C

Come let's see how deep it is!!

Team Emertxe

EMERTXE

Introduction

SDLC - A Quick Introduction

Requirement

Design

Code

Test

- Understand the requirement properly
- Consider all the possible cases like inputs and outputs
- Know the boundary conditions
- Get it verified

Requirement

Design

Code

Test

- Have a proper design plan
- Use some algorithm for the requirement
  - Use paper and pen method
- Use a flow chart if required
- Make sure all the case are considered

Advanced C

SDLC - A Quick Introduction

Requirement

Design

Code

Test

- Implement the code based on the derived algorithm
- Try to have modular structure where ever possible
- Practice neat implementation habits like
  - Indentation
  - Commenting
  - Good variable and function naming's
  - Neat file and function headers

Requirement

Design

Code

Test

- Test the implementation thoroughly
- Capture all possible cases like
  - Negative and Positive case
- Have neat output presentation
- Let the output be as per the user requirement

Problem Solving – What?

- An approach which could be taken to reach to a solution
- The approach could be ad hoc or generic with a proper order
- Sometimes it requires a creative and out of the box thinking to reach to perfect solution

- Polya's rule
  - Understand the problem
  - Devise a plan
  - Carryout the Plan
  - Look back

How should I proceed with my Application?

- Never jump to implementation. Why?
  - You might not have the clarity of the application
  - You might have some loose ends in the requirements
  - Complete picture of the application could be missing

  and many more...

Algorithm – What?

- A procedure or formula for solving a problem
- A sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- Algorithms is needed to generate correct output in finite time in a given constrained environment
  - Correctness of output
  - Finite time
  - Better Prediction

Algorithm – How?

- Natural Language
- Pseudo Codes
- Flowcharts etc.,

Algorithm – Daily Life Example

- Let's consider a problem of reaching this room
- The different possible approach could be thought of
  - Take a Walk
  - Take a Bus
  - Take a Car
  - Let's Pool
- Lets discuss the above approaches in bit detail

The steps could be like

1. Start a 8 AM
2. Walk through street X for 500 Mts
3. Take a left on main road and walk for 2 KM
4. Take a left again and walk 200 Mts to reach

- Pros
  - You might say walking is a good exercise :)
  - Might have good time prediction
  - Save some penny
- Cons
  - Depends on where you stay (you would choose if you stay closer)
  - Should start early
  - Would get tired
  - Freshness would have gone

Algorithm – Reaching this Room – Take a Bus

The steps could be like

1. Start a 8 .30 AM
2. Walk through street X for 500 Mts
3. Take a left on main road and walk for 100 Mts to bus stop
4. Take Bus No 111 and get down at stop X and walk for 100 Mts
5. Take a left again and walk 200 Mts to reach

- Pros
  - You might save some time
  - Less tiredness comparatively
- Cons
  - Have to wait walk to the bus stop
  - Have to wait for the right bus (No prediction of time)
  - Might not be comfortable on rush hours

The steps could be like

1. Start a 9 AM
2. Drive through street X for 500 Mts
3. Take a left on main road and drive 2 KM
4. Take a left again and drive 200 Mts to reach

- Pros
  - Proper control of time and most comfortable
  - Less tiresome
- Cons
  - Could have issues on traffic congestions
  - Will be costly

The steps could be like

1. Start a 8.45 AM
2. Walk through street X for 500 Mts
3. Reach the main road wait for you partner
4. Drive for 2 KM on the main road
5. Take a left again and drive 200 Mts to reach

- Pros
  - You might save some time
  - Less costly comparatively
- Cons
  - Have to wait for partner to reach
  - Could have issues on traffic congestions

- All the above solution eventually will lead you to this room
- Every approach some pros and cons
- It would our duty as designer to take the best approach for the given problem

Advanced C

Algorithm – A Computer Example1

- Let's consider a problem of adding two numbers
- The steps involved :

  Start

  Read the value of A and B

  Add A and B and store in SUM

  Display SUM

  Stop
- The above 5 steps would eventually will give us the expected result

Algorithm – A Computer Example1 – Pseudo Code

- Let's consider a problem of adding two numbers
- The steps involved :

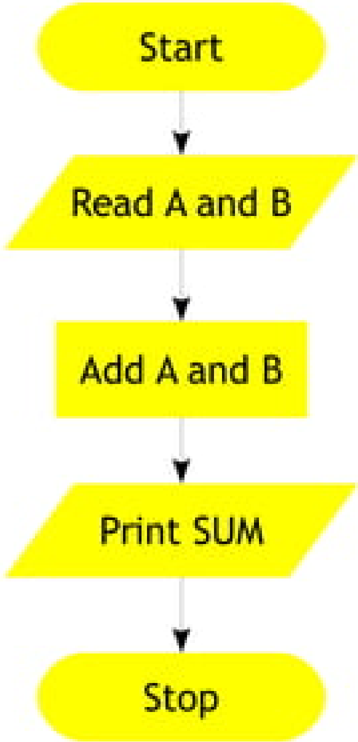
```
BEGIN
Read A, B
SUM = A + B
Print SUM
END
```

- The above 5 steps would eventually will give us the expected result

Advanced C

Algorithm – A Computer Example1 – A Flow Chart

- Let's consider a problem of adding two numbers



- Find the largest of given 2 numbers
- Find the average of given set of numbers
- Find out the given letter is vowel or not
- Find the sum of N natural numbers
- Find out the given number is odd or even
- Find out the given number is prime or not
- Find the largest of given 3 numbers
- Print the digits of a given number

- Write an algorithm to find Armstrong numbers between 0 and 999
  – An Armstrong number of three digits is an integer such that the sum of the cubes of its digits is equal to the number itself
  – For example, 371 is an Armstrong number since 3^3 + 7^3 + 1^3 = 371
- Generate all prime numbers <= N (given)
  – Example : if N = 98 then prime numbers 2,3,5,7 … 97 shall be printed on the screen

- Write an algorithm to print number pyramid

```
1234554321
1234__4321
123____321
12______21
1________1
```

Advanced C

Algorithm - DIY - Pattern (challenge)

- Print rhombus

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

Advanced C

Algorithm – DIY – Pattern (challenge)

- Print nested squares

```
*  *  *  *  *  *  *  *  *  *  *
*                             *
*     *  *  *  *  *  *  *     *
*     *                 *     *
*     *     *  *  *     *     *
*     *     *     *     *     *
*     *     *  *  *     *     *
*     *                 *     *
*     *  *  *  *  *  *  *     *
*                             *
*  *  *  *  *  *  *  *  *  *  *
```

Advanced C

Have you ever pondered how
- powerful it is?
- efficient it is?
- flexible it is?
- deep you can explore your system?

if [NO]
    Wait!! get some concepts right before you dive into it
else
    You shouldn't be here!!

Advanced C
Where is it used?

- System Software Development
- Embedded Software Development
- OS Kernel Development
- Firmware, Middle-ware and Driver Development
- File System Development

And many more!!

Advanced C

Language – What?

- A stylized communication technique
- Language has collection of words called "Vocabulary"
  - Rich vocabulary helps us to be more expressive
- Language has finite rules called "Grammar"
  - Grammar helps us to form infinite number of sentences
- The components of grammar :
  - The ***syntax*** governs the structure of sentences
  - The ***semantics*** governs the meanings of words and sentences

Advanced C

Language - What?

- A stylized communication technique
- It has set of words called “keywords”
- Finite rules (Grammar) to form sentences (often called expressions)
  - Expressions govern the behavior of machine (often a computer)
- Like natural languages, programming languages too have :
  - Syntactic rules (to form expressions)
  - Semantic rules (to govern meaning of expressions)

Advanced C

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

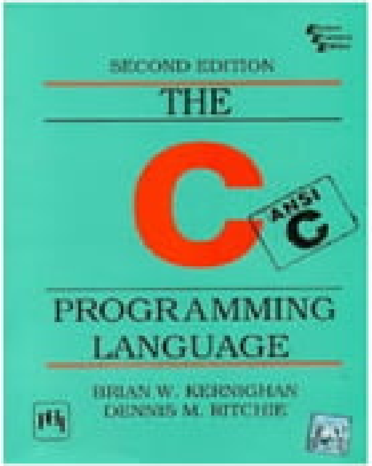
Portable, efficient and easy to use language was a dream.

Advanced C

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

Advanced 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

Advanced C

Important Characteristics

- Considered as a middle level language
- 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

- 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

Chapter 1

- 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 - Categories

- Data Types
- Qualifiers
- Loop
- Storage Class
- Decision
- Jump
- Derived
- User Defined
- Others

Advanced C

Keywords – Data Types

char

- The basic data type supported for storing characters of one byte size

int

- A variable would hold a integer type value
- A simple and efficient type for arithmetic operations
- Usually a word size of the processor although the compiler is free to choose the size
- ANSI C does not permit an integer, which is less than 16 bits

float

- A variable would hold a single precision value
- ANSI C does not specify any representation

double

- A variable would hold a double precision value
- The implementations may follow IEEE formats to represent floats and doubles, and double occupies 8-bytes in memory, Supports 0, ∞ (-/+) and NaN

signed	• A variable can hold both positive and negative value • Most Significant Bit (MSB) of the variable decides the sign
unsigned	• A variable can hold only positive value • Larger range of positive values within the same available space
short	• Fairly small integer type of value
long	• Fairly large integer type of value

- ANSI C says that the size of short and long implementation defined, but ensures that the non-decreasing order of char, short, int, and long is preserved i.e.,

char ≤ short ≤ int ≤ long

Advanced C
Keywords – Qualifiers

const	• It specifies the value of a field or a local variable that cannot be modified
volatile	• Instructs the complier not to optimize the variable qualified with it • Indicates that the variable is asynchronous, and system or external sources may change its value

for	• Could be used when the number of passes is known in advance (but not necessarily)
while	• Could be used when the number of passes is not known in advance (but not necessarily) • Entry controlled looping mechanism
do	• Could be used when the number of passes is not known in advance (but not necessarily) • Exit controlled looping mechanism

Advanced C

Keywords – Storage Class

auto	• Storage:	Memory
	• Default Initial value:	Unpredictable
	• Scope:	Local
	• Life:	Within the block
register	• Storage:	CPU Registers (if available)
	• Default Initial value:	Garbage
	• Scope:	Local
	• Life:	Within the block
static	• Storage:	Memory
	• Default Initial value:	Zero
	• Scope:	Local
	• Life:	Across function calls
	• Others	Limits the function's scope to the current file
extern	• Storage:	Memory
	• Default Initial value:	Zero
	• Scope:	Global
	• Life:	Program Life

Advanced C

Keywords – Decision

if	• Simple conditional branching statement • Any non-zero value is treated as a true value and will lead to execution of this statement
else	• Used with if statements • Else part is executed when the condition in if becomes false
switch	• A specialized version of an if-else cascade • Equality checks only with integral type constants • Simplified control-flow by generating much faster code • Break keyword is must to end the current case
case	• Used in switch statements for selecting a particular case • The case should be followed by a constant integral expression.
default	• This label is used in switch statements • The statements after this label will be executed only when there is no match found in the case labels.

goto	• Take the control to required place in the program
break	• Force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop
continue	• Take the control to the beginning of the loop bypassing the statements inside the loop

struct
- struct keyword provides support for aggregate types

unions
- Can be considered a special case of structures
- The syntax for both is mostly the same and only the semantics differ
- Memory is allocated such that it can accommodate the biggest member
- There is no in-built mechanism to know which union member is currently used to store the value

Advanced C
Keywords – User Defined

typedef
- Do not create new types - they just add new type names
- Helpful in managing complex declarations
- Increase portability of the code
- Obey scoping rules and are not textual replacements (as opposed to #defines)

enum
- A set of named constants that are internally represented as integers
- Makes the code more readable and self-documenting

void	• Non-existent or empty set of values • It is used in the case of void pointers as a generic pointer type in C • Return type of a function to specify that the function returns nothing • You cannot have objects of type void, and hence this type is sometimes called a pseudo-type.
return	• To return control back from the called method
sizeof	• Used to obtain the size of a type or an object • Can be used for portable code, since the size of a data type may differ depending on the implementation.

Advanced C

Typical C Code Contents

Documentation

Preprocessor Statements

Global Declaration

The Main Code:

Local Declarations
Program Statements
Function Calls

One or many Function(s):

The function body

- A typical code might contain the blocks shown on left side
- It is generally recommended to practice writing codes with all the blocks

Advanced C

Anatomy of a Simple C Code

`/* My first C code */`	File Header
`#include <stdio.h>`	Preprocessor Directive
`int main()`	The start of program
`{`	
`    /* To display Hello world */`	Comment
`    printf("Hello world\n");`	Statement
`    return 0;`	Program Termination
`}`	

- Assuming your code is ready, use the following commands to compile the code
- On command prompt, type

  $ gcc <file_name>.c

- This will generate a executable named a.out
- But it is recommended that you follow proper conversion even while generating your code, so you could use

  $ gcc <file_name>.c -o <file_name>

- This will generate a executable named <file_name>

- To execute your code you shall try

  ```
  $ ./a.out
  ```

- If you have named you output file as your <file_name> then

  ```
  $ ./<file_name>
  ```

- This should the expected result on your system

Chapter 2

Basic Refreshers

Advanced C

Number Systems

- A number is generally represented as
  - Decimal
  - Octal
  - Hexadecimal
  - Binary

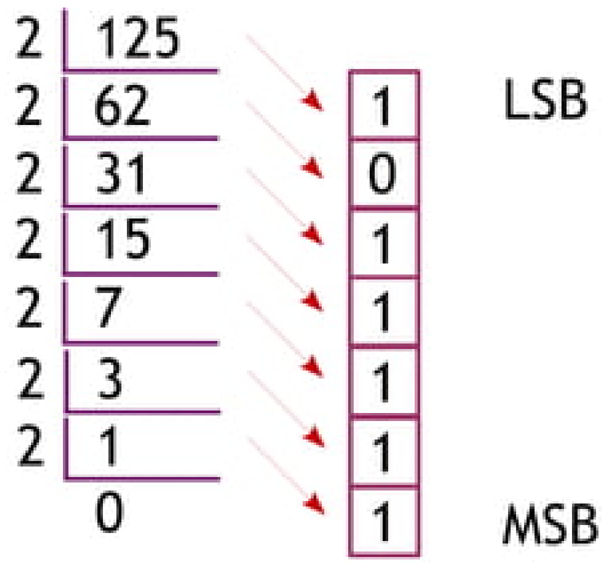
Type	Range (8 Bits)
Decimal	0 - 255
Octal	000 - 0377
Hexadecimal	0x00 - 0xFF
Binary	0b00000000 - 0b11111111

Type	Dec	Oct	Hex	Bin			
Base	10	8	16	2			
	0	0	0	0	0	0	0
	1	1	1	0	0	0	1
	2	2	2	0	0	1	0
	3	3	3	0	0	1	1
	4	4	4	0	1	0	0
	5	5	5	0	1	0	1
	6	6	6	0	1	1	0
	7	7	7	0	1	1	1
	8	10	8	1	0	0	0
	9	11	9	1	0	0	1
	10	12	A	1	0	1	0
	11	13	B	1	0	1	1
	12	14	C	1	1	0	0
	13	15	D	1	1	0	1
	14	16	E	1	1	1	0
	15	17	F	1	1	1	1

Advanced C

Number Systems – Decimal to Binary

- 125_{10} to Binary

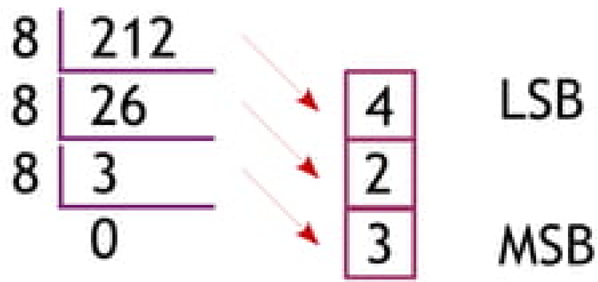


- So 125_{10} is 1111101_2

Advanced C

Number Systems - Decimal to Octal

- 212_{10} to Octal



- So 212_{10} is 324_8

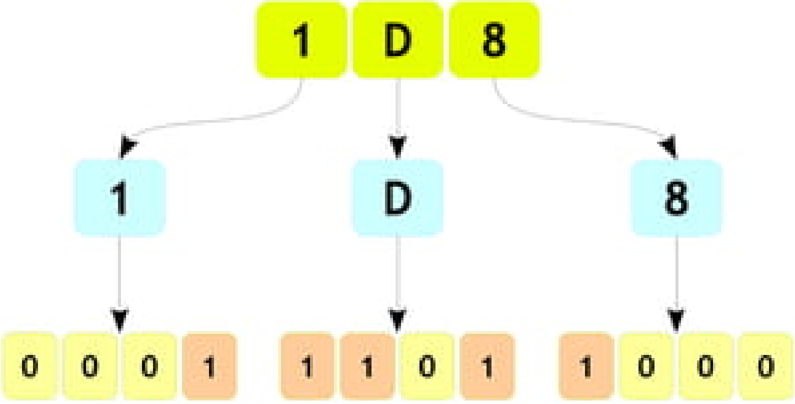
- 472_{10} to Hexadecimal

16	472		
16	29	8	LSB
16	1	13	
	0	1	MSB

Representation	Substitutes															
Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

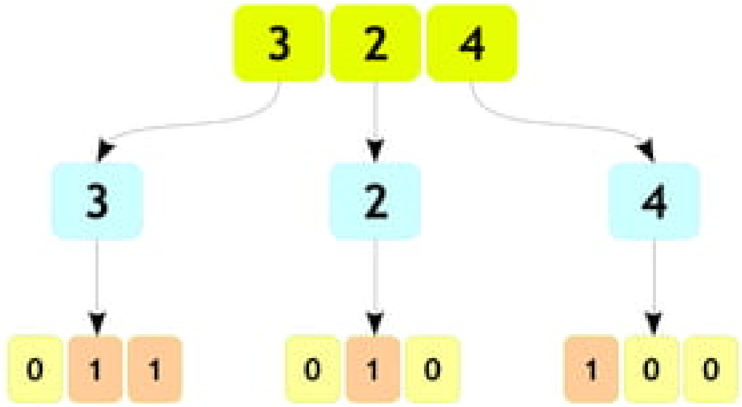
- So 472_{10} is $1D8_{16}$

- $1D8_{16}$ to Binary



- So $1D8_{16}$ is 000111011000_2 which is nothing but 111011000_2

- 324_8 to Binary

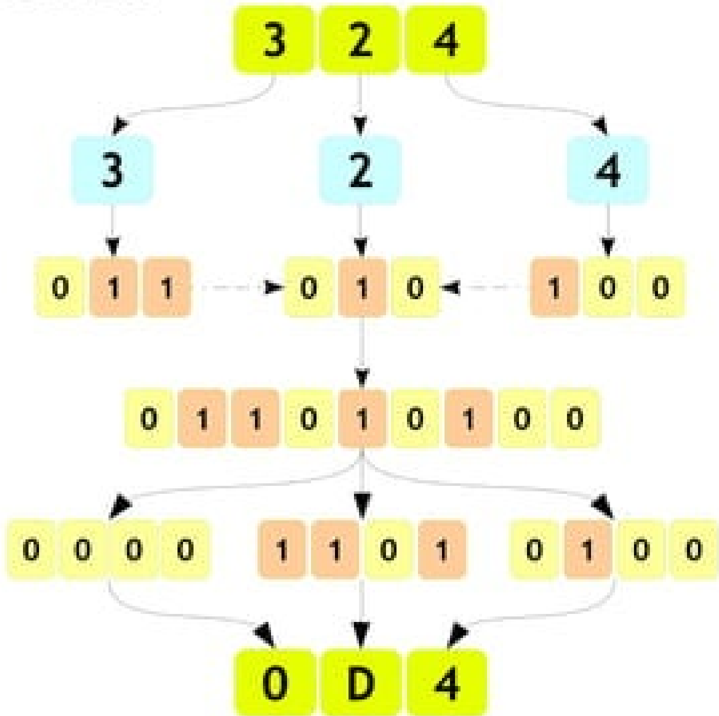


- So 324_8 is 011010100_2 which is nothing but 11010100_2

Advanced C

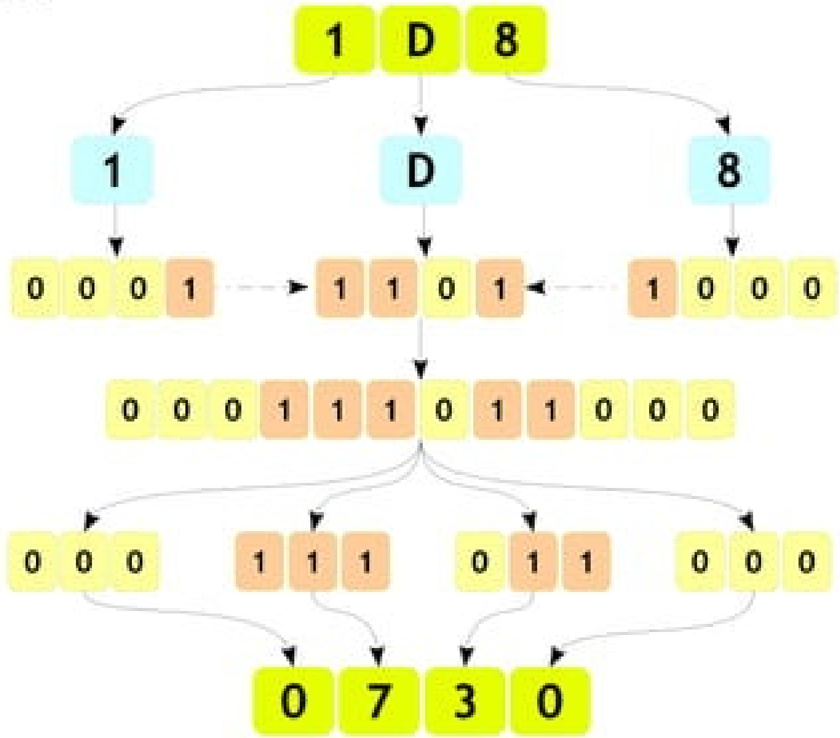
Number Systems – Octal to Hexadecimal

- 324_8 to Hexadecimal



- So 324_8 is $0D4_{16}$ which is nothing but $D4_{16}$

- $1D8_{16}$ to Octal



- So $1D8_{16}$ is 0730_8 which is nothing but 730_8

- 111011000_2 to Decimal

	1	1	1	0	1	1	0	0	0
Bit Position	8	7	6	5	4	3	2	1	0
	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$256 + 128 + 64 + 0 + 16 + 8 + 0 + 0 + 0 = 472$$

- So 111011000_2 is 472_{10}

- Literally computer understand only two states HIGH and LOW making it a binary system
- These states are coded as 1 or 0 called binary digits
- "**Bi**nary Digi**t**" gave birth to the word "**Bit**"
- Bit is known a basic unit of information in computer and digital communication

Value	No of Bits
0	0
1	1

- A unit of digital information
- Commonly consist of 8 bits
- Considered smallest addressable unit of memory in computer

Value	No of Bits							
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1

- One byte represents one unique character like 'A', 'b', '1', '$' ...
- Its possible to have 256 different combinations of 0s and 1s to form a individual character
- There are different types of character code representation like
  - ASCII → American Standard Code for Information Interchange - 7 Bits (Extended - 8 Bits)
  - EBCDIC → Extended BCD Interchange Code - 8 Bits
  - Unicode → Universal Code - 16 Bits and more

- ASCII is the oldest representation
- Please try the following on command prompt to know the available codes

  $ man ascii

- Can be represented by char datatype

Value	No of Bits							
0	0	1	1	0	0	0	0	0
A	0	1	0	0	0	0	0	1

Advanced C

Data Representation - word

- Amount of data that a machine can fetch and process at one time
- An integer number of bytes, for example, one, two, four, or eight
- General discussion on the bitness of the system is references to the word size of a system, i.e., a 32 bit chip has a 32 bit (4 Bytes) word size

Value	No of 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 0 0 0 0 0
1	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 0 0 0 1

EMERTXE

Advanced C

Integer Number - Positive

- Integers are like whole numbers, but allow negative numbers and no fraction
- An example of 13_{10} in 32 bit system would be

Bit	No of Bits																															
Position	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
Value	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

Advanced C

Integer Number - Negative

- Negative Integers represented with the 2's complement of the positive number
- An example of -13_{10} in 32 bit system would be

Bit	No of Bits																															
Position	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
Value	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
1's Compli	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	0
Add 1	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	0	0	0	1
2's Compli	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

- Mathematically : $-k \equiv 2^n - k$

- A formulaic representation which approximates a real number
- Computers are integer machines and are capable of representing real numbers only by using complex codes
- The most popular code for representing real numbers is called the IEEE Floating-Point Standard

	Sign	Exponent	Mantissa
Float (32 bits) Single Precision	1 bit	8 bits	23 bits
Double (64 bits) Double Precision	1 bit	11 bits	52 bits

Float Point Number – 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 part into binary format.
  - Converting the fractional part into binary format.

  The integral part is converted with the techniques examined previously.

  The fractional part can be converted by multiplying it with 2.
- STEP 2: 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.

$$\text{Float} : V = (-1)^S * 2^{(E-127)} * 1.F$$

$$\text{Double} : V = (-1)^S * 2^{(E-1023)} * 1.F$$

Advanced C

Float Point Number - Conversion - Example 1

Convert 2.5 to IEEE 32-bit floating point format

Step 1:

0.5	× 2	1.0	1

$2.5_{10} = 10.1_2$

Step 2:

Normalize: $10.1_2 = 1.01_2 \times 2^1$

Mantissa is 01000000000000000000000

Exponent is $1 + 127 = 128 = 1000\ 0000_2$

Sign bit is 0

Bit	S	Exponent								Mantissa																						
Position	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
Value	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

EMERTXE

Float Point Number – Conversion - Example 2

Convert 0.625 to IEEE 32-bit floating point format

Step 1:

0.625	× 2	1.25	1
0.25	× 2	0.5	0
0.5	× 2	1.0	1

$0.625_{10} = 0.101_2$

Step 2:

Normalize: $0.101_2 = 1.01_2 \times 2^{-1}$

Mantissa is 01000000000000000000000

Exponent is $-1 + 127 = 126 = 01111110_2$

Sign bit is 0

Bit	S	Exponent								Mantissa																						
Position	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
Value	0	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Advanced C

Float Point Number - Conversion - Example 3

Convert 39887.5625 to IEEE 32-bit floating point format

Step 1:

0.5625	× 2 =	1.125	1
0.125	× 2 =	0.25	0
0.25	× 2 =	0.5	0
0.5	× 2 =	1.0	1

$39887.5625_{10} =$
1001101111001111.1001_2

Step 2:

Normalize:
$1001101111001111.1001_2 =$
$1.0011011110011111001_2 \times 2^{15}$

Mantissa is 00110111100111110010000
Exponent is $15 + 127 = 142 = 10001110_2$
Sign bit is 0

Bit	S	Exponent								Mantissa																						
Position	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
Value	0	1	0	0	0	1	1	1	0	0	0	1	1	0	1	1	1	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0

Advanced C

Float Point Number - Conversion - Example 5

Convert -13.3125 to IEEE 32-bit floating point format

Step 1:

0.3125	× 2 =	0.625	0
0.625	× 2 =	1.25	1
0.25	× 2 =	0.5	0
0.5	× 2 =	1.0	1

$13.3125_{10} = 1101.0101_2$

Step 2:

Normalize:

$1101.0101_2 = 1.1010101_2 \times 2^3$

Mantissa is 10101010000000000000000

Exponent is $3 + 127 = 130 = 1000\ 0010_2$

Sign bit is 1

Bit	S	Exponent								Mantissa																						
Position	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
Value	1	1	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Advanced C

Float Point Number - Conversion - Example 6

Convert 1.7 to IEEE 32-bit floating point format

Step 1:

0.7	× 2 =	1.4	1
0.4	× 2 =	0.8	0
0.8	× 2 =	1.6	1
0.6	× 2 =	1.2	1
0.2	x 2 =	0.4	0
0.4	× 2 =	0.8	0
0.8	× 2 =	1.6	1
0.6	× 2 =	1.2	1

Step 2:

Normalize:

$1.10110011001100110011001_2 =$

$1.10110011001100110011001_2 \times 2^0$

Mantissa is 10110011001100110011001

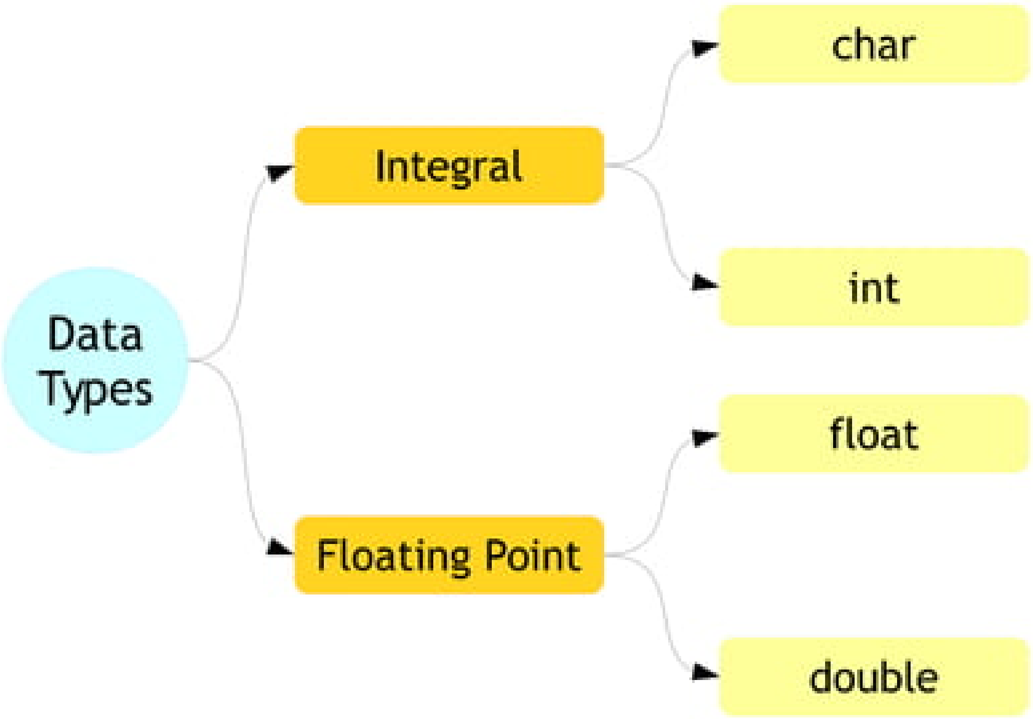
Exponent is $0 + 127 = 127 = 01111111_2$

