}$

\---

	MSB																															LSB
	3rd Byte								2nd Byte								1st Byte								0th Byte							
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bits	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1

Note : Mathematically : -k ≡ 2 ^ n - k

ΣMERTXE

Byte Encoding

- 1Nibble = 4 bits
- 1Byte = 8 bits
  - Binary : 00000000_2 - 11111111_2
  - Decimal : 0_{10} - 255_{10}
  - Octal : 0_8 - 377_8
  - Hexadecimal: 00_{16} - FF_{16}
- Base 16 number representation
- Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- Write $FA1D37B_{16}$ in C as 0xFA1D37B

Hex	Dec	Bin
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Byte Encoding
Quiz

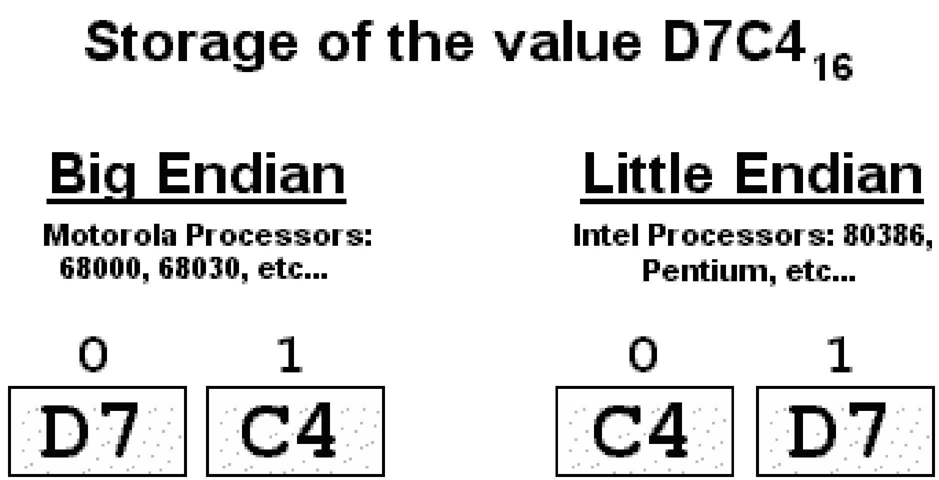
1. Convert 103 from decimal to base 2
2. Convert 1011 0110 0011 from binary to hexadecimal
3. Convert 240 from hexadecimal to binary
4. Convert 011 111 111 from binary to base 8
5. Convert 14 from base 8 to binary

Byte ordering

Machine Type	Ordering of Bytes	Examples
Big Endian	Least significant byte has Highest address	Sun, Mac
Little Endian	Least significant byte has lowest address	Alphas

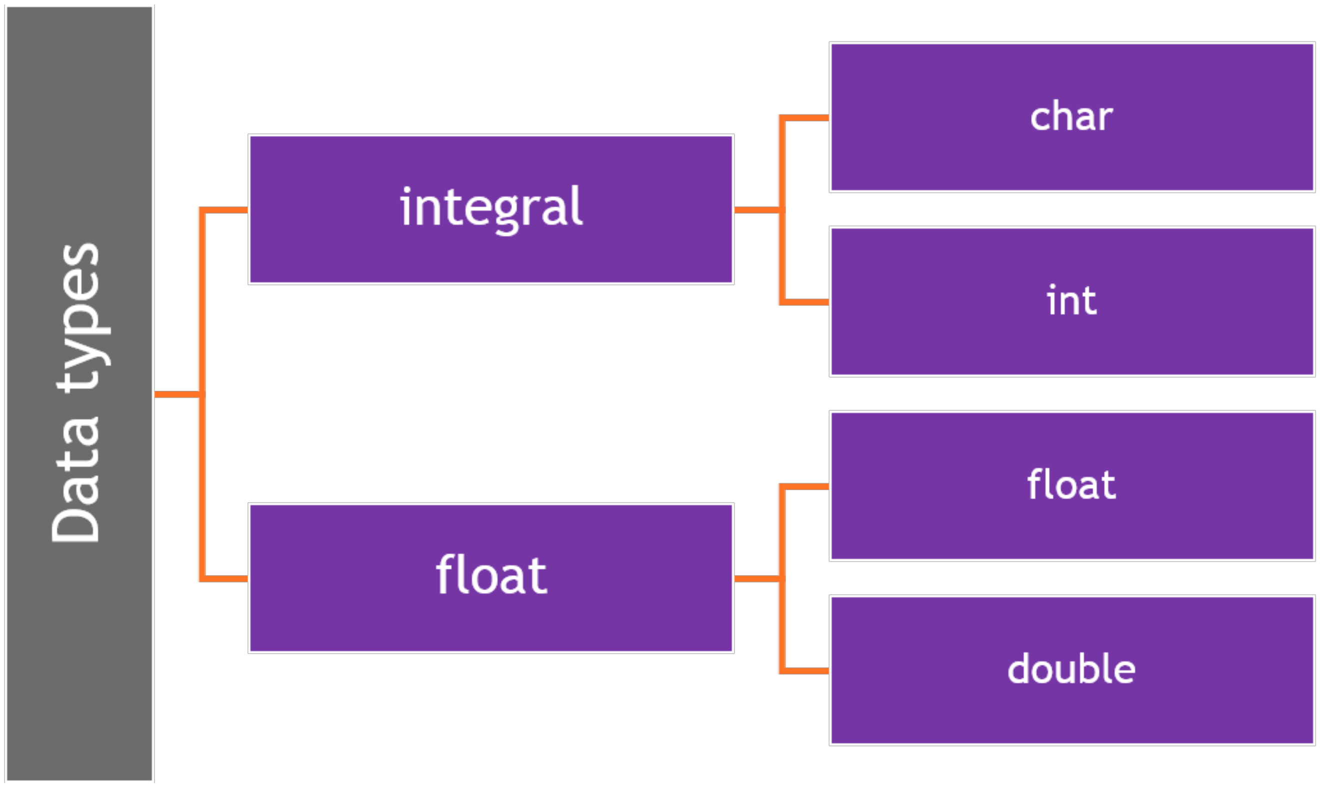
Example:

- Variable x has 2-byte representation D7C4



Basic Data Types

Basic Data Types



Sizes : Basic Data Types

- Type int is supposed to represent a machine's natural word size
- The size of character is always 1 byte
- 1 byte = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- float = 4 bytes
- double = 8 bytes
- pointer = address word, mostly same as word

Sizes : Basic Data Types

- In C sizes of datatypes are not Fixed

OS	32-Bit	64-Bit
Windows	LP32	LLP64
Linux	LP32	LP64

Basic Data Types: char

- Smallest addressable unit of the machine that can contain basic character set.
- It is an integer type.
- Actual type can be either signed or unsigned depending on the *implementation.*
- *Example:*
  - char ch = 'a';
  - char ch = 97;

Think

char ch = 'a' + 10 is possible?

Basic Data Types: char

- Different ways of reading / scanning the character

Function	Example
scanf	scanf(“%c”, &ch);
getchar	ch = getchar();
getc	ch = getc(stdin);
fgetc	ch = fgetc(stdin);

```c
#include <stdio.h>

int main()
{
	char ch;
	printf("Enter the character");

	//Using scanf()
	scanf("%c", &ch);

	//using getchar()
	ch = getchar();

	//using getc()
	ch = getc(stdin);

	//using fgetc()
	ch = fgetc(stdin);

	return 0;
}

```

Basic Data Types: int

- The most basic and commonly used integral type.
- Format specifiers for ints are either %d or %i, for either printf or scanf.
- *Example:*
  - int i = 100;

Real Numbers: Representation

IEEE float: 32 bits	1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. Also called single precision.
IEEE double: 64 bits	1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. Also called double precision.

	sign	Exponent	mantissa
Float (32 bits)	1 bit	8 bits	23 bits
Double (64 bits)	1 bit	11 bits	52 bits

Float to binary
Conversion Procedure

STEP 1: Convert the absolute value of the number to binary, perhaps with a fractional part after the binary point. This can be done by converting the integral and fractional parts separately. The integral part is converted with the techniques examined previously. The fractional part can be converted by multiplication.

STEP 2: Append × (2 ^ 0) to the end of the binary number (which does not change its value).

STEP 3: Normalize the number. Move the binary point so that it is one bit from the left. Adjust the exponent of two so that the value does not change.

Float to binary

Conversion Procedure

STEP 4: Place the mantissa into the mantissa field of the number. Omit the leading one, and fill with zeros on the right.

STEP 5: Add the bias to the exponent of two, and place it in the exponent field. The bias is 2 ^ (k−1)−1, where k is the number of bits in the exponent field. For the eight-bit format, k = 3, so the bias is 2 ^ (3−1) − 1 = 3. For IEEE 32-bit, k = 8, so the bias is 2^(8−1) − 1 = 127.

STEP 6: Set the sign bit, 1 for negative, 0 for positive, according to the sign of the original number.

Float to binary
Conversion Example

Convert 0.1015625 to IEEE 32-bit floating point format.

STEP 1

0.1015625	× 2 =	0.203125	0	Generate 0 and continue.
0.203125	× 2 =	0.40625	0	Generate 0 and continue.
0.40625	× 2 =	0.8125	0	Generate 0 and continue.
0.8125	× 2 =	1.625	1	Generate 1 and continue with the rest.
0.625	× 2 =	1.25	1	Generate 1 and continue with the rest.
0.25	× 2 =	0.5	0	Generate 0 and continue.
0.5	× 2 =	1.0	1	Generate 1 and nothing remains.

So $0.1015625_{10} = 0.0001101_2$.

ΣMERTXE

Float to binary
Conversion Example

Convert 0.1015625 to IEEE 32-bit floating point format.

STEP 2

Normalize: $0.0001101_2 = 1.101_2 \times 2^{-4}$.

STEP 3

Mantissa is 10100000000000000000000, exponent is -4 + 127 = 123 = 011110112, sign bit is 0

So 0.1015625 is 00111101110100000000000000000000 = $3dd00000_{16}$

	s	Exponent								Mantissa																						
Bit No.	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bits	0	0	1	1	1	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ΣMERTXE

Data Type & Variable Qualifiers - I

const

- The const keyword is used to create a read only variable.
- Once initialised, the value of the variable cannot be changed but can be used just like any other variable.
- *Example:*
  - const float pi = 3.14;

register

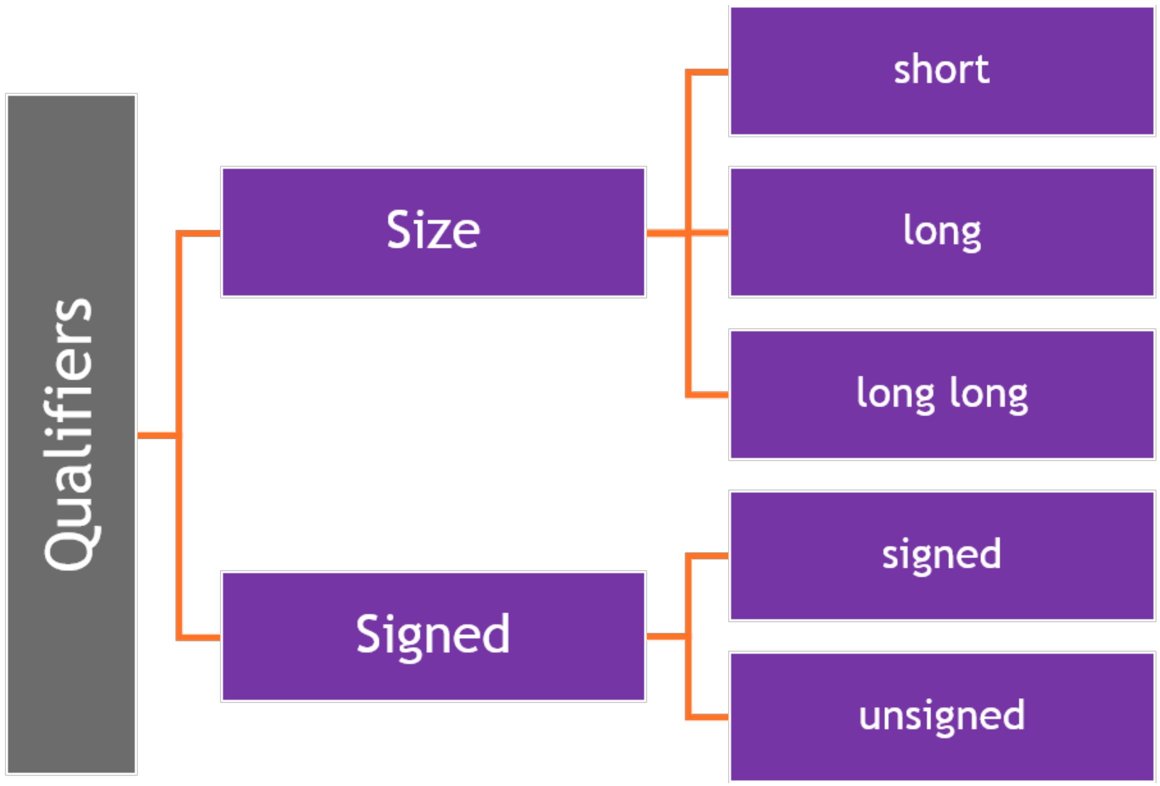
- Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword.

- *Syntax*
  register data_type variable;

- *Example:*

  - register int i;

Type Qualifiers



DIY:

WAP to find the sizes of all basic data types along with qualifiers

Range

Size Qualifiers

Qualifier	Bytes	Range
short	2	-32768 to 32767
long	8	-2^8 to $+(2^8 - 1)$

Signed Qualifiers

Qualifier	Bytes	Range
signed int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295

Examples:

1. char ch; // Default qualifier is implementation dependent

2. int a; // Always it is signed

DIY: Calculate the range for long long if the size is 12 bytes

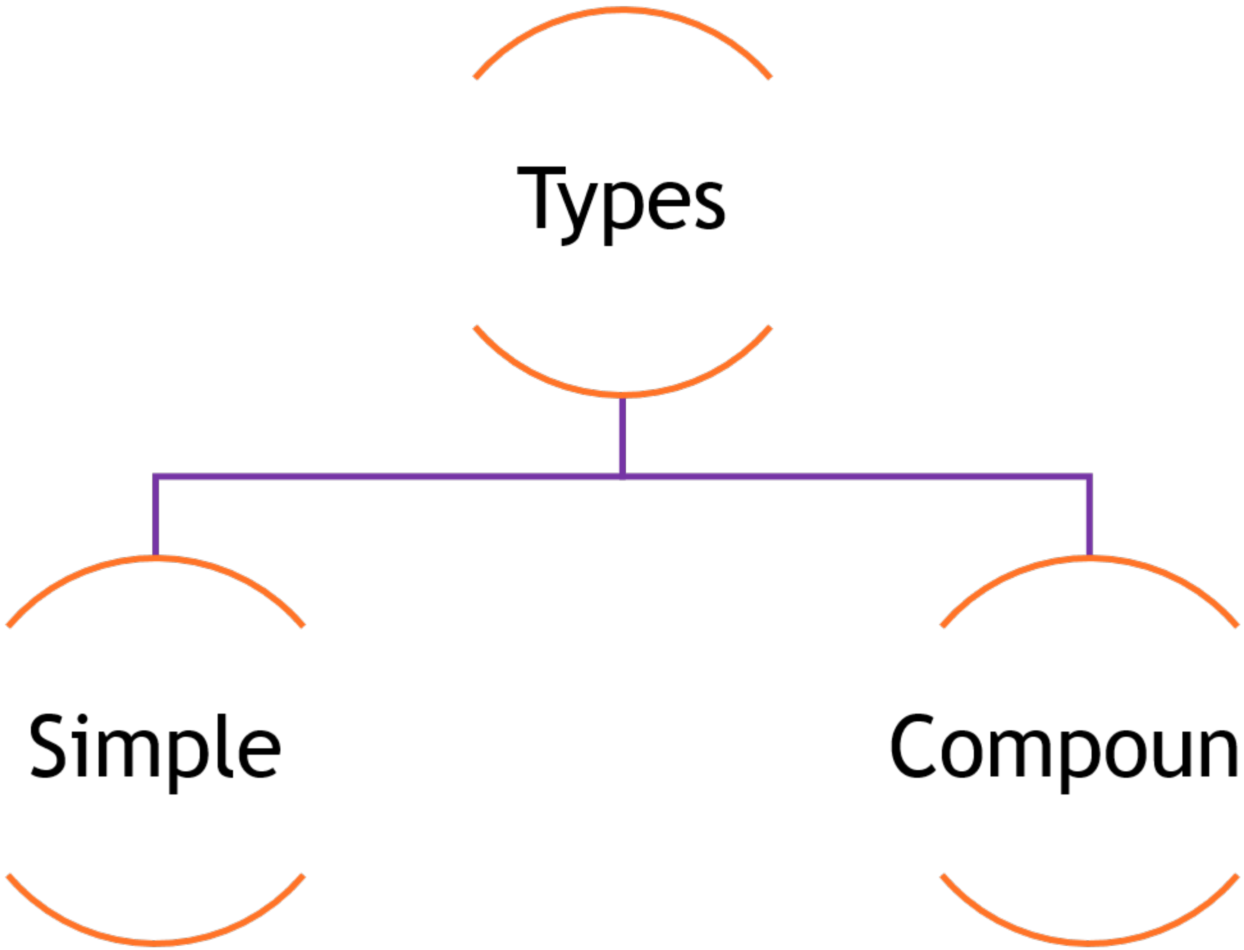
Common Mistakes

Unsigned int x = -1;

Short count = 9999999;

Statements

Statement Types



Simple Statements

- May be expression / function call terminated by semicolon(;)

Example	Description
A = 2;	Assignment statement
x = a + 3;	The result of (a + 3) assign to x
4 + 5;	has no effect, will be discarded by smartcompilers

; is a valid statement as it is a part of the statement, and not just a statement terminator as in **Pascal.**

Compound Statements
And its need

- Compound statements come in two varieties: conditionals and loops.