Sign bit is 1

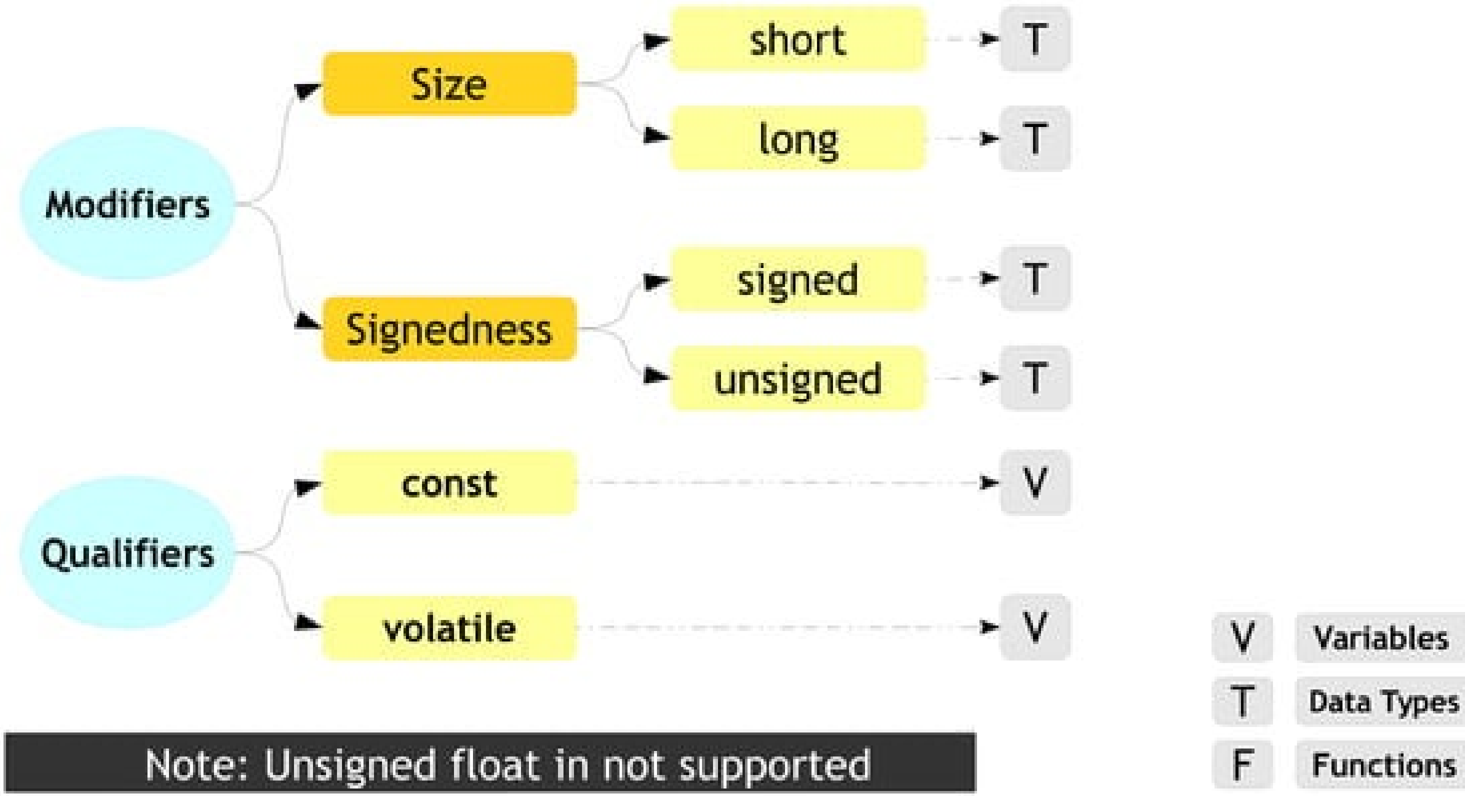
$1.7_{10} = 1.10110011001100110011001_2$

Bit	S	Exponent								Mantissa																						
Position	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
Value	0	0	1	1	1	1	1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1

ΣMERTXE


Data Types
Integral
Floating Point
char
int
float
double

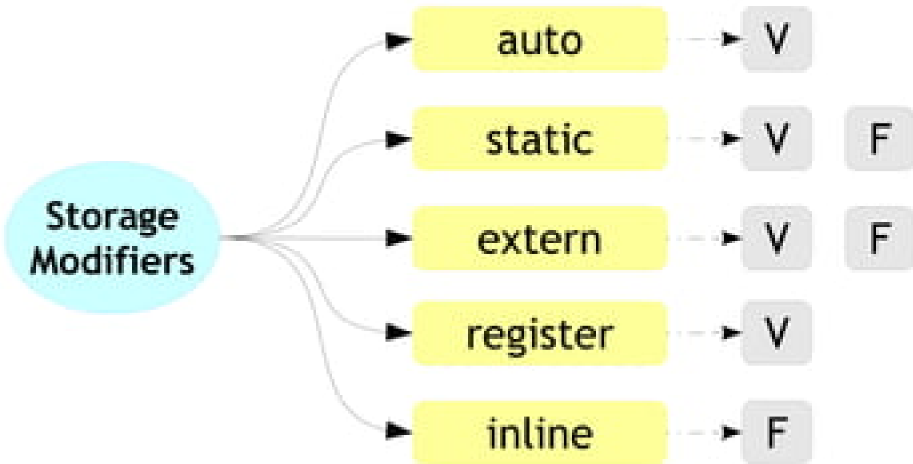
Data Type Modifiers and Qualifiers



Note: Unsigned float in not supported

Advanced C

Data Type and Function storage modification



V	Variables
T	Data Types
F	Functions

Advanced C
Code Statements - Simple

```c
int main()
{
    number = 5;
    3; +5;
    sum = number + 5;
    4 + 5;
    ;
}
```

Assignment statement

Valid statement, But smart compilers might remove it

Assignment statement. Result of the number + 5 will be assigned to sum

Valid statement, But smart compilers might remove it

This valid too!!

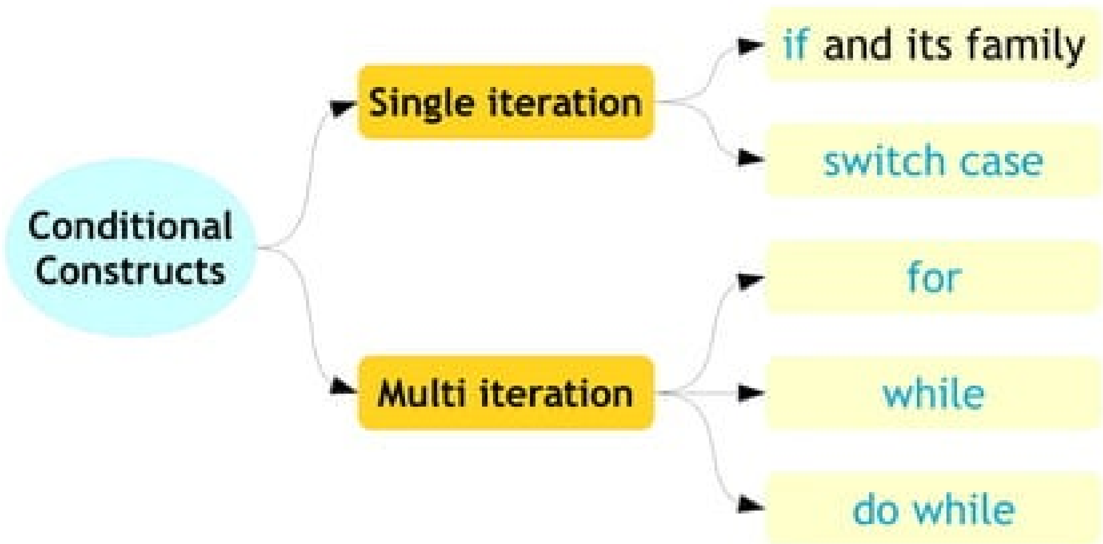
Advanced C

Code Statements - Compound

```
int main()
{
    ...
    if (num1 > num2)
    {
        if (num1 > num3)
        {
            printf("Hello");
        }
        else
        {
            printf("World");
        }
    }
    ...
}
```

If conditional statement

Nested if statement



Conditional Constructs
Single iteration
Multi iteration
if and its family
switch case
for
while
do while

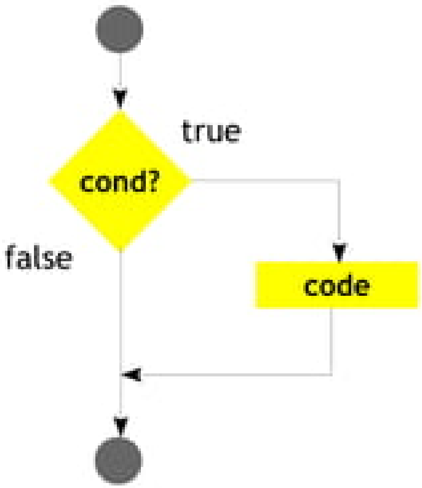
Advanced C

Conditional Constructs - if

Syntax

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

Flow



Example

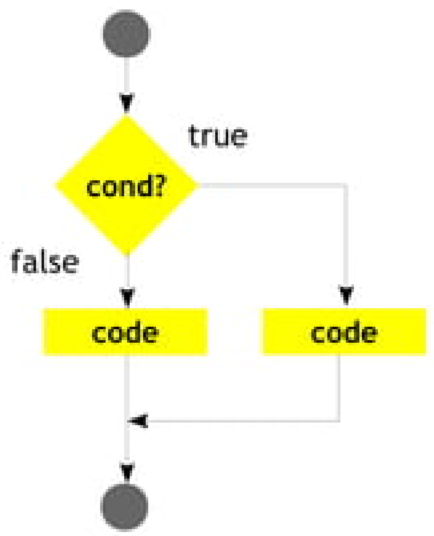
```c
#include <stdio.h>

int main()
{
    int num = 2;

    if (num < 5)
    {
        printf("num < 5\n");
    }
    printf("num is %d\n", num);

    return 0;
}
```

Conditional Constructs - if else

Syntax

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

Flow



Example

```
#include <stdio.h>

int main()
{
    int num = 10;

    if (num < 5)
    {
        printf("num is smaller than 5\n");
    }
    else
    {
        printf("num is greater than 5\n");
    }

    return 0;
}
```

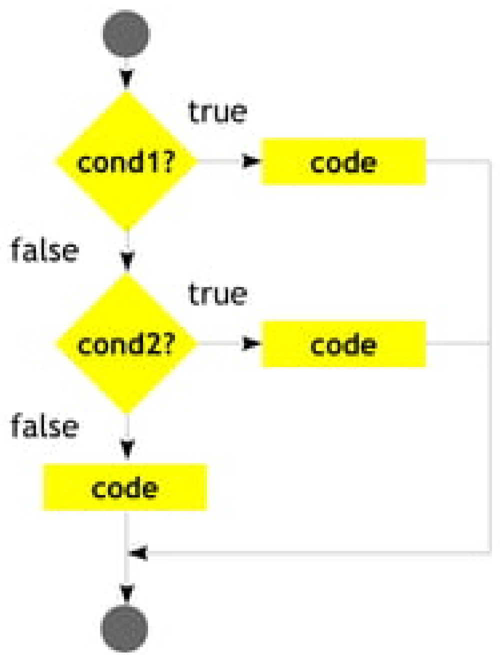
Advanced C

Conditional Constructs - if else if

Syntax

```
if (condition1)
{
    statement(s);
}
else if (condition2)
{
    statement(s);
}
else
{
    statement(s);
}
```

Flow



Conditional Constructs - if else if

Example

```
#include <stdio.h>

int main()
{
    int num = 10;

    if (num < 5)
    {
        printf("num is smaller than 5\n");
    }
    else if (num > 5)
    {
        printf("num is greater than 5\n");
    }
    else
    {
        printf("num is equal to 5\n");
    }


    return 0;
}
```

- WAP to find the max of two numbers
- WAP to find the greatest of given 3 numbers
- WAP to check whether character is
  - Upper case
  - Lower case
  - Digit
  - No of the above
- WAP to find the middle number (by value) of given 3 numbers

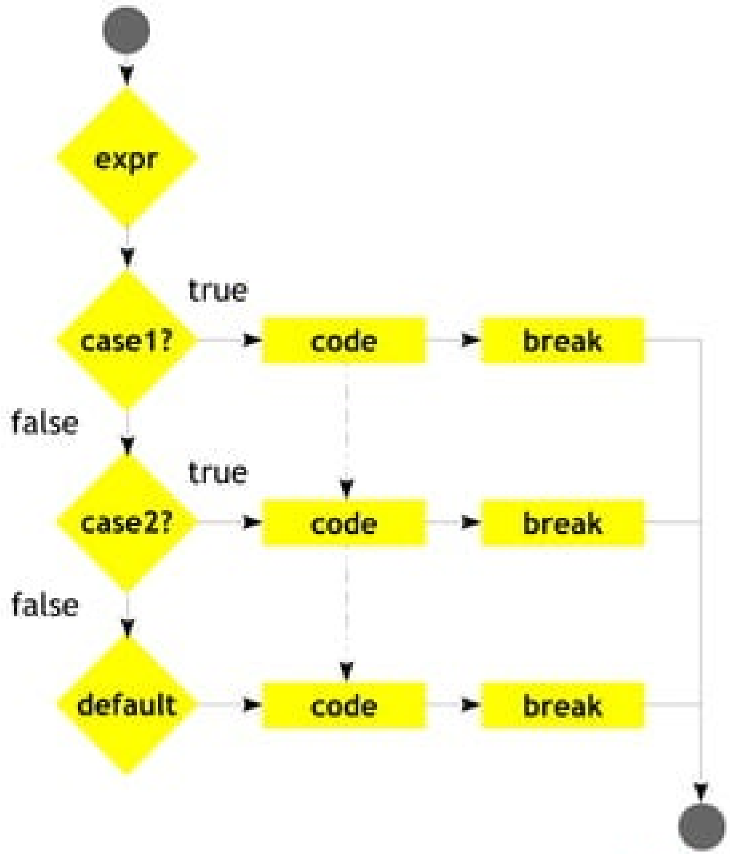
Advanced C

Conditional Constructs - switch

Syntax

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

Flow



Advanced C

Conditional Constructs - switch

Example

```
#include <stdio.h>

int main()
{
    int option;
    printf("Enter the value\n");
    scanf("%d", &option);

    switch (option)
    {
        case 10:
            printf("You entered 10\n");
            break;
        case 20:
            printf("You entered 20\n");
            break;
        default:
            printf("Try again\n");
    }

    return 0;
}
```

ΣMERTXE

- W.A.P to check whether character is
  - Upper case
  - Lower case
  - Digit
  - None of the above
- W.A.P for simple calculator

Advanced C

Conditional Constructs - while

Syntax

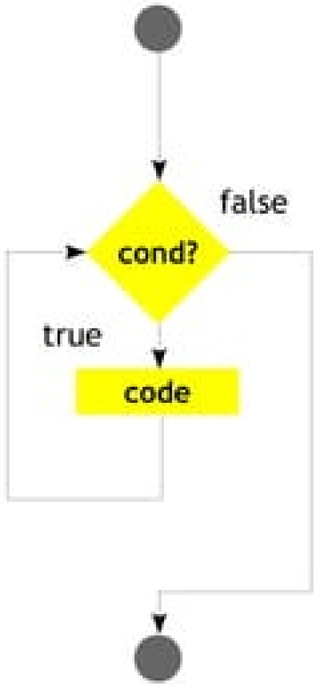
```
while (condition)
{
    statement(s);
}
```

- **Controls** the loop.
- Evaluated **before** each execution of loop body

Example

```
#include <stdio.h>

int main()
{
    int iter;

    iter = 0;
    while (iter < 5)
    {
        printf("Looped %d times\n", iter);
        iter++;
    }

    return 0;
}
```

Flow



Advanced C

Conditional Constructs - do while

Syntax

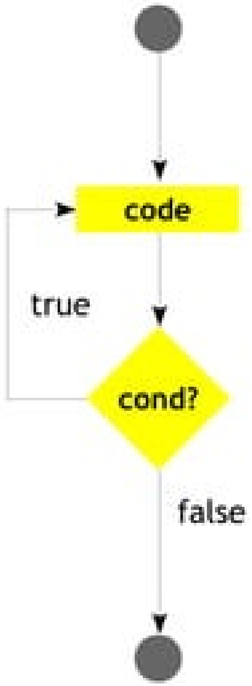
```
do
{
    statement(s);
} while (condition);
```

- **Controls** the loop.
- Evaluated **after** each execution of loop body

Example

```
#include <stdio.h>

int main()
{
    int iter;

    iter = 0;
    do
    {
        printf("Looped %d times\n", iter);
        iter++;
    } while (iter < 10);

    return 0;
}
```

Flow



ΣMERTXE

Advanced C

Conditional Constructs - for

Syntax

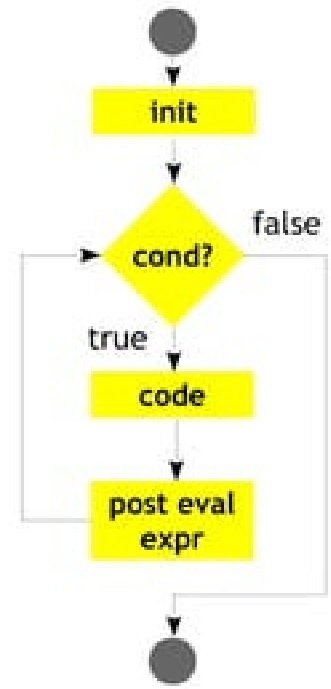
```
for (init; condition; post evaluation expr)
{
    statement(s);
}
```

- **Controls** the loop.
- Evaluated **before** each execution of loop body

Example

```c
#include <stdio.h>

int main()
{
    int iter;

    for (iter = 0; iter < 10; iter++)
    {
        printf("Looped %d times\n", iter);
    }

    return 0;
}
```

Flow



- W.A.P to print the power of two series using for loop
  - $2^1, 2^2, 2^3, 2^4, 2^5 \ldots$
- W.A.P to print the power of N series using Loops
  - $N^1, N^2, N^3, N^4, N^5 \ldots$
- W.A.P to multiply 2 nos without multiplication operator
- W.A.P to check whether a number is palindrome or not

- WAP to print line pattern
  - Read total (n) number of pattern chars in a line (number should be “odd”)
  - Read number (m) of pattern char to be printed in the middle of line (“odd” number)
  - Print the line with two different pattern chars
  - Example - Let's say two types of pattern chars '$' and '*' to be printed in a line. Total number of chars to be printed in a line are 9. Three '*' to be printed in middle of line.
    - Output ==> $$$* * *$$$

- Based on previous example print following pyramid

```
      *
    * * *
  * * * * *
* * * * * * *
```

- Print rhombus using for loops

```
      *
    * * *
  * * * * *
* * * * * * *
  * * * * *
    * * *
      *
```

Advanced C

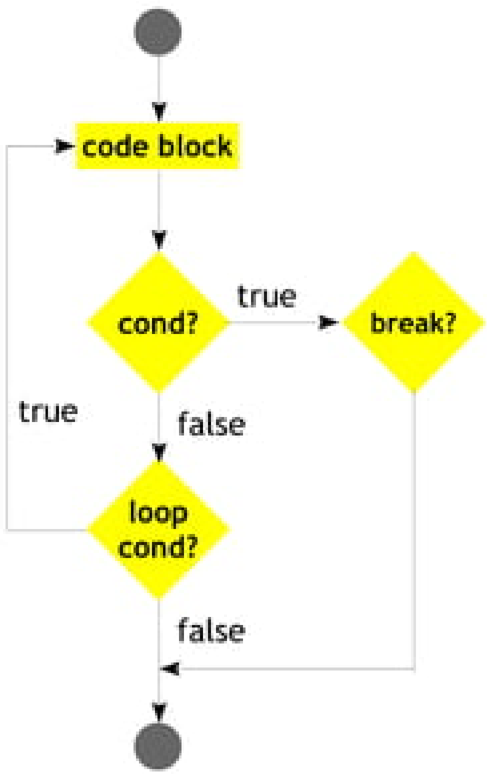
Conditional Constructs - break

- A break statement shall appear only in “switch body” or “loop body”
- *“break”* is used to exit the loop, the statements appearing after break in the loop will be skipped

Syntax

```
do
{
    conditional statement
        break;
} while (condition);
```

Flow



Example

```c
#include <stdio.h>

int main()
{
    int iter;

    for (iter = 0; iter < 10; iter++)
    {
        if (iter == 5)
        {
            break;
        }
        printf("%d\n", iter);
    }

    return 0;
}
```

Advanced C

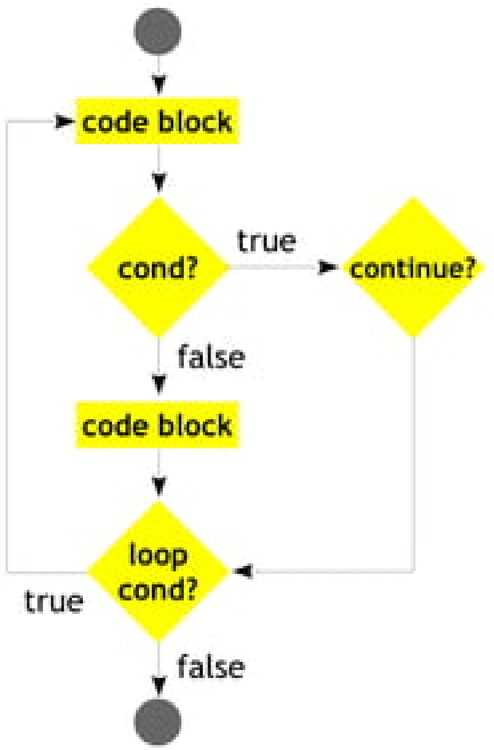
Conditional Constructs - continue

- A *continue* statement causes a jump to the loop-continuation portion, that is, to the end of the loop body
- The execution of code appearing after the continue will be skipped
- Can be used in any type of multi iteration loop

Syntax

```
do
{
    conditional statement
        continue;
} while (condition);
```

Flow



Advanced C

Conditional Constructs - continue

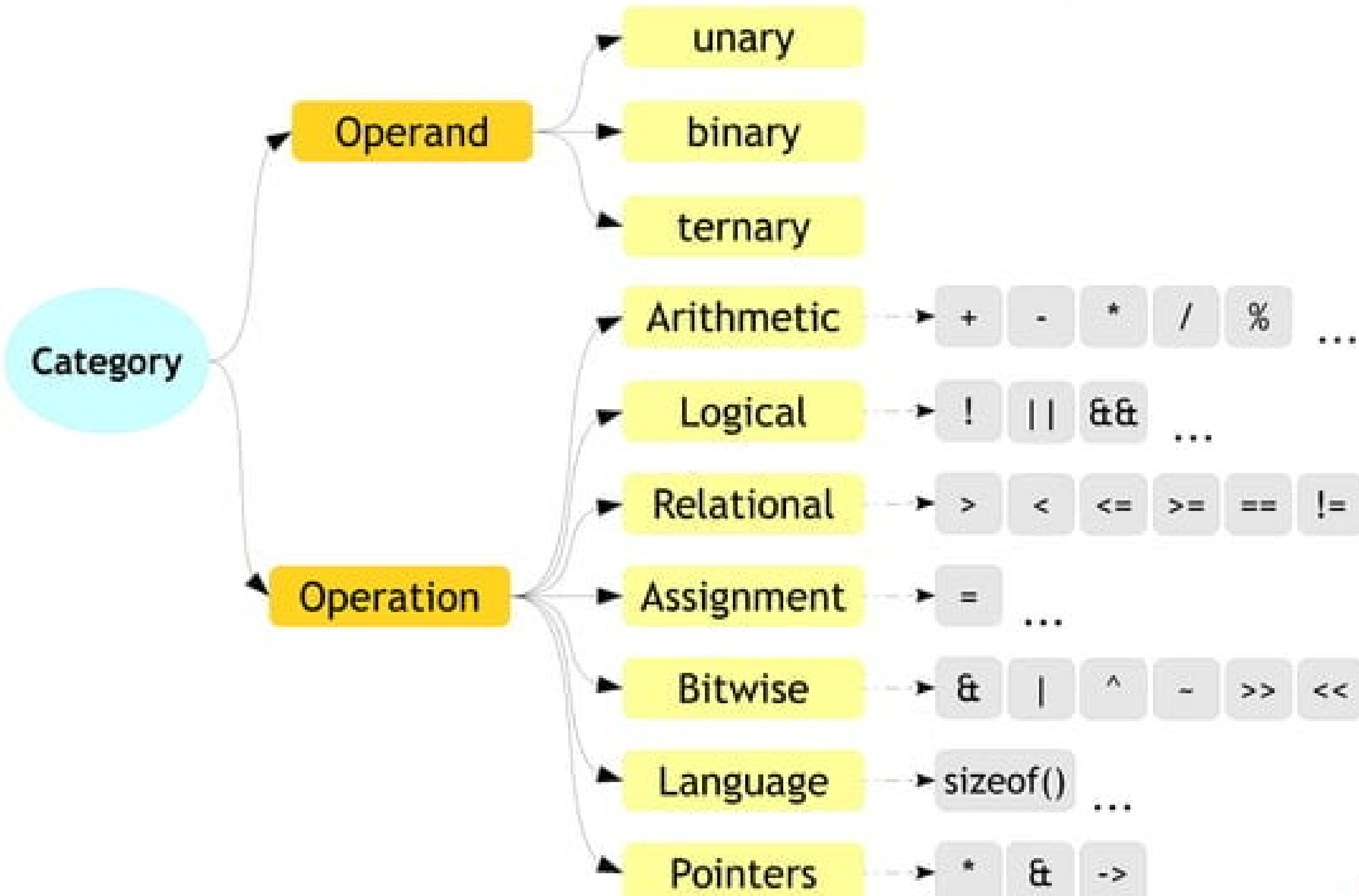
Example

```
#include <stdio.h>

int main()
{
    int iter;

    for (iter = 0; iter < 10; iter++)
    {
        if (iter == 5)
        {
            continue;
        }
        printf("%d\n", iter);
    }

    return 0;
}
```

- Symbols that instructs the compiler to perform specific arithmetic or logical operation on operands
- All C operators do 2 things
  - Operates on its operands
  - Returns a value

Advanced C

Operators



Advanced C

Operators - Precedence and Associativity

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

Note:

post ++ and -- operators have higher precedence than pre ++ and -- operators

(Rel-99 spec)

EMERTXE

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

Example

```
#include <stdio.h>

int main()
{
    int num;

    Num = 7 - 4 * 3 / 2 + 5;

    printf("Result is %d\n", num);

    return 0;
}
```

What will be the output?

Advanced C

Operators - Language - sizeof()

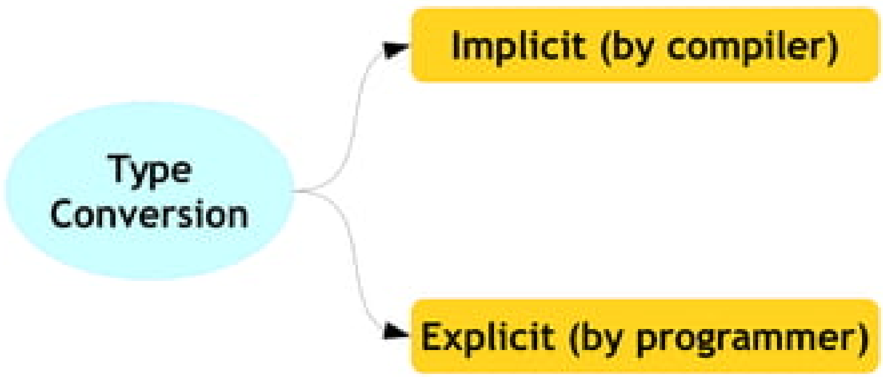
Example

```
#include <stdio.h>

int main()
{
    int num = 5;

    printf("%u:%u:%u\n", sizeof(int), sizeof num, sizeof 5);

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    int num1 = 5;
    int num2 = sizeof(++num1);

    printf("num1 is %d and num2 is %d\n", num1, num2);

    return 0;
}
```

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


Type Conversion
Implicit (by compiler)
Explicit (by programmer)

Advanced C

Type Conversion Hierarchy

- Automatic Unary conversions
  - The result of + and - are promoted to int if operands are char and short
  - The result of ~ and ! is integer
- 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 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.