Output?

```
if(a > b)
        if(a > c)
                printf("Hello");
        else
                printf("Hai");


_
```

Declaration & Definition

Declaration

- A declaration specifies a type, and contains a list of one or more variables of that type.

- Example

```
int lower, upper, step;
char ch;
```

Declaration is an announcement and can be done 1 or more times.

Definition

- A definition associates an object name with a memory.

- Example

```
int lower = 10;
char ch = 'A';
```

Definition is an actual execution and should be done exactly once.

Conditional Constructs

Types

- Types
  - Single
    - If and its family
    - switch & case
  - Repetitive
    - for
    - while
    - do-while

if

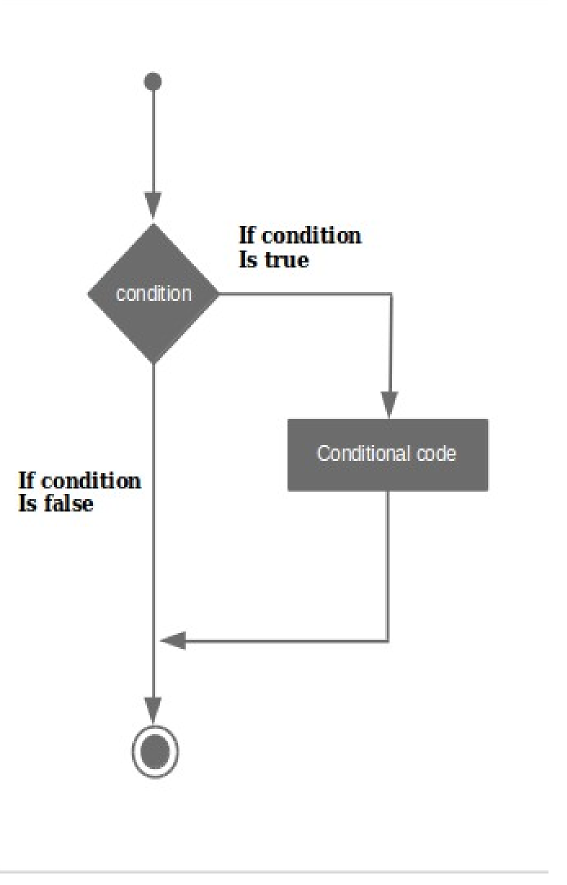
Syntax

```
if(condition)
{
        statement(s);
}

```

Example

```
int a = 2;
if( a < 5 )
{
        printf("a is less than 5\n" );
}
printf("value of a is : %d\n", a);
```

Flow



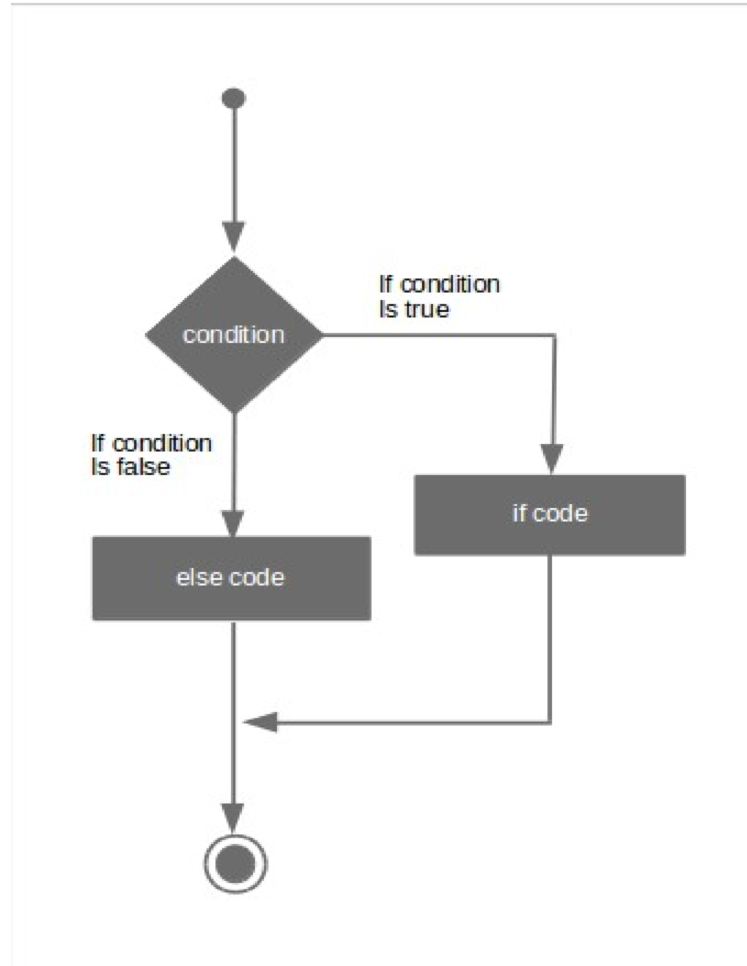
Note:
zero or null is assumed as false value apart from this
any non-zero and non-null values are assumed as true .

If - else

Syntax

```
if(condition)
{
        statement(s);
}
else
{
        statement(s);
}
```

Flow



Note:
zero or null is assumed as false value apart from this
any non-zero and non-null values are assumed as true .

If – else…

Example

```
int a = 10;
if( a < 2 )
{
        printf("a is less than 2\n" );
}
else
{
        printf("a is not less than 2\n" );
}
```

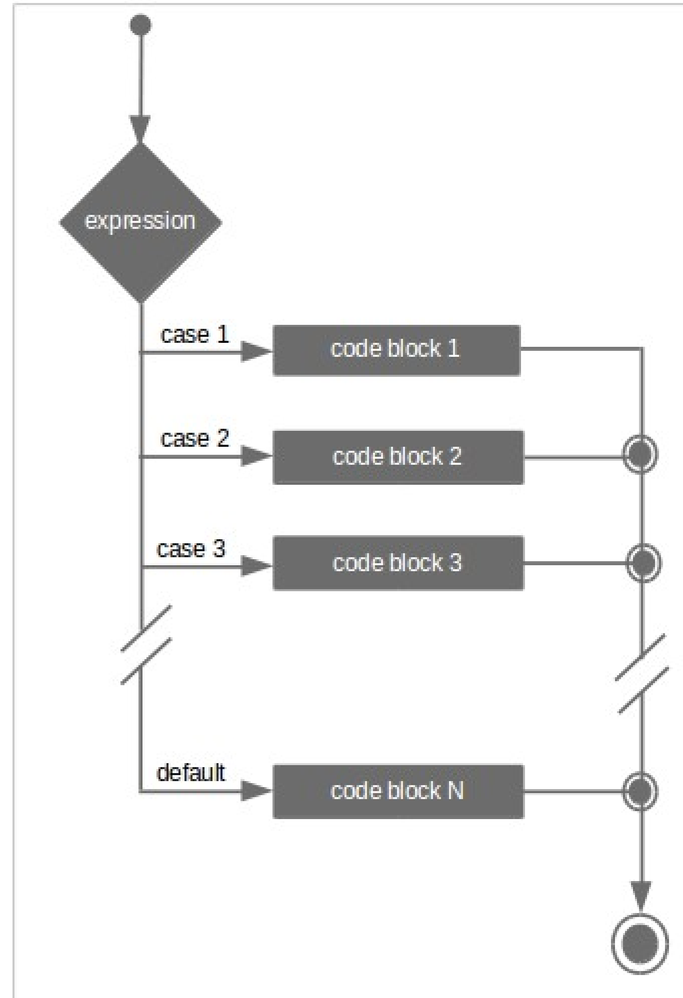
DIY:
WAP to find the biggest of two given integers

switch

Syntax

```
switch(expression)
{
        case constant:
                        statement(s);
                        break;
        case constant:
                        statement(s);
                        break;
        case constant:
                        statement(s);
                        break;
        :
        :
        :
        default:
                        statement(s);
}

```

Flow



switch

Example

```
int a;
printf("Enter the value");
scanf("%d", &a);

switch(a)
{
        case 10:
                printf("You entered 10");
                break;
        case 20:
                printf("You entered 20");
                break;
        default:
                printf("Try again");
}
```

DIY:
Implement simple arithmetic calculator

for loop

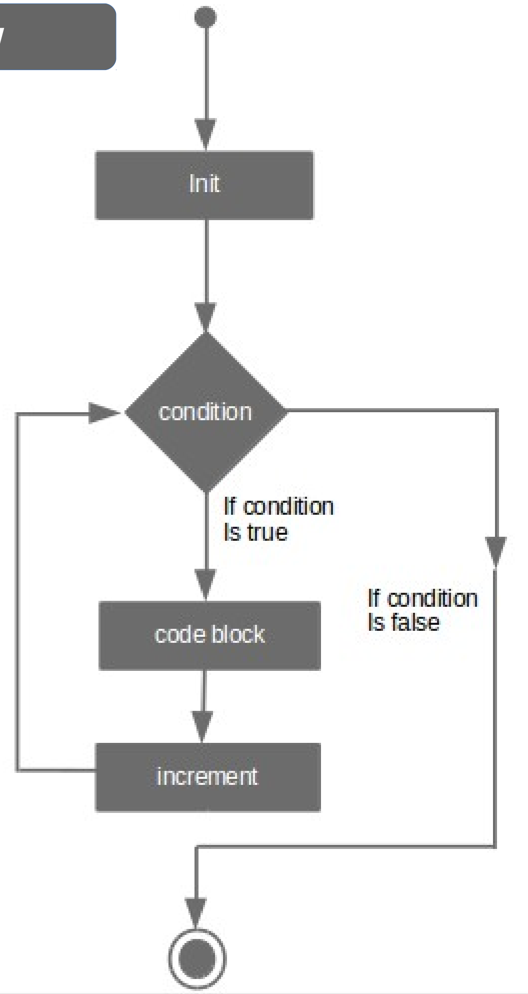
Syntax

```
for(init; condition; increment)
{
        //body
}

```

Example

```
int i;
for(i = 0; i < 10; i++)
{
        printf("%d", i);
}

```

Flow



DIY:
Write a program to find the sum of first n natural numbers where n is entered by user

while loop

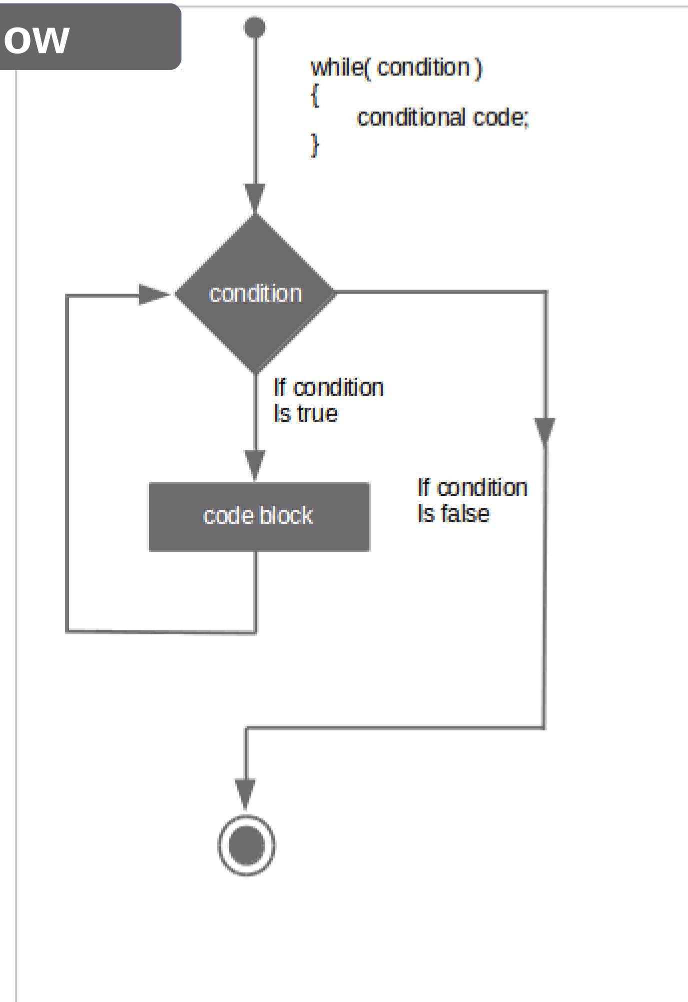
Syntax

```c
while(condition)
{
            //body
}

```

Example

```c
int i;

while(i < 10)
{
        printf("%d", i);
        i++;
}

```

Flow



DIY:
Write a program to find the factorial of a given number where 'number' is entered by user

do-while loop

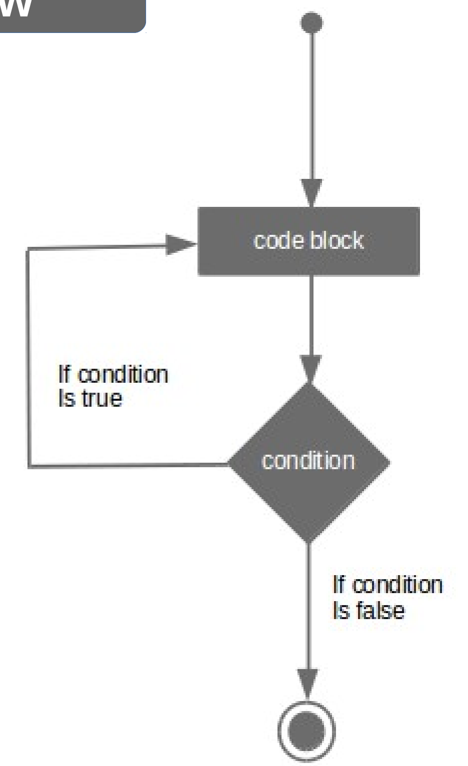
Syntax

```
do
{
        //body
}while(condition);
```

Example

```
int i = 1;
do
{
        printf("%d", i);
        i++;
}while(i < 10);
```

Flow



DIY:
Write a C program to add all the numbers entered by a user until user enters 0

continue Statement

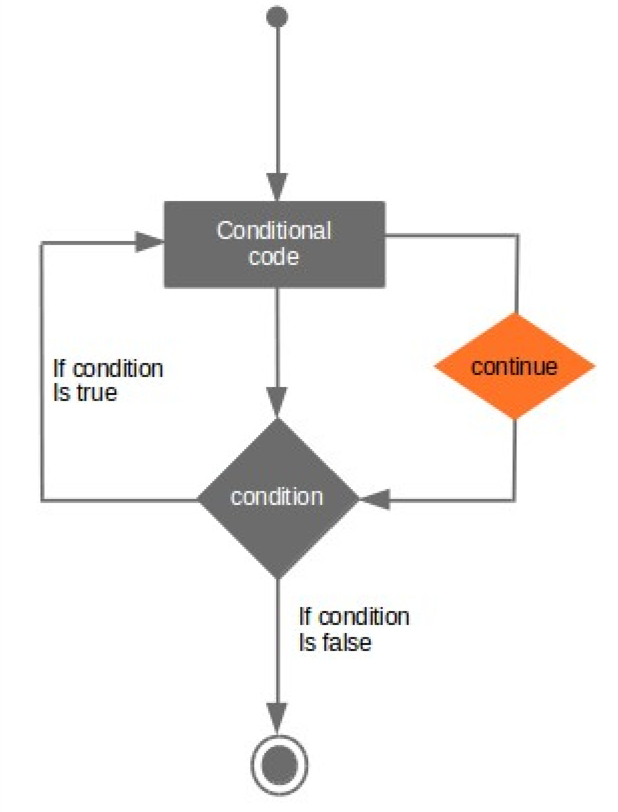
continue statements are used to skip some statements inside the loop.

Syntax

```
continue;
```

Example

```
int i;
while(i < 100)
{
        if(i == 50)
        {
                continue;
        }
        printf("%d", i);
}
```



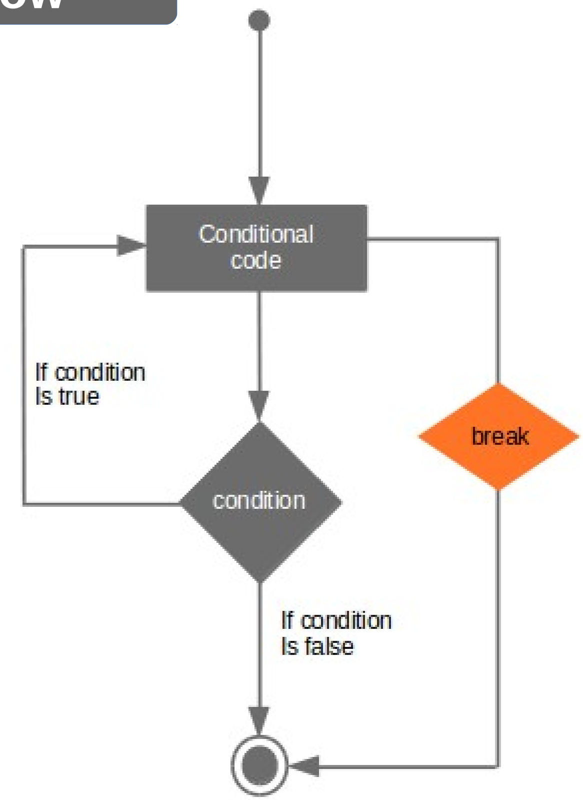
break Statement

break is used in terminating the loop immediately after it is encountered.