Advanced C

Type Conversion - Explicit (Type Casting)

Syntax

```
(data type) expression
```

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 5, num2 = 3;

    float num3 = (float) num1 / num2;

    printf("nun3 is %f\n", num3);

    return 0;
}
```

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

Example

```
#include <stdio.h>

int main()
{
    int num1 = 1, num2 = 0;

    if (++num1 || num2++)
    {
        printf("num1 is %d num2 is %d\n", num1, num2);
    }
    num1 = 1, num2 = 0;
    if (num1++ && ++num2)
    {
        printf("num1 is %d num2 is %d\n", num1, num2);
    }
    else
    {
        printf("num1 is %d num2 is %d\n", num1, num2);
    }
    return 0;
}
```

What will be the output?

- Have the ability to “short circuit” a calculation if the result is definitely known, this can improve efficiency
  - Logical AND operator (&&)
    - If one operand is false, the result is false.
  - Logical OR operator (||)
    - If one operand is true, the result is true.

Advanced C

Operators - Relational

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

Example

```
#include <stdio.h>

int main()
{
    float num1 = 0.7;

    if (num1 == 0.7)
    {
        printf("Yes, it is equal\n");
    }
    else
    {
        printf("No, it is not equal\n");
    }

    return 0;
}
```

What will be the output?

ΣMERTXE

Advanced C

Operators - Assignment

Example

```
#include <stdio.h>

int main()
{
    int num1 = 1, num2 = 1;
    float num3 = 1.7, num4 = 1.5;

    num1 += num2 += num3 += num4;

    printf("num1 is %d\n", num1);

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    float num1 = 1;

    if (num1 = 1)
    {
        printf("Yes, it is equal!!\n");
    }
    else
    {
        printf("No, it is not equal\n");
    }

    return 0;
}
```

- Bitwise operators perform operations on bits
- The operand type shall be integral
- Return type is integral value

&	**Bitwise AND**	Bitwise ANDing of all the bits in two operands

Operand	Value								
A	0x61	0	1	1	0	0	0	0	1
B	0x13	0	0	0	1	0	0	1	1
A & B	0x01	0	0	0	0	0	0	0	1

|	**Bitwise OR**	Bitwise ORing of all the bits in two operands

Operand	Value								
A	0x61	0	1	1	0	0	0	0	1
B	0x13	0	0	0	1	0	0	1	1
A | B	0x73	0	1	1	1	0	0	1	1

Advanced C

Operators - Bitwise

			Operand	Value								
^	Bitwise XOR	Bitwise XORing of all the bits in two operands	A	0x61	0	1	1	0	0	0	0	1
			B	0x13	0	0	0	1	0	0	1	1
			A ^ B	0x72	0	1	1	1	0	0	1	0

			Operand	Value								
~	Compliment	Complimenting all the bits of the operand	A	0x61	0	1	1	0	0	0	0	1
			~A	0x9E	1	0	0	1	1	1	1	0

ΣMERTXE

Syntax

Left Shift : shift-expression << additive-expression

(left operand) (right operand)

Right Shift : shift-expression >> additive-expression

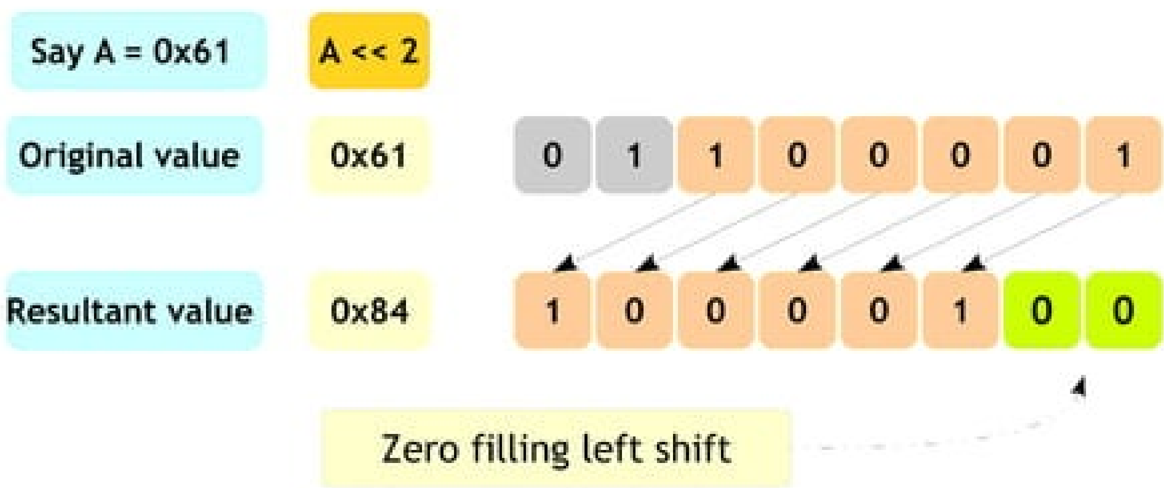
(left operand) (right operand)

Advanced C

Operators - Bitwise - Left Shift

'Value' << 'Bits Count'

- **Value** : Is shift-expression on which bit shifting effect to be applied
- **Bits count** : Is additive-expression, by how many bit(s) the given “Value” to be shifted

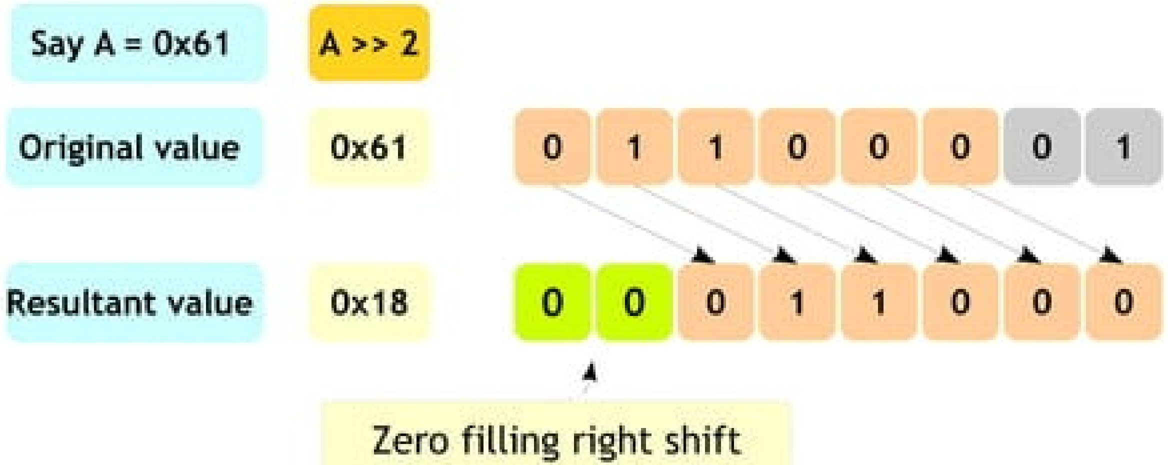


Advanced C

Operators - Bitwise - Right Shift

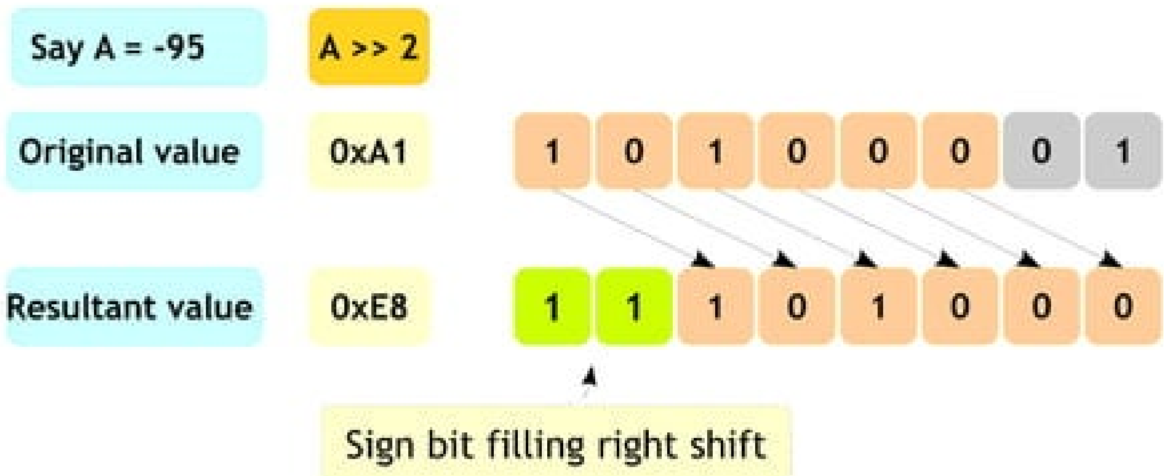
'Value' >> 'Bits Count'

- **Value** : Is shift-expression on which bit shifting effect to be applied
- **Bits count** : Is additive-expression, by how many bit(s) the given "Value" to be shifted



“Signed Value' >> 'Bits Count'

- Same operation as mentioned in previous slide.
- But the sign bits gets propagated.



Advanced C

Operators - Bitwise

Example

```
#include <stdio.h>

int main()
{
	int count;
	unsigned char iter = 0xFF;

	for (count = 0; iter != 0; iter >>= 1)
	{
		if (iter & 01)
		{
			count++;
		}
	}

	printf("count is %d\n", count);

	return 0;
}
```

EMERTXE

- Each of the operands shall have integer type
- The integer promotions are performed on each of the operands
- If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined
- Left shift (<<) operator : If left operand has a signed type and nonnegative value, and (left_operand * (2^n)) is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined

- The below example has undefined behaviour

Example

```c
#include <stdio.h>

int main()
{
    int x = 7, y = 7;

    x = 7 << 32;
    printf("x is %x\n", x);

    x = y << 32;
    printf("x is %x\n", x);

    return 0;
}
```

- W.A.P to count set bits in a given number
- W.A.P to print bits of given number
- W.A.P to swap nibbles of given number

- Set bit
- Get bit
- Clear bit

Operators - Ternary

Syntax

```
Condition ? Expression 1 : Expression 2;
```

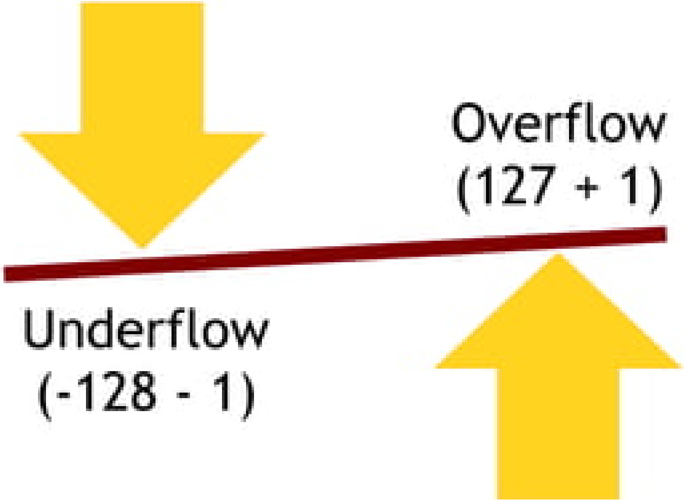
Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3;

    if (num1 > num2)
    {
        num3 = num1;
    }
    else
    {
        num3 = num2;
    }
    printf("%d\n", num3);

    return 0;
}
```

```c
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3;

    num3 = num1 > num2 ? num1 : num2;
    printf("Greater num is %d\n", num3);

    return 0;
}
```

- The left operand of a comma operator is evaluated as a void expression (result discarded)
- Then the right operand is evaluated; the result has its type and value
- Comma acts as separator (not an operator) in following cases -
  - Arguments to function
  - Lists of initializers (variable declarations)
- But, can be used with parentheses as function arguments such as -
  - `foo ((x = 2, x + 3)); // final value of argument is 5`

- 8-bit Integral types can hold certain ranges of values
- So what happens when we try to traverse this boundary?



Advanced C

Overflow - Signed Numbers

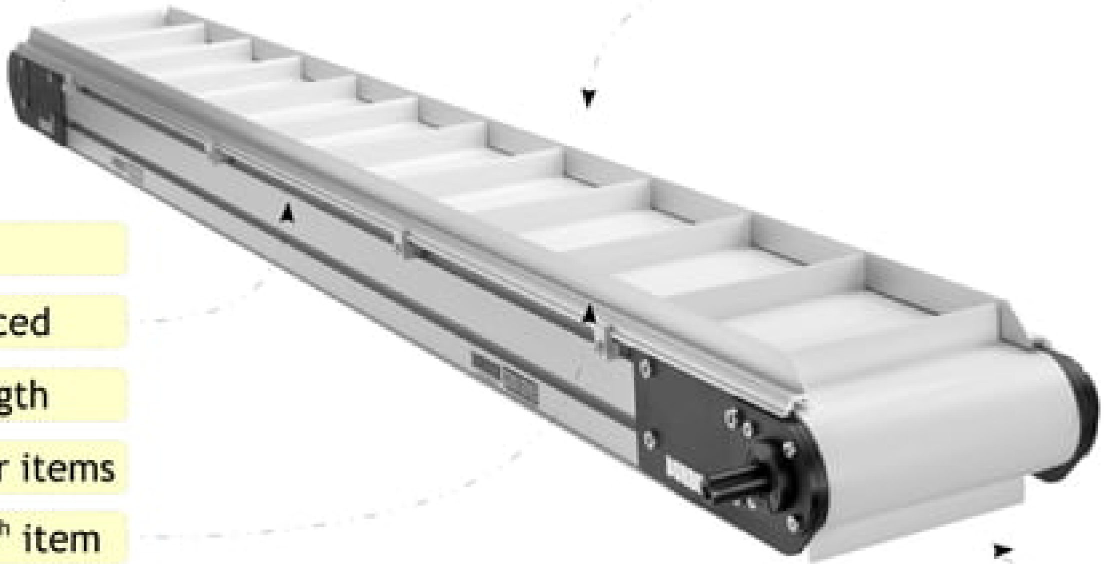
Say A = +127

Original value	0x7F	0	1	1	1	1	1	1	1
Add	1	0	0	0	0	0	0	0	1
Resultant value	0x80	1	0	0	0	0	0	0	0
		sign bit							

Advanced C

Underflow - Signed Numbers

Advanced C
Arrays - Know the Concept



A conveyor belt

Starts here

Equally spaced

Defined length

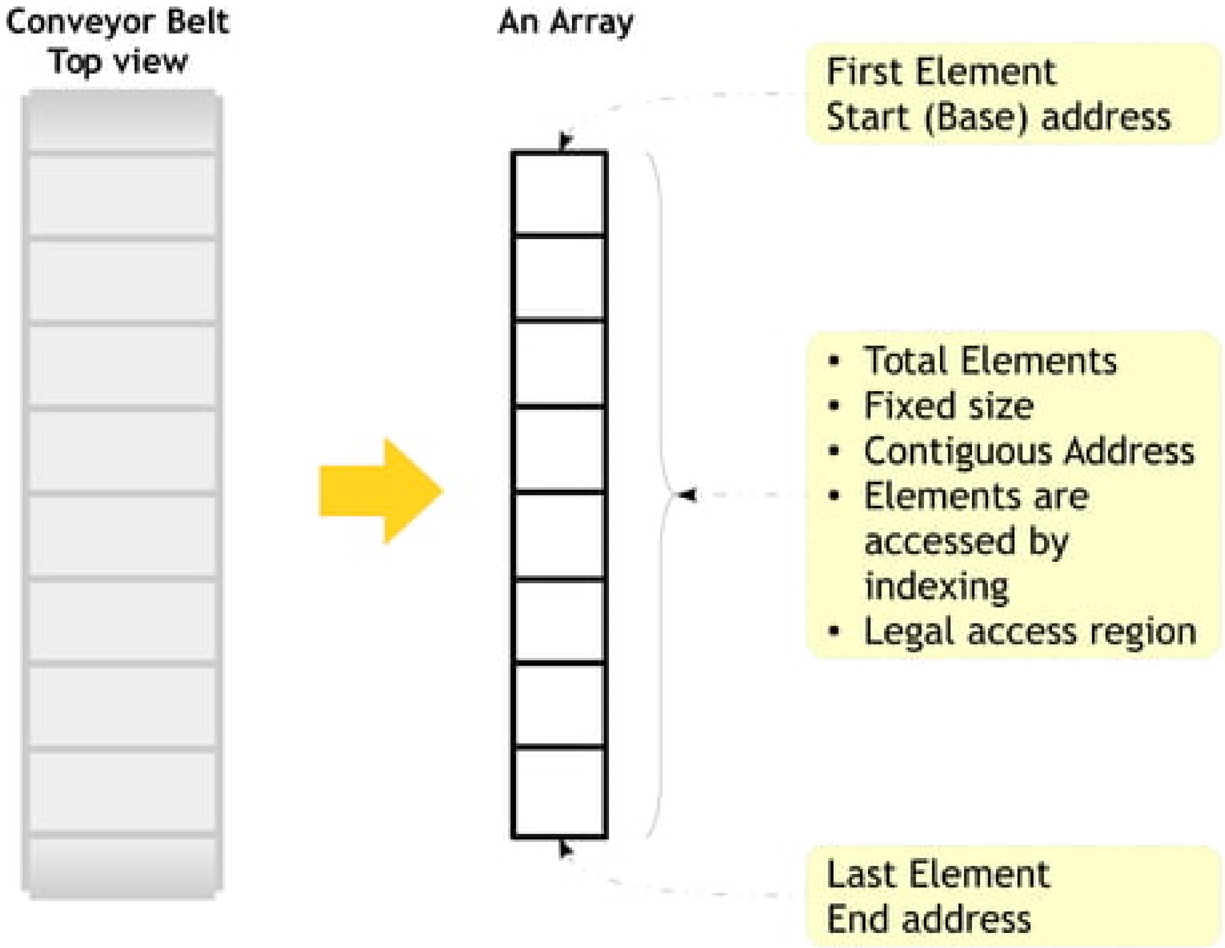
Carry similar items

Index as 10^{th} item

Ends here

Advanced C

Arrays – Know the Concept



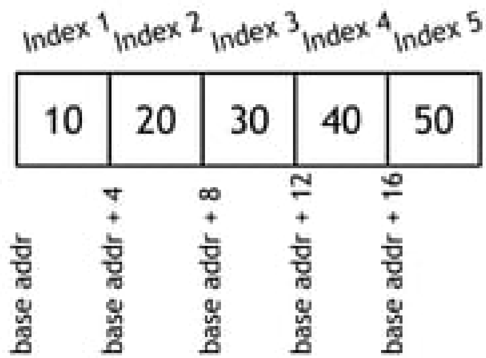
What is the type of conveyor belt?

Syntax

```
data_type name[SIZE];

Where SIZE represents number of elements
Memory occupied by array = (number of elements * size of an element)
   = SIZE * <size of data_type>
```

Example

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



- An array is a collection of similar data type
- Elements occupy consecutive memory locations (addresses)
- First element with lowest address and the last element with highest address
- Elements are indexed from 0 to SIZE - 1. Example : 5 elements array (say array[5]) will be indexed from 0 to 4
- Accessing out of range array elements would be “illegal access”
  - Example : Do not access elements array[-1] and array[SIZE]
- Array size can't be altered at run time

Advanced C

Arrays – Why?

Example

```
#include <stdio.h>

int main()
{
    int num1 = 10;
    int num2 = 20;
    int num3 = 30;
    int num4 = 40;
    int num5 = 50;

    printf("%d\n", num1);
    printf("%d\n", num2);
    printf("%d\n", num3);
    printf("%d\n", num4);
    printf("%d\n", num5);

    return 0;
}
```

```
#include <stdio.h>

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

    for (index = 0; index < 5; index++)
    {
        printf("%d\n", num_array[index]);
    }

    return 0;
}
```

Arrays - Reading

Example

```
#include <stdio.h>

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    int index;

    index = 0;
    do
    {
        printf("Index %d has Element %d\n", index, array[index]);
        index++;
    } while (index < 5);

    return 0;
}
```

Arrays - Storing

Example

```
#include <stdio.h>

int main()
{
    int array[5];
    int index;

    for (index = 0; index < 5; index++)
    {
        scanf("%d", &num_array[index]);
    }

    return 0;
}
```

Advanced C

Arrays - Initializing

Example

```
#include <stdio.h>

int main()
{
    int array1[5] = {1, 2, 3, 4, 5};
    int array2[5] = {1, 2};
    int array3[] = {1, 2};
    int array4[]; /* Invalid */

    printf("%u\n", sizeof(array1));
    printf("%u\n", sizeof(array2));
    printf("%u\n", sizeof(array3));

    return 0;
}
```

- Can we copy 2 arrays? If yes how?

Example

```
#include <stdio.h>

int main()
{
    int array_org[5] = {1, 2, 3, 4, 5};
    int array_bak[5];
    int index;

    array_bak = array_org;

    if (array_bak == array_org)
    {
        printf("Copied\n");
    }

    return 0;
}
```

Waow!! so simple?
But can I do this

- No!! its not so simple to copy two arrays as put in the previous slide. C doesn't support it!
- Then how to copy an array?
- It has to be copied element by element

- W.A.P to find the average of elements stored in a array.
  - Read value of elements from user
  - For given set of values : { 13, 5, -1, 8, 17 }
  - Average Result = 8.4
- W.A.P to find the largest array element
  - Example 100 is the largest in {5, 100, -2, 75, 42}

- W.A.P to compare two arrays (element by element).
  - Take equal size arrays
  - Arrays shall have unique values stored in random order
  - Array elements shall be entered by user
  - Arrays are compared "EQUAL" if there is one to one mapping of array elements value
  - Print final result "EQUAL" or "NOT EQUAL"

Example of Equal Arrays :

- A[3] = {2, -50, 17}
- B[3] = {17, 2, -50}

Advanced C

Arrays – Oops!! what is this now?

Chapter 3

Functions

Functions – What?

An activity that is natural to or the purpose of a person or thing.

"bridges perform the function of providing access across water"

A relation or expression involving one or more variables.

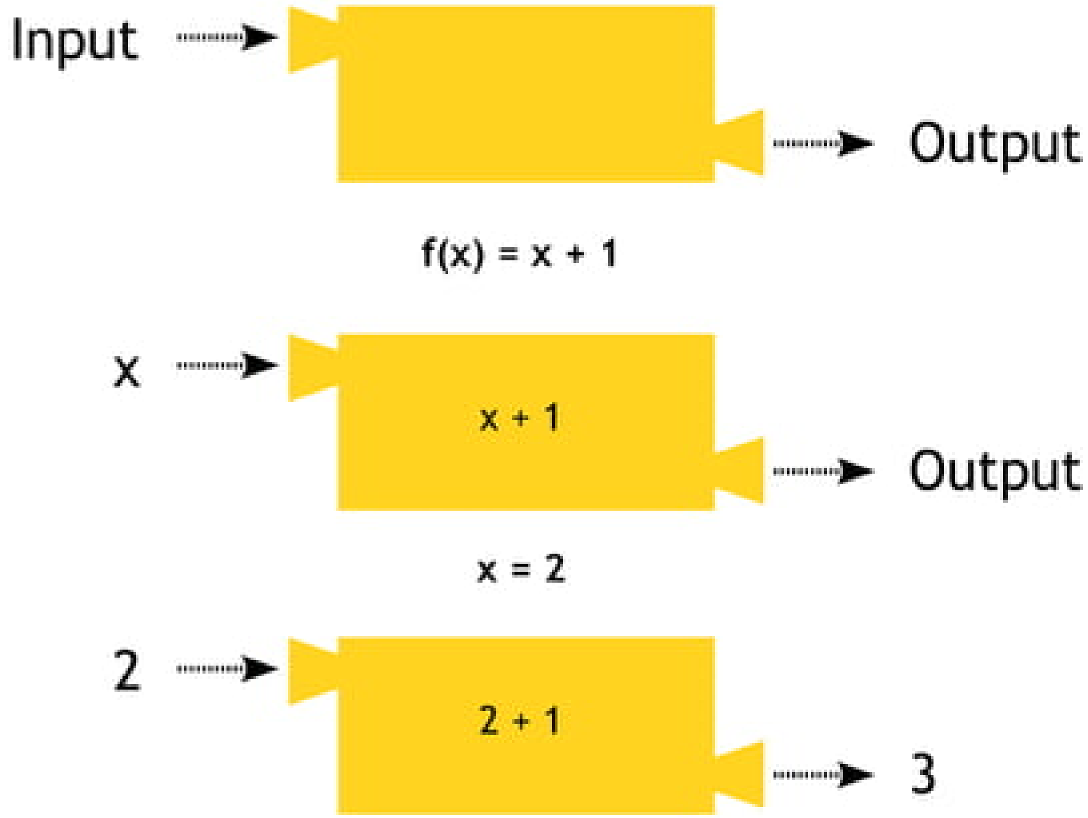
"the function (bx + c)"

Source: Google

- In programing languages it can be something which performs a specific service
- Generally a function has 3 properties
  - Takes Input
  - Perform Operation
  - Generate Output

Advanced C

Functions – What?



Functions - How to write

Syntax

```
return_data_type function_name(arg_1, arg_2, ..., arg_n)
{
    /* Function Body */
}
```

Example

Return data type as **int**

First parameter with **int** type

```
int foo(int arg_1, int arg_2)
{

}
```

Second parameter with **int** type

$y = x + 1$

Example

```
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

Return from function

Advanced C

Functions - How to call

Example

```
#include <stdio.h>

int main()
{
    int x, y;

    x = 2;

    y = foo(x);
    printf("y is %d\n", y);

    return 0;
}
```

```
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

The function call

Functions – Why?

- Re usability
  - Functions can be stored in library & re-used
  - When some specific code is to be used more than once, at different places, functions avoids repetition 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

Advanced C

Functions - A complete look

Example

```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

The main function

The function call

Actual arguments

Return type

Formal arguments

Function

Return from function

ΣMERTXE

Advanced C

Functions - Ignoring return value

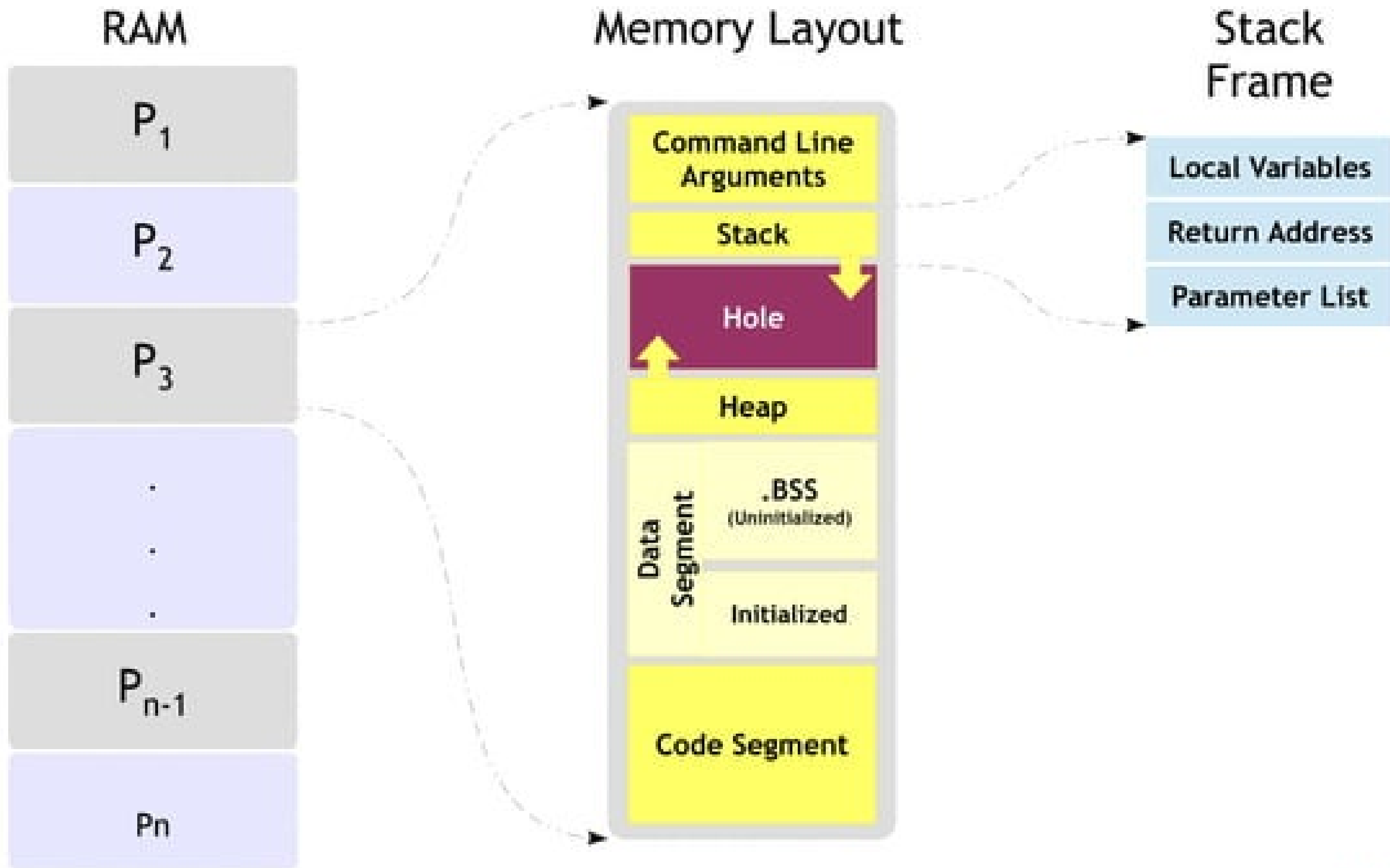
Example

```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Ignored the return from function
In C, it is up to the programmer to capture or ignore the return value

- Write a function to calculate square a number
  - y = x * x
- Write a function to convert temperature given in degree Fahrenheit to degree Celsius
  - C = 5/9 * (F - 32)
- Write a program to check if a given number is even or odd. Function should return TRUE or FALSE



RAM
P_1
P_2
P_3
.
.
.
P_{n-1}
Pn
Memory Layout
Command Line Arguments
Stack
Hole
Heap
Data Segment
.BSS
(Uninitialized)
Initialized
Code Segment
Stack Frame
Local Variables
Return Address
Parameter List

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.