Syntax

```
break;
```

Flow

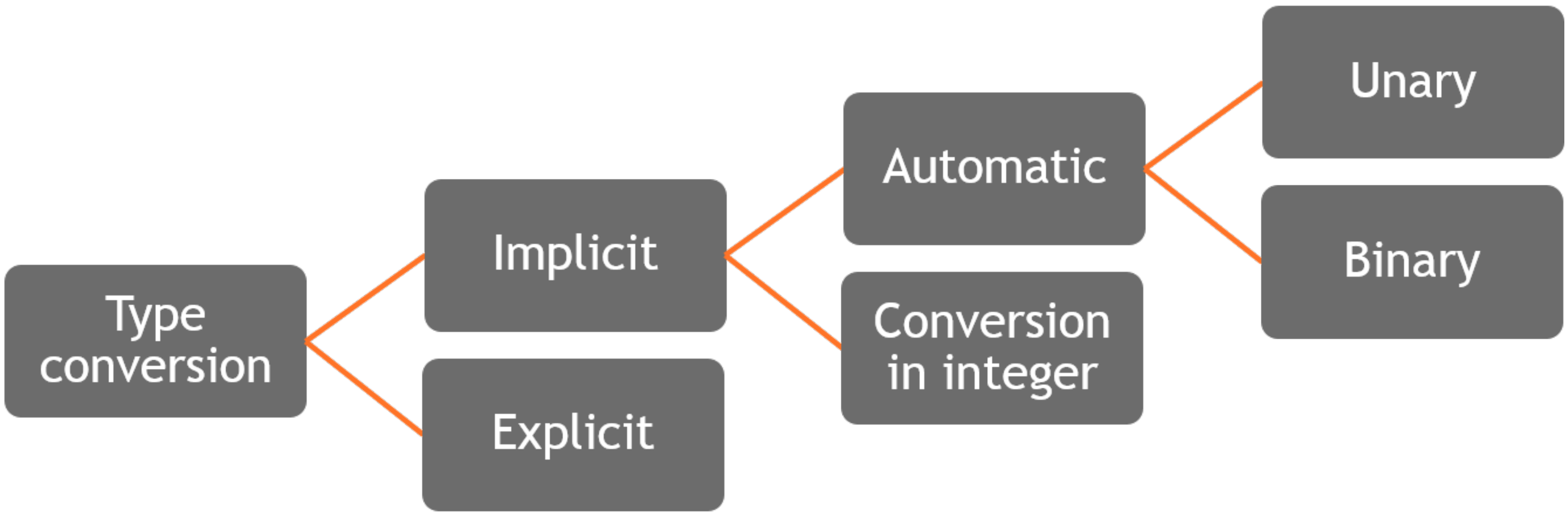


Example

```
int i;
while(i < 100)
{
        if(i == 50)
        {
                break;
        }
        printf("%d", i);
}

```

Type Conversions

Type Conversion: Types



Type Conversion: Hierarchy

- Long double
- Double
- Float
- Unsigned long long
- Signed long long
- Unsigned long
- Signed long
- Unsigned int
- Signed int
- Unsigned short
- Signed short
- Unsigned char
- Signed char

Type Conversion: Implicit

- **Automatic Unary conversions**
  - All operands of type char & short are converted to int before any operations

- **Automatic Binary conversions**

  - If one operand is of LOWER RANK data type & other is of HIGHER RANK data type then LOWER RANK will be converted to HIGHER RANK while evaluating the expression.

  - **Example**

  If one operand is int & other is float then, int is converted to float.

Type Conversion: Implicit

- **Type conversions in assignments**

- The type of right hand side operand is converted to type of left hand side operand in assignment statements.
- If type of operand on right hand side is LOWER RANK data type & left hand side is of HIGHER RANK data type then LOWER RANK will be promoted to HIGHER RANK while assigning the value.

- If type of operand on right hand side is HIGHER RANK data type & left hand side is of LOWER RANK data type then HIGHER RANK will be demoted to LOWER RANK while assigning the value.
- **Example**

  Fractional part will be truncated during conversion of **float** to **int**.

EMERTXE

Type Conversion:
Explicit Or Type Casting

Output ?

```c
#include <stdio.h>

int main()
{
        int a = 5, b = 3;

        float f = a / b;

        printf("%f", f);

        return 0;
}

```

Type Conversion:
Explicit Or Type Casting

```c
#include <stdio.h>

int main()
{
	int a = 5, b = 3;

	float f = (float)a / b;

	printf("%f", f);

	return 0;
}

```

Type casting

Syntax

(datatpye) expression

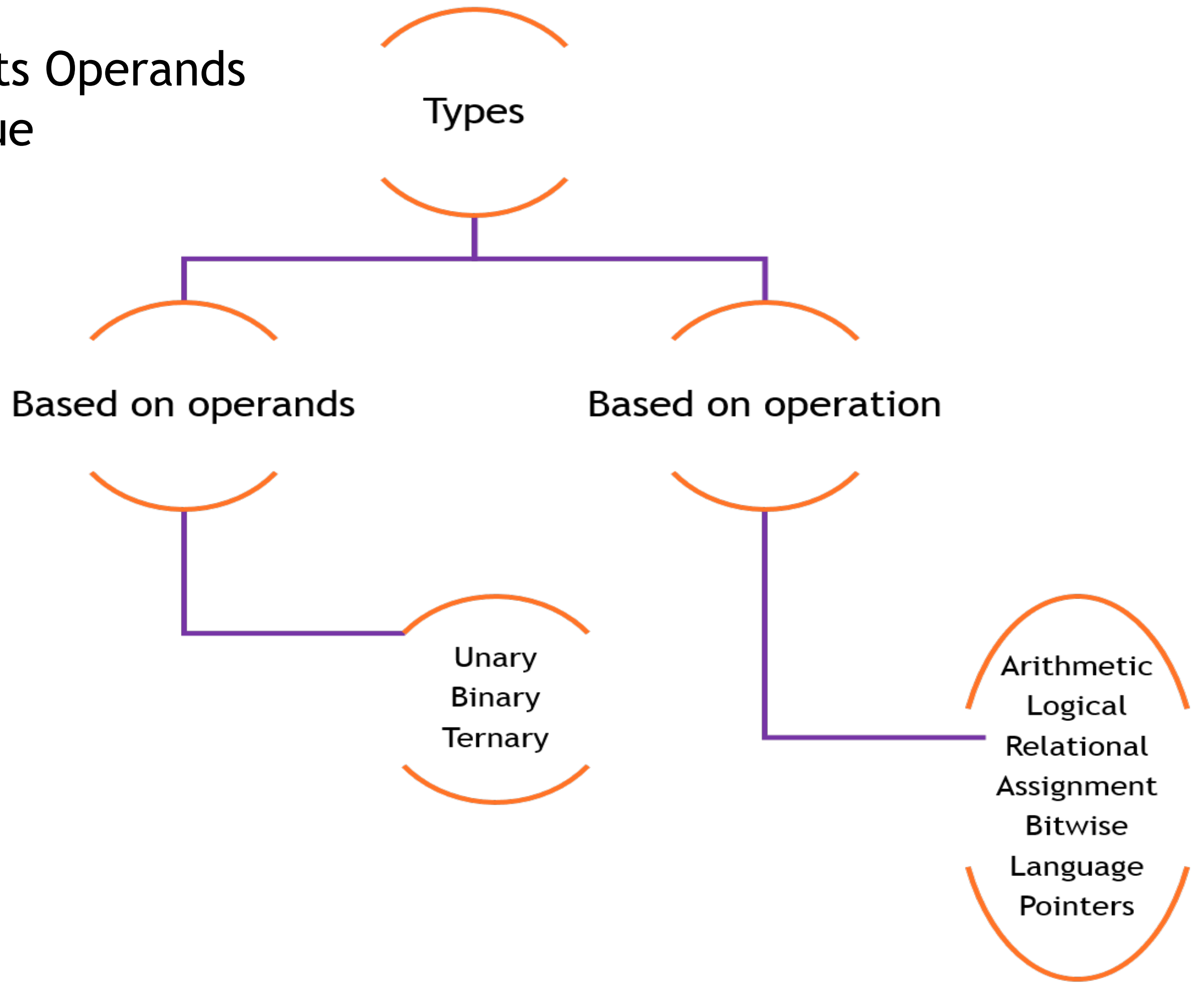
DIY:
WAP to convert farenheit to celcius

Operators

Introduction

All C operators do 2 things:

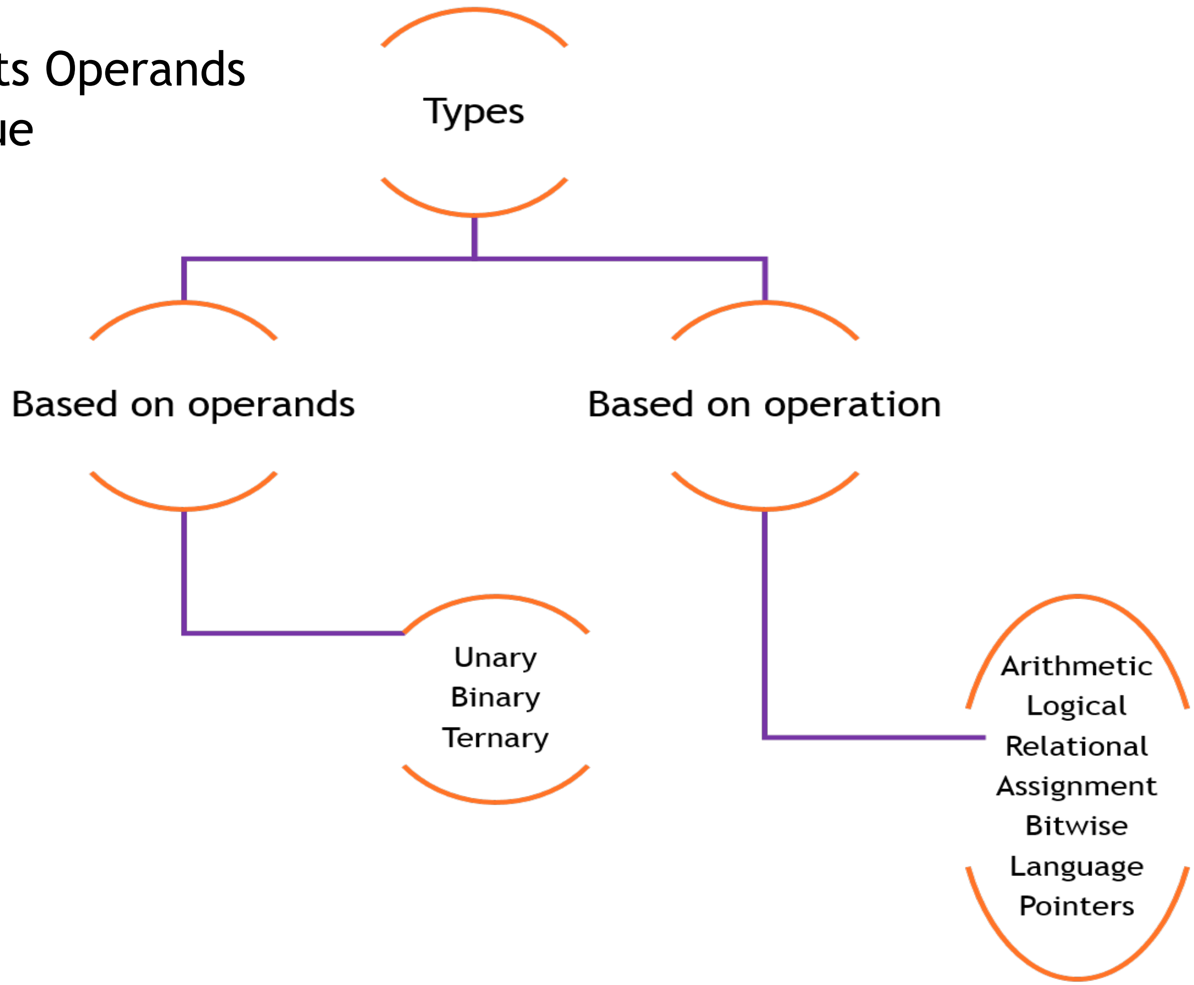
- Operates on its Operands
- Returns a value



Introduction

All C operators do 2 things:

- Operates on its Operands
- Returns a value



Arithmetic Operators

Operator	Description	Associativity
* / %	Multiplication Division Modulo	L to R
+ -	Addition Subtraction	L to R

Output ?

```c
int main()
{
        int i = 0, j = 0;
        printf("%d", i++ + ++j);
        return 0;
}
```

Logical Operators

Operator	Description	Associativity
!	Logical Not	R to L
&& ||	Logical AND Logical OR	L to R

Output ?

```c
int main()
{
        int a = 1, b = 0;
        if (++a || ++b)
                printf("In first if a = %d, b = %d", a, b);
        a = 1, b = 0;
        if (b++ && ++a)
                printf("In second if a = %d, b = %d", a, b);
        else
                printf("In second if a = %d, b = %d", a, b);
        return 0;
}
```

Relational Operators

Operator	Description	Associativity
>	Greater than	L to R
>=	Greater than or equal to	
<	Less than	
<=	Less than or equal to	
==	Equal	L to R
!=	Not equal	

```c
int main()
{
        float f = 0.7;
        if(f == 0.7)
                printf("Equal");
        else
                printf("Not Equal");
}

```

Output ?

Assignment Operators

- An assignment operator is used to assign a constant / value of one variable to another

Output ?

```
int main()
{
        int a = 1, c = 1, e = 1;
        float b = 1.5, d = 1.5, f = 1.5;
        a += b += c += d += e += f;
}
```

Output ?

```
int main()
{
        int x = 0;
        if (x = 5)
                printf("Its equal");
        else
                printf("No! it's not!!!");
}
```

Bitwise Operators

Operator	How to read	Description	Example
&	Bitwise AND	Bitwise ANDing all the bits in two operands	A = 60 = 0011 1100 B = 13 = 0000 1101 (A & B) = 12 = 0000 1100
|	Bitwise OR	Bitwise ORing all the bits in two operands	A = 60 = 0011 1100 B = 13 = 0000 1101 (A | B) = 61 = 0011 1101
^	Bitwise XOR	Bitwise XORing all the bits in two operands	A = 60 = 0011 1100 B = 13 = 0000 1101 (A ^ B) = 49 = 0011 0001
~	Compliment	Complimenting all the bits in given operand	A = 60 = 0011 1100 (~A) = -61 = 1100 0011
>>	Bitwise Right shift	Shift all the bits right n times by introducing zeros left	A = 60 = 0011 1100 (A << 2) = 240 = 1111 0000
<<	Bitwise Left shift	Shift all the bits left n times by introducing zeros right	A = 60 = 0011 1100 (A >> 2) = 15 = 1111 0000

Bit wise operators are very powerful, extensively used in low level programming. This helps to deal with hardware (Ex: registers) efficiency.

Sizeof() Operator

- The sizeof operator returns the size of its operand in bytes.

Output ?

```c
int main()
{
        int x = 5;
        printf("%u:%u:%u\n", sizeof(int), sizeof x, sizeof 5);
}
```

Output ?

```c
int main()
{
        int i = 0;
        int j = sizeof(++i);
        printf("%d:%d\n", i, j);
        /* Assume sizeof int is 4 bytes */
}
```

The only C operators, which operate during compilation itself,
i.e. a compile-time operator

Sizeof() Operator...

- 3 reasons for why sizeof is not a function:
  - Any type of operands,
  - Type as an operand,
  - No brackets needed across operands

Output ?

```
int main()
{
        int i;
        int array[5] = {0, 2, 4, 1, 3};
        for(i = -1; i < sizeof(array) / sizeof(int) - 1; i++)
                printf("%x\n", array[i + 1]);
}
```

Ternary Operator

- Operates on 3 operands

Syntax

Condition ? Expression1 : Expression2;

Example

```
int a = 10;
int b = 20;

if(a > b)
{
        return 1;
}
else
{
        return 0;
}
```

```
int a = 10;
int b = 20;

int c = (a > b) ? 1: 0;
```

Precedence & Associativity Of Operators

Operators	Associativity	Precedence
() [] -> .	L - R	HIGH
! ~ ++ -- + - * & (type) sizeof	R - L	
/ % *	L - R	
+ -	L - R	
<< >>	L - R	
< <= > >=	L - R	
== !=	L - R	
&	L - R	
^	L - R	
|	L - R	
&&	L - R	
||	L - R	
?:	R - L	
= += -= *= /= %= &= ^= |= <<= >>=	R - L	
,	L - R	LOW

Circuit logical operators

- These operators are similar to the & and | operators that are applied to Boolean type

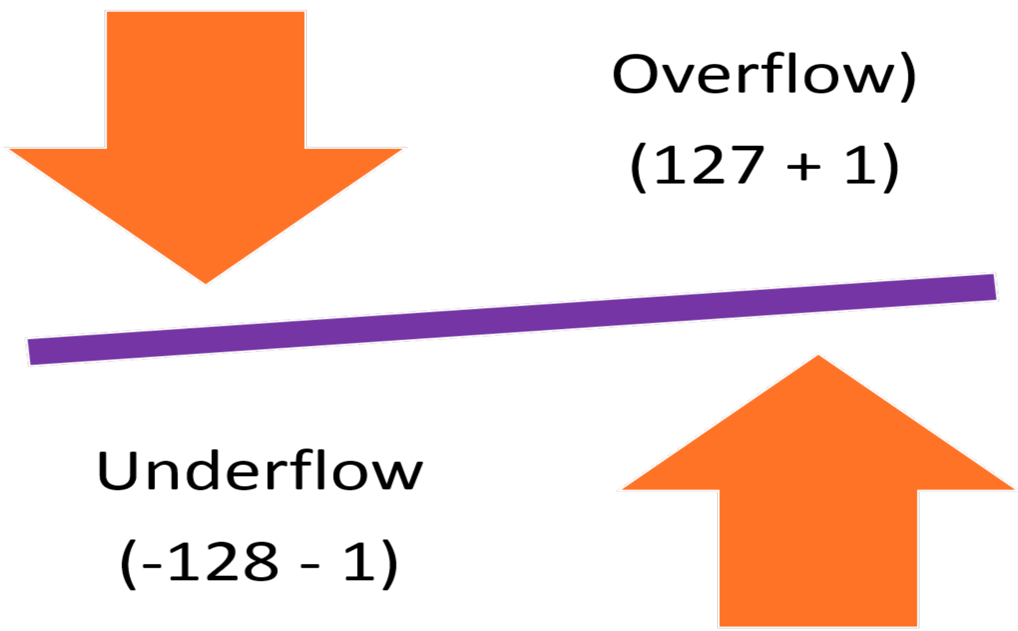
- Have the ability to “short circuit” a calculation if the result is definitely known, this can improve efficiency

  - AND operator &&
  - If one operand is false, the result is false.
  - OR operator ||
  - If one operand is true, the result is true.

Underflows & Overflows

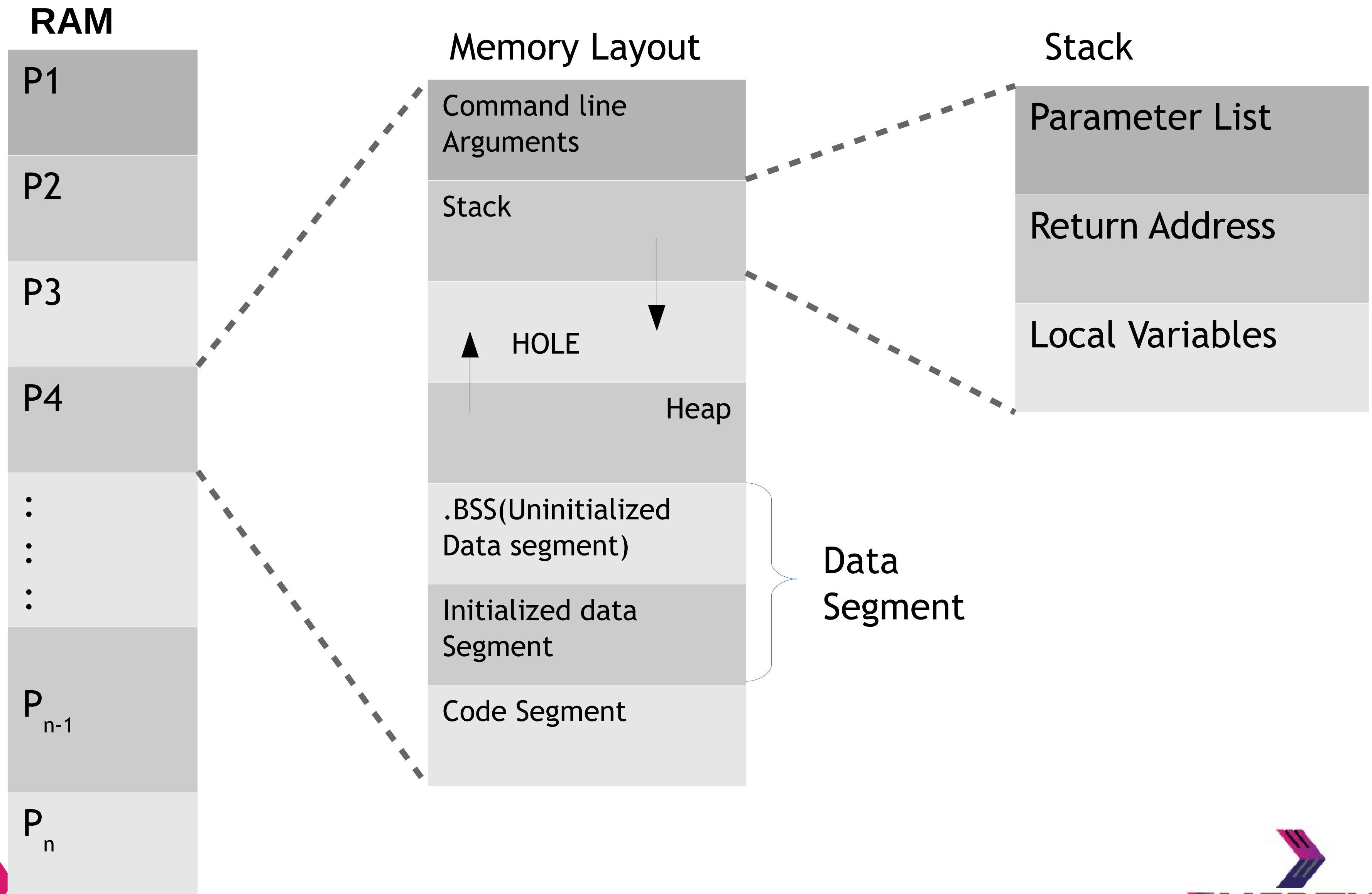
8-bit Integral types can hold certain ranges of values.

So what happens when we try to traverse this boundary?



Program Memory Layout

Program Segments



Run time Layout

- ✓ **Run-time memory includes four (or more) segments**
  - Text area: program text
  - Global data area: global & static variables
  - Allocated during whole run-time
- ✓ **Stack: local variables & parameters**
  - A stack entry for a functions
  - Allocated (pushed) - When entering a function
  - De-allocated (popped) - When the function returns
- ✓ **Heap**
  - Dynamic memory
  - Allocated by malloc()
  - De-allocated by free()

Details

- ✓**Text Segment:** The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions.

- ✓**Initialized Data Segment:** This segment contains global variables which are initialized by the programmer.

- ✓**Uninitialized Data Segment:** Also named “BSS" (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute.

Details

- ✓ **The Stack:** The stack is a collection of stack frames which will be described in the next section. When a new frame needs to be added (as a result of a newly called function), the stack grows downward

- ✓ **The Heap:** Most dynamic memory, whether requested via C's malloc() . The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows upward

Storage Classes

Storage Classes

Storage Class	**Scope**	**Lifetime**	**Memory Allocation**
auto	Within the block / Function	Till the end of the block / function	Stack
Register	Within the block / Function	Till the end of the block / function	Register
Static local	Within the block / Function	Till the end of the program	Data Segment
Static global	File	Till the end of the program	Data segment
extern	Program	Till the end of the program	Data segment

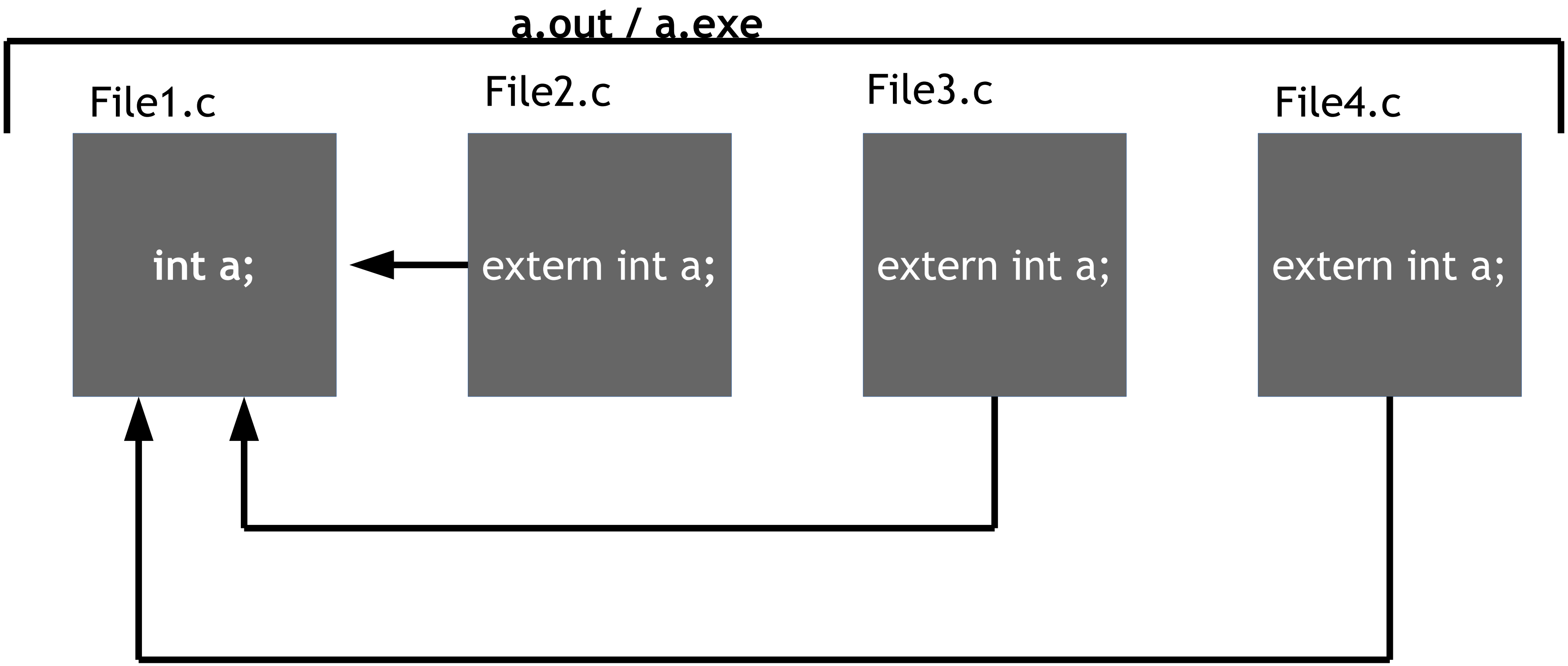
Storage Classes...

```c
#include <stdio.h>

int global;
int global1 = 10;

static int global2;
static int global3 = 20;

int main()
{
        int local;
        static int local1;
        static int local2 = 30;

        :
        :
        :

        register int i;
        for(i = 0; i < 100; i++)
        {
                //do something
        }
}
```

Variable	Storage Class	Memory Allocation
global	No	.BSS
global1	No	Initialized data segment
global2	Static global	.BSS
global3	Static global	Initialized data segment
local	auto	stack
local1	Static local	.BSS
local2	Static local	Initialized data segment
i	Register	Registers

Storage Classes...

Extern



Embedded
Data storage

- Non-portability
- Solution: User typedefs
- Size utilizations
  - Compiler Dependency
  - Architecture Dependency
  - u8, s8, u16, s16, u32, s32, u64, s64
- Non-ANSI extensions
  - bit

Typedefs

- Typedef are the alternative names given to the existing names.
- Mostly used with user defined data types when names of the existing data types gets complicated.

General Syntax:

```
typedef existing_name new_name
```

Example:

```
typedef unsigned long int size_t
```

Quiz

???

```c
/*Is this code is valid?*/

int main()
{
        3;+5;
        ;
}

```

Functions

Why Functions?

- **Reusability**
  - Functions can be stored in library & re-used
  - When some specific code is to be used more than once, at different places, functions avoids repeatition of the code.
- **Divide & Conquer**
  - A big & difficult problem can be divided into smaller sub-problems and solved using divide & conquer technique
- **Modularity** can be achieved.
- Code can be easily **understandable** & **modifiable.**
- Functions are easy to **debug** & **test**.
- One can suppress, how the task is done inside the function, which is called **Abstraction**.

Parameters, Arguments and Return Values

Parameters, Arguments and Return Values

- Parameters are also commonly referred to as arguments, though arguments are more properly thought of as the actual values or references assigned to the parameter variables when the subroutine is called at runtime.

- When discussing code that is calling into a subroutine,any values or references passed into the subroutine are the arguments, and the place in the code where these values or references are given is the parameter list.

- When discussing the code inside the subroutine definition, the variables in the subroutine’s parameter list are the parameters, while the values of the parameters at runtime are the arguments.

Parameters, Arguments and Return Values

Example

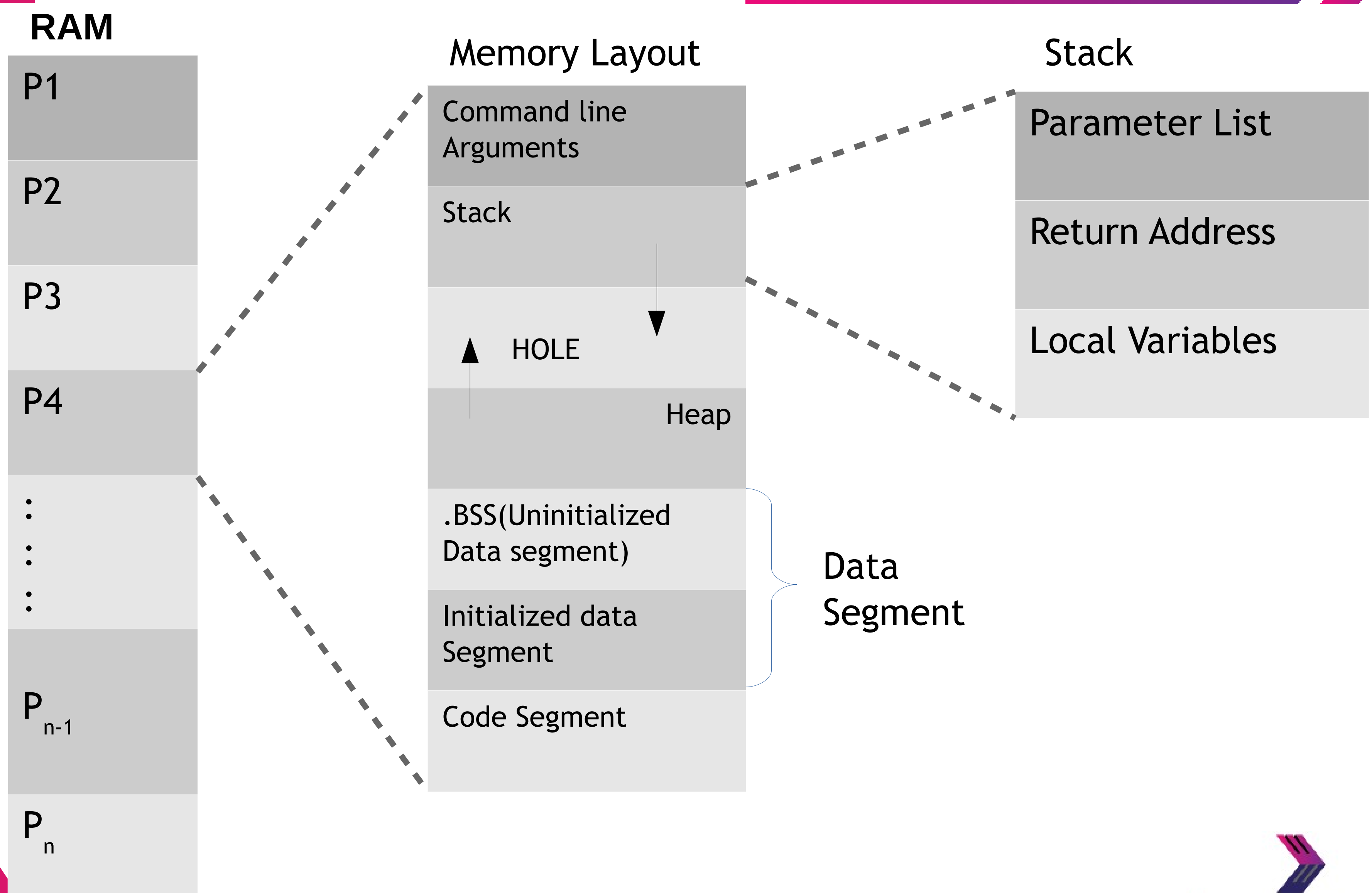
```c
#include <stdio.h>
//Function declaration
int add(int, int);
int main()
{
        int a = 10;
        int b = 20;
        int sum;

        //Function call
        sum = add(a, b);

        printf("Sum : %d\n", sum);
}
//Function definition
int add(int a, int b)
{
        int res;

        res = a + b;

        return res;
}

```

Actual Arguments

Formal Arguments

Return type

Function and the Stack



Various Passing Mechanisms

Various Passing Mechanisms

Pass by Value	Pass by reference
• This method copies the actual value of an argument into the formal parameter of the function. • In this case, changes made to the parameter inside the function have no effect on the actual argument.	• This method copies the address of an argument into the formal parameter. • Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Pass by value

Example

```
#include <stdio.h>

void modify(int);

int main()
{
        int a = 10;

        printf("Before modify");
        printf("a : %d", a);

        modify(a);

        printf("After modify");
        printf("a : %d", a);
}

void modify(int a)
{
        a = a + 1;
}
```

O / P ???

EMERTXE

Pass by reference

Example

```
#include <stdio.h>

void modify(int*);

int main()
{
		int a = 10;

		printf("Before modify");
		printf("a : %d", a);

		modify(&a);

		printf("After modify");
		printf("a : %d", a);
}

void modify(int *a)
{
		*a = *a + 1;
}
```

O / P ???

EMERTXE

Pass by value &
Pass by reference

DIY

Problem 1:

Write a function to swap two integers passed from main function.

Problem 2:

Write a function to perform both sum & product of two different integers passed from main function.