Advanced C

Functions – Pass by Value

Example

```c
#include <stdio.h>

int add_numbers(int num1, int num2);

int main()
{
    int num1 = 10, num2 = 20, sum;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Advanced C

Functions – Pass by Value

Example

```
#include <stdio.h>

void modify(int num1)
{
    num1 = num1 + 1;
}

int main()
{
    int num1 = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num1);

    modify(num1);

    printf("After Modification\n");
    printf("num1 is %d\n", num1);

    return 0;
}
```

Are you sure you understood the previous problem?

Are you sure you are ready to proceed further?

Do you know the prerequisite to proceed further?

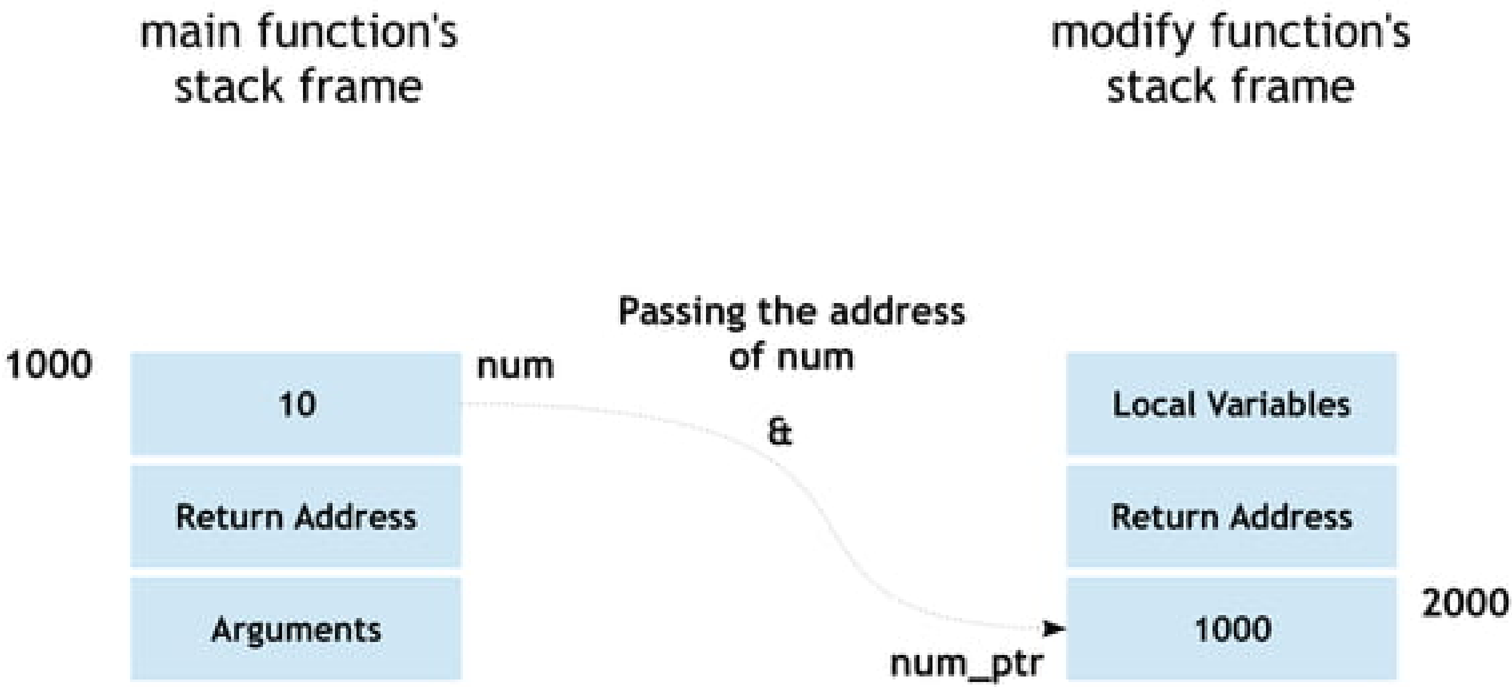
If no let's get it cleared

Advanced C

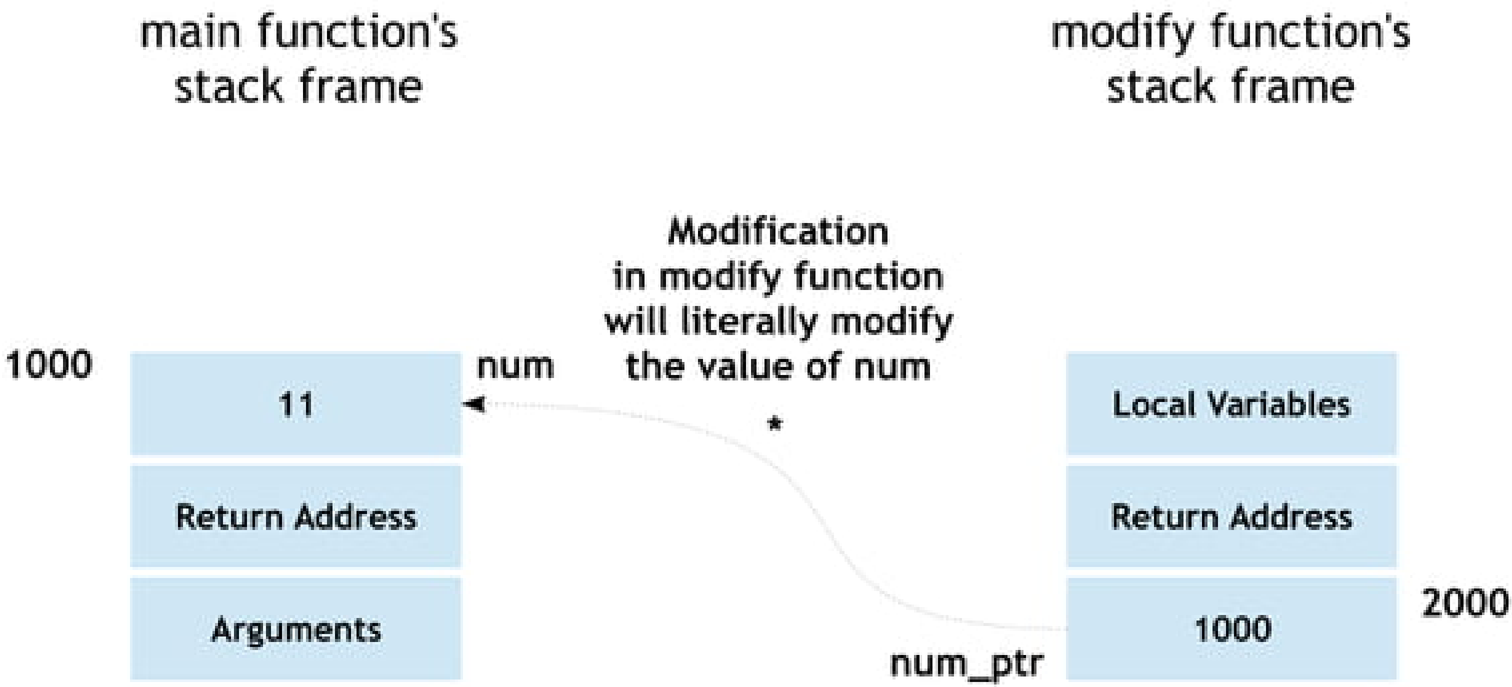
Functions - Pass by Reference

Example

```
#include <stdio.h>

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

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```



main function's stack frame
modify function's stack frame
Passing the address of num
1000
10
num
&
Return Address
Arguments
Local Variables
Return Address
1000
2000
num_ptr



main function's stack frame
modify function's stack frame
Modification in modify function will literally modify the value of num
1000
num
11
*
Return Address
Arguments
Local Variables
Return Address
2000
1000
num_ptr

Functions - Pass by Reference - Advantages

- Return more than one value from a function
- Copy of the argument is not made, making it fast, even when used with large variables like arrays etc.
- Saving stack space if argument variables are larger (example - user defined data types)

Advanced C
Functions - DIY

- Write a program to find the square of a number
- Write a program to find the square and cube of a number
- Write a program to swap two numbers
- Write a program to find the sum and product of 2 numbers

- As mentioned in previous slide passing an array to function can be faster
- But before you proceed further it is expected you are familiar with some pointer rules
- If you are OK with your concepts proceed further, else please know the rules first

Advanced C

Functions - Passing Array

Example

```c
#include <stdio.h>

void print_array(int array[]);

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

    print_array(array);

    return 0;
}
```

```c
void print_array(int array[])
{
    int iter;

    for (iter = 0; iter < 5; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

EMERTXE

Advanced C

Functions - Passing Array

Example

```
#include <stdio.h>

void print_array(int *array);

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

    print_array(array);

    return 0;
}
```

```
void print_array(int *array)
{
    int iter;

    for (iter = 0; iter < 5; iter++)
    {
        printf("Index %d has Element %d\n", iter, *array);
        array++;
    }
}
```

Advanced C

Functions - Passing Array

Example

```c
#include <stdio.h>

void print_array(int *array, int size);

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

    print_array(array, 5);

    return 0;
}
```

```c
void print_array(int *array, int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, *array++);
    }
}
```

Advanced C

Functions - Returning Array

Example

```c
#include <stdio.h>

int *modify_array(int *array, int size);
void print_array(int array[], int size);

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

    new_array_val = modify_array(array, 5);
    print_array(new_array_val, 5);

    return 0;
}
```

```c
int *modify_array(int *array, int size)
{
    int  iter;

    for (iter = 0; iter < size; iter++)
    {
        *(array + iter) += 10;
    }

    return array;
}
```

```c
void print_array(int array[], int size)
{
    int  iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

Advanced C

Functions - Returning Array

Example

```
#include <stdio.h>

int *return_array(void);
void print_array(int *array, int size);

int main()
{
    int *array_val;

    array_val = return_array();
    print_array(array_val, 5);

    return 0;
}
```

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

    return array;
}
```

```
void print_array(int *array, int size)
{
    int iter;

    for (iter = 0; iter < size; iter++)
    {
        printf("Index %d has Element %d\n", iter, array[iter]);
    }
}
```

- Write a program to find the average of 5 array elements using function
- Write a program to square each element of array which has 5 elements

- Need of function prototype
- Implicit int rule

Functions – return type

- Local return
- Void return

Advanced C

Functions

Advanced C
Functions – Recursive

- Recursion is the process of repeating items in a self-similar way
- In programming a function calling itself is called as recursive function
- Two steps

  Step 1: Identification of base case

  Step 2: Writing a recursive case

EMERTXE

Advanced C

Functions - Recursive - Example

Example

```
#include <stdio.h>

/* Factorial of 3 numbers */

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

int main()
{
    int ret;

    ret = factorial(3);
    printf("Factorial of 3 is %d\n", ret);

    return 0;
}
```

n	!n
0	1
1	1
2	2
3	6
4	24

- A variable that stores the pointer to a function.

Syntax

```
datatype (*foo)(datatype, ...);
```

- 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 data_type function_name(parameter list, ...);
```

- 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.

  Example
```
int foo(int a, ...)
{
    /* Function 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.

- 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

Advanced C

Functions - Variadic - Example

Example

```c
#include <stdio.h>

int main()
{
    int ret;

    ret = add(3, 2, 4, 4);
    printf("Sum is %d\n", ret);

    ret = add(5, 3, 3, 4, 5, 10);
    printf("Sum is %d\n", ret);

    return 0;
}
```

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

    /* Initilize the arg list */
    va_start(ap, count);

    sum = 0;
    for (iter = 0; iter < count; iter++)
    {
        /* Extract args */
        sum += va_arg(ap, int);
    }

    /* Cleanup */
    va_end(ap);

    return sum;
}
```

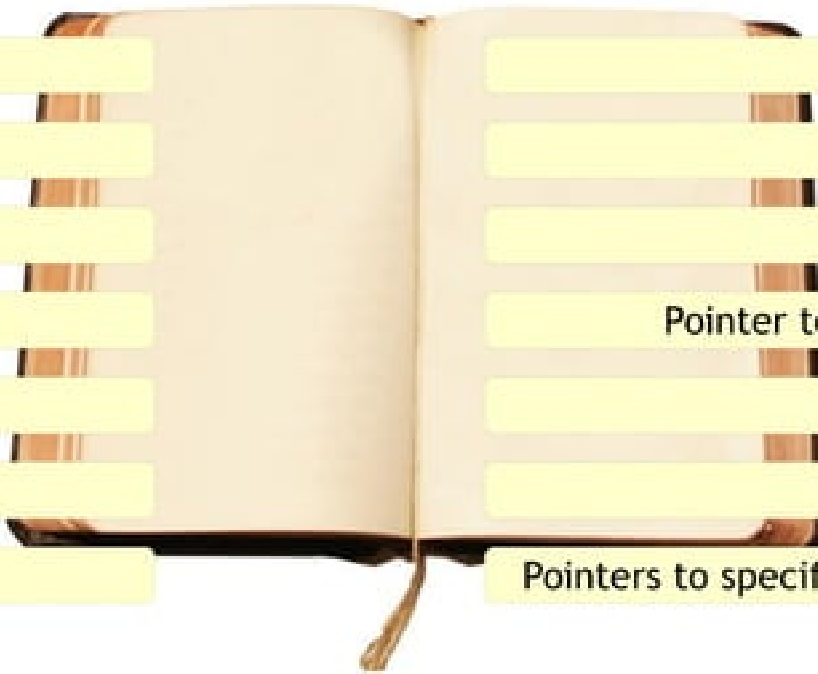
Chapter 6

Pointers and Strings

- What's a Jargon?
  - Jargon may refer to terminology used in a certain profession, such as computer jargon, or it may refer to any nonsensical language that is not understood by most people.
  - Speech or writing having unusual or pretentious vocabulary, convoluted phrasing, and vague meaning.
- Pointer are perceived difficult
  - Because of jargonification
- So, let's dejargonify & understand them

Advanced C

Pointers – Analogy with Book



A Book	System Memory
Contains pages	Memory Pages
Front matter	Boot Code
Table of Contents	Pointer to different programs
Chapters	Code Segments
Notes Section	Data & Stack
Indexes	Pointers to specific memory locations

- Just like a book analogy, Computers contains different different sections (Code) in the memory
- All sections have different purposes
- Every section has a address and we need to point to them whenever required
- In fact everything (Instructions and Data) in a particular section has a address!!
- So the pointer concept plays a big role here

Advanced C

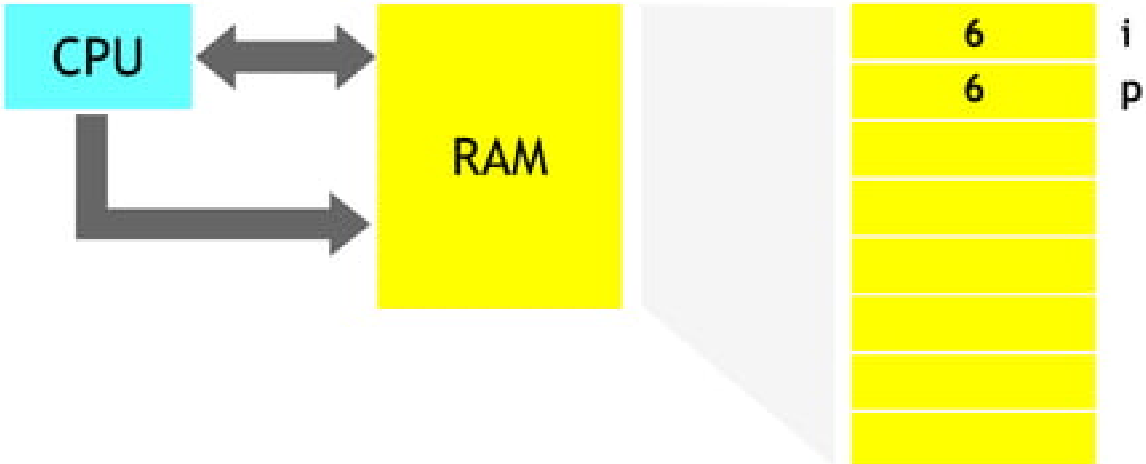
Pointers – Why?

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

- Rule 1 - Pointer is an Integer
- Rule 2 - Referencing and De-referencing
- Rule 3 - Pointing means Containing
- Rule 4 - Pointer Type
- Rule 5 - Pointer Arithmetic
- Rule 6 - Pointing to Nothing
- Rule 7 - Static vs Dynamic Allocation

Advanced C

Pointers - The 7 Rules - Rule 1



Integer i;
Pointer p;
Say:

    i = 6;
    p = 6;

- Whatever we put in data bus is Integer
- Whatever we put in address bus is Pointer
- So, at concept level both are just numbers. May be of different sized buses
- Rule: "Pointer is an Integer"
- Exceptions:
  - May not be address and data bus of same size
  - Rule 2 (Will see why? while discussing it)

Advanced C

Pointers - Rule 1 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```

	x	
1000	?	Say 4 bytes

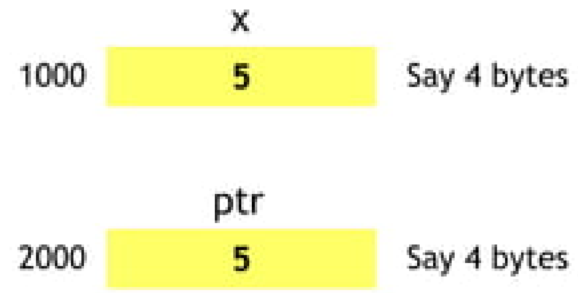
	ptr	
2000	?	Say 4 bytes

Advanced C

Pointers – Rule 1 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = 5;

    return 0;
}
```



- So pointer is an integer
- But remember the “They may not be of same size”

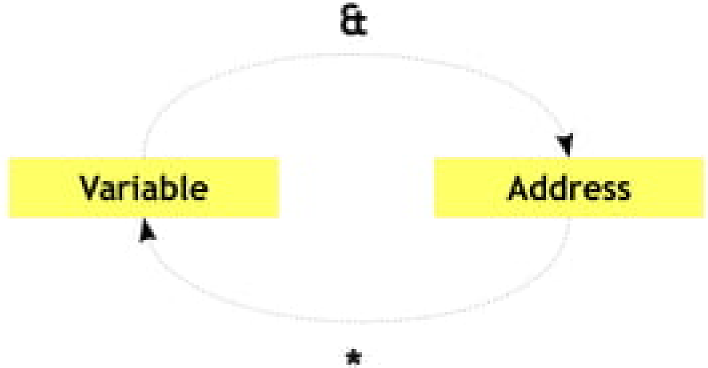
  32 bit system = 4 Bytes

  64 bit system = 8 Bytes

Advanced C

Pointers - The 7 Rules - Rule 2

- Rule : “Referencing and Dereferencing”

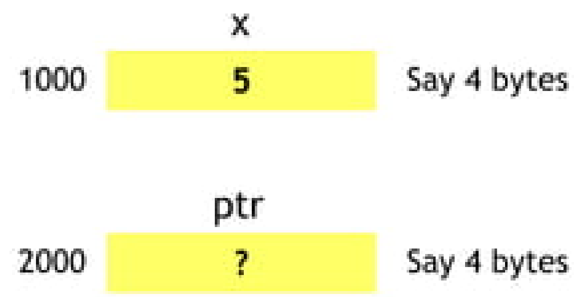


Advanced C

Pointers - Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

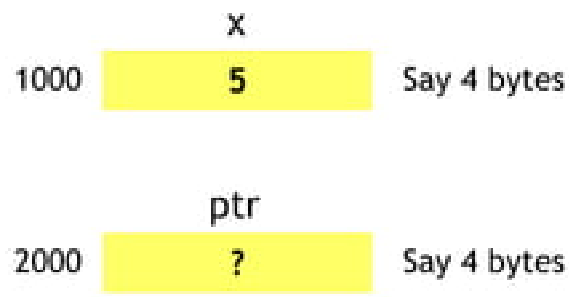
  * 1000

Advanced C

Pointers - Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;

    return 0;
}
```



- Considering the image, What would the below line mean?

  * 1000

- Goto to the location 1000 and fetch its value, so

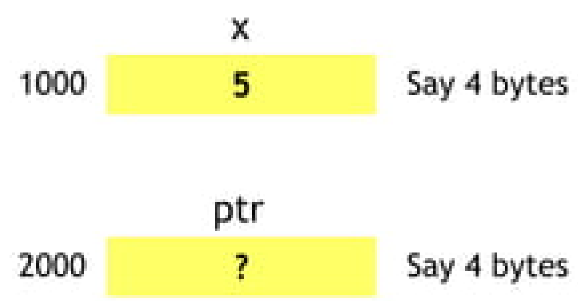
  * 1000 → 5

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x

    return 0;
}
```



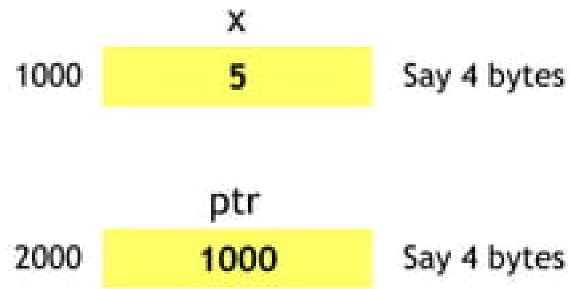
- What should be the change in the above diagram for the above code?

Advanced C

Pointers - Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x

    return 0;
}
```



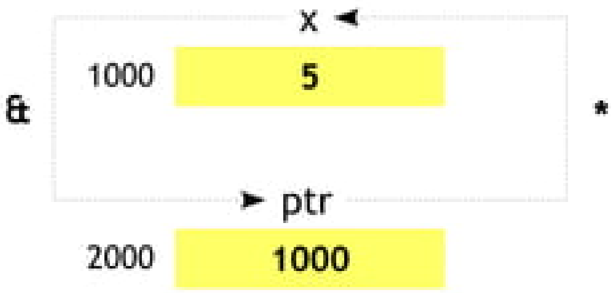
- So pointer should contain the address of a variable
- It should be a valid address

Advanced C

Pointers – Rule 2 in detail

Example

```c
#include <stdio.h>

int main()
{
    int x;
    int *ptr;

    x = 5;
    ptr = &x

    return 0;
}
```



"Add a & on variable to store its address in a pointer"

"Add a * on the pointer to extract the value of variable it is pointing to"

Advanced C

Pointers – Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("Address of number is %p\n", &number);
    printf("ptr contains %p\n", ptr);

    return 0;
}
```

Advanced C

Pointers - Rule 2 in detail

Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

Advanced C

Pointers - Rule 2 in detail

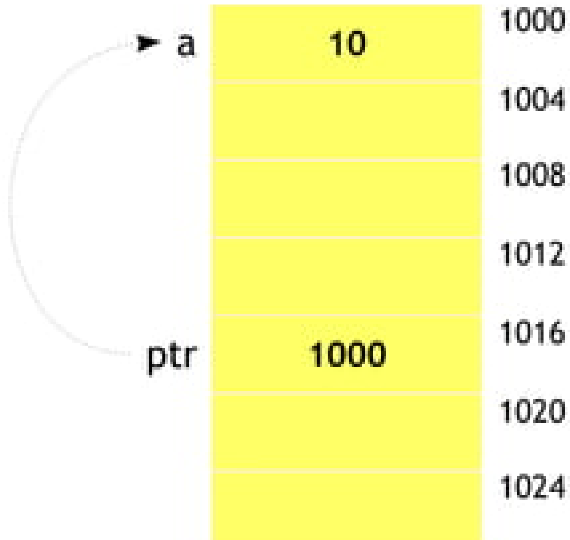
Example

```
#include <stdio.h>

int main()
{
    int number = 10;
    int *ptr;

    ptr = &number;
    *ptr = 100;

    printf("number contains %d\n", number);
    printf("*ptr contains %d\n", *ptr);

    return 0;
}
```

- By compiling and executing the above code we can conclude

“*ptr <=> number”

- Pointer pointing to a Variable = Pointer contains the Address of the Variable
- Rule: "Pointing means Containing"
- int *ptr; int a = 5; ptr = &a;



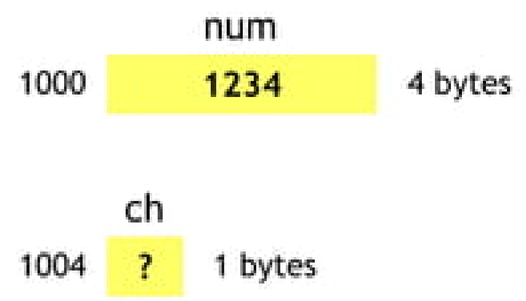
- So, why do we need types attached to pointers?
  - Only for 'De-referencing'
  - Pointer arithmetic
- Summarized as the following rule:

Rule: "Pointer of type t = t Pointer = (t *) = A variable, which contains an address, which when dereferenced returns a variable of type t, starting from that address"

- Does address has a type?

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    return 0;
}
```

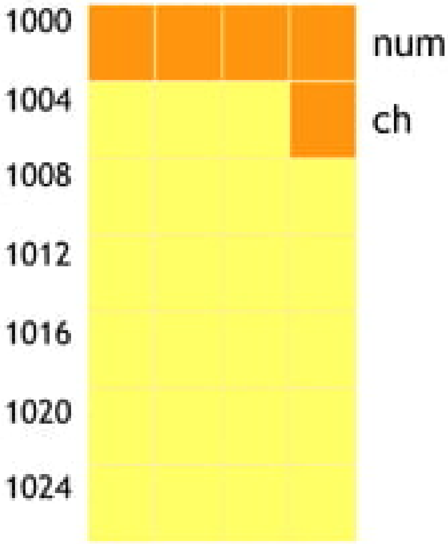


- So, from the above above diagram can we say &num → 4 bytes and &ch → 1 byte?

Advanced C

Pointers - Rule 4 in detail

- The answer is no!!, it does not depend on the type of the variable
- The size of address remains the same, and it depends on the system we use
- Then a simple questions arises is why types to pointers?

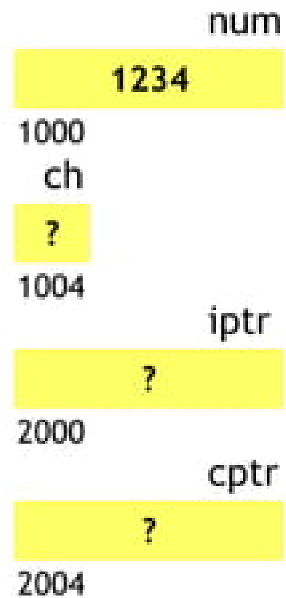


Advanced C

Pointers - Rule 4 in detail

Example

```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr;
    char *cptr;

    return 0;
}
```



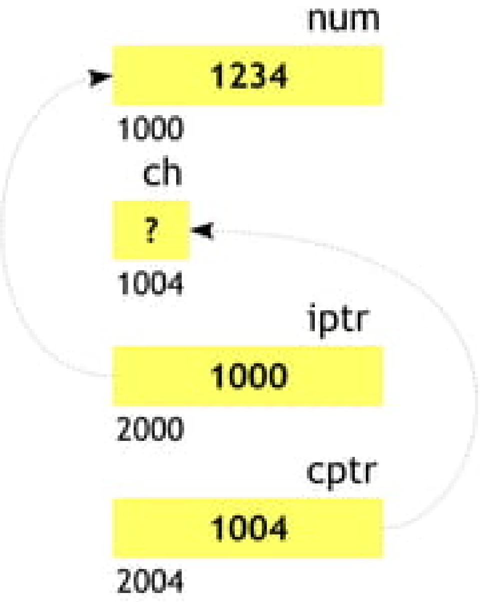
- Lets consider the above examples to understand it
- Say we have an integer and a character pointer

Advanced C

Pointers - Rule 4 in detail

Example

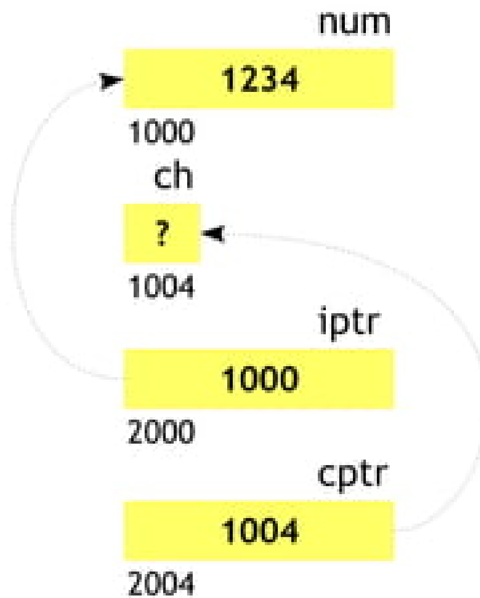
```c
#include <stdio.h>

int main()
{
    int num = 1234;
    char ch;

    int *iptr = &num;
    char *cptr = &ch;

    return 0;
}
```



- Lets consider the above examples to understand it
- Say we have an integer and a character pointer

Advanced C
Pointers - Rule 4 in detail

- With just the address, can know what data is stored?
- How would we know how much data to fetch for the address it is pointing to?
- Eventually the answer would be NO!!
- So the type of the pointer is required while
  - Dereferencing it
  - Doing pointer arithmetic



Advanced C

Pointers - Rule 4 in detail

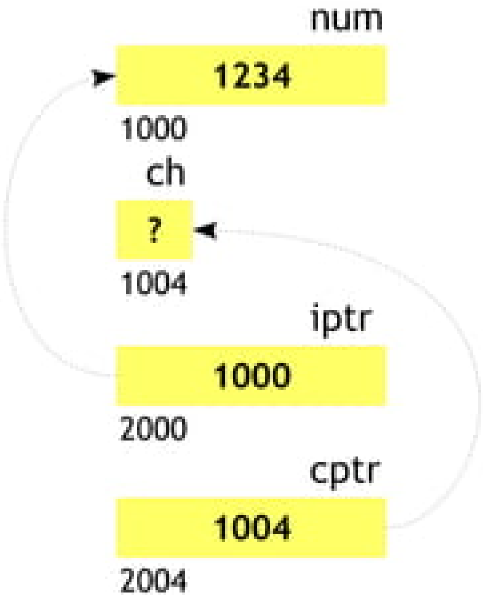
- When we say while dereferencing, how does the pointer know how much data it should fetch at a time
- From the diagram right side we can say

  *cptr fetches a single byte

  *iptr fetches 4 consecutive bytes

- So as conclusion we can say

type * → fetch sizeof(type) bytes



Advanced C

Pointers – Rule 4 in detail - Endianness

- Since the discussion is on the data fetching, its better we have knowledge of storage concept of machines
- The Endianness of the machine
- What is this now!!?
  - Its nothing but the byte ordering in a word of the machine
- There are two types
  - Little Endian – LSB in Lower order Memory Address
  - Big Endian – MSB in Lower Memory Address

- LSB
  - The byte of a multi byte number with the least importance
  - The change in it would have least effect on complete number
- MSB
  - The byte of a multi byte number with the most importance
  - The change in it would have more effect on complete change number