Ignoring Function’s Return Value

Ingnoring the function's return value

```c
#include <stdio.h>

int read_int(void);

int main()
{
        int i = read_int();
        :
        :
        :
        return 0;
}
```

- Function read_int returning an int And collecting in variable i in the main function.

```c
int read_int()
{
        int i;
        :
        :
        :
        return i;
}
```

- Main function ignoring the return value.

```c
int main()
{
        read_int();
        :
        :
        :
        return 0;
}
```

In C tradition, you can choose to ignore to capture the return value from a method:

Introduction : Arrays

Introduction

- An array is a collection of data of similar data type.
- Example:

  If we want to store the ages of 5 different people, then we can use array instead of using 5 different variables.

- The mory alloaction will be contiguos.
- A specific element in an array is accessed by an index.

Array: Declaration

Syntax

type arrayName [SIZE];

Example
To declare an array to store ages of 5 people
int age[5];

Total Memory

```
Total Memory = SIZE * sizeof(dataType)
             =  5 * sizeof( int ) // Assuming sizeof(int) = 4
             =  5 * 4
             =  20 bytes
```

Array: Definition

Example
To declare an array to store ages of 5 people
int age[5] = {10, 20, 30, 40, 50};

Memory Allocation

age[0]	age[1]	age[2]	age[3]	age[4]
10	20	30	40	50
100	104	108	112	116

DIY : Write a program to add all the elements of the array.

Array: Passing to function

```c
#include <stdio.h>

void print_array(int a[]);

int main()
{
        int a[5] = {10, 20, 30, 40, 50};

        //Function call
        print_array(a, 5);

        return 0;
}
```

```c
void print_array(int a[], int size)
{
        int i;
        for(i = 0; i < size; i++)
        {
                printf("a[%d] : %d\n", i, a[i]);
        }
}

```

DIY: WAF to sort the elements of the array in asc / des order.

Returning the array from the function

Array: Passing to function

```c
#include <stdio.h>

int* return_array(void);

int main()
{

	//Function call
	int *a = return_array();
	:
	:
	:
	return 0;
}
```

```c
int* return_array(void)
{
	static int a[5] = {10, 20, 30, 40, 50};
	:
	:
	:
	return a;
}
```

Static keyword is used to retain the values between the function calls

Variadic functions

Variadic Functions:
Introduction

- Variadic functions can be called with any number of trailing arguments.

  For example,
  printf(), scanf() are common variadic funtions

- Variadic functions can be called in the usual way with individual arguments.

syntax

return_type function_name(parameter list, ...);

Variadic Functions:
How are defined & used

Defining and using a variadic function involves three steps:

step 1: Variadic functions are defined using an ellipsis ('…') in the argument list, and using special macros to access the variable arguments.

For example,

```
int foo(int a, ...)
{
        //body
}
```

step 2: Declare the function as variadic, using a prototype with an ellipsis ('…'), in all the files which call it.

Step 3: Call the function by writing the fixed arguments followed by the additional variable arguments.

Variadic Functions:
Arguments access macros

- *Descriptions of the macros used to retrieve variable arguments.*

- *These macros are defined in the header file **stdarg.h***

Type/ Macros	Description
va_list	The type va_list is used for argument pointer variables
va_start	This macro initializes the argument pointer variable ap to point to the first of the optional arguments of the current function; last-required must be the last required argument to the function
va_arg	The va_arg macro returns the value of the next optional argument, and modifies the value of ap to point to the subsequent argument. Thus, successive uses of va_arg return successive optional arguments
va_end	This ends the use of ap

Variadic Functions:
Example

```c
int main (void)
{
        /* This call prints 16. */
        printf ("%d\n", add( 3, 5, 5, 6));

        /* This call prints 55. */
        printf ("%d\n", add(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

        return 0;
}
```

```c
int add(int count,...)
{
        va_list ap;
        int i, sum;

        /* Initialize the argument list. */
        va_start (ap, count);
        sum = 0;
        for (i = 0; i < count; i++)
        /* Get the next argument value. */
                sum += va_arg (ap, int);
        /* Clean up */
        va_end (ap);
        return sum;
}
```

Recursive functions

Recursive Function

- Function calling itself

<u>*Steps:*</u>

1. Identification of the base case.

2. Writing the recursive case.

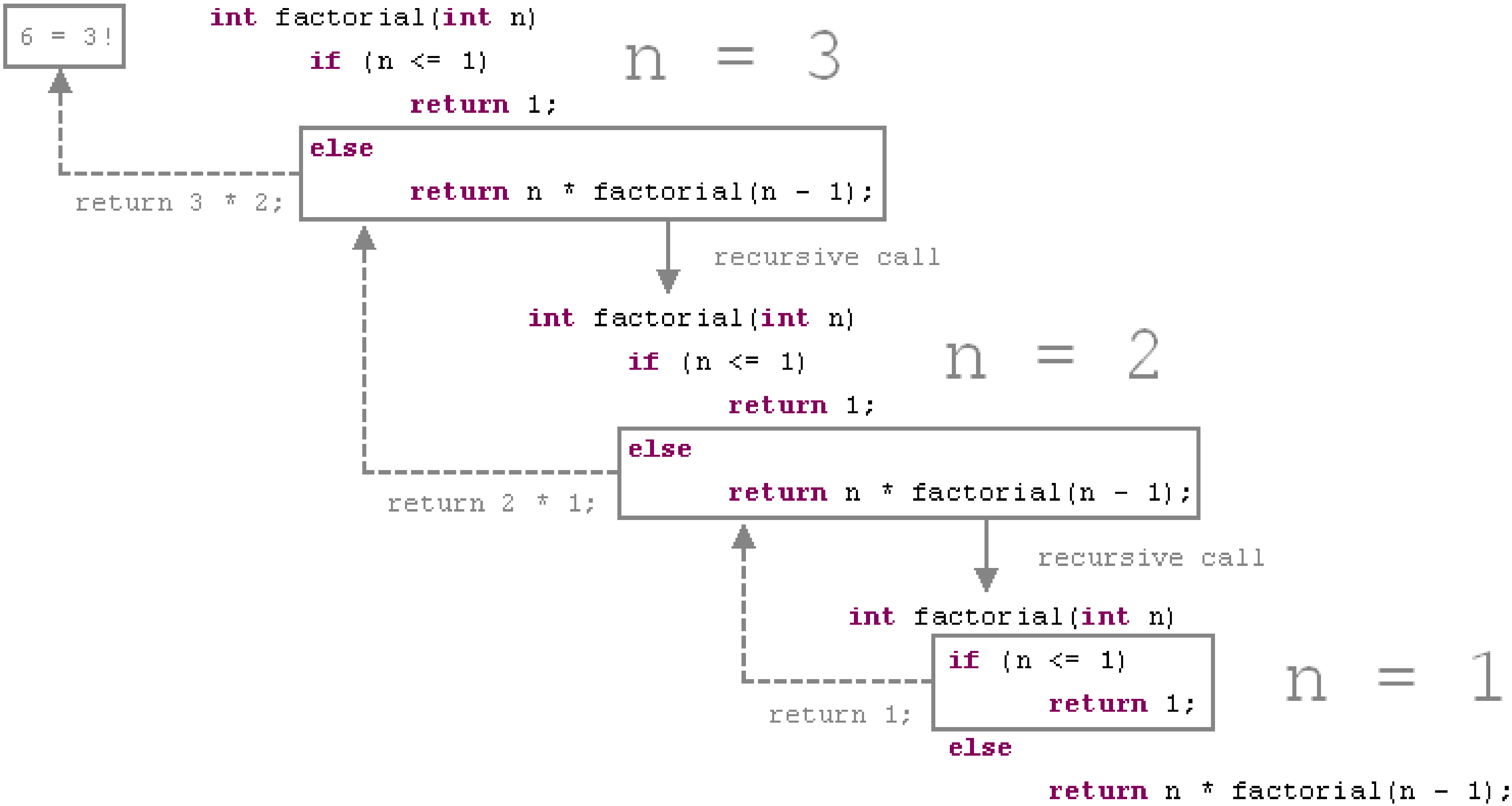
Example Factorial of a number

$$n\,! = \begin{cases} 1 & \textbf{n = 0 (Base case)} \\ n * (n - 1)\,! & \textbf{n > 0 (Recursive case)} \end{cases}$$

Recursive Functions:
Example

```c
#include <stdio.h>

int factorial(int n)
{
	if (n <= 1)
		return 1;
	else
		return n * factorial(n - 1);
}

int main()
{
	int res = factorial(3);

	printf("Res : %d", res);
}

```

Recursive Functions:
Example



File Handlings

Files : What & Why?

File is a sequence of bytes.

- Persistent storage.
- Theoretically unlimited size.
- Flexibility of putting any data type into it.

Files : Operations

- Opening the file
- Reading / Writing / Appending
- Closing the file

STEPS for processing the FILE

- File pointer declaration
- Opening the file using fopen()
- Processing the file using appropriate function
- Closing the file using fclose()

Opening a file

- The **fopen()** function is used to open the existing file or to create the new file.

Syntax

```
FILE *fopen(const char *path, const char *mode);
```

Description:

- ✔ The fopen() function opens the file whose name is the string pointed to by **path** and associates a stream with it.

- ✔ **Modes**

r	Open text file for reading. The stream is positioned at the beginning of the file.
w	Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
a	Open for appending (writing at end of file).The file is created if it does not exist.The stream is positioned at the end of the file.

For more details regarding the modes,
refer man pages of fopen()

Closing a file

- The **fclose()** function is used to close the stream.

Syntax

```
int fclose(FILE *fp);
```

Description:

✔ The fclose() function flushes the stream pointed to by fp and closes

the underlying file descriptor.

File Program:
Structure

```
#include <stdio.h>

int main()
{
        FILE *fp;
        fp = fopen("filename", "mode");
        :
        :
        :
        fclose(fp);
        return 0;
}

```

DIY:
WAP to display the contents of the given file

Input / Output Functions

Formatted & unformatted, block IO Diagram

UnFormatted : fgetc, fputc, getc, putc
Fputs, fgets

Formatted : fprintf, fscanf

Block: fwrite, fread

Random access to files: fseek, ftell, rewind

Unformatted I / O

- ***Reading from a file character by character***

fgetc() is the simplest funcion to read a single character from a file

```
int fgetc( FILE * fp );
```

Description:

fp = file pointer.

The return value is the character read, or in case of any error it returns EOF.

Unformatted I / O

- ***Writing to a file character by character***

fputc() is the simplest funcion to wrie a single character to a file

```
int fputc( int c, FILE *fp );
```

Description:

fp = file pointer.

✔ The function fputc() writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error.

Example

- *Snippet to demonstrate reading & writing charaters using fgetc() & fputc()*

```c
#include <stdio.h>

int main()
{
        :
        :
        :
        //Reading from a file and
        //writing to standard output
        while((ch = fgetc(fp)) != EOF)
        {
                fputc(ch, stdout);
        }
        :
        :
        return 0;

}
```

Exercise - 1

Program 1:

To count number of words from the given file.

Program 2:

To delete a specific line from a given file

Unformatted I / O

- ***Reading a string from a file***

fgets() is the simplest funcion to read a string from a file

```
char *fgets( char *buf, int n, FILE *fp );
```

Description:

The functions fgets() reads up to 'n' characters from the input stream referenced by fp. It copies the read string into the buffer buf, appending a null character to terminate the string.

Unformatted I / O

- ***Writing a string to a file***

fputs() is the simplest funcion to write a string to a file

```
int fputs( const char *s, FILE *fp );
```

Description:

The function fputs() writes the string s to the output stream referenced by fp. It returns a non-negative value on success, otherwise EOF is returned in case of any error.

Example

- *Reading & Writing a string using fgets & fputs functions*

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

int main()
{
	FILE *fptr;
	char str[80];

	//Open the file in read mode
	//Check for error
	if ((fptr = fopen("test", "r")) == NULL)
	{
		printf("Error: In opening the file\n");
		exit(1);
	}

	//Reading the string & writing
	while (fgets(str, 80, fptr) != NULL)
	{
		fputs(str, stdout);
	}

	fclose(fptr);
	return 0;
}

```

Exercise - 2

Program 1:

To append the content of file at the end of another

Program 2:

Program that merges lines alternatively from 2 files & print result

Block I / O

- ***Reading a block of data from a file using fread() function***

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Description:

The function fread() reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

Return value:

On success, fread() return the number of items read or written. This number equals the number of bytes transferred only when size is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).

fread() does not distinguish between end-of-file

Block I / O

- *Reading a block of data from a file using fread() function*

Examples:

1. To read a single float value from the file and storing in the variable var.

```
fread(&var, sizeof(float), 1, fp);
```

2. To read array of ints from the file and storing in the array arr.

```
fread(arr, sizeof(arr), 1, fp);
```

3. To read 6 ints from the file and storing in the array arr.

```
fread(arr, sizeof(arr), 5, fp);
```

Block I / O

- ***Writing a block of data to a file using fwrite() function***

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,FILE *stream);
```

Description:

The function fwrite() writes nmemb elements of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

Return value:

On success, fwrite() return the number of items read or written. This number equals the number of bytes transferred only when size is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero).

Block I / O

- ***Writing a block of data to a file using fwrite() function***

Examples:

1. To write a single float value to a file, stored in the variable var.

```
fwrite(&var, sizeof(float), 1, fp);
```

2. To write an array of ints to a file, stored in the array arr.

```
fwrite(arr, sizeof(arr), 1, fp);
```

3. To write only 6 ints to the file, stored in the array arr.

```
fwrite(arr, sizeof(arr), 6, fp);
```

Exercise - 3

Program 1:

To compare two binary files, printing the first byte position where they differ.

Random Access to file

- fseek( )
- ftell( )
- rewind( )

Random Access to file

fseek() : This function is used for setting the file pointer at the specified byte.

Syntax:

```
int fseek(FILE *stream, long offset, int whence);
```

Description:

- ✔ The fseek() function sets the file position indicator for the stream pointed to by stream.
- ✔ The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence.
- ✔ If whence is set to SEEK_SET, SEEK_CUR, or SEEK_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.

Random Access to file

fseek() : Examples

1. fseek(fp, 5L, 1);

fp is moved 5 bytes forward from the current position

2. fseek(fp, -6L, SEEK_END);

Fp is moved 6 bytes backward from the end of the file

3. fseek(fp, 0L, 0);

After this statement, fp is positioned to the begining of the file.

Random Access to file

ftell() : This function returns the current position of the file pointer.

Syntax:

```
long ftell(FILE *stream);
```

Description:

- ✔ The ftell() function obtains the current value of the file position indicator for the stream pointed to by stream.
- ✔ ftell() returns the current offset.

Random Access to file

rewind : This function is used to move the file pinter to the begining of the file.

Syntax:

```
void rewind(FILE *stream);
```

rewind(fp) <=> fseek(fp, 0L, 0)

Pointers : Sharp Knives
Handle with Care!

Pointers : Why

- To have C as a low level language being a high level language.
- To have the dynamic allocation mechanism.
- To achieve the similar results as of ”pass by variable” parameter passing mechanism in function, by passing the reference.
- Returning more than one value in a function.

Pointers & Seven rules

Rule #1

Pointer as a integer variable

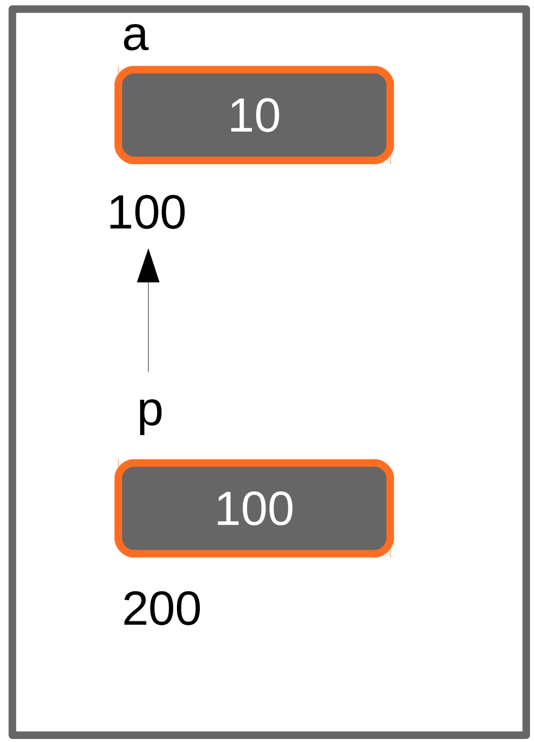
Syntax

*dataType *pointer_name;*

Example

```c
int a = 10;

int *p;

p = &a;

```

Pictorial Representation

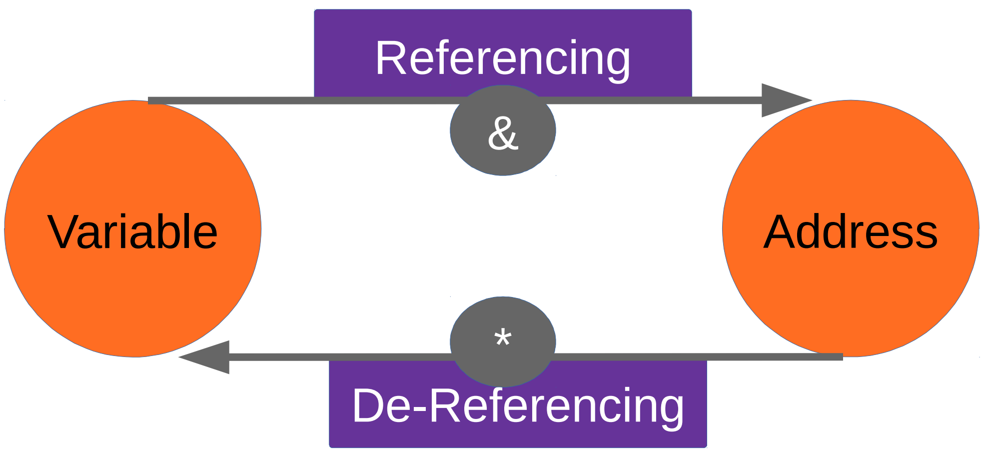


DIY:
Declare the float pointer & assign the address of float variable

EMERTXE

Rule #2

Referencing & Dereferencing



Example

```c
int a = 10;

int *ptr = &a;

int value = *ptr;

```

DIY:

Print the value of double using the pointer.

Rule #3

Type of a pointer

- Pointer of type t ≡ t Pointer ≡ (t *) ≡ A variable which contains an address,which when dereferenced becomes a variable of type t

All pointers are of same size.

Rule #4

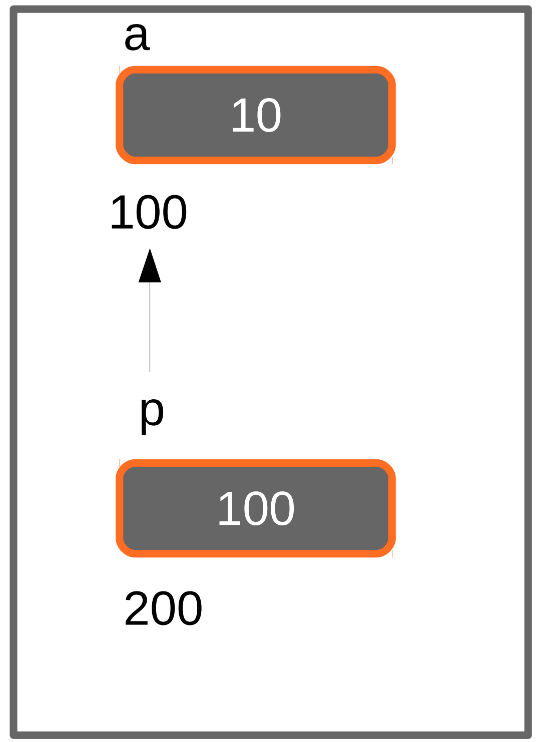
Value of a pointer

- Pointer pointing to a variable ≡
- Pointer contains the address of the variable

Example

```c
int a = 10;

int *p;

p = &a;

```

Pictorial Representation



Pointing means Containing

Rule #5

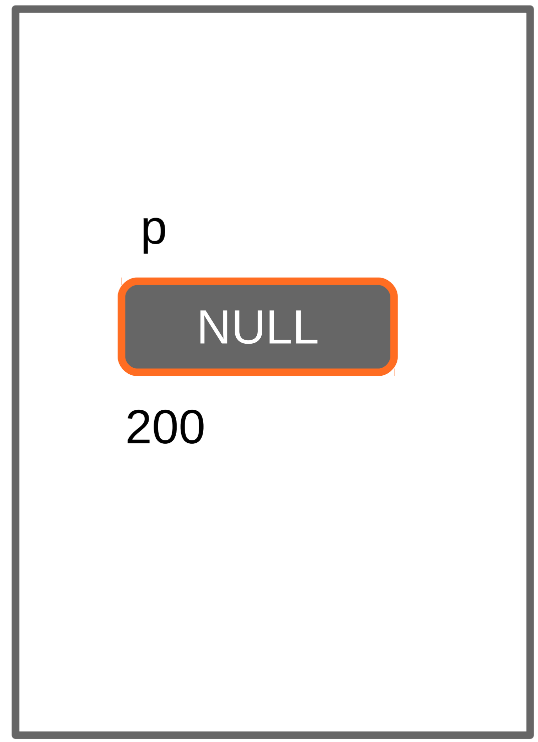
NULL pointer

- Pointer Value of zero ≡ Null Addr ≡ NULL pointer ≡ Pointing to nothing

Example

```
int *ptr = NULL;
```

Pictorial Representation



Not pointing to anywhere

Segmentation fault

A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed.

Example

```
#include <stdio.h>

int main()
{
        int a;
        printf("Enter the number\n");
        scanf("%d", a);
}

```

Fault occurs, while attempting to write to a read-only location, or to overwrite part of the operating system

Bus error

A ***bus error*** is a fault raised by hardware, notifying an operating system (OS) that a process is trying to access memory that the CPU cannot physically address: an invalid address for the address bus, hence the name.

Example

```c
int main()
{
        char a[sizeof(int) + 1];
        int *x, *y;
        x = &a[0];
        y = &a[1];
        scanf("%d%d", x, y);
}
```

DIY: Write a similar code which creates bus error

Rule #6:

Arithmetic Operations with Pointers & Arrays

- value(p + i) ≡ value(p) + value(i) * sizeof(*p)

- **Array → Collection of variables vs Constant pointer variable**
- short sa[10];
- &sa → Address of the array variable
- sa[0] → First element
- &sa[0] → Address of the first array element
- sa → Constant pointer variable

- **Arrays vs Pointers**
- Commutative use
- (a + i) ≡ i + a ≡ &a[i] ≡ &i[a]
- *(a + i) ≡ *(i + a) ≡ a[i] ≡ i[a]
- constant vs variable

Rule #7:

Static & Dynamic Allocation

- Static Allocation ≡ Named Allocation -
- Compiler's responsibility to manage it – Done internally by compiler,
- when variables are defined

- Dynamic Allocation ≡ Unnamed Allocation -
- User's responsibility to manage it – Done using malloc & free

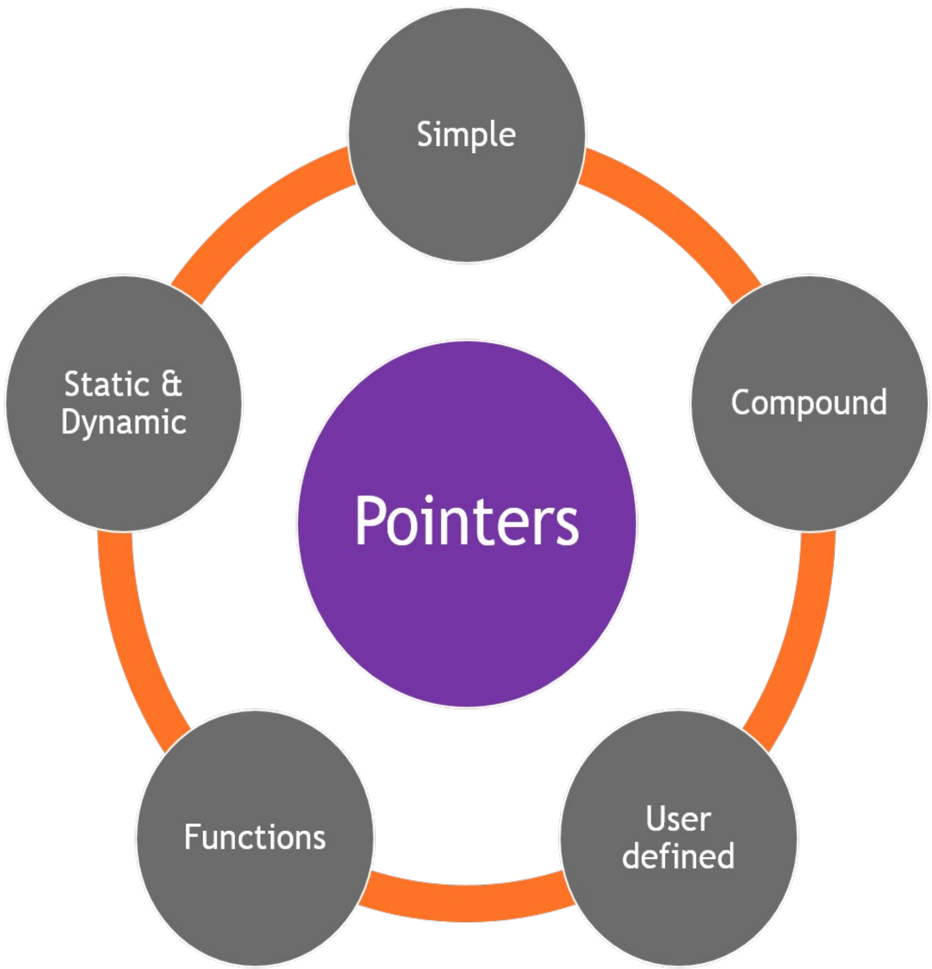
- **Differences at program segment level**

- Defining variables (data & stack segmant) vs Getting & giving it from
- the heap segment using malloc & free

- int x, int *xp, *ip;
- xp = &x;
- ip = (int*)(malloc(sizeof(int)));

Pointers: Big Picture

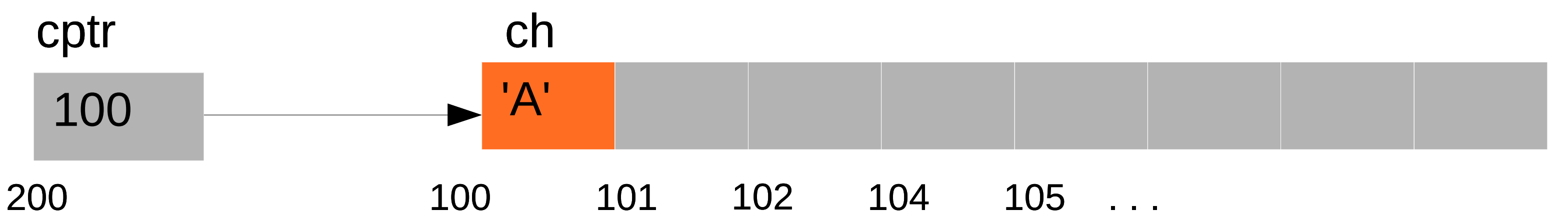
Pointers: Big Picture



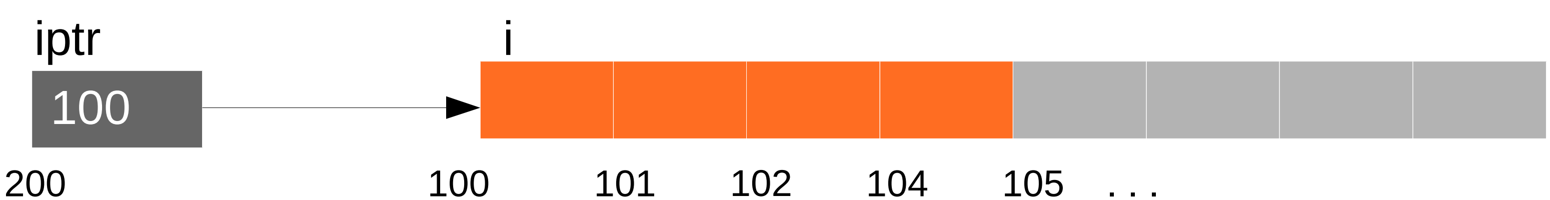
Pointers :

Simple data Types

```c
char ch = 'A';
char *cptr = &ch;

```




```c
int i = 10;
int *iptr = &i;

```

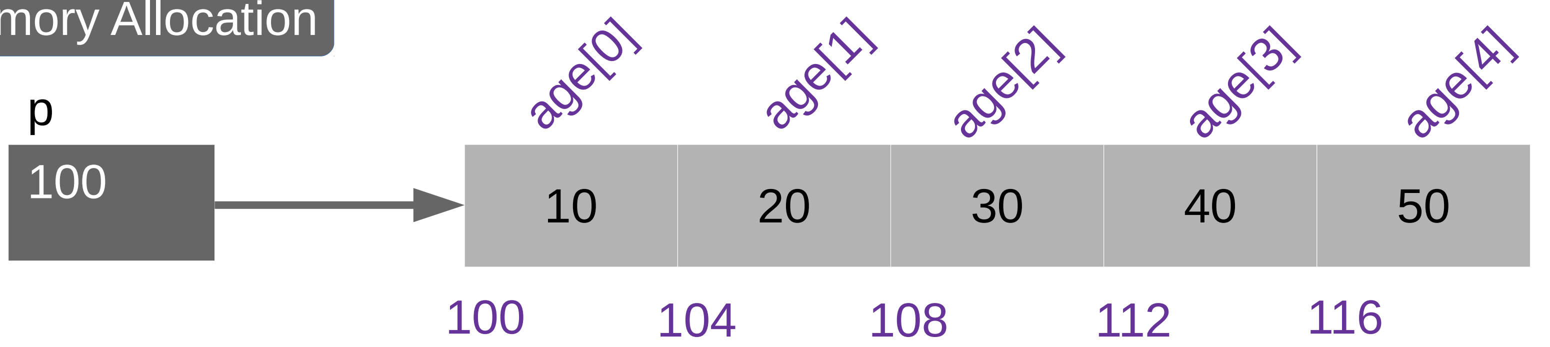


Pointers :
Compound data Types

Example: Arrays & Strings (1D arrays)

```
int age[ 5 ] = {10, 20, 30, 40, 50};
int  *p = age;
```

Memory Allocation



age[i] ≡ *(age + i)
≡ ≡
i[age] ≡ *(i + age)

age[2] = *(age + 2) = *(age + 2 * sizeof(int))
= *(age + 2 * 4)
= *(100 + 2 * 4)
= *(108)
= 30 = *(p + 2) = p[2]

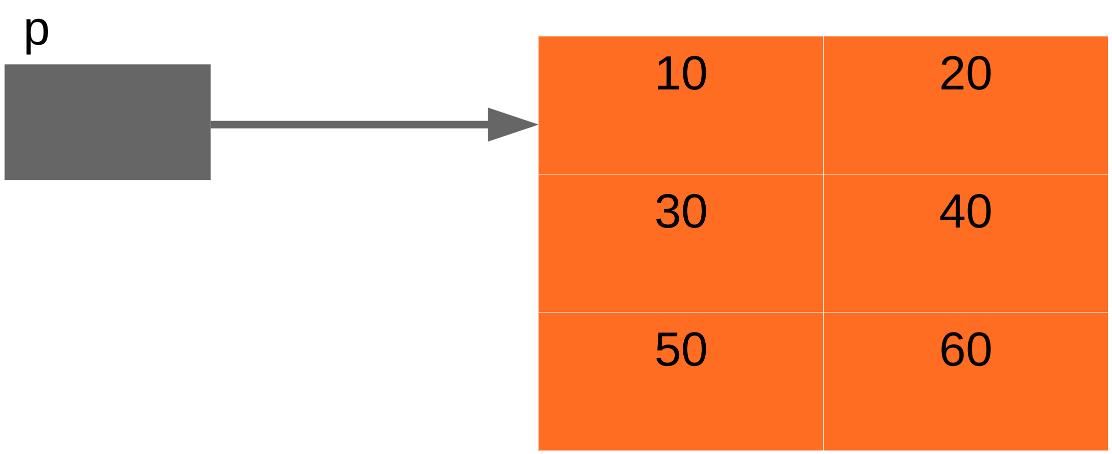
DIY : Write a program to add all the elements of the array.

Pointers :
Compound data Types

Example: Arrays & Strings (2D arrays)

```
int a[ 3 ][ 2 ] = {10, 20, 30, 40, 50, 60};
int  ( * p) [ 2 ] = a;
```

Memory Allocation



Pointers :
Compound data Types

Example: Arrays & Strings (2D arrays)

```
int a[ 3 ][ 2 ] = {10, 20, 30, 40, 50, 60};
int  ( * p) [ 2 ] = a;
```

a[2][1] = *(*(age + 2) + 1) =   *(*(a + 2 * sizeof(1D array)) + 1 * sizeof(int))
= *(*(a + 2 * 8) + 1 * 4)
= *(*(100 + 2 * 8) + 4)
= *(*(108) + 4)
= *(108 + 4)
= *(112)
= 40 = p[2][1]

In general :
a[i][j] ≡ *(a[i] + j) ≡ *(*(a + i) + j) ≡ (*(a + i))[j] ≡ j[a[i]] ≡ j[i[a]] ≡ j[*(a + i)]

DIY : Write a program to print all the elements of the 2D array.

Dynamic Memory Allocation

- In C functions for dynamic memory allocation functions are
- declared in the header file <stdlib.h>.

- In some implementations, it might also be provided
- in <alloc.h> or <malloc.h>.

- **malloc**
- **calloc**
- **realloc**
- **free**

Malloc

- The malloc function allocates a memory block of size size from dynamic memory and returns pointer to that block if free space is available, other wise it returns a null pointer.

- **<u>Prototype</u>**

- **void *malloc(size_t size);**

Calloc

- The calloc function returns the memory (all initialized to zero)
- so may be handy to you if you want to make sure that the memory
- is properly initialized.

- calloc can be considered as to be internally implemented using
- malloc (for allocating the memory dynamically) and later initialize
- the memory block (with the function, say, memset()) to initialize it to zero.

- **Prototype**

- **void *calloc(size_t n, size_t size);**

Realloc

- **The function realloc has the following capabilities**
- 1. To allocate some memory (if p is null, and size is non-zero,
- then it is same as malloc(size)),
- 2. To extend the size of an existing dynamically allocated block
- (if size is bigger than the existing size of the block pointed by p),
- 3. To shrink the size of an existing dynamically allocated block
- (if size is smaller than the existing size of the block pointed by p),
- 4. To release memory (if size is 0 and p is not NULL
- then it acts like free(p)).
- **Prototype**
- **void *realloc(void *ptr, size_t size);**

free

The free function assumes that the argument given is a pointer to the memory that is to be freed and performs no heck to verify that memory has already been allocated.

1. if free() is called on a null pointer, nothing happens.

2. if free() is called on pointer pointing to block other than the one allocated by dynamic allocation, it will lead to undefined behavior.

3. if free() is called with invalid argument that may collapse the memory management mechanism.