Advanced C

Pointers - Rule 4 in detail - Endianness

Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;

    return 0;
}
```

- Let us consider the following example and how it would be stored in both machine types

Big Endian

1000	12	34	56	78	num
1004					

→

1000	1001	1002	1003
12	34	56	78

Little Endian

1000	78	56	34	12	num
1004					

→

1000	1001	1002	1003
78	56	34	12

*Little Endian - Lower order byte at Lower address

Advanced C

Pointers - Rule 4 in detail - Endianness

- OK Fine. What now? How is it going to affect to fetch and modification?
- Let us consider the same example put in the previous slide

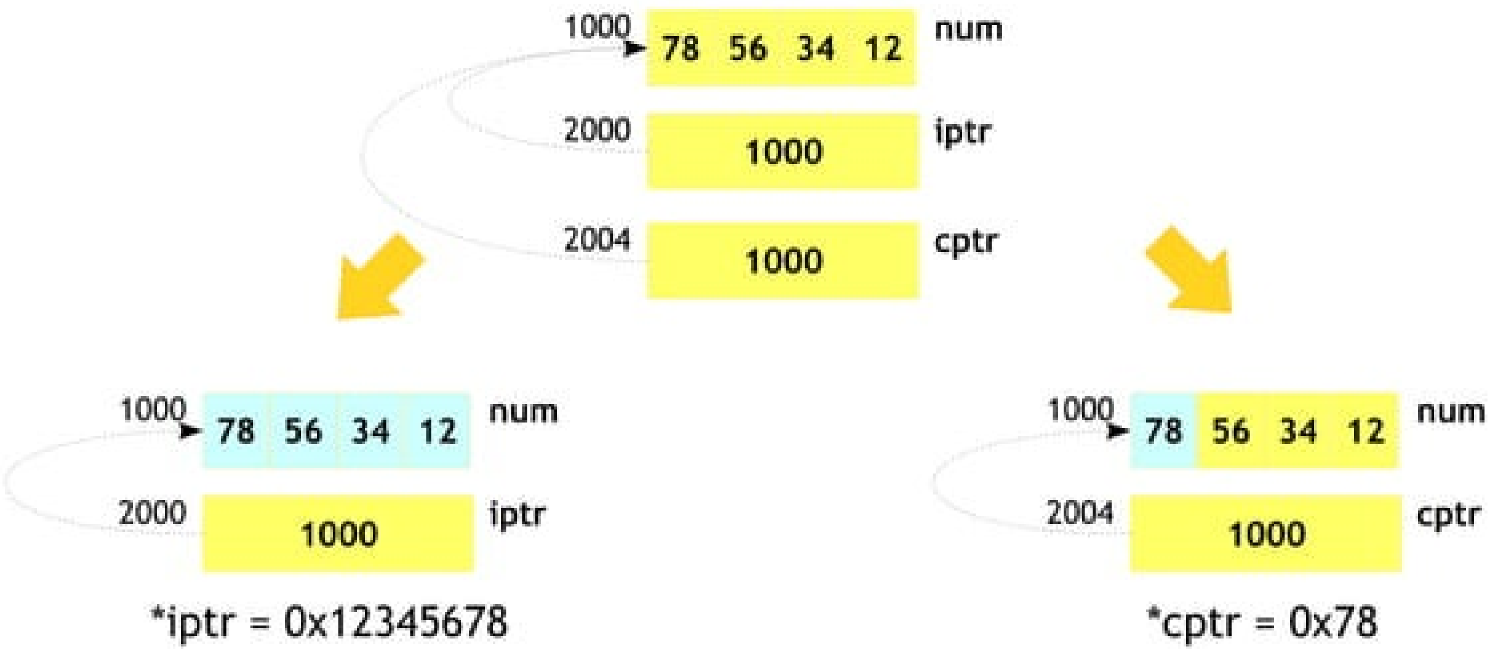
Example

```
#include <stdio.h>

int main()
{
    int num = 0x12345678;
    int *iptr, char *cptr;

    iptr = &num;
    cptr = &num;


    return 0;
}
```

- First of all is it possible to access a integer with character pointer?
- If yes, what should be the effect on access?
- Let us assume a Litte Endian system

Advanced C

Pointers - Rule 4 in detail - Endianness



- So from the above diagram it should be clear that when we do cross type accessing, the endianness should be considered

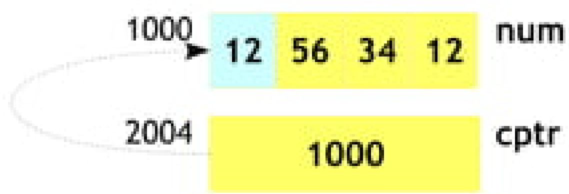
Advanced C

Pointers - The 7 Rules - Rule 5

Example

```c
#include <stdio.h>

int main()
{
    int num = 0x12345678;
    char ch;

    int *iptr = &num;
    char *cptr = &num;

    *cptr = 0x12;

    return 0;
}
```

- So changing *cptr will change only the byte its pointing to



*cptr = 0x12

- So *iptr would contain 0x12345612 now!!

- In conclusion, the type of a pointer represents it's ability to perform read or write operations on number of bytes (data) its pointing to
- So the size of the pointer for different types remains the same

Example

```
#include <stdio.h>

int main()
{
    if (sizeof(char *) == sizeof(long long *))
    {
        printf("Yes its Equal\n");
    }

    return 0;
}
```

- Pointer Arithmetic

  Rule: "Value(p + i) = Value(p) + i * sizeof(*p)"

- Before proceeding further let us understand an array interpretation

  (1) Original Big Variable (bunch of variables, whole array)

  (2) Constant Pointer to the 1st Small Variable in the bunch (base address)

- When first interpretation fails than second interpretation applies

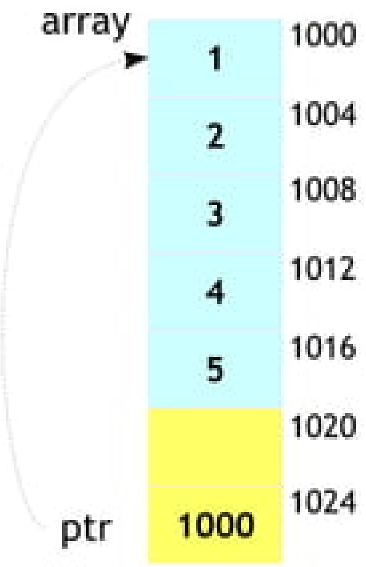
- Cases when first interpretation applies
  - When name of array is operand to sizeof operator
  - When "address of operator (&)" is used the with name of array while performing pointer arithmetic
- The following are the case when first interpretation fails:
  - When we pass array name as function argument
  - When we assign an array variable to pointer variable

Advanced C

Pointers - The Rule 5 in detail

Example

```
#include <stdio.h>

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    int *ptr = array;

    return 0;
}
```



- So,

  Address of array = 1000

  Base address = 1000

  &array[0] = 1 → 1000

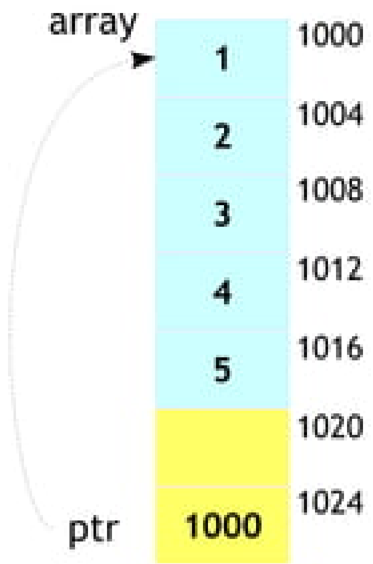
  &array[1] = 2 → 1004

Advanced C

Pointers - The Rule 5 in detail

Example

```
#include <stdio.h>

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    int *ptr = array;

    printf("%d\n", *ptr);

    return 0;
}
```



- This code should print 1 as output since its points to the base address
- Now, what should happen if we do

  ptr = ptr + 1;

Advanced C

Pointers – The Rule 5 in detail

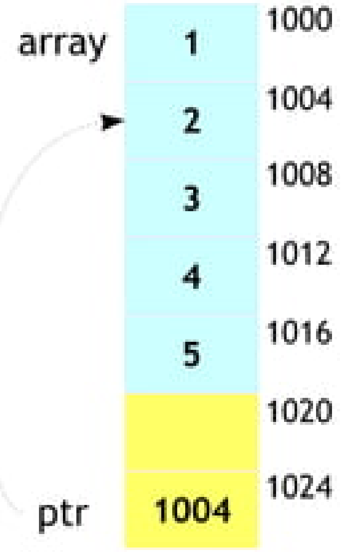
- ptr = ptr + 1;
- The above line can be described as follows
- ptr = ptr + 1 * sizeof(data type)
- In this example we have an integer array, so
- ptr = ptr + 1 * sizeof(int)

  = ptr + 1 * 4

  = ptr + 4
- Here ptr = 1000 so

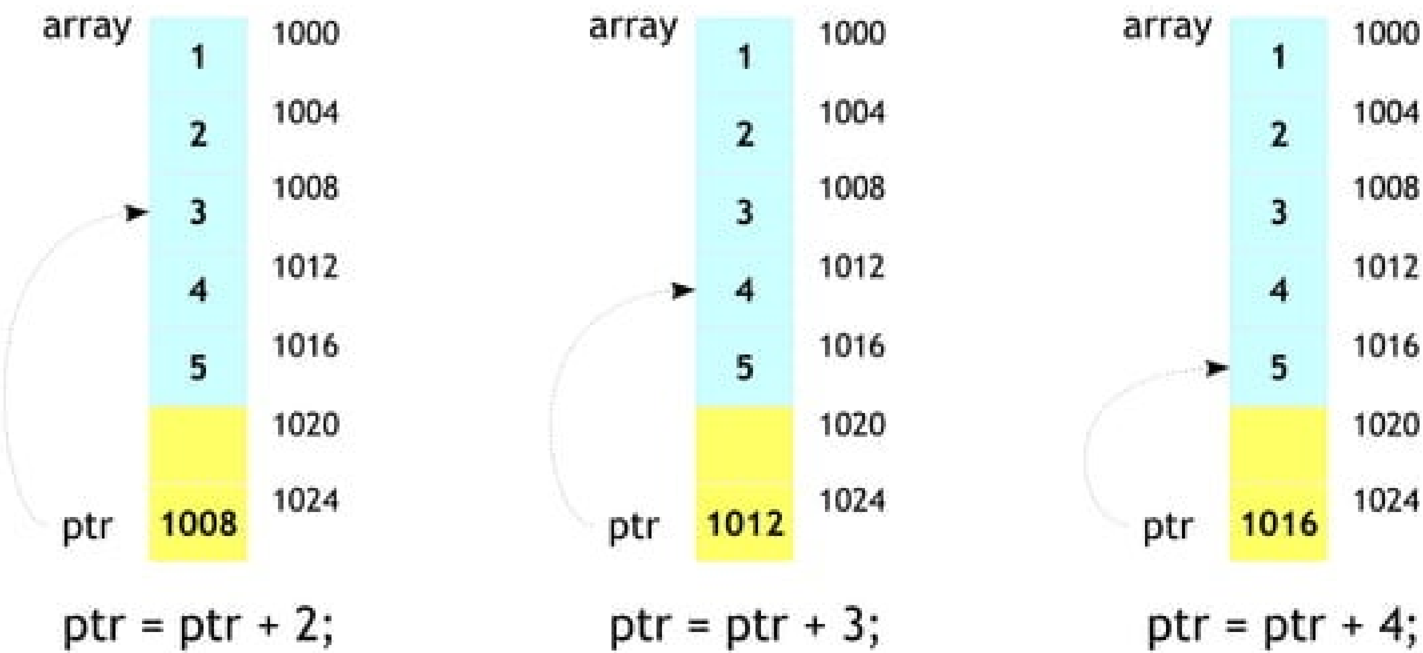
  = 1000 + 4

  = 1004



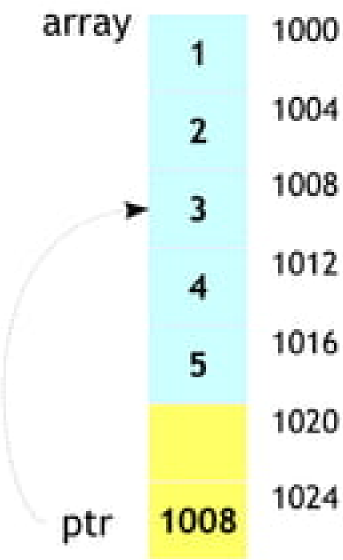
Advanced C

Pointers – The Rule 5 in detail



- Why does the compiler does this?. Just for convenience

Advanced C
Pointers - The Rule 5 in detail



ptr = ptr + 2;

- Relation with array can be explained as

  ptr + 2

  ptr + 2 * sizeof(int)

  1000 + 2 * 4

  1008 → &array[2]

- So,

  ptr + 2 → 1008 → &array[2]

  *(ptr + 2) → *(1008) → array[2]

- So to access an array element using a pointer would be

$$*(ptr + i) \rightarrow array[i]$$

- This can be written as following too!!

$$array[i] \rightarrow *(array + i)$$

- Which results to

$$ptr = array$$

- So as summary the below line also becomes valid because of second array interpretation

```
int *ptr = array;
```

- Wait can I write

*(ptr + i) → *(i + ptr)

- Yes. So than can I write

array[i] → i[array]

- Yes. You can index the element in both the ways

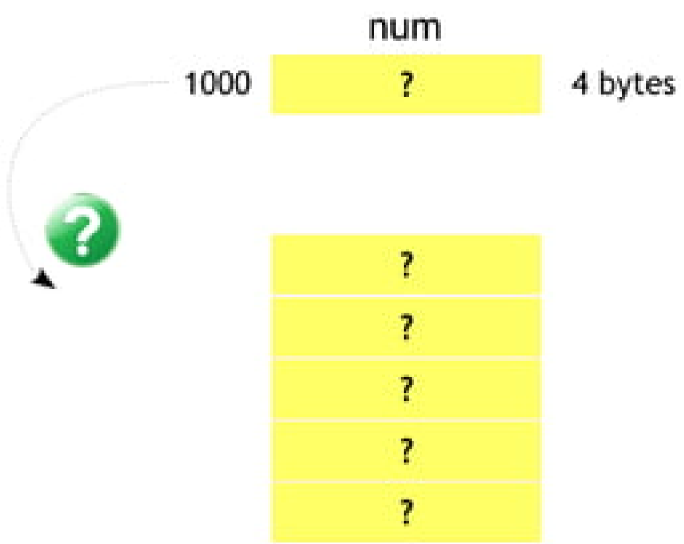
- Rule: “Pointer value of NULL or Zero = Null Addr = Null Pointer = Pointing to Nothing”

Advanced C

Pointers - Rule 6 in detail - NULL Pointer

Example

```
#include <stdio.h>

int main()
{
    int *num;

    return 0;
}
```



Where am I pointing to?

What does it Contain?

Can I read or write wherever I am pointing?

- Is it pointing to the valid address?
- If yes can we read or write in the location where its pointing?
- If no what will happen if we access that location?
- So in summary where should we point to avoid all this questions if we don't have a valid address yet?
- The answer is Point to Nothing!!

- Now, what is “Pointing to Nothing”?
- A permitted location in the system will always give predictable result!
- It is possible that we are pointing to some memory location within our program limit, which might fail any time! Thus making it bit difficult to debug.
- An act of initializing pointers to 0 (generally, implementation dependent) at definition.
- 0??, Is it a value zero? So a pointer contain a value 0?
- Yes. On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system

Advanced C

Pointers – Rule 6 in detail – NULL Pointer

- So by convention if a pointer is initialized to zero value, it is logically understood to be "pointing to nothing".
- And now, in the pointer context, 0 is called as **NULL**
- So a pointer that is assigned NULL is called a Null Pointer which is **Pointing to Nothing**
- So dereferencing a NULL pointer is illegal and will always lead to segment violation, which is better than pointing to some unknown location and failing randomly!

Pointers - Rule 6 in detail - NULL Pointer

- Need for Pointing to 'Nothing'
  - Terminating Linked Lists
  - Indicating Failure by malloc, ...and string functions
- Solution
  - Need to reserve one valid value
  - Which valid value could be most useless?
  - In wake of OSes sitting from the start of memory, 0 is a good choice
  - As discussed in previous slides it is implementation dependent

Advanced C

Pointers – Rule 6 in detail – NULL Pointer

Example

```
#include <stdio.h>

int main()
{
    int *num;

    num = NULL;

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    int *num = NULL;

    return 0;
}
```

- A pointer with incomplete type
- Void pointer can't be dereferenced. You MUST use type cast operator (type) to dereference.
- Exercise -
  - W.A.P to swap any given data type

- The void type comprises an empty set of values; it is an incomplete type that cannot be completed
- Hence pointer arithmetic can NOT be performed on void pointer

Note: To make standard compliant, compile using
- gcc -pedantic-errors or -Werror-pointer-arith

GCC Extension :

6.22 Arithmetic on void- and Function-Pointers

In GNU C, addition and subtraction operations are supported on "pointers to void" and on "pointers to functions". This is done by treating the size of a void or of a function as 1.

A consequence of this is that sizeof is also allowed on void and on function types, and returns 1.

The option **-Wpointer-arith** requests a warning if these extensions are used

- Rule: "Static Allocation vs Dynamic Allocation"

Example

```
#include <stdio.h>

int main()
{
    char array[5];

    return 0;
}
```

Example

```
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(5);

    return 0;
}
```

- Named vs Unnamed Allocation = Named/Unnamed Houses

Ok, House 1, I should go that side ←

Ok, House 1, I should go??? Oops

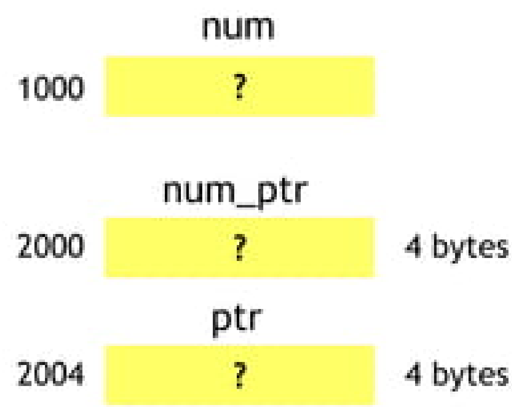
- Managed by Compiler vs User
- Compiler
  - The compiler will allocate the required memory
  - Required memory is computed at compile time
  - Memory is allocated in data segment or stack
- User
  - The user has to allocate the memory whenever required and deallocate whenever required
  - This is done by using malloc and free functions
  - Memory is allocated in heap

Advanced C
Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(4);

    return 0;
}
```



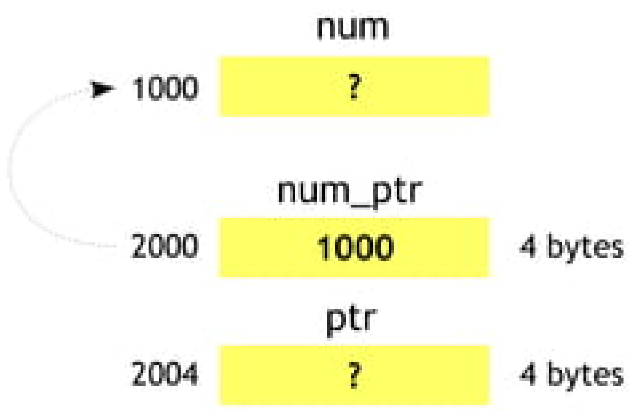
Advanced C

Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```c
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(4);

    return 0;
}
```



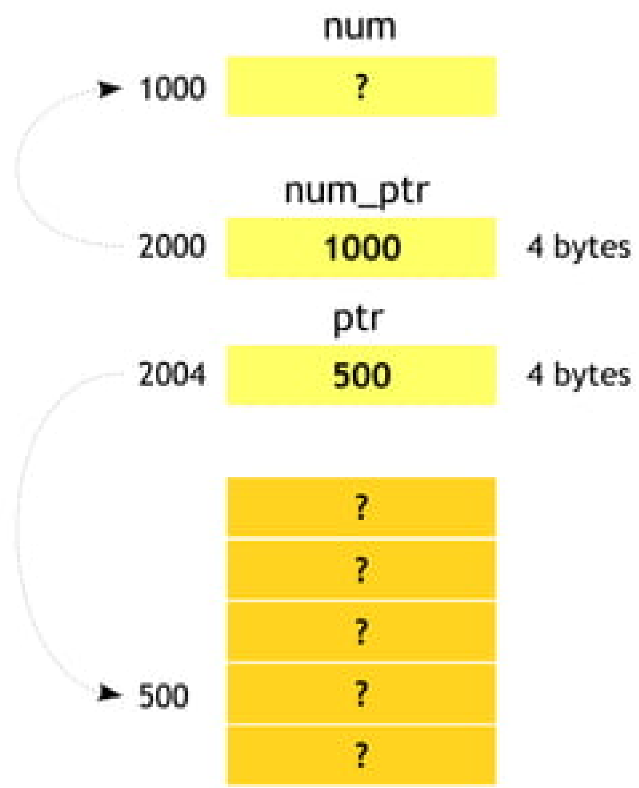
Advanced C

Pointers - Rule 7 in detail

- Static vs Dynamic

Example

```
#include <stdio.h>

int main()
{
    int num, *num_ptr, *ptr;

    num_ptr = &num;

    ptr = malloc(4);

    return 0;
}
```



- The need
  - You can decide size of the memory at run time
  - You can resize it whenever required
  - You can decide when to create and destroy it

Prototype

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

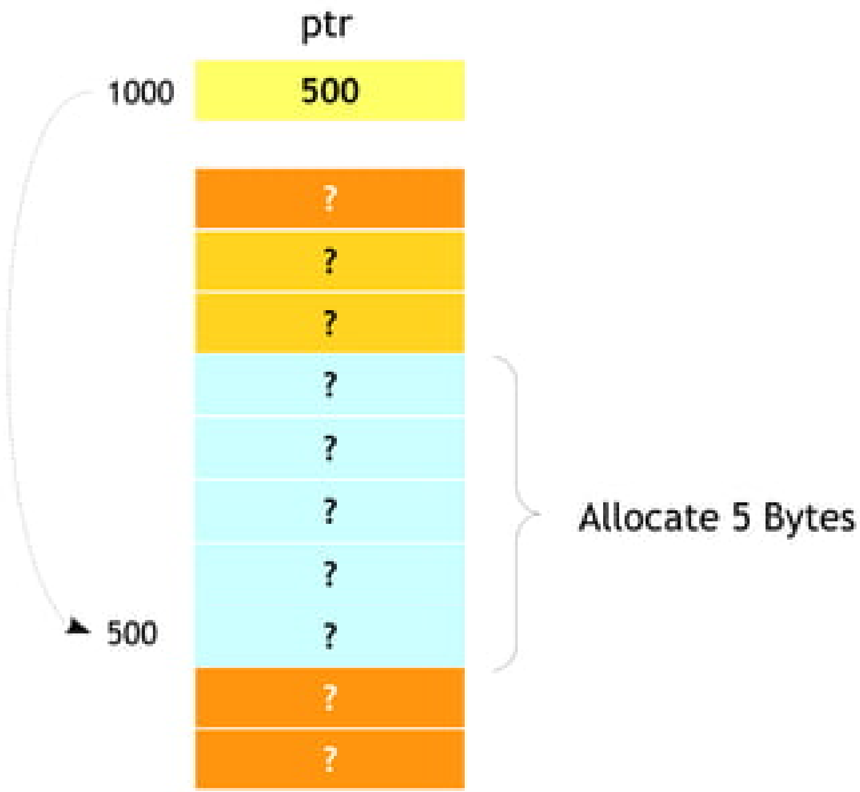
- Allocates the requested size of memory from the heap
- The size is in bytes
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers - Rule 7 - Dynamic Allocation - malloc

Example

```c
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(5);

    return 0;
}
```

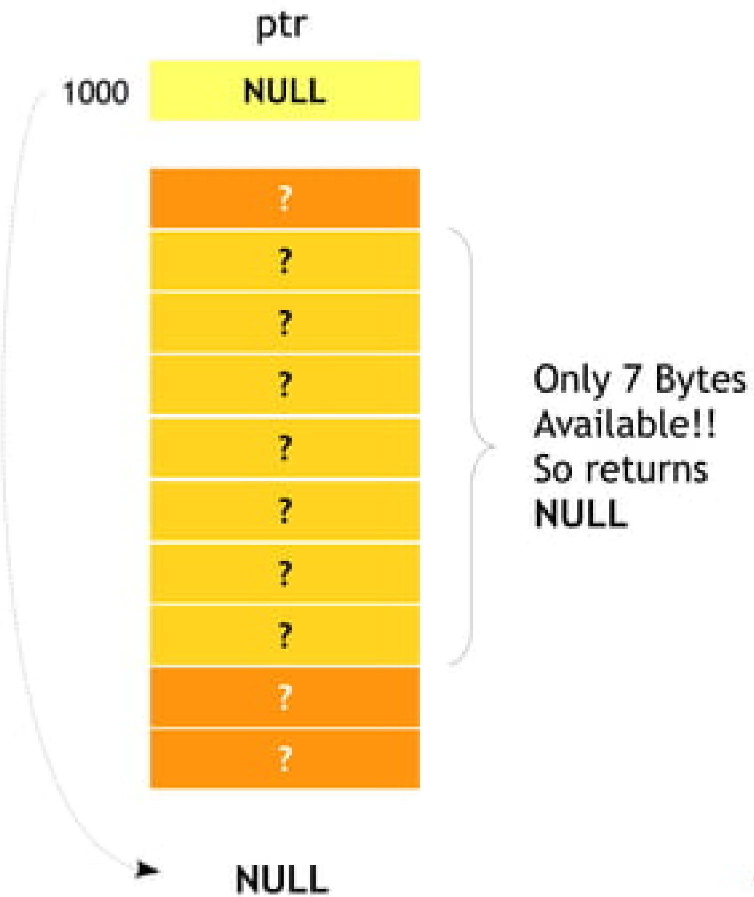


Advanced C

Pointers - Rule 7 - Dynamic Allocation - malloc

Example

```
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(10);

    return 0;
}
```



Prototype

```
void *calloc(size_t nmemb, size_t size);
```

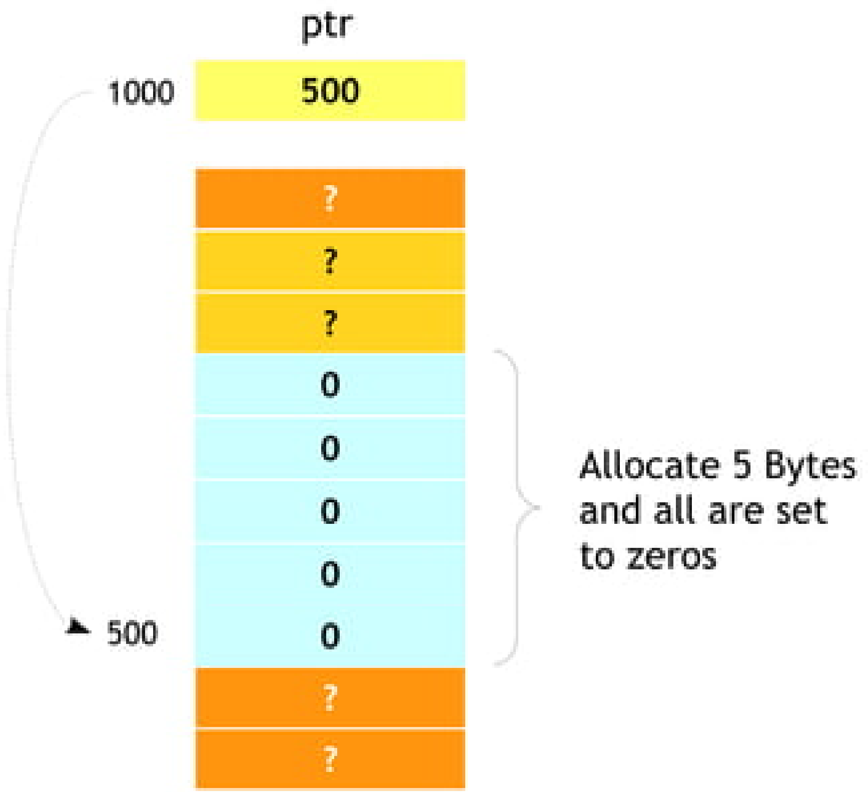
- Allocates memory blocks large enough to hold "n elements" of "size" bytes each, from the heap
- The allocated memory is set with 0's
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers - Rule 7 - Dynamic Allocation - calloc

Example

```c
#include <stdio.h>

int main()
{
    char *str;

    str = calloc(5, 1);

    return 0;
}
```



Prototype

```
void *realloc(void *ptr, size_t size);
```

- Changes the size of the already allocated memory by malloc or calloc
- Returns the pointer of the allocated memory on success, else returns NULL pointer

Advanced C

Pointers - Rule 7 - Dynamic Allocation - realloc

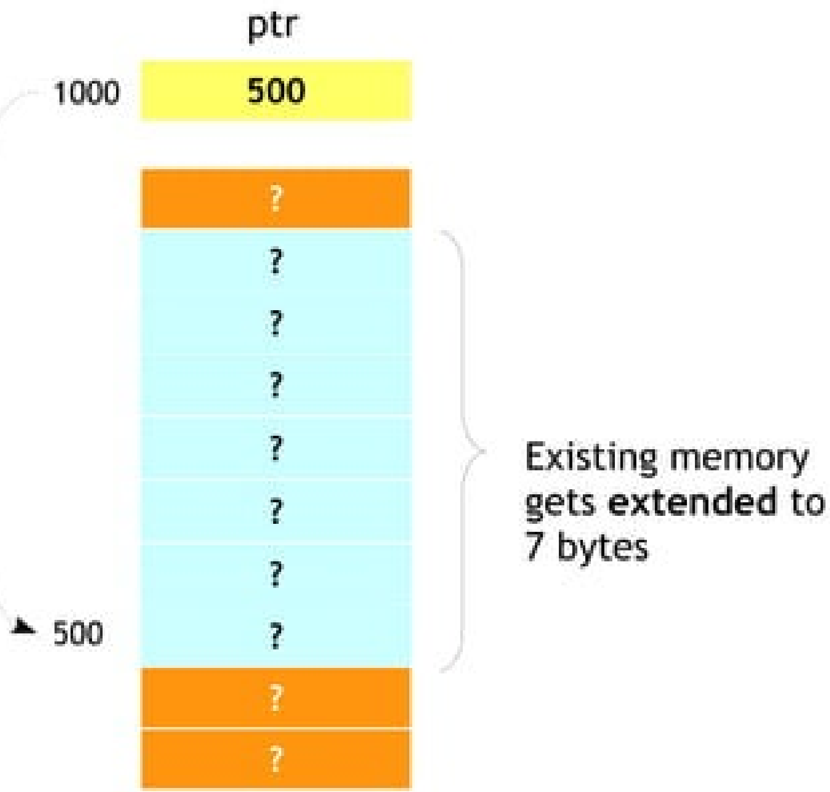
Example

```c
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(5);

    str = realloc(str, 7);
    str = realloc(str, 2);

    return 0;
}
```

Pointers – Rule 7 – Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(5);

    str = realloc(str, 7);
    str = realloc(str, 2);

    return 0;
}
```



Advanced C

Pointers - Rule 7 - Dynamic Allocation - realloc

Example

```c
#include <stdio.h>

int main()
{
    char *str;

    str = malloc(5);

    str = realloc(str, 7);
    str = realloc(str, 2);

    return 0;
}
```

- Points to be noted
  - Reallocating existing memory will be like deallocating the allocated memory
  - If the requested chunk of memory cannot be extended in the existing block, it would allocate in a new free block starting from different memory!
  - If new memory block is allocated then old memory block is automatically freed by realloc function

Pointers – Rule 7 – Dynamic Deallocation - free

Prototype

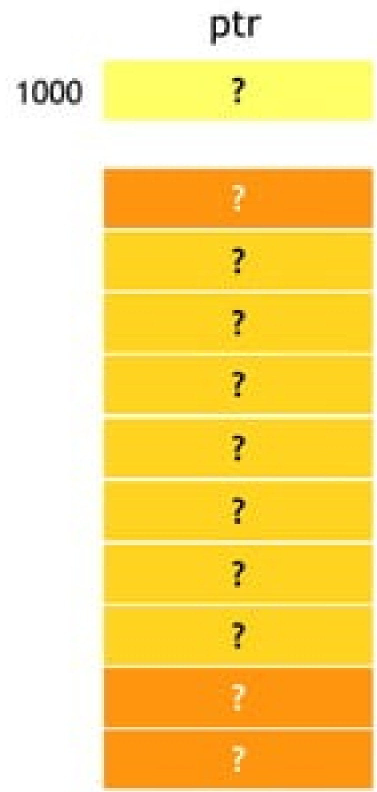
```
void free(void *ptr);
```

- Frees the allocated memory, which must have been returned by a previous call to malloc(), calloc() or realloc()
- Freeing an already freed block or any other block, would lead to undefined behavior
- Freeing NULL pointer has no abnormal effect.
- If free() is called with invalid argument, might collapse the memory management mechanism
- If free is not called after dynamic memory allocation, will lead to memory leak

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    for (iter = 0; iter < count; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```

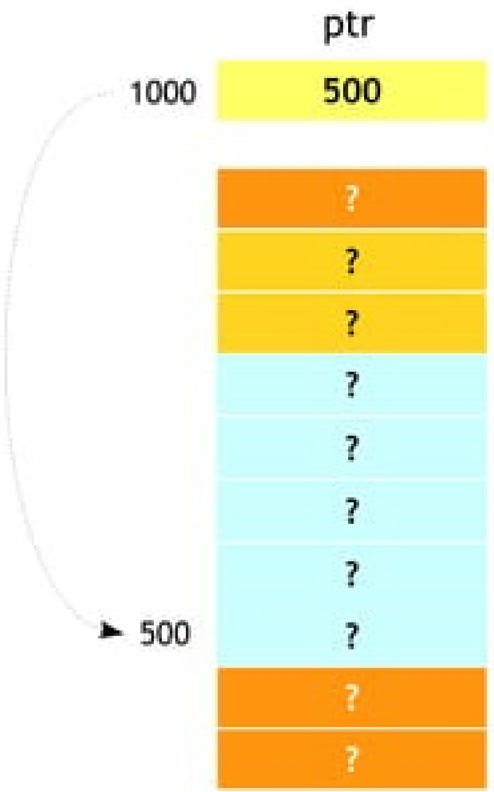


Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

```c
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    for (iter = 0; iter < count; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```

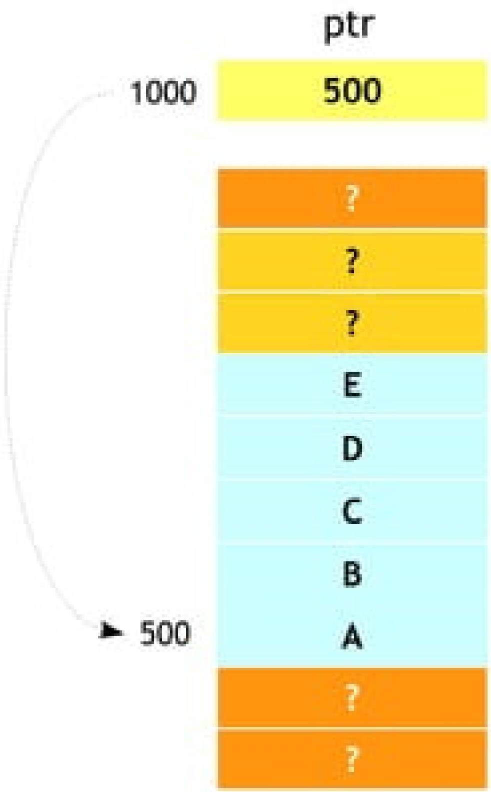


Advanced C

Pointers – Rule 7 – Dynamic Deallocation - free

Example

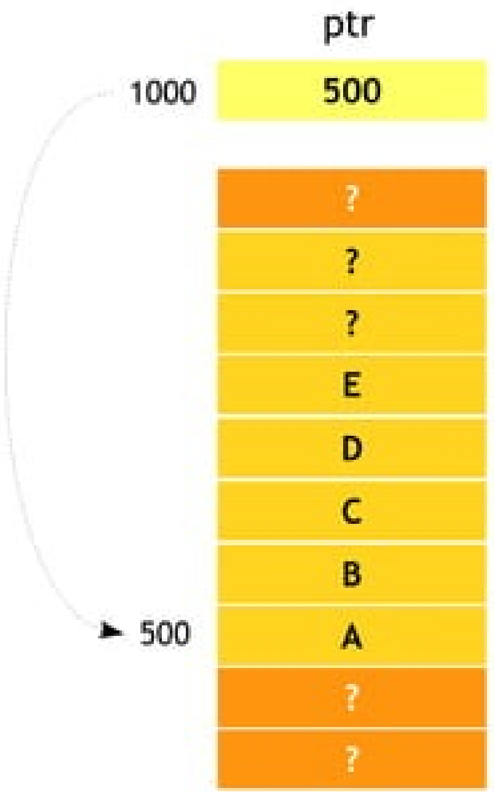
```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    for (iter = 0; iter < count; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```



Pointers – Rule 7 – Dynamic Deallocation - free

Example

```
#include <stdio.h>

int main()
{
    char *ptr;

    ptr = malloc(5);

    for (iter = 0; iter < count; iter++)
    {
        ptr[iter] = 'A' + iter;
    }

    free(ptr);

    return 0;
}
```



- Points to be noted
  - Free releases the allocated block, but the pointer would still be pointing to the same block!!, So accessing the freed block will have undefined behavior
  - This type of pointer which are pointing to freed locations are called as Dangling Pointers
  - Doesn't clear the memory after freeing

- A pointer, pointing to another pointer which can be pointing to others pointers and so on is know as multilevel pointers.
- We can have any level of pointers.
- As the depth of the level increase we have to bit careful while dealing with it.

Example

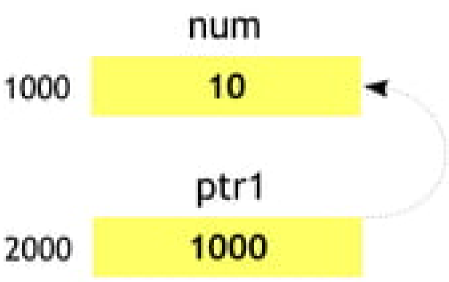
```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

num

1000 | 10

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

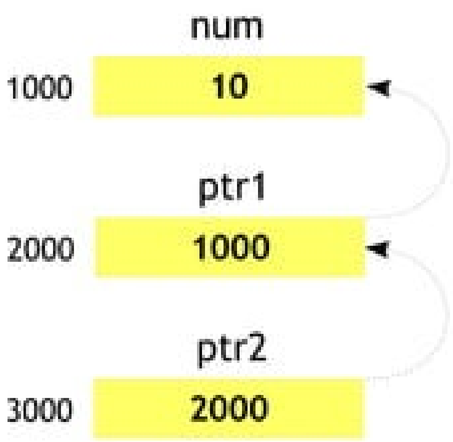


Advanced C

Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

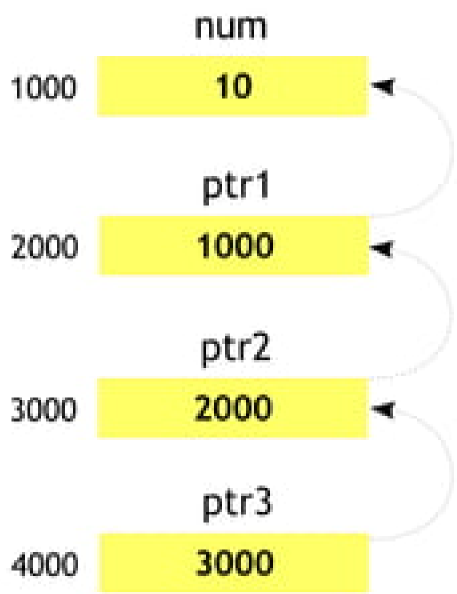


Advanced C

Pointers – Multilevel

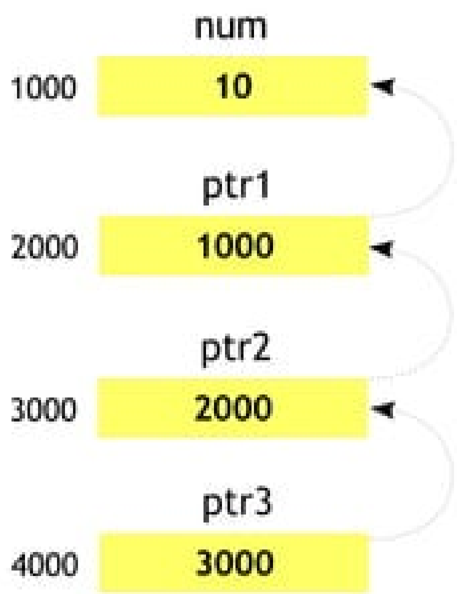
Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%p", ptr3);
    printf("%p", *ptr3);
    printf("%p", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%p", ptr3);
    printf("%p", *ptr3);
    printf("%p", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



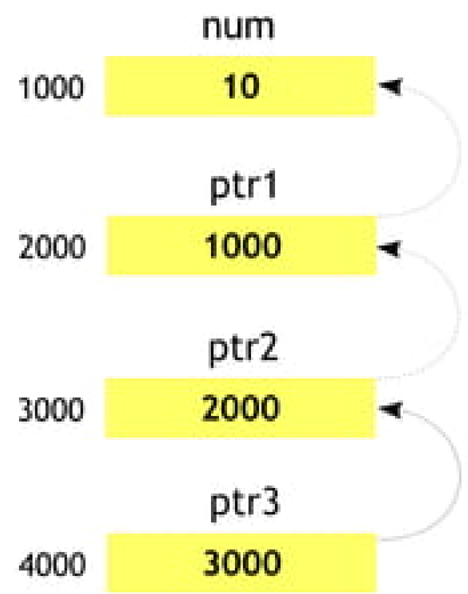
Output → 3000

Advanced C

Pointers - Multilevel

Example

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

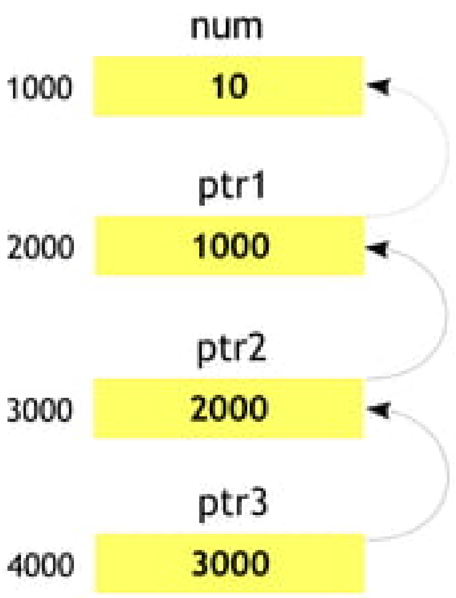


Output → 2000

Advanced C
Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

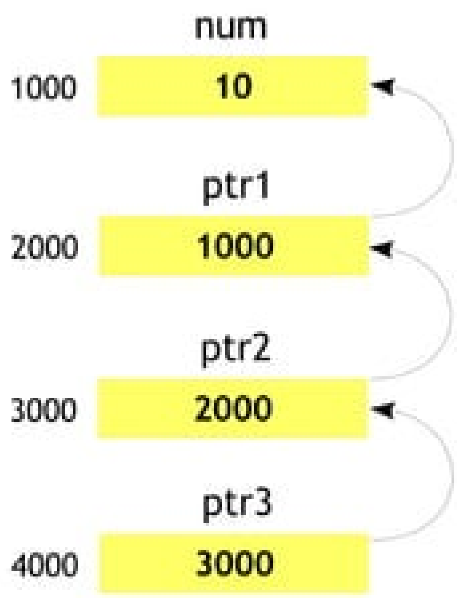


Output → 1000

ΣMERTXE

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



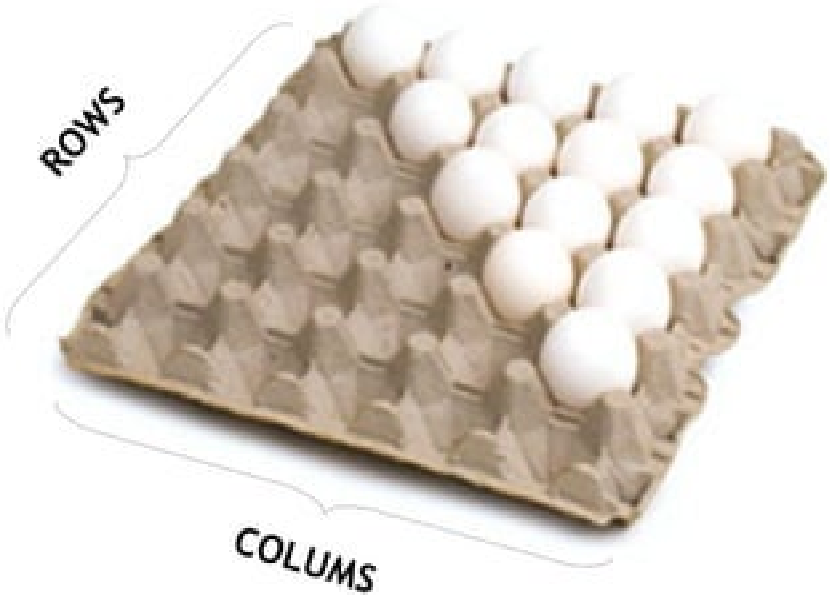
Output → 10

- A read only pointer
- Valid address MUST be assigned while defining constant pointer
- Once address assigned, it can't be changed

int arr[5];

- What is the interpretation of arr?
  - Whole array or Base address
- What is (arr + i) ?
  - Address of ith element
- What is *(arr + i)
  - Content of ith element

- What is the type of array?
  - Type of array is derived from it's elements
- What *(arr + i) would return?
  - It would depend on dimension of arr.
  - Dereferencing n-dimension array returns n-1 dimension array
  - 1 D array would return primitive data value
- What is *(arr + i) equivalent to?
  - arr[i]


ROWS
COLUMS

Advanced C

Pointers - 2D Array

$$
\begin{array}{c|c}
 & \text{COLS} \\
\text{ROWS} & \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}
\end{array}
\Rightarrow
\begin{array}{cccc}
 & C0 & C1 & C2 \\
R0 & \multicolumn{3}{c}{\multirow{2}{*}{$\begin{bmatrix} 00 & 01 & 02 \\ 10 & 11 & 12 \end{bmatrix}$}} \\
R1 & & &
\end{array}
$$

Example

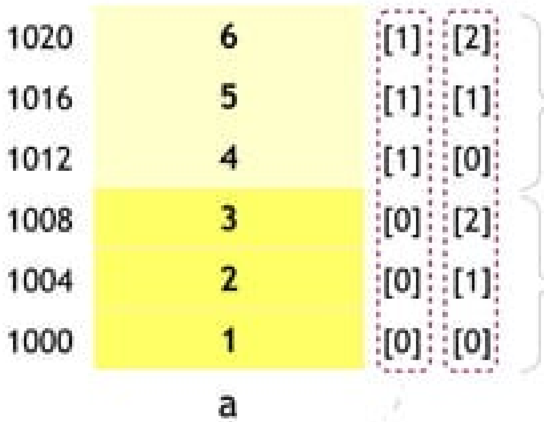
```
#include <stdio.h>

int main()
{
    int a[2][3] = {1, 2, 3, 4, 5, 6};

    return 0;
}
```

```
Total Memory: ROWS * COLS * sizeof(datatype) Bytes
```

Advanced C

Pointers – 2D Array - Referencing

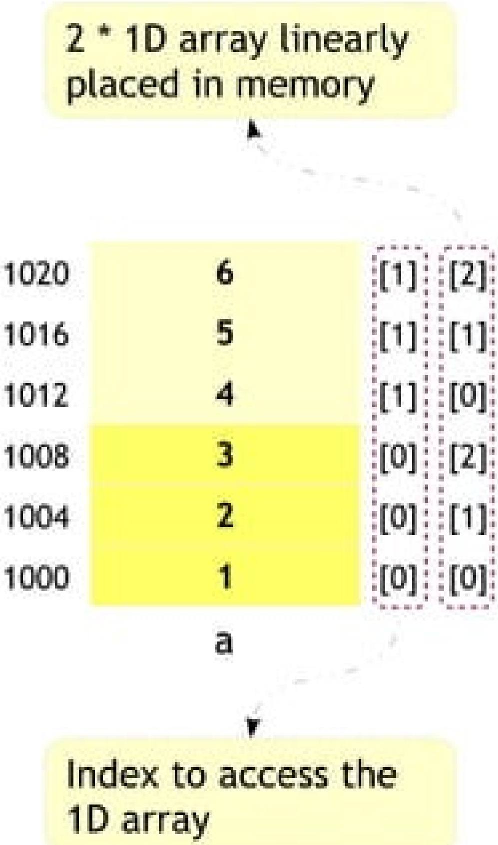
2 * 1D array linearly placed in memory



2nd 1D Array with base address 1012
a[1] = &a[1][0] = a + 1 → 1012

1st 1D Array with base address 1000
a[0] = &a[0][0] = a + 0 → 1000

Index to access the 1D array



Example 1: Say **a[0][1]** is to be accessed, then decomposition happens like,

```
a[0][1] = *(a[0] + 1 * sizeof(type))
        = *(*(a + 0 * sizeof(1D array)) + 1 * sizeof(type))
        = *(*(1000 + 0 * 12) + 1 * 4)
        = *(*(1000 + 0) + 4)
        = *(*(1004))
        = 2
```

Example 2: Say **a[1][2]** is to be accessed, then decomposition happens like,

```
a[1][2] = *(a[1] + 2 * sizeof(type))
        = *(*(a + 1 * sizeof(1D array)) + 2 * sizeof(type))
        = *(*(1000 + 1 * 12) + 2 * 4)
        = *(*(1000 + 12) + 8)
        = *(*(1020))
        = 6
```

Advanced C

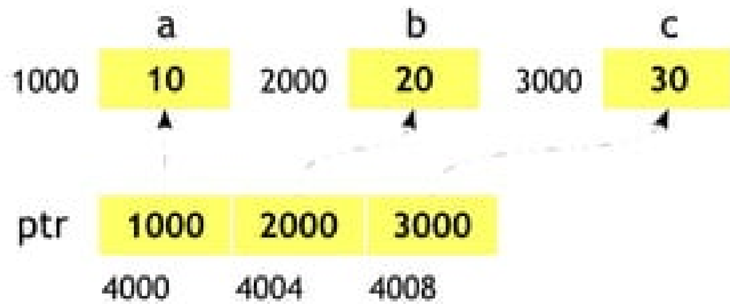
Pointers – Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

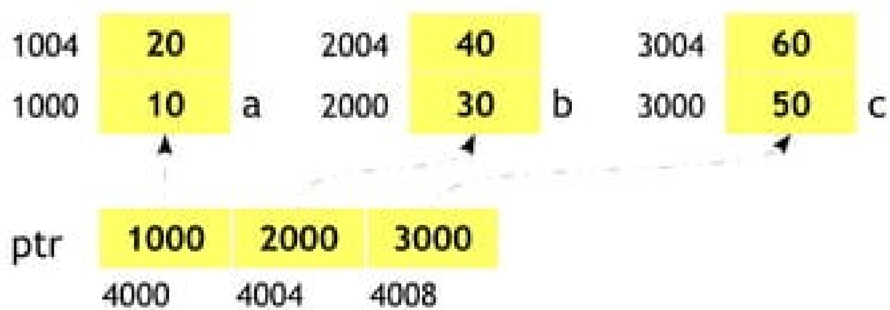
Example

```
int a = 10;
int b = 20;
int c = 30;

int *ptr[3] = {&a, &b, &c};
```



Example

```
int a[2] = {10, 20};
int b[2] = {30, 40};
int c[2] = {50, 60};

int *ptr[3] = {a, b, c};
```

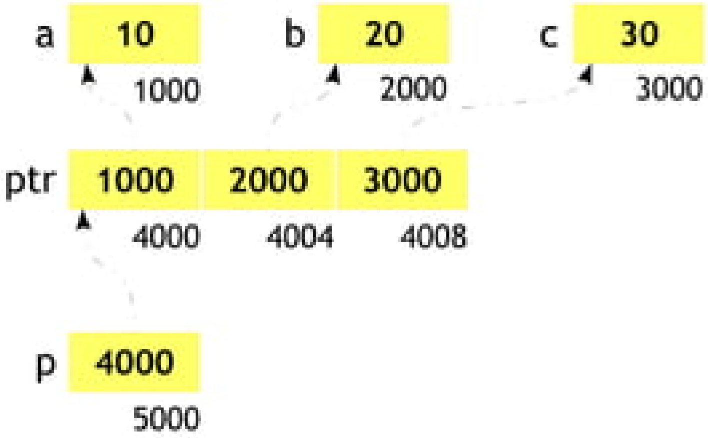


Advanced C

Pointers - Array of pointers

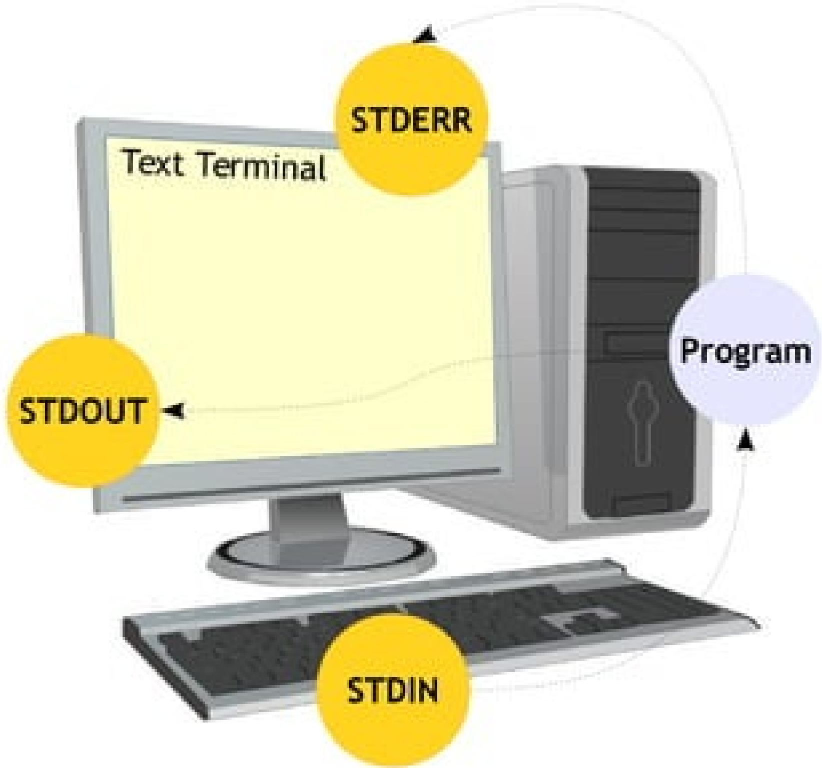
Example

```
#include <stdio.h>

void print_array(int **p)
{
    int i;

    for (i = 0; i < 3; i++)
    {
        printf("%d ", *p[i]);
        printf("at %p\n", p[i]);
    }
}

int main()
{
    int a = 10;
    int b = 20;
    int c = 30;
    int *ptr[3] = {&a, &b, &c};

    print_array(ptr);

    return 0;
}
```



Chapter 4

Standard Input / Output


STDERR
Text Terminal
Program
STDOUT
STDIN

- OS uses 3 file descriptors to provide standard input output functionalities
  - 0 → stdin
  - 1 → stdout
  - 2 → stderr
- These are sometimes referred as “The Standard Streams”
- The IO access example could be
  - stdin → Keyboard, pipes
  - stdout → Console, Files, Pipes
  - stderr → Console, Files, Pipes

Advanced C

Standard I/O - The File Descriptors

- Wait!!, did we see something wrong in previous slide? Both stdout and stderr are similar ?
- If yes why should we have 2 different streams?
- The answer is convenience and urgency.
  - Convenience : the stderr could be used while doing diagnostics. Example - we can separate error messages from low priority informatory messages
  - Urgency : serious error messages shall be displayed on the screen immediately
- So how the C language help us in the standard IO?.

- You need to refer input/output library function

  `#include <stdio.h>`

- When the reference is made with "<*name*>" the search for the files happen in standard path

Advanced C

Standard I/O - Unformatted (Basic)

- Internal binary representation of the data directly between memory and the file
- Basic form of I/O, simple, efficient and compact
- Unformatted I/O is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor
- getchar() and putchar() are two functions part of standard C library
- Some functions like getch(), getche(), putch() are defined in conio.h, which is not a standard C library header and is not supported by the compilers targeting Linux / Unix systems

Example

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

int main()
{
    int ch;

    for ( ; (ch = getchar()) != EOF; )
    {
        putchar(toupper(ch));
    }

    puts("EOF Received");

    return 0;
}
```

- Data is formatted or transformed
- Converts the internal binary representation of the data to ASCII before being stored, manipulated or output
- Portable and human readable, but expensive because of the conversions to be done in between input and output
- The printf() and scanf() functions are examples of formatted output and input

Advanced C

Standard I/O - printf()

Example

```
#include <stdio.h>

int main()
{
    char *a = "Emertxe";

    printf(a);

    return 0;
}
```

- What will be the output of the code on left side?
- Is that syntactically OK?
- Lets understand the printf() prototype
- Please type

  man printf

  on your terminal

Standard I/O - printf()

Prototype

```
int printf(char *format, arg1, arg2, ...);
or
int printf("format string", [variables]);

where format string arguments can be
%[flags][width][.precision][length]specifier

%format_specifier is mandatory and others are optional
```

- Converts, formats, and prints its arguments on the standard output under control of the format
- Returns the number of characters printed

Prototype

```
int printf(char *format, arg1, arg2, ...);
```

What is this!?

Prototype

```
int printf(char *format, arg1, arg2, ...);
```

What is this!?

- Is called as ellipses
- Means, can pass any number of arguments of any type i.e 0 or more
- So how to complete the below example?

Example

What should be wrriten here and how many?

```
int printf("%c %d %f", );
```

Advanced C

Standard I/O - printf()

Example

```
int printf("%c %d %f", arg1, arg2, arg3);
```

Now, how to you decide this!?

Based on the number of format specifiers

- So the number of arguments passed to the printf function should exactly match the number of format specifiers
- So lets go back the code again

Advanced C

Standard I/O - printf()

Example

```
#include <stdio.h>

int main()
{
    char *a = "Emertxe";

    printf(a);

    return 0;
}
```

Isn't this a string?

And strings are nothing but array of characters terminated by null

So what get passed, while passing a array to function?

```
int printf(char *format, arg1, arg2, ...);
```

Isn't this a pointer?

So a pointer hold a address, can be drawn as

format	0x1000
	0x2000

E	m	e	r	t	x	e	\0
0x1000							

So the base address of the array gets passed to the pointer, Hence the output

Note: You will get a warning while compiling the above code.
So this method of passing is not recommended

ΣMERTXE

Advanced C

Standard I/O - printf() - Type Specifiers

Specifiers	Example	Expected Output
%c	printf("%c", 'A')	A
%d %i	printf("%d %i", 10, 10)	10 10
%o	printf("%o", 8)	10
%x %X	printf("%x %X %x", 0xA, 0xA, 10)	a A a
%u	printf("%u", 255)	255
%f %F	printf("%f %F", 2.0, 2.0)	2.000000 2.000000
%e %E	printf("%e %E", 1.2, 1.2)	1.200000e+00 1.200000E+00
%a %A	printf("%a", 123.4) printf("%A", 123.4)	0x1.ed9999999999ap+6 0X1.ED9999999999AP+6
%g %G	printf("%g %G", 1.21, 1.0)	1.21 1
%s	printf("%s", "Hello")	Hello

Advanced C

Standard I/O - printf() - Type Length Specifiers

Length specifier	Example	Expected Output
%[h]X	printf("%hX", 0xFFFFFFFF)	FFFF
%[l]X	printf("%lX", 0xFFFFFFFFl)	FFFFFFFF
%[ll]X	printf("%llX", 0xFFFFFFFFFFFFFFFF)	FFFFFFFFFFFFFFFF
%[L]f	printf("%Lf",1.23456789L)	1.234568

Width	Example	Expected Output
%[x]d	printf("%3d %3d", 1, 1) printf("%3d %3d", 10, 10) printf("%3d %3d", 100, 100)	1 1 10 10 100 100
%[x]s	printf("%10s", "Hello") printf("%20s", "Hello")	Hello Hello
%*[specifier]	printf("%*d", 1, 1) printf("%*d", 2, 1) printf("%*d", 3, 1)	1 1 1

Advanced C

Standard I/O - printf() - Precision

Precision	Example	Expected Output
%[x].[x]d	printf("%3.1d", 1) printf("%3.2d", 1) printf("%3.3d", 1)	1 01 001
%0.[x]f	printf("%0.3f", 1) printf("%0.10f", 1)	1.000 1.0000000000
%[x].[x]s	Printf("%12.8", "Hello World")	Hello Wo

Advanced C

Standard I/O - printf() - Flags

Flag	Example	Expected Output
%[#]x	printf("%#x %#X %#x", 0xA, 0xA, 10) printf("%#o", 8)	0xa 0XA 0xa 010
%[-x]d	printf("%-3d %-3d", 1, 1) printf("%-3d %-3d", 10, 10) printf("%-3d %-3d", 100, 100)	1 1 10 10 100 100
%[ ]3d	printf("% 3d", 100) printf("% 3d", -100)	100 -100

Advanced C

Standard I/O - printf() - Escape Sequence

Escape Sequence	Meaning	Example	Expected Output
\n	New Line	**printf(“Hello World\n”)**	Hello World (With a new line)
\r	Carriage Return	**printf(“Hello\rWorld”)**	World
\t	Tab	**printf(“Hello\tWorld”)**	Hello World
\b	Backspace	**printf(“Hello\bWorld”)**	HellWorld
\v	Vertical Tab	**printf(“Hello\vWorld”)**	Hello World
\f	Form Feed	**printf(“Hello World\f”)**	Might get few extra new line(s)
\e	Escape	**printf(“Hello\eWorld”)**	Hellooorld
\\		**printf(“A\\B\\C”)**	A\B\C
\"		**printf(“\“Hello World\””)**	“Hello World”

EMERTXE

- So in the previous slides we saw some 80% of printf's format string usage.