4. if free() is not called on the dynamically allocated memory block after its use, it will lead to memory leaks.

<u>Prototype</u>

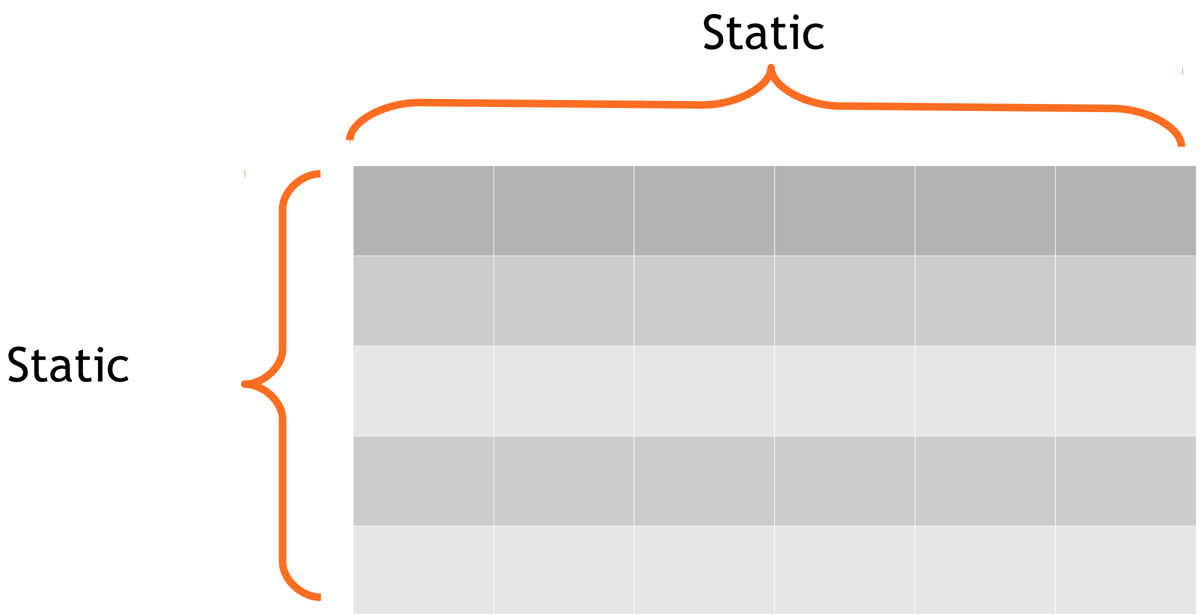
void free(void *ptr);

2D Arrays

- ✓Each Dimension could be static or Dynamic
- ✓Various combinations for 2-D Arrays (2x2 = 4)
  - C1: Both Static (Rectangular)
  - C2: First Static, Second Dynamic
  - C3: First Dynamic, Second Static
  - C4: Both Dynamic
- ✓2-D Arrays using a Single Level Pointer

C1: Both static

- ✓ Rectangular array
- ✓ int rec [5][6];
- ✓ Takes totally 5 * 6 * sizeof(int) bytes

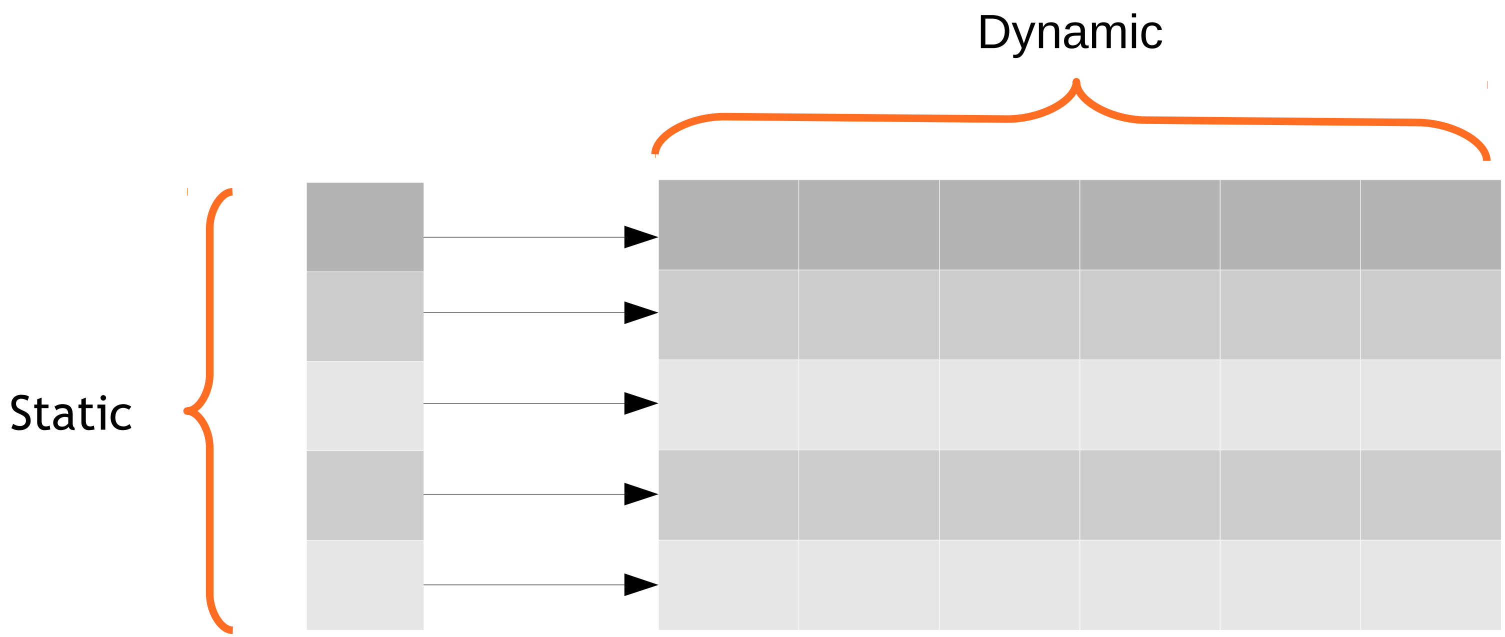


C2: First static, Second dynamic

- ✓ One dimension static, one dynamic (Mix of Rectangular & Ragged)

```
int *ra[5];
for( i = 0; i < 5; i++)
    ra[i] = (int*) malloc( 6 * sizeof(int));
```

- ✓ Total memory used : 5 * sizeof(int *) + 6 * 5 * sizeof(int) bytes

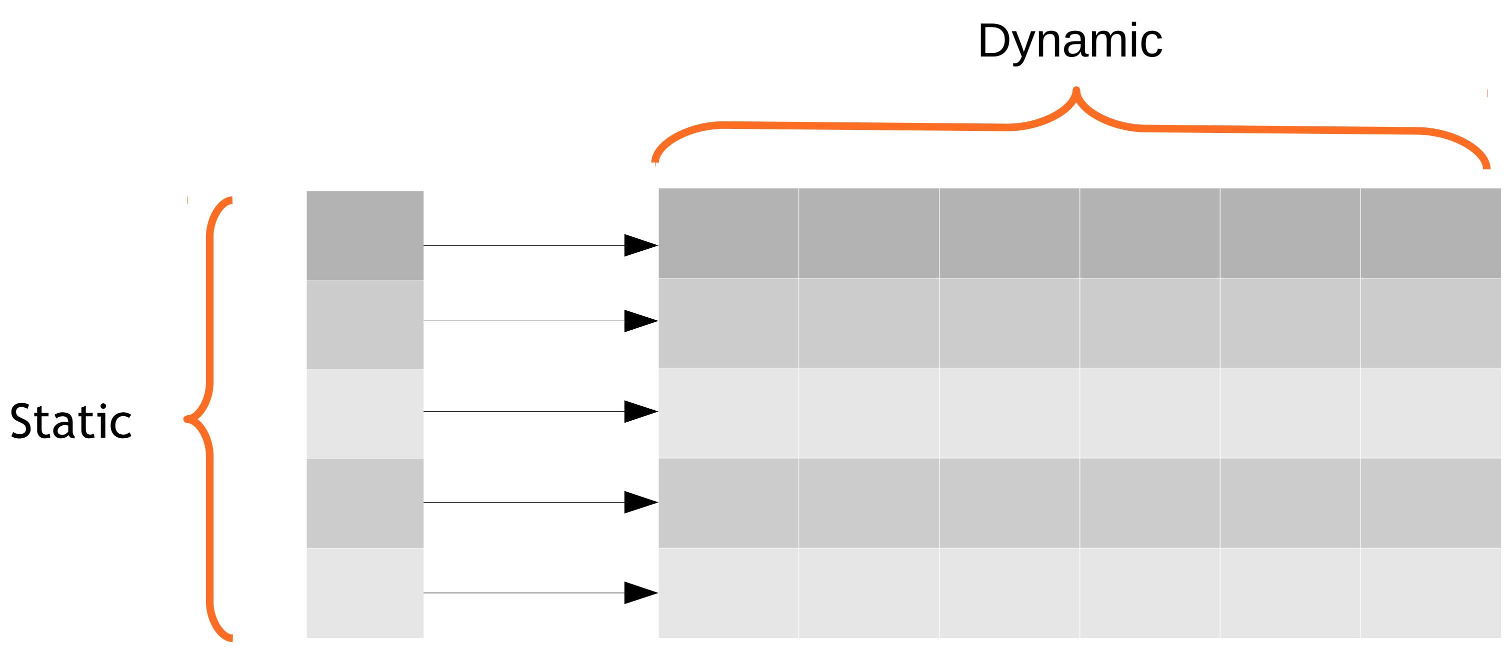


C2: First static, Second dynamic

- ✓ One dimension static, one dynamic (Mix of Rectangular & Ragged)

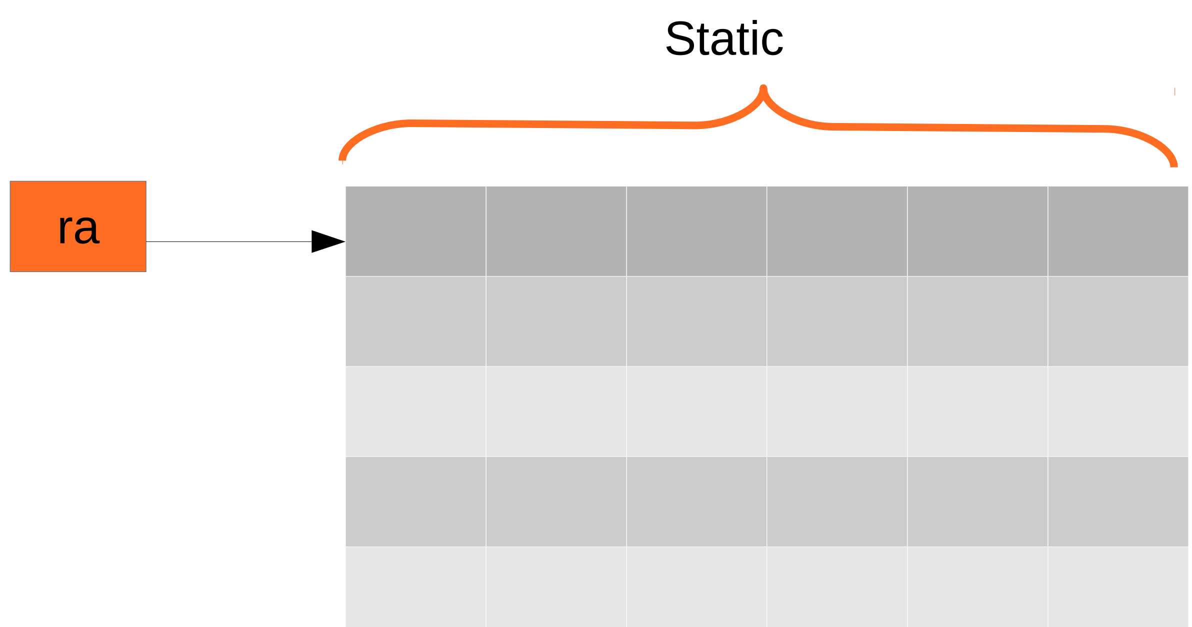
```
int *ra[5];
for( i = 0; i < 5; i++)
    ra[i] = (int*) malloc( 6 * sizeof(int));
```

- ✓ Total memory used : 5 * sizeof(int *) + 6 * 5 * sizeof(int) bytes



C3: Second static, First dynamic

- ✓ One static, One dynamic
  - int (*ra)[6]; (Pointer to array of 6 integer)
  - ra = (int(*)[6]) malloc(5 * sizeof(int[6]));
- ✓ Total memory used : sizeof(int *) + 6 * 5 * sizeof(int) bytes



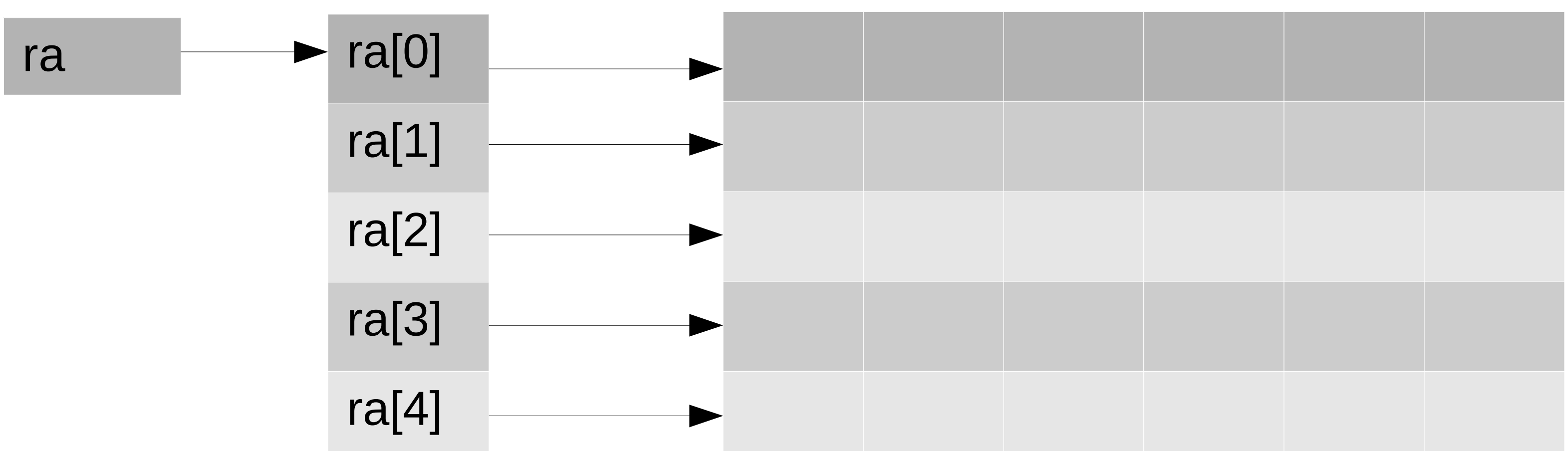
C4: Both dynamic

✓ Ragged array

```
int **ra;
ra = (int **) malloc (5 * sizeof(int*));
for(i = 0; i < 5; i++)
        ra[i] = (int*) malloc( 6 * sizeof(int));
```

✓ Takes 5 * sizeof(int*) for first level of indirection

✓ Total memory used : 1 * sizeof(int **) + 5 * sizeof(int *) + 5 * 6 * sizeof(int) bytes



Function Pointers

Function pointers : Why

- Chunk of code that can be called independently and is standalone
- Independent code that can be used to iterate over a collection of objects
- Event management which is essentially asynchronous where there may be several objects that may be interested in ”Listening” such an event
- ”Registering” a piece of code and calling it later when required.

Function Pointers: Declaration

Syntax

*return_type (*ptr_name)(type1, type2, type3, ...)*

Example

```
float (*fp)( int );
```

Description:

fp is a pointer that can point to any function that returns a float value and accepts an int as an argument.

Function Pointers: Example

```
#include <stdio.h>

int main()
{
        int a = 10, b = 20;

        int (*fp)(int, int) = add;

        /*Invoking the function through
        the function pointer*/
        int res = (*fp)(a, b);

        printf("Res : %d\n", res);

        return 0;
}

int add(int a, int b)
{
        return (a + b);
}

```

Function Pointers: As an argument

```
#include <stdio.h>
void foo(char, void(*fp)(float));
void fool(float);

int main()
{
        //Calling foo()
        foo('A', fool);
        return 0;
}

void foo(char ch, void (*fp)(float))
{
        //Invoking fool() throu' pointer
        (*fp)(8.5);
}

void fool(float f)
{
        printf("%f", f);
}

```

Function Pointers: More examples

- The bsearch function in the standard header file <stdlib.h>

```
void *bsearch(void *key, void *base, size_t num, size_t width,
int (*compare)(void *elem1, void *elem2));
```

- The last parameter is a function pointer.
- It points to a function that can compare two elements (of the sorted array, pointed by base) and return an int as a result.
- This serves as general method for the usage of function pointers. The bsearch function does not know anything about the elements in the array and so it cannot decide how to compare the elements in the array.
- To make a decision on this, we should have a separately function for it and pass it to bsearch.
- Whenever bsearch needs to compare, it will call this function to do it. This is a simple usage of function pointers as callback methods.

Function Pointers: More examples

- Function pointers can be registered & can be called when the program exits

```
#include <stdio.h>

int atexit(int (*)void);

int my_function ()
{
	printf("Exiting the program \\n");
	return 0;
}

int main()
{
	printf("Inside main\n");
	atexit(my_function);
	printf("About to quit\n");
}

```

Output ?

Preprocessors

Preprocessor: what ?