  What?? Ooh man!!. Now how to print 80%??

Standard I/O - printf() - Example

Example

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string[] = "Hello World";

    printf("%d %c %f %s\n",num1 , ch, num2, string);
    printf("%+05d\n", num1);
    printf("%.2f %.5s\n", num2, string);

    return 0;
}
```

Standard I/O - printf() - Return

Example

```
#include <stdio.h>

int main()
{
    int ret;
    char string[] = "Hello World";

    ret = printf("%s", string);

    printf("The previous printf() printed %d chars\n", ret);

    return 0;
}
```

Standard I/O - sprintf() - Printing to string

Prototype

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

- Similar to printf() but prints to the buffer instead of stdout
- Formats the arguments in arg1, arg2, etc., according to format specifier
- buffer must be big enough to receive the result

Standard I/O - sprintf() - Example

Example

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string1[] = "sprintf() Test";
    char string2[100];

    sprintf(string2, "%d %c %f %s\n",num1 , ch, num2, string1);
    printf("%s", string2);

    return 0;
}
```

Prototype

```
int scanf(char *format, ...);
or
int scanf("string", [variables]);
```

- Reads characters from the standard input, interprets them according to the format specifier, and stores the results through the remaining arguments.
- All most all the format specifiers are similar to printf() except changes in few
- Each of the other argument must be a pointer

- It returns as its value the number of successfully matched and assigned input items.
- On the end of file, EOF is returned. Note that this is different from 0, which means that the next input character does not match the first specification in the format string.
- The next call to scanf() resumes searching immediately after the last character already converted.

Standard I/O - scanf() - Example

Example

```
#include <stdio.h>

int main()
{
    int num1;
    char ch;
    float num2;
    char string[10];

    scanf("%d %c %f %s", &num1 , &ch, &num2, string);
    printf("%d %c %f %s\n", num1 , ch, num2, string);

    return 0;
}
```

Advanced C

Standard I/O - scanf() - Format Specifier

Flag	Examples	Expected Output
%*[specifier]	**scanf("%d%*c%d%*c%d", &h, &m, &s)**	User Input → HH:MM:SS Scanned Input → HHMMSS User Input → 5+4+3 Scanned Input → 543
%[]	**scanf("%[a-z A-Z]", name)**	User Input → Emertxe Scanned Input → Emertxe User Input → Emx123 Scanned Input → Emx
	scanf("%[0-9]", &id)	User Input → 123 Scanned Input → 123 User Input → 123XYZ Scanned Input →123

Standard I/O - scanf() - Return

Example

```
#include <stdio.h>

int main()
{
    int num, int ret;

    printf("The enter a number [is 100 now]: ");
    ret = scanf("%d", &num);

    if (ret != 1)
    {
        printf("Invalid input. The number is still %d\n", num);
        return 1;
    }
    else
    {
        printf("Number is modified with %d\n", num);
    }

    return 0;
}
```

Standard I/O - sscanf() - Reading from string

Prototype

```
int sscanf(char *string, char *format, ...);
```

- Similar to sscanf() but read from string instead of stdin
- Formats the arguments in arg1, arg2, etc., according to format

Standard I/O - sscanf() - Example

Example

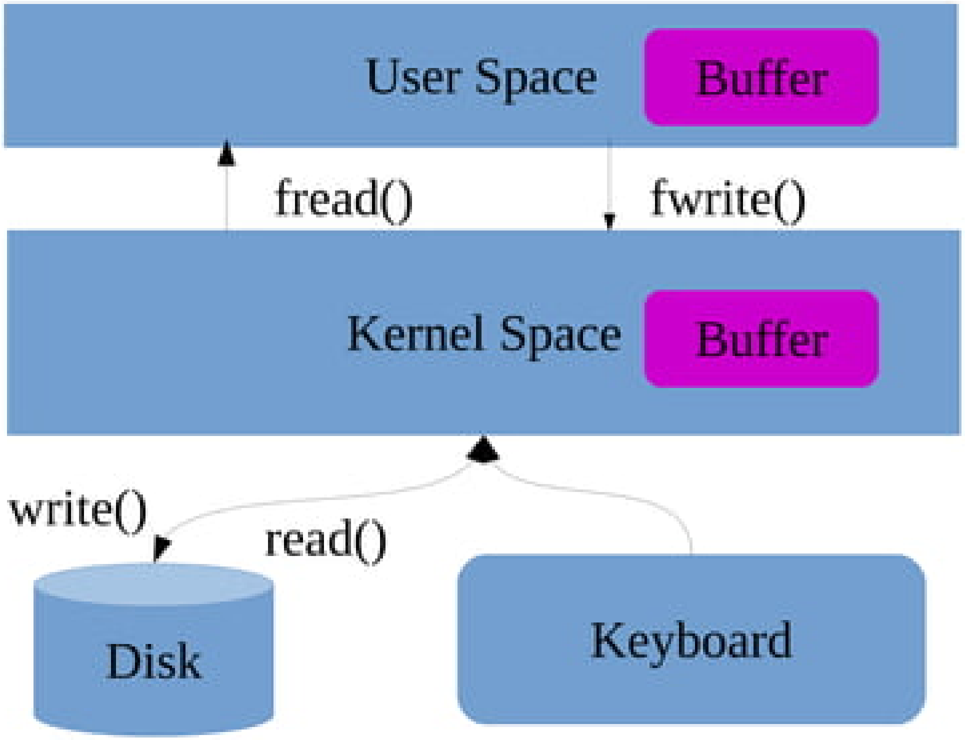
```
#include <stdio.h>

int main()
{
    int age;
    char array_1[10];
    char array_2[10];

//  sscanf("I am 30 years old", "%s %s %d", array_1, array2, &age);
    sscanf("I am 30 years old", "%*s %*s %d", &age);
    printf("OK you are %d years old\n", age);

    return 0;
}
```

Advanced C

Standard I/O - Buffering

Advanced C

Standard I/O - Buffering



- Buffer is a consecutive memory block used for temporary storage
- The standard I/O, except stderr, uses the buffer to store the data
- The input would be stored in the user space before providing it to your process
- The main idea is to reduce the context switching between user and kernel space which leads to better I/O efficiency

- For example, consider a process that writes one character at a time to a file. This is obviously inefficient: Each write operation corresponds to a write() system call
- Similarly, imagine a process that reads one character at a time from a file into memory!! This leads to read() system call
- I/O buffers are temporary memory area(s) to moderate the number of transfers in/out of memory by assembling data into batches

- The output buffer get flushed out due to the following reasons
  - Normal Program Termination
  - '\n' in a printf
  - Read
  - fflush call
  - Buffer Full

Chapter 6

Program Segments & Storage Classes

Advanced C

Memory Segments

Linux OS

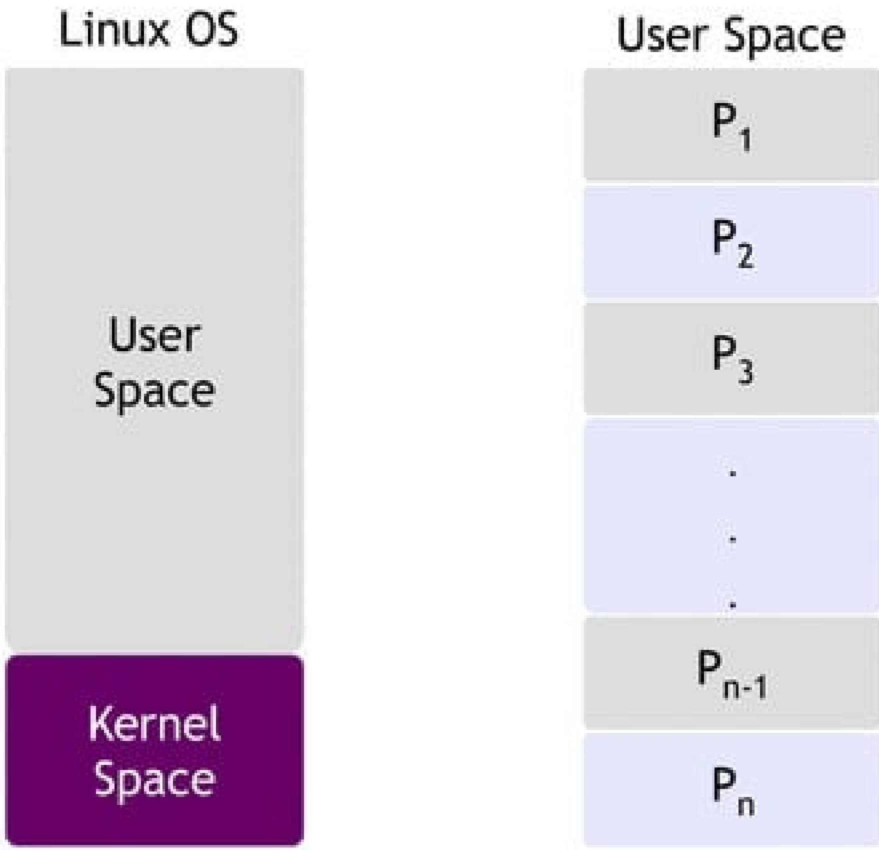
The Linux OS is divided into two major sections

- User Space
- Kernel Space

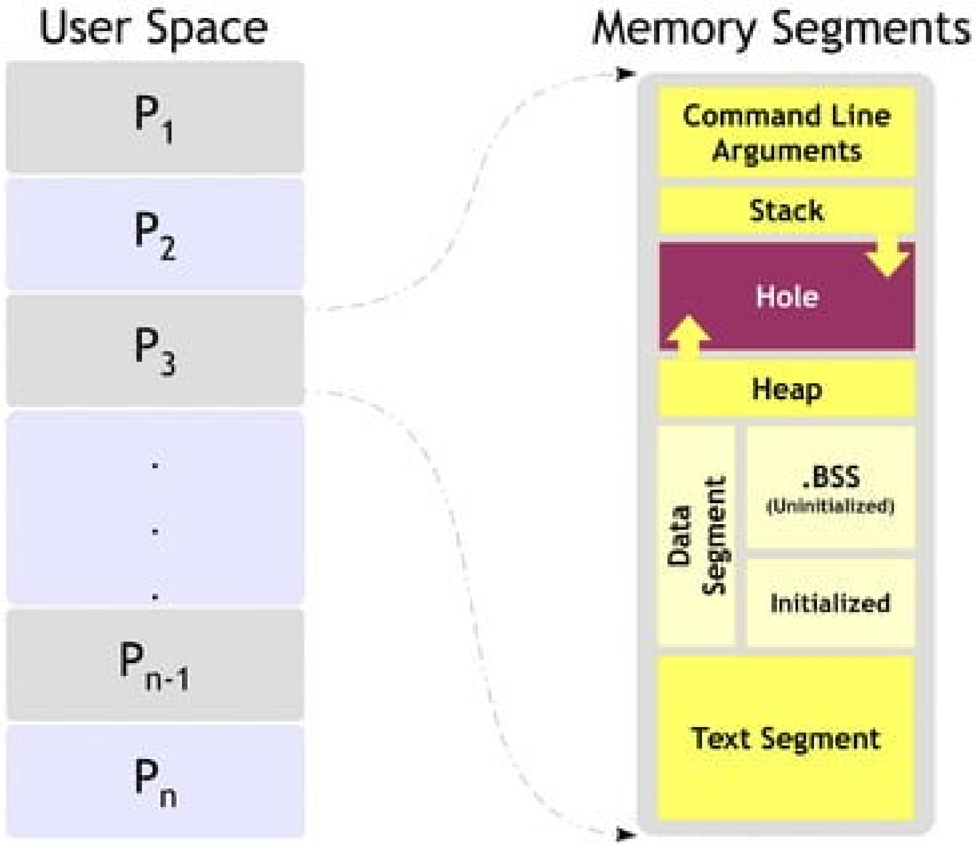
The user programs cannot access the kernel space. If done will lead to segmentation violation

Let us concentrate on the user space section here

Advanced C
Memory Segments



- The User space contains many processes
- Every process will be scheduled by the kernel
- Each process will have its memory layout discussed in next slide



The memory segment of a program contains four major areas.

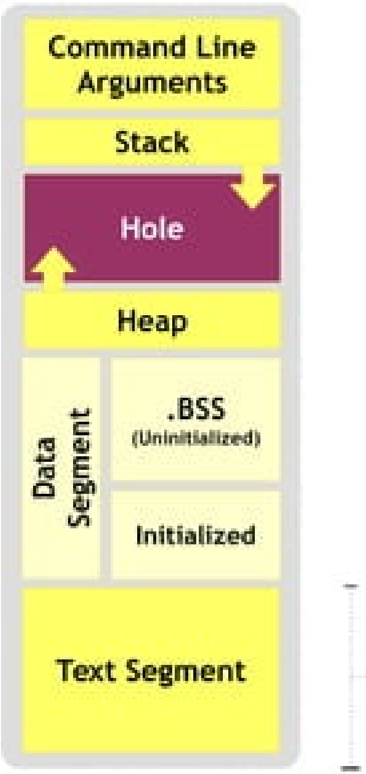
- Text Segment
- Stack
- Data Segment
- Heap

.BSS - Block Started by Symbol

Advanced C

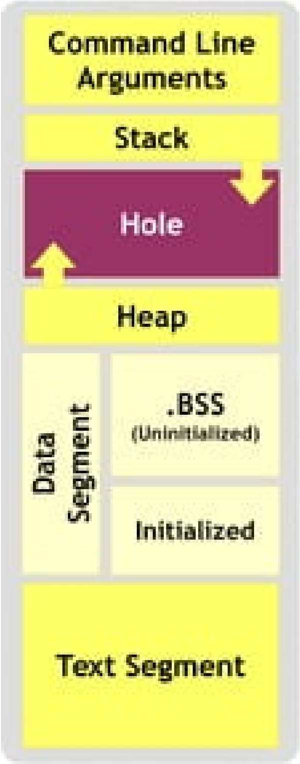
Memory Segments - Text Segment

Memory Segments



- Also referred as Code Segment
- Holds one of the section of program in object file or memory
- In memory, this is place below the heap or stack to prevent getting over written
- Is a read only section and size is fixed

Memory Segments

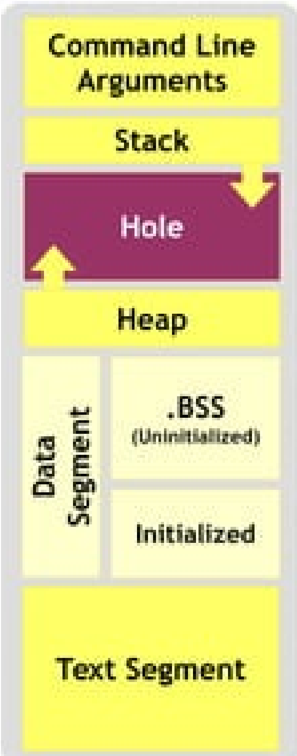


- Contains 2 sections as initialized and uninitialized data segments
- Initialized section is generally called as Data Segment
- Uninitialized section is referred as BSS (Block Started by Symbol) usually filled with 0s

Advanced C

Memory Segments - Data Segment

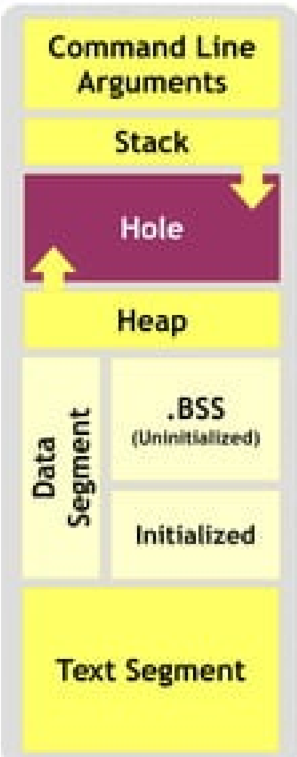
Memory Segments



Dynamic memory allocation takes place here

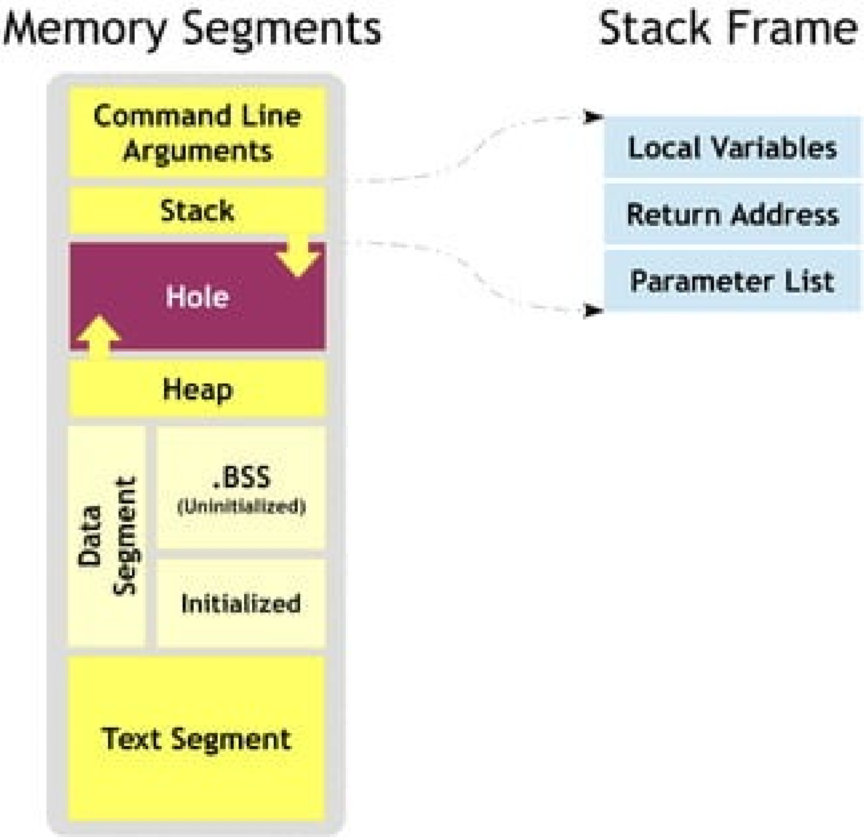
Begins at the end of BSS and grows upward from there

Memory Segments



- Adjoins the heap area and grow in opposite area of heap when stack and heap pointer meet (Memory Exhausted)
- Typically loaded at the higher part of memory
- A "stack pointer" register tracks the top of the stack; it is adjusted each time a value is "pushed" onto the stack
- The set of values pushed for one function call is termed a "stack frame"

Advanced C
Memory Segments - Stack Segment



A stack frame contain at least of a return address

```c
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int n1, int n2)
{
    int s = 0;

    s = n1 + n2;

    return s;
}
```

Stack Frame

Stack Frame	
num1 = 10 num2 = 20 sum = 0	main()
Return Address to the caller	
s = 0	add_numbers()
Return Address to the main()	
n1 = 10 n2 = 20	

Advanced C

Memory Segments - Runtime

- 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()

- 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 (for pointers) before the program begins to execute

- The Stack: The stack is a collection of stack frames. 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

Advanced C

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

Example

```c
#include <stdio.h>

int global_1;
int global_2 = 10;

static int global_3;
static int global_4 = 10;

int main()
{
    int local_1;
    static int local_1;
    static int local_2 = 20;

    register int iter;
    for (iter = 0; iter < 0; iter++)
    {
        /* Do Something */
    }

    return 0;
}
```

Variable	Storage Class	Memory Allocation
global_1	No	.BSS
global_2	No	Initialized data segment
global_3	Static global	.BSS
global_4	Static global	Initialized data segment
local_1	auto	stack
local_2	Static local	.BSS
local_3	Static local	Initialized data segment
iter	Register	Registers

Advanced C

Declaration

```
extern int num1;
extern int num1;

int main();

int main()
{
    int num1, num2;
    char short_opt;

    ...
}
```

Declaration specifies type to the variables

Its like an announcement and hence can be made 1 or more times

Declaration about num1

Declaration about num1 yet again!!

Declaration about main function

Advanced C

Storage Classes - extern

file1.c

```c
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
        func_2();
    }

    return 0;
}
```

file2.c

```c
#include <stdio.h>

extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

file3.c

```c
#include <stdio.h>

extern int num;

int func_2()
{
    printf("num is %d from file3\n", num);

    return 0;
}
```

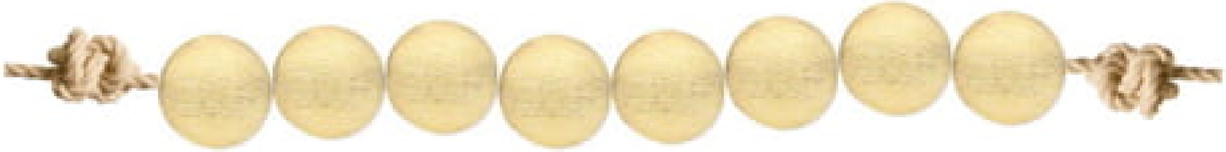
ΣMERTXE

Chapter 6

Strings

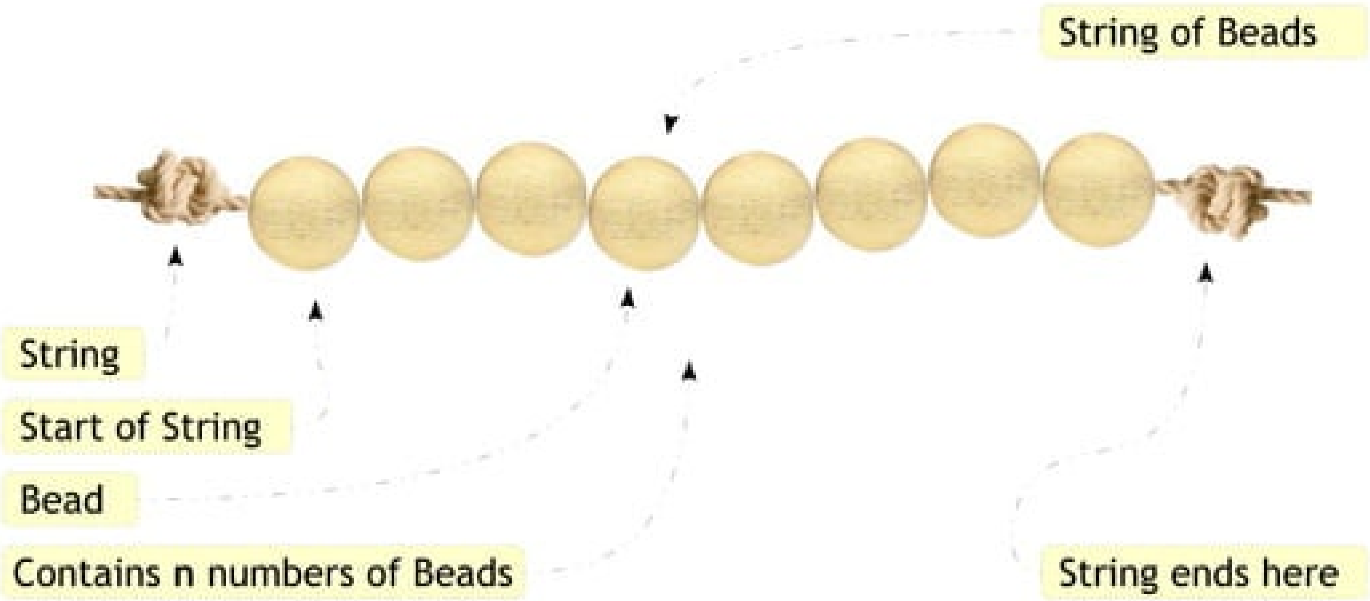
Advanced C

S______s - Fill in the blanks please ;)



A set of things tied or threaded together on a thin cord

Source: Google



- Contiguous sequence of characters
- Stores printable ASCII characters and its extensions
- End of the string is marked with a special character, the null character '\0'
- '\0' is implicit in strings enclosed with “”
- Example

“You know, now this is what a string is!”

Advanced C

Strings - Initializations

Examples

`char char_array[5] = {'H', 'E', 'L', 'L', 'O'};`	Character Array
`char str[6] = {'H', 'E', 'L', 'L', 'O', '\0'};`	String
`char str[] = {'H', 'E', 'L', 'L', 'O', '\0'};`	Valid
`char str[6] = {"H", "E", "L", "L", "O"};`	Invalid
`char str[6] = {"H" "E" "L" "L" "O"};`	Valid
`char str[6] = {"HELLO"};`	Valid
`char str[6] = "HELLO";`	Valid
`char str[] = "HELLO";`	Valid
`char *str = "HELLO";`	Valid

ΣMERTXE

Examples

```c
#include <stdio.h>

int main()
{
    char char_array_1[5] = {'H', 'E', 'L', 'L', O'};
    char char_array_2[] = "Hello";

    sizeof(char_array_1);
    sizeof(char_array_2);

    return 0;
}
```

The size of the array
Is calculated So,

5, 6

```c
int main()
{
    char *str = "Hello";

    sizeof(str);

    return 0;
}
```

The size of pointer is
always constant so,

4 (32 Bit Sys)

Example

```
#include <stdio.h>

int main()
{
    if (sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
    {
        printf("WoW\n");
    }
    else
    {
        printf("HuH\n");
    }


    return 0;
}
```

Examples

```
char str1[6] = "Hello";
char str2[6];

str2 = "World";
```

Not possible to assign a string to a array since its a constant pointer

```
char *str3 = "Hello";
char *str4;

str4 = "World";
```

Possible to assign a string to a pointer since its variable

```
Str1[0] = 'h';
```

Valid. str1 contains "hello"

```
Str3[0] = 'w';
```

Invalid. str3 might be stored in read only section. Undefined behaviour

Example

```
#include <stdio.h>

int main()
{
    char *str1 = "Hello";
    char *str2 = "Hello";

    if (str1 == str2)
    {
        printf("Hoo. They share same space\n");
    }
    else
    {
        printf("No. They are in different space\n");
    }

    return 0;
}
```

Puropse	Prototype	Return Values
Length	**size_t strlen(const char *str)**	String Length
Compare	**int strcmp(const char *str1, const char *str2)**	str1 < str2 → < 0 str1 > str2 → > 0 str1 = str2 → = 0
Copy	**char *strcpy(char *dest, const char *src)**	Pointer to dest
Check String	**char *strstr(const char *haystack, const char *needle)**	Pointer to the beginning of substring
Check Character	**char *strchr(const char *s, int c)**	Pointer to the matched char else NULL
Merge	**char *strcat(char *dest, const char *src)**	Pointer to dest

- Can we copy 2 strings like, str1 = str2?
- Why don't we pass the size of the string to string functions?
- What will happen if you overwrite the '\0' of string? Will you still call it a string?
- What is the difference between char *s and char s[]?

Advanced C

Strings - DIY

- WAP to calculate length of the string
- WAP to reverse a string
- WAP to check a given string is palindrome or not
- WAP to compare two strings
- WAP to compare string2 with string1 up to n characters
- WAP to copy a strings
- WAP to concatenate two strings

"strlen, strcpy, strcmp, strcat, strstr, strtok"

- WAP to print user information -
  - Read : Name, Age, ID, Mobile number
  - Print the information on monitor
  - Print error "Invalid Mobile Number" if length of mobile number is not 10
- WAP to read user name and password and compare with stored fields. Present a puzzle to fill in the banks
- Use strtok to separate words from string "www.emertxe.com/bangalore"

Chapter 7

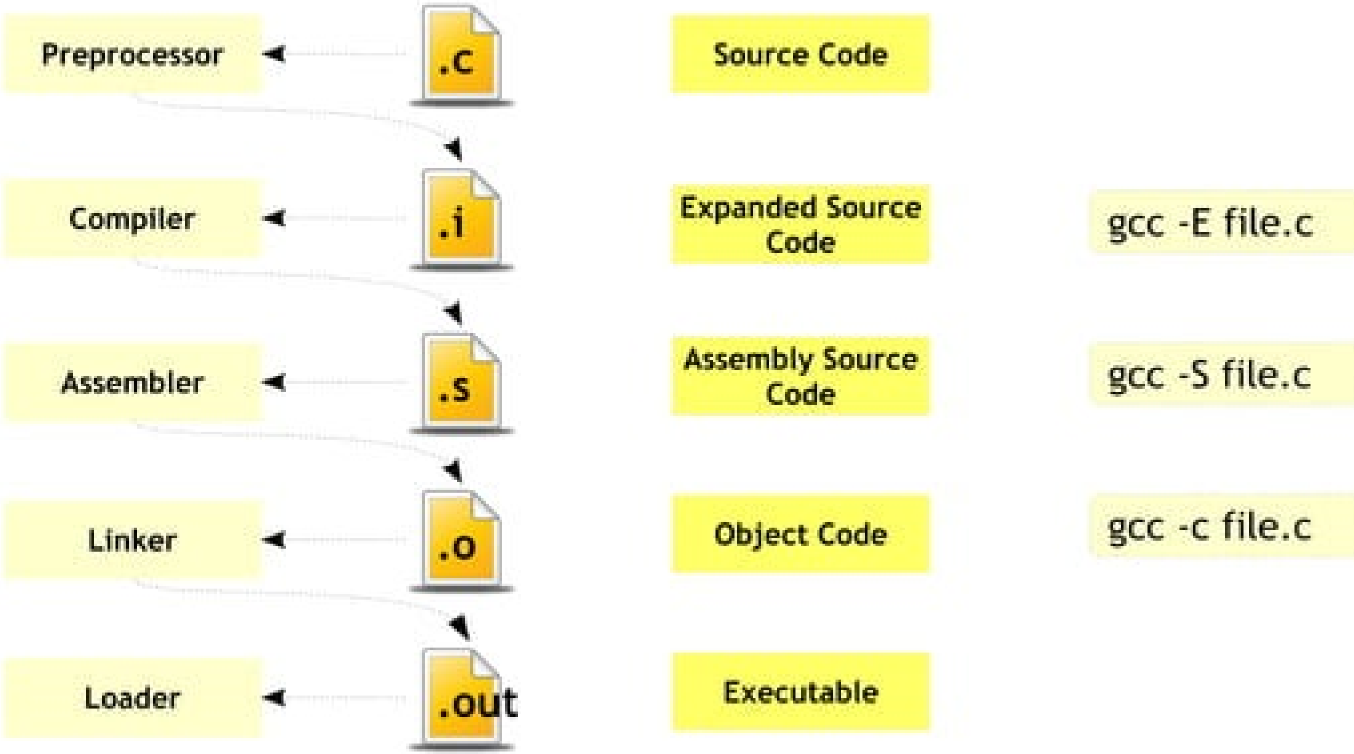
Preprocessor

- One of the step performed before compilation
- Is a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- Instructions given to preprocessor are called preprocessor directives and they begin with “#” symbol
- Few advantages of using preprocessor directives would be,
  - Easy Development
  - Readability
  - Portability

- Before we proceed with preprocessor directive let's try to understand the stages involved in compilation
- Some major steps involved in compilation are
  - Preprocessing (Textual replacement)
  - Compilation (Syntax and Semantic rules checking)
  - Assembly (Generate object file(s))
  - Linking (Resolve linkages)
- The next slide provide the flow of these stages

Advanced C

Preprocessor - Compilation Stages



gcc -save-temps file.c would generate all intermediate files

- #include
- #define
- #undef
- #ifdef
- #ifndef
- #else
- #endif
- #if
- #else
- #elif
- #error
- #warning
- #line
- #pragma
- #
- ##

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- Has to be included using C preprocessing directive '#include'
- Header files serve two purposes.
  - Declare the interfaces to parts of the operating system by supplying the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program.

Advanced C

Preprocessor – Header Files vs Source Files

VS

- Declarations
- Sharable/reusable
  - #defines
  - Datatypes
- Used by more than 1 file

- Function and variable definitions
- Non sharable/reusable
  - #defines
  - Datatypes

Advanced C

Preprocessor - Header Files - Syntax

Syntax

```
#include <file.h>
```

- System header files
- It searches for a file named ***file*** in a standard list of system directories

Syntax

```
#include "file.h"
```

- Local (your) header files
- It searches for a file named ***file*** first in the directory containing the current file, then in the quote directories and then the same directories used for <file>

Advanced C
Preprocessor - Header Files - Operation

file2.c

```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

file2.h

```
char *test(void);
```

file1.c

```
int num;

#include "file2.h"

int main()
{
    puts(test());

    return 0;
}
```

```
int num;

char *test(void);

int main()
{
    puts(test());

    return 0;
}
```

Advanced C

Preprocessor - Header Files - Search Path

- On a normal Unix system GCC by default will look for headers requested with #include <file> in:
  - /usr/local/include
  - libdir/gcc/target/version/include
  - /usr/target/include
  - /usr/include
- You can add to this list with the -I <*dir*> command-line option

"gcc -print-prog-name=cc1 -v" would search the path info

- If a header file happens to be included twice, the compiler will process its contents twice causing an error
- E.g. when the compiler sees the same structure definition twice
- This can be avoided like

Example

```
#ifndef NAME
#define NAME

/* The entire file is protected */

#endif
```

- An object-like macro is a simple identifier which will be replaced by a code fragment
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants

Syntax

```
#define SYMBOLIC_NAME     CONSTANTS
```

Example

```
#define BUFFER_SIZE       1024
```

- Function-like macros can take arguments, just like true functions
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like

Example

```
#include <stdio.h>

#define SET_BIT(num, pos)     (num | (1 << pos))

int main()
{
    SET_BIT(0, 2);

    return 0;
}
```

- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
- When the macro is expanded, however, it will all come out on one line

Example

```c
#include <stdio.h>

#define SWAP(a, b)              \
{                               \
    int temp = a;               \
    a = b;                      \
    b = temp;                   \
}

int main()
{
    int num1 = 10, num1= 20;

    SWAP(num1, num2);

    printf("%d %d\n", num1, num2);

    return 0;
}
```

Advanced C

Preprocessor – Macro – Stringification

Example

```
#include <stdio.h>

#define WARN_IF(EXP)                                    \
do                                                      \
{                                                       \
    x--;                                                \
    if (EXP)                                            \
    {                                                   \
        fprintf(stderr, "Warning: " #EXP "\n");         \
    }                                                   \
} while (x);

int main()
{
    int x = 5;

    WARN_IF(x == 0);

    return 0;
}
```

- You can convert a macro argument into a string constant by adding #

Advanced C

Preprocessor - Macro - Standard Predefined

- Several object-like macros are predefined; you use them without supplying their definitions.
- Standard are specified by the relevant language standards, so they are available with all compilers that implement those standards

Example

```
#include <stdio.h>

int main()
{

    printf("Program: \"%s\" ", __FILE__);
    printf("was compiled on %s at %s. ", __DATE__, __TIME__);
    printf("This print is from Function: \"%s\" at line %d\n", __func__, __LINE__);

    return 0;
}
```

- A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler
- Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator
- A conditional in the C preprocessor resembles in some ways an if statement in C with the only diffrence being it happens in compile time
- Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Advanced C

Preprocessor - Conditional Compilation

- There are three general reasons to use a conditional.
  - A program may need to use different code depending on the machine or operating system it is to run on
  - You may want to be able to compile the same source file into two different programs, like one for debug and other as final
  - A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference

Syntax

```
#ifdef MACRO

/* Controlled Text */

#endif
```

Syntax

```
#if defined (ERROR) && (WARNING)

/* Controlled Text */

#endif
```

Advanced C

Preprocessor - Conditional Compilation - if

Syntax

```
#if expression

/* Controlled Text */

#endif
```

Advanced C

Preprocessor - Conditional Compilation - else

Syntax

```
#if expression

/* Controlled Text if true */

#else

/* Controlled Text if false */

#endif
```

Advanced C

Preprocessor - Conditional Compilation - elif

Syntax

```
#if DEBUG_LEVEL == 1

/* Controlled Text*/

#elif  DEBUG_LEVEL == 2

/* Controlled Text */

#else

/* Controlled Text */

#endif
```

Syntax

```
#if 0

/* Deleted code while compiling */
/* Can be used for nested code comments */
/* Avoid for general comments */

#endif
```

- The directive '#error' causes the preprocessor to report a fatal error. The tokens forming the rest of the line following '#error' are used as the error message
- The directive '#warning' is like '#error', but causes the preprocessor to issue a warning and continue preprocessing. The tokens following '#warning' are used as the warning message

Chapter 8

User Defined Datatype

Advanced C

UDDs – Structures (Composite data types)

- Sometimes it becomes tough to build a whole software that works only with integers, floating values, and characters.
- In circumstances such as these, you can create your own data types which are based on the standard ones
- There are some mechanisms for doing this in C:
  - Structures
  - Unions
  - Typedef
  - Enums
- Hoo!!, let's not forget our old friend _r_a_ which a user defined data type too!!.

Composite (or compound) data type :

- Any data type which can be constructed from primitive data types and other composite types
- It is sometimes called a structure or aggregate data type
- Primitives types – int, char, float, double

Syntax

```
struct StructureName
{
    /* Group of data types */
};
```

- If we consider the Student as an example, The admin should have at least some important data like name, ID and address.

- So if we create a structure of the above requirement, it would look like,

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};
```

Advanced C

UDDs - Structures - Declaration and definition

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    return 0;
}
```

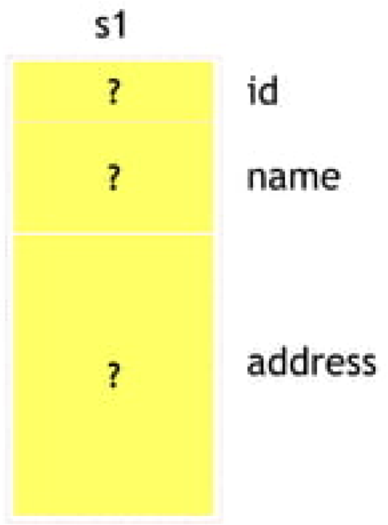
- Name of the **datatype.** Note it's **struct Student** and not Student
- Are called as **fields** or **members** of of the structure
- Declaration **ends** here
- The memory is not yet allocated!!
- **s1** is a **variable** of **type** **struct Student**
- The memory is allocated now

EMERTXE

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    return 0;
}
```

- What does s1 contain?
- How can we draw it's memory layout?



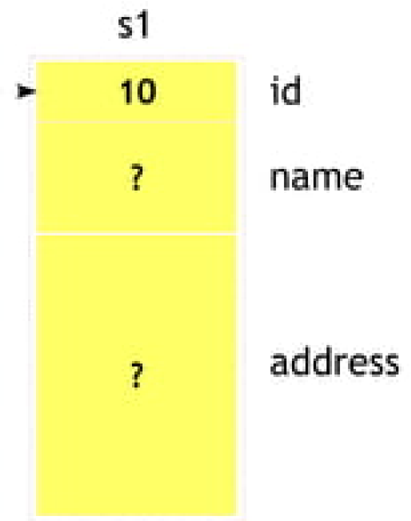
Advanced C

UDDs - Structures - Access

Example

```c
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    s1.id = 10;

    return 0;
}
```

- How to write into id now?
- It's by using "." (Dot) operator (member access operator)



- Now please assign the name for s1

Advanced C

UDDs - Structures - Initialization

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    return 0;
}
```

Advanced C

UDDs - Structures - Copy

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};
    struct Student s2;

   s2 = s1;

    return 0;
}
```

Structure name does not represent its address. (No correlation with arrays)

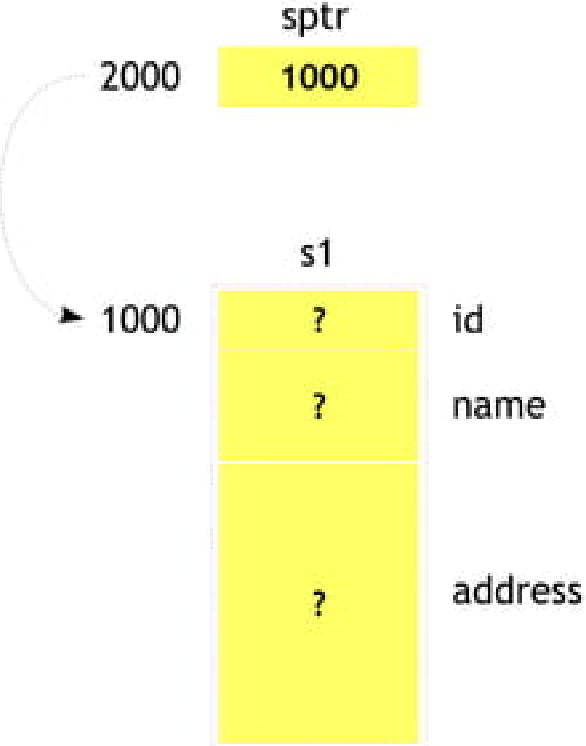
Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    printf("Struture starts at %p\n", &s1);
    printf("Member id is at %p\n", &s1.id);
    printf("Member name is at %p\n", s1.name);
    printf("Member address is at %p\n", s1.address);

    return 0;
}
```

- Pointers!!!. Not again ;). Fine don't worry, not a big deal
- But do you any idea how to create it?
- Will it be different from defining them like in other data types?

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    return 0;
}
```

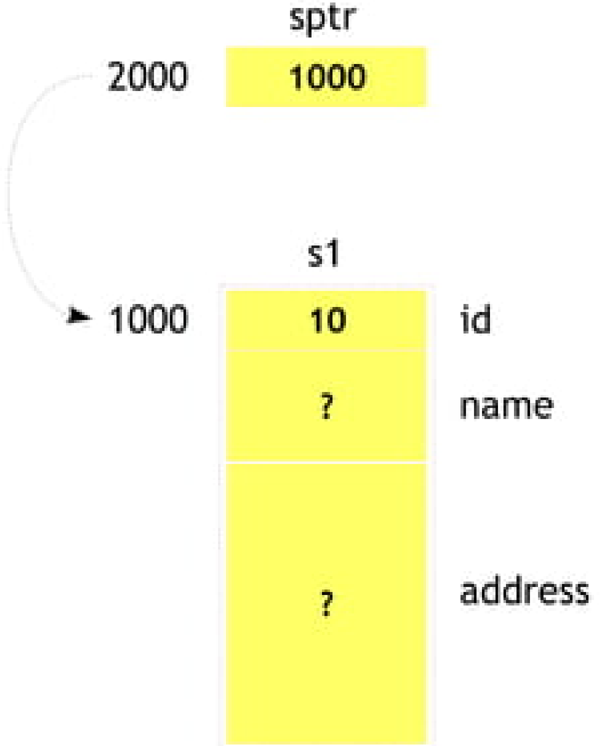


Advanced C

UDDs - Structures - Pointer - Access

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    (*sptr).id = 10;

    return 0;
}
```

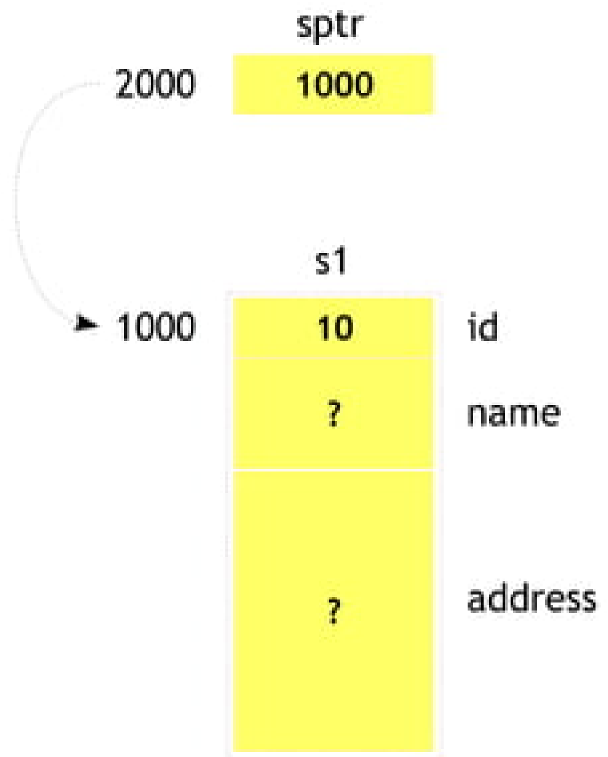


Advanced C

UDDs - Structures - Pointer - Access - Arrow

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    sptr->id = 10;

    return 0;
}
```



Note: we can access the structure pointer as seen in the previous slide. The Arrow operator is just convinence and frequently used

- The structures can be passed as parameter and can be returned from a function
- This happens just like normal datatypes.
- The parameter passing can have two methods again as normal
  - Pass by value
  - Pass by reference

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student s)
{
    s.id = 10;
}

int main()
{
    struct Student s1;

    data(s1);

    return 0;
}
```

Not recommended on larger structures

Advanced C

UDDs - Structures - Functions - Pass by Reference

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};

void data(struct Student *s)
{
    s->id = 10;
}

int main()
{
    struct Student s1;

    data(&s1);

    return 0;
}
```

Recommended on larger structures

Advanced C

UDDs - Structures - Functions - Return

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};

struct Student data(struct Student s)
{
    s.id = 10;

    return s;
}

int main()
{
    struct Student s1;

    s1 = data(s1);

    return 0;
}
```

- Adding of few extra useless bytes (in fact skip address) in between the address of the members are called structure padding.
- What!!?, wasting extra bytes!!, Why?
- This is done for Data Alignment.
- Now!, what is data alignment and why did this issue suddenly arise?
- No its is not sudden, it is something the compiler would internally while allocating memory.
- So let's understand data alignment in next few slides

- A way the data is arranged and accessed in computer memory.
- When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (4 bytes in 32 bit system) or larger.
- The main idea is to increase the efficiency of the CPU, while handling the data, by arranging at a memory address equal to some multiple of the word size
- So, Data alignment is an important issue for all programmers who directly use memory.

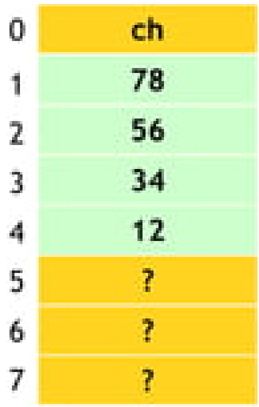
- If you don't understand data and its address alignment issues in your software, the following scenarios, in increasing order of severity, are all possible:
  - Your software will run slower.
  - Your application will lock up.
  - Your operating system will crash.
  - Your software will silently fail, yielding incorrect results.

Advanced C

Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

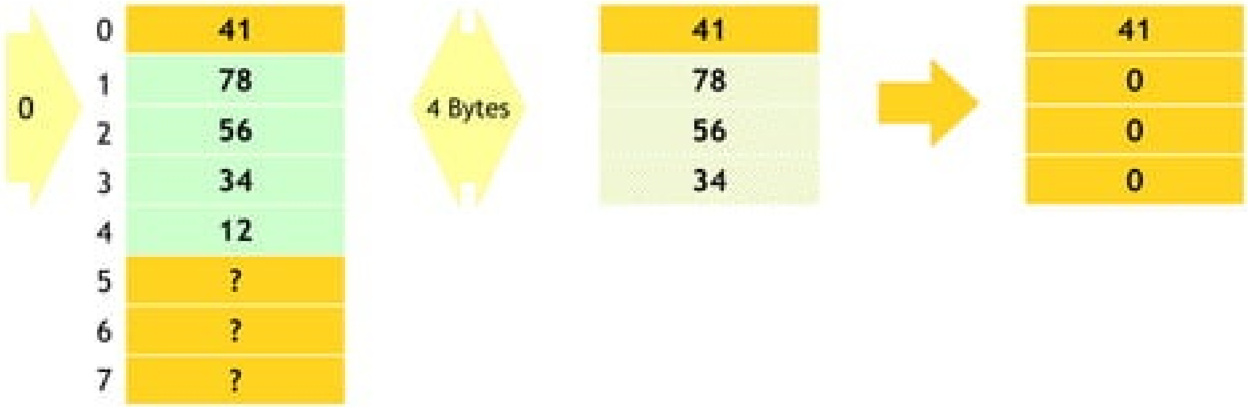


- Lets consider the code as given
- The memory allocation we expect would be like shown in figure
- So lets see how the CPU tries to access these data in next slides

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the character by the CPU will be like shown below



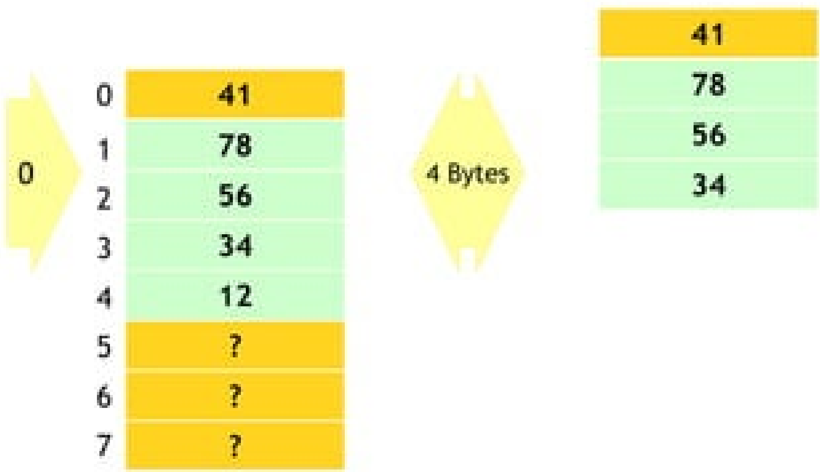
Advanced C

Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below

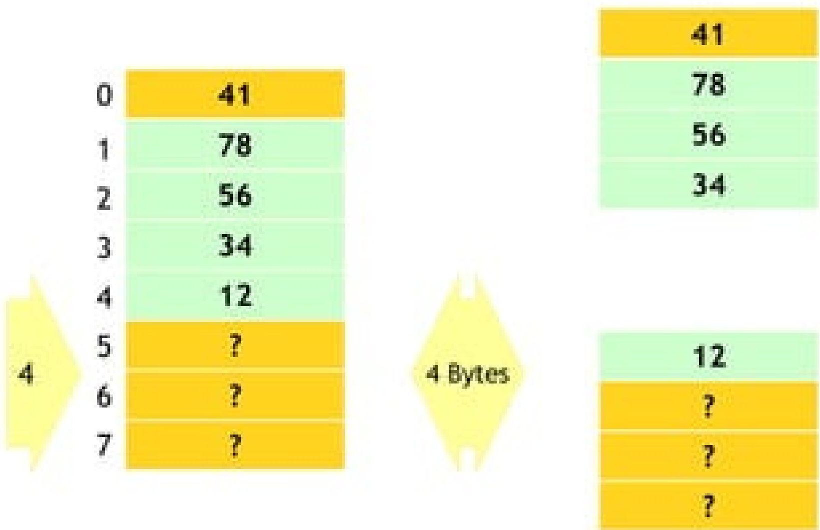


Advanced C
Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



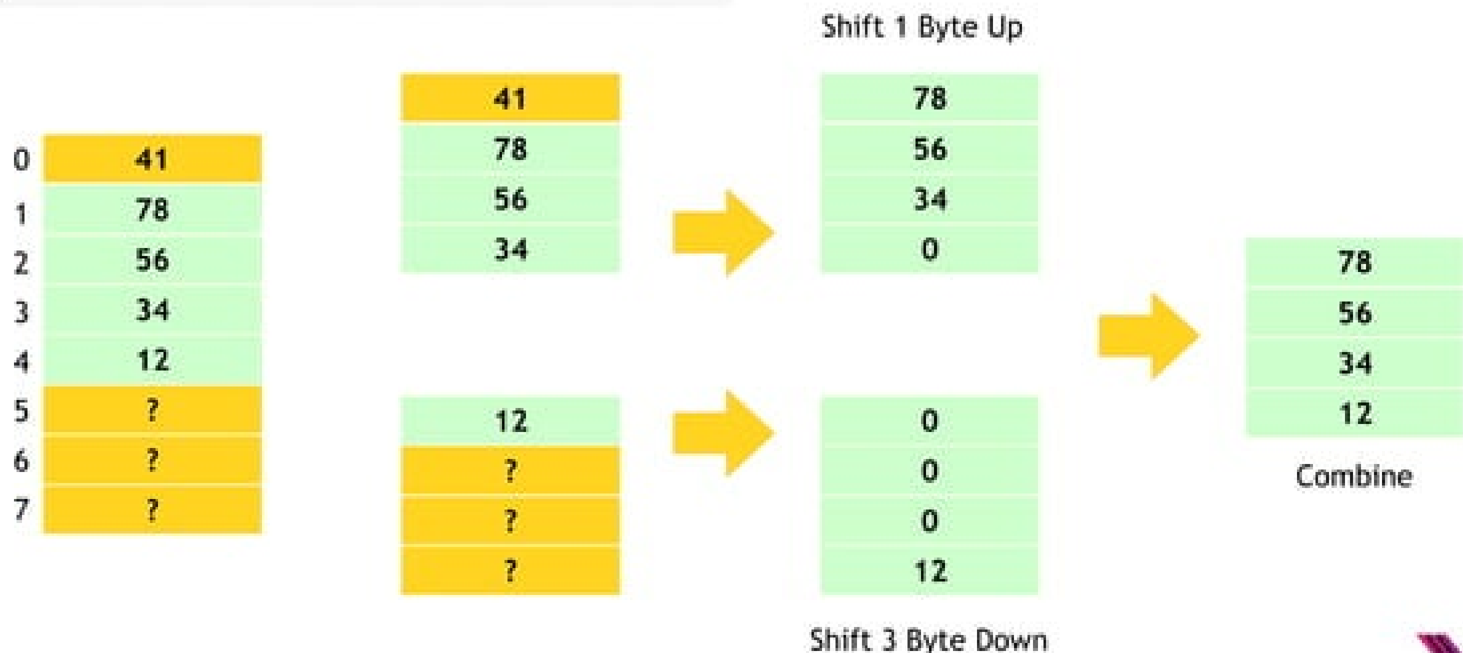
Advanced C

Data Alignment

Example

```c
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



Advanced C

UDTs - Structures - Data Alignment - Padding

- Because of the data alignment issue, structures uses padding between its members if the don't fall under even address.
- So if we consider the following structure the memory allocation will be like shown in below figure

Example

```
struct Test
{
    char ch1;
    int num;
    char ch2;
}
```

0	ch1
1	pad
2	pad
3	pad
4	num
5	num
6	num
7	num
8	ch2
9	pad
A	pad
B	pad

- You can instruct the compiler to modify the default padding behavior using #pragma pack directive

Example

```c
#pragma pack(1)

struct main()
{
    char ch1;
    int num;
    char ch2;
}
```

0	ch1
1	num
2	num
3	num
4	num
5	ch2

- The compiler generally gives the memory allocation in multiples of bytes, like 1, 2, 4 etc.,
- What if we want to have freedom of having getting allocations in bits?!.
- This can be achieved with bit fields.
- But not that
  - The minimum memory allocation for a bit field member would be a byte that can be broken in max of 8 bits
  - The maximum number of bits assigned to a member would depend on the length modifier
  - The default size is equal to word size

Example

```
struct Nibble
{
    unsigned char upper    : 4;
    unsigned char lower    : 4;
};
```

- The above structure divides a char into two nibbles
- We can access these nibbles independently

Advanced C

UDDs - Unions

- Like structures, unions may have different members with different data types.
- The major difference is, the structure members get different memory allocation, and in case of unions there will be single memory allocation for the biggest data type

Example

```
union DummyVar
{
    char option;
    int id;
    double height;
};
```

- The above union will get the size allocated for the type double
- The size of the union will be 8 bytes.
- All members will be using the same space when accessed
- The value the union contain would be the latest update
- So as summary a single varible can store different type of data as required

- Typedef is used to create a new name to the existing types.
- K&R states that there are two reasons for using a typedef.
  - First, it provides a means to make a program more portable. Instead of having to change a type everywhere it appears throughout the program's source files, only a single typedef statement needs to be changed.
  - Second, a typedef can make a complex definition or declaration easier to understand.

Advanced C

UDTs - Enums

- Set of named integral values

Examples

```
enum Boolean
{
    e_false,
    e_true
};
```

```
typedef enum
{
    e_red = 1,
    e_blue = 4,
    e_green
} Color;
```

```
typedef enum
{
    red,
    blue
} Color;

int blue;
```

- The above example has two members with its values starting from 0. i.e, e_false = 0 and e_true = 1.
- The member values can be explicitly initialized
- The is no constraint in values, it can be in any order and same values can be repeated
- Enums does not have name space of its own, so we cannot have same name used again in the same scope.

Chapter 5

File Input / Output

- Sequence of bytes
- Could be regular or binary file
- Why?
  - Persistent storage
  - Theoretically unlimited size
  - Flexibility of storing any type data

File Input / Output - Via Redirection

- General way for feeding and getting the output is using standard input (keyboard) and output (screen)
- By using redirection we can achieve it with files i.e

```
./a.out < input_file > output_file
```

- The above line feed the input from input_file and output to output_file
- The above might look useful, but its the part of the OS and the C doesn't work this way
- C has a general mechanism for reading and writing files, which is more flexible than redirection

- C abstracts all file operations into operations on streams of bytes, which may be "input streams" or "output streams"
- No direct support for random-access data files
- To read from a record in the middle of a file, the programmer must create a stream, seek to the middle of the file, and then read bytes in sequence from the stream
- Let's discuss some commonly used file I/O functions

- stdio.h is used for file I/O library functions
- The data type for file operation generally is

Type
`FILE *fp;`

- FILE pointer, which will let the program keep track of the file being accessed
- Operations on the files can be
  - Open
  - File operations
  - Close

File Input / Output - Functions

Prototype

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *filename);

Where mode are:
r  - open for reading
w  - open for writing (file need not exist)
a  - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)
```

Example

```
#include <stdio.h>

int main()
{
    FILE *fp;

    fp = fopen("test.txt", "r");
    fclose(fp);

    return 0;
}
```

Example

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

int main()
{
    FILE *input_fp;

    input_fp = fopen("text.txt", "r");

    if (input_fp == NULL)
    {
        exit(1);
    }

    fclose(input_fp);

    return 0;
}
```

Prototype

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

Example

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

int main()
{
    FILE *input_fp;
    char ch;

    input_fp = fopen("text.txt", "r");

    if (input_fp == NULL)
    {
        exit(1);
    }

    while ((ch = fgetc(input_fp)) != EOF)
    {
        fputc(ch, stdout);
    }

    fclose(input_fp);

    return 0;
}
```

Advanced C

File Input / Output - Functions

Prototype

```
int fprintf(char *string, char *format, ...);
```

Example

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

int main()
{
    FILE *input_fp;

    input_fp = fopen("text.txt", "r");

    if (input_fp == NULL)
    {
        fprintf(stderr, "Can't open input file text.txt!\n");
        exit(1);
    }

    fclose(input_fp);

    return 0;
}
```

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

Emertxe Information Technologies,

No-1, 9th Cross, 5th Main,
Jayamahal Extension,
Bangalore, Karnataka 560046

T: +91 80 6562 9666
E: training@emertxe.com

https://www.facebook.com/Emertxe

https://twitter.com/EmertxeTweet

https://www.slideshare.net/EmertxeSlides

ΣMERTXE

Thank You