- Preprocessor is a powerful tool with raw power
- Preprocessor is often provided as a separate tool with the C compilers
- Preprocessor is a program that processes the code before it passes through the compiler

Preprocessor:
Functionalities

- Inclusion of header files
- Macro expansion
- Conditional compilation
- Removing comments from the source code
- Processing of trigraph sequence.

Preprocessor:
Built-in Defines

__FILE__ : Represents the current source file name in which it appears.

__LINE__ : Represents the current line number during the preprocessing in the source file.

__DATE__ : Represents the current date during the preprocessing.

__TIME__ : Represents the current time at that point of preprocessing.

__STDC__ : This constant is defined to be true if the compiler conforms to ANSI/ISO C standard.

__func__ : Represents the current function name in which it appears

Preprocessor:
Built-in Defines Example

```c
#include<stdio.h>

int main()
{
    printf("Error in %s @ %d on %s @ %s @ %s @ %d \n",__FILE__,
    __LINE__, __DATE__, __TIME__, __func__, __STDC__);

    return 0;
}

```

Output

```
Error in test.c @ 5 on Feb 16 2015 @ 15:42:46 @ main @ 1
```

Preprocessor:
Directives

- Preprocessing directives are lines in your program that start with `#'.
- The `#' is followed by an identier that is the directive name.

For example,

#include <stdio.h>

- #include
- #define
- #undef
- #ifdef, #ifndef, #else, #endif
- #error
- #line
- #pragma

Directive:
#include

Copy of a specified file included in place of the directive

#include <filename>

- Searches standard library for file
- Use for standard library files

#include "filename"

- Searches current directory, then standard library
- Use for user-defined files

Used for:

- Programs with multiple source files to be compiled together

Header file – has common declarations and definitions (classes, structures, function prototypes)

Directive:
#define

- This is used to create symbolic constants and macros
- When program is preprocessed, all occurrences of symbolic constant replaced with replacement text

Syntax

`#define` *identifier replacement-text*

Example

```
#define PI 3.14159
```

PI is a Macro

Macros with Arguments

- Operation defined in **#define**
- A macro without arguments is treated like a symbolic constant
- A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
- Performs a text substitution – no data type checking

Example

#define CIRCLE_AREA(x) (PI * (x) * (x))

Would cause

area = CIRCLE_AREA(4);

To become

area = (3.14159 * (4) * (4));

Macros with Arguments

DIY

1. Write a macro to find max of two given nos.

2. Write a macro to set nth bit of given character.

Directive: #undefine, #ifdef, #ifndef, #else, #endif

```
#include<stdio.h>
#define ABC 25
#ifdef ABC
        #undef ABC
        #define ABC 50
#else
        #define ABC 100
#endif

int main()
{
        printf("%d",ABC);
        return 0;
}
```

Directive:
#error

Syntax

#error tokens

- Tokens are sequences of characters separated by spaces **"Error cannot process"** has **3** tokens
- Displays a message including the specified tokens as an error message
- Stops preprocessing and prevents program compilation

Directive:
#error

Output ?

```c
#include <stdio.h>

#ifndef MAX
        #error Define MAX first and Compile
#endif

int main()
{
        printf("%d\n", MAX);
        return 0;
}
```

ΣMERTXE

Directive:
#line

- Renumbers subsequent code lines, starting with integer value
- File name can be included
- **#line 100 "myFile.c"**
  - Lines are numbered from **100** beginning with next source code file
  - Compiler messages will think that the error occurred in **"myfile.C"**
  - Makes errors more meaningful
  - Line numbers do not appear in source file

Preprocessor Operators

(stringization operator)

- Causes a replacement text token to be converted to a string surrounded by quotes

- The statement

#define HELLO(x) printf(“Hello, ” #x “\n”);

would cause

HELLO(John)

to become

printf(“Hello, ” “John” “\n”);

Strings separated by whitespace are concatenated when using **printf**

Preprocessor Operators

(concatenation operator)

- **Concatenates two tokens**

- The statement

  #define TOKENCONCAT(x, y) x ## y

  would cause

  TOKENCONCAT(O, K)

  to become

  OK

Preprocessor Operators

Example

```c
#define print(expr) printf(#expr "=%d", expr);
#define CAT(x, y) (x##y)
#define STRFY(x) #x

int main()
{
        int CAT(x, 0);
        CAT(x, 0) = 5;
        printf(STRFY(CAT(Hello, World)));
        print(CAT(x, 0));
}

```

Advantages & Disadvantages

Macros are not type-checked

<u>Examples:</u>

```
int k = max_macro(i, j);
/* works with int */

float max_float = max_macro(10.0, 20.0);
/* also works with float constants */

int k = max_function(i, j);
/* works with int */

float max_float = max_function (10.0, 20.0);
/* does not work - you can pass only integral values */
```

Advantages & Disadvantages

Macros have side effects during textual replacement whereas functions does not have

Examples:

```
int k = max_macro (i++, j);
/* we are in trouble as i++ is evaluated twice */
/* int k = (i++ > j) ? i++ : j */


int k = max_function (i++, j);
/* no problem, as it is not expanded, but a call to max_function is made */
```

Macros might result in faster code as textual replacement is done and no function call overhead is involved.

User Defined Datatypes

Structures

Structures:
What

What ?

- It is user defined data type.
- It is used to group together different types of variables under the same name.

Example

To create the student record, which consist of

- ✔ Name
- ✔ Roll number
- ✔ Age
- ✔ Marks etc...

Structures:
why

Why ?

- It helps to construct a complex data type in more meaningful manner.
- Organising the data in a better & efficient way.

Arrays & Structures

Arrays	Structures
Collection of similar data types	Collections of different data types

Structures:
Declaration

Declaration

```
struct tagname
{
        dataType member1;
        dataType member2;
        dataType member3;
        dataType member4;
        :
        :
        :
};
```

Example

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};

```

DIY:

Declare the structure called date, which contains the memebers
day, month, year

Structures:
Declaration of variables

Declaration of structures variables can be done in two ways

- With structure declaration
- Using structure tag

With structure declaration	Using structure tag

`struct student`
`{`
`        char name[20];`
`        int age;`
`        int roll_num;`
`        float marks;`
`}s1, s2, s3;`
	`struct student`
	`{`
	`        char name[20];`
	`        int age;`
	`        int roll_num;`
	`        float marks;`
	`};`
	`struct student s1, s2, s3;`

Declaring the structure variable ***reserves*** the space in memory

Structures:
Initialization of variables

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
}s1 = {"Kenneth", 25, 43, 67.25};

```

With structure declaration

Using structure tags

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};
struct student s1 = {"Kenneth", 25, 43, 67.25};

```

The number, order, type of these variables should be same as in the structure template

Structures:
Initialization of variables

Invalid Initialization

```c
struct student
{
        char name[20];
        int age;
        int roll_num = 45;//Invalid Initialization
        float marks;
};
_
```

Members of the structures cannot be initialize while defining the structure

Structures:
Partial Initialization

Example

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};
struct student s = {"Kenneth"};

```

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};
struct student s = {"Kenneth", 0, 0, 0.0};
```

Structures:
Accessing the members

Format for accessing the members

variable.member

```
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};
struct student s1 = {"Kenneth", 25, 43, 67.25};
```

Example

s1.name
s1.age
s1.roll_num

Dot(.) operator is also called as period or membership operator

EMERTXE

Structures:
Accessing the members

Format for accessing the members

variable.member

```c
struct student
{
        char name[20];
        int age;
        int roll_num;
        float marks;
};
struct student s1 = {"Kenneth", 25, 43, 67.25};
```

Example

s1.name
s1.age
s1.roll_num

Dot(.) operator is also called as period or membership operator

Structures:

Assignment of structure variables

We can assign values of a structure variable to another structure variable, if both variables are of the *same structure type*.

Example

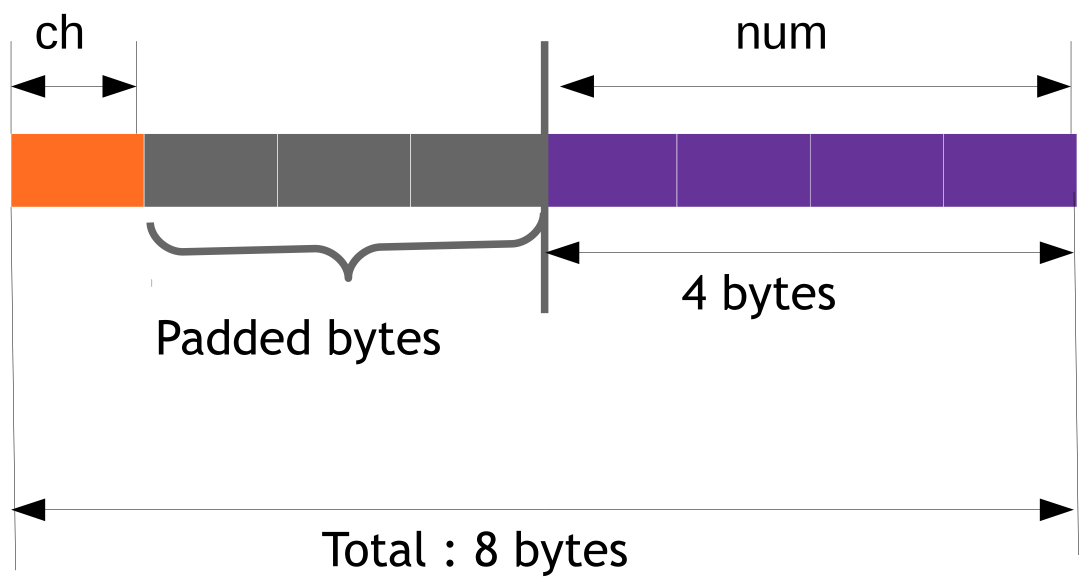
```
struct student
{
            char name[20];
            int age;
            int roll_num;
            float marks;
};
struct student s1 = {"Kenneth", 25, 43, 67.25};

//Copies s1 to s2
struct student s2 = s1;
```

Structures:

Storage Allocation & Padding

Let us consider an example,

```c
struct sample
{
        char ch;
        int num;
}var;
```



sizeof() operator can be used to find the total size of this structure

Structures:

Array of structures

Example:

```
struct student
{
            char name[20];
            int age;
            int roll_num;
            float marks;
};
struct student s[3] =  {
                         {"Kenneth", 25, 43, 67.25},
                         {"John", 24, 35, 76.50}
                         {"Richie", 26, 36, 80.00}
                        };
_
```

S [0]				S [1]				S [2]			
name	age	roll_ num	marks	name	age	roll_num	marks	name	age	roll_num	marks
Kenneth	25	43	67.25	John	24	35	76.50	Richie	26	36	80.00

Total : 32 * 3 = 96 bytes of memory

All the structures of an array are stored in consecutive memory location

Structures:

Nested structures

Structures within structure

```
struct tag1
{
        member1;
        member2;
        :
        :
        struct tag2
        {
                member1;
                member2;
                :
                :
        }var1;
        :
}var2;
```

Example

```
struct date
{
        int day;
        int month;
        int year;
        struct time
        {
                int hour;
                int min;
                int secs;
        }t;
}dob;
```

Accessing the members within structure

var1.var2.member1

Example: dob.t.day;

Structures:
Pointers to structures

Example

```c
struct student
{
            char name[20];
            int age;
            int roll_num;
            float marks;
};
struct student s1 = {"Kenneth", 25, 43, 67.25};

```

Defining the pointer

student *sptr = &s1;

Accessing the members thru' ptr

struct

syntax

pointer -> member;

Example

sptr->name, sptr->age

Use -> operator to access the members using pointers

Structures:

Functions & structures

Passing structure members as Arguments

```c
struct student
{
        char name[20];
        int age;
        float marks;
};
```

Calling function

```c
int main()
{
        struct student s =  {"Kenneth", 25, 67.25};
        //Calling the function
        display(s.name, s.age, s.marks);
        :
        :
}
```

Called function

```c
void display(char *name, int age, float marks)
{
        :
        :
        :
}
```

Structures:

Functions & structures

Passing structure variables as Arguments

```
struct student
{
        char name[20];
        int age;
        float marks;
};
```

Calling function

```
int main()
{
        struct student s =  {"Kenneth", 25, 67.25};
        //Calling the function
        display(s);
        :
        :
}
```

Called function

```
void display(struct student s)
{
        :
        :
        :
}
```

Structures:
Functions & structures

Passing pointers to structures as Arguments

```c
struct student
{
        char name[20];
        int age;
        float marks;
};
```

Calling function

```c
int main()
{
        struct student s =  {"Kenneth", 25, 67.25};
        //Calling the function
        display(&s);
        :
        :
}
```

Called function

```c
void display(struct student *s)
{
        :
        :
        :
}
```

Structures:
Functions & structures

Returning structure variable from function

```c
struct student
{
        char name[20];
        int age;
        float marks;
};
```

Calling function

```c
int main()
{
        struct student s =  {"Kenneth", 25, 67.25};
        //Calling the function
        s = change(s);
        :
        :
}
```

Called function

```c
struct student change(struct student s)
{
        s.marks = s.marks + 2.00;
        :
        :
        return s;
}
```

EMERTXE

Structures:
Functions & structures

Returning a pointer to structure from function

```
struct student
{
        char name[20];
        int age;
        float marks;
};
```

Calling function

```
int main()
{
        struct student s =  {"Kenneth", 25, 67.25};
        //Calling the function
        struct student *sptr;
        sptr = change(&s);
        :
        :
}
```

Called function

```
struct student *change(struct student *s)
{
        s->marks = s->marks + 2.00;
        :
        :
        return s;
}
```

Structures:
Self referential structures

A structure that contains pointers to structures of its own type.

Syntax:

```
struct tag
{
      datatype member1;
      datatype member2;
      :
      :
      struct tag *ptr1;
      struct tag *ptr2;
};
```

Example:

```
struct node
{
        int data;
        struct node *link;
};
```

More details will be dealt in Data structures

Unions

Unions:
Introduction

- User defined data type
- The syntax used for declaration of a union, declaration of variables and accessing the members are all similar to that of structures, except the keyword ***'union'*** intead of ***'struct'***
- The main difference between ***union*** & ***structures*** is the way the memory is allocated for the members.
- In structures, each member has its own memory, whereas in the union members share the same memory locations.
- Compiler allocates sufficient memory to hold the largest member in the union.

Unions:
Examples

- Example to compare the memory allocation for the members of the unions & structures.

structures	unions

`struct sample`
`{`
`        char c;`
`        int i;`
`        float f;`
`};`
	`union sample`
	`{`
	`        char c;`
	`        int i;`
	`        float f;`
	`};`

sizeof(struct sample) = 12 bytes	sizeof(union sample) = 4 bytes Since highest data type among the members is ***'float'***

Unions:
Union inside structure

Example

```
struct result
{
        char name[20];
        int roll_num;
        union res
        {
                int percent;
                float gpa;
        }performance;
}data;
```

This structure has three members,

1. Name
2. Roll number
3. Union member performance

Union will take only one value at a time, either a percent rounded of to whole number or gpa

typedefs

typedefs:
Introduction

- The purpose of typedef is to form complex types from more-basic machine types and assign simpler names to such combinations.

- They are most often used when a standard declaration is cumbersome, potentially confusing, or likely to vary from one implementation to another.

- Under C convention (such as in the C standard library or POSIX), types declared with typedef end with '_t' (e.g., size_t, time_t). Such type names are reserved by POSIX for future extensions and should generally be avoided for user defined types.

typedefs:
Examples

Syntax

$$\underbrace{\textbf{typedef}}_{\text{Keyword}}\ \underbrace{\textbf{data\_type}}_{\text{Existing data type}}\ \underbrace{\textbf{new\_name;}}_{\text{Identifier}}$$

Example

typdef unsigned int long ul_t;

Now, ul_t type can be used to declare the variables of the type unsigned int long.

Typedefs & pointers
Examples

```
typedef  int*  iptr;
```

Now, iptr is synonym for the int * or pointer to int.

```
iptr p, q; // p, q are of type int*
```

```
iptr *p; // Here, p is declared as pointer to pointer to int
```

Typedefs & arrays
Examples

```
typedef int intarr [ 10 ];
```

Now, intarr is synonym for the integer array of size 10.

```
intarr a;
```

Above example is equivalent to int a [10];

```
intarr a [ 5 ];
```

Above example is equivalent to int a [5][10];

enums

Enums:
Introduction

- Set of named integer constants

Syntax:

```
enum tag { member1, member2, member3,...};
```

Example:

```
enum month { Jan, Feb, Mar, Apr, May, June};
```

- Internally the compiler treats these enumerators as integer constant.
- Enumerators are automatically assigned integer values begining from 0,1,2,...
- In the above example, Jan-0, Feb-1, Mar-2,...

Bit-fields

Bit-fields
Introduction

- A bit field is set up with a structure declaration that labels each field and determines its width.

Syntax:

```
struct tag
{
        datatype member1: width;
        datatype member2: width;
        datatype member3: width;
};
```

Example:

```
typedef struct control
{
        unsigned int code:15;
        unsigned int reset:3;
        unsigned int enable:1;
        unsigned int flags:12;
        unsigned int priority:1;
}control;
```

Stay connected

About us: Emertxe is India's one of the top IT finishing schools & self learning kits provider. Our primary focus is on Embedded with diversification focus on Java, Oracle and Android areas

Branch Office:
Emertxe Information Technologies,
No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046

Corporate Headquarters:
Emertxe Information Technologies,
83, Farah Towers, 1st Floor,
MG Road,
Bangalore, Karnataka - 560001
T: +91 809 555 7333 (M), +91 80 41289576 (L)
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

ΣMERTXE

THANK YOU