Come let's see how deep it is!!

Team Emertxe



Chapter 1



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





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!!



Language - What?



- A stylized communication technique
- The more varied vocabulary it has, the more expressive it becomes.
- Grammar makes meaningful communication
- Similarly, a programming language is a stylized communication technique intended to be used for controlling the behavior of a machine (often a computer)
- Like the natural languages, programming languages too, have syntactic rules (to form words) and semantic rules (to form sentences), used to define the meaning



Brief History



- Prior to C, most of the computer languages (such as Algol) were academic oriented, unrealistic and were generally defined by committees.
- Since such languages were designed having application domain in mind, they could not take the advantages of the underlying hardware and if done, were not portable or efficient under other systems.
- It was thought that a high-level language could never achieve the efficiency of assembly language

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



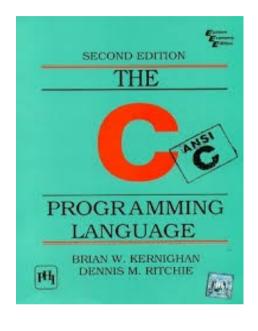
Brief History



- It was a revolutionary language and shook the computer world with its might. With just 32 keywords, C established itself in a very wide base of applications.
- It has lineage starting from CPL, (Combined Programming Language) a never implemented language
- Martin Richards implemented BCPL as a modified version of CPL. Ken Thompson further refined BCPL to a language named as B
- Later Dennis M. Ritchie added types to B and created a language, what we have as C, for rewriting the UNIX operating system



Standard



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



Important Characteristics



- 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



Important Characteristics



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



Keywords



- 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





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 8bytes in memory, Supports 0, ∞ (-/+) and NaN



Keywords - Data Types Modifiers



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



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



Keywords - Loops



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



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

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.



Keywords - Jump



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



Keywords - Derived



struct unions

- struct keyword provides support for aggregate types
- 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



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



Keywords - Others



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



Anatomy of a Simple C Code



```
File Header
#include <stdio.h> <------
                         Preprocessor Directive
int main() ◄··-----
                         The start of program
  Comment
  printf("Hello world\n"); <------</pre>
                         Statement
  Program Termination
```



Advanced C Compilation



- 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>



Execution



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

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

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

This should the expected result on your system



Introduction

How should I proceed with my Application?



- Never jump to implementation. Why?
 - You might 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...



SDLC - A Quick Introduction



Requirement

Design

Code

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



Advanced C SDLC - A Quick Introduction





Design

Code

- 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



SDLC - A Quick Introduction





Design

Code

- 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



SDLC - A Quick Introduction





Design

Code

- 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



Problem Solving - How?

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



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.



Algorithm - Need?

- Faster Processing
- 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



Algorithm - Reaching this Room - Take a Walk



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



Algorithm - Reaching this Room - Take a Walk



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



Algorithm - Reaching this Room - Take a Bus



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



Algorithm - Reaching this Room - Take a Car



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



Algorithm - Reaching this Room - Take a Car



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



Algorithm - Reaching this Room - Let's Pool



- 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





Algorithm - Reaching this Room - Let's Pool



- You might save some time
- Less costly comparatively

Cons

- Have to wait for partner to reach
- Could have issues on traffic congestions



Algorithm - Daily Life Example - Conclusion



- 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



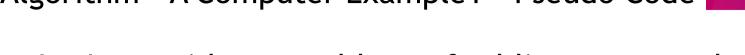
Algorithm - A Computer Example1



- Let's consider a problem of adding two numbers
- The steps involved in it would be as simple as given below
 - 1. Start
 - 2. Read the value of A and B.
 - 3. Add A and B and store in SUM
 - 4. Display SUM.
 - 5. 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 in it would be as simple as given below

BEGIN

Read A, B

SUM = A + B

Print SUM

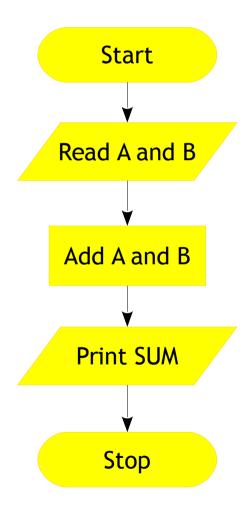
END

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



Algorithm - A Computer Example1 - A Flow Chart

Let's consider a problem of adding two numbers





Algorithm - DIY

- Finding largest of 2 numbers
- Find the average of given set of numbers
- To check whether given letter is vowel or not
- To find the sum of n natural numbers
- To find the number is prime or not
- Convert the number from decimal to binary and display



Chapter 2 Basic Refreshers

Number Systems

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

Туре	Range (8 Bits)
Decimal	0 - 255
Octal	<mark>0</mark> 00 - <mark>0</mark> 377
Hexadecimal	0x00 - 0xFF
Binary	<mark>0b</mark> 00000000 - <mark>0b</mark> 11111111

Dec	Oct	Hex		В	in	
10	8	16		4	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	В	1	0	1	1
12	14	C	1	1	0	0
13	15	D	1	1		
14	16	E	1	1	1	0
15	17	F	1	1	1	1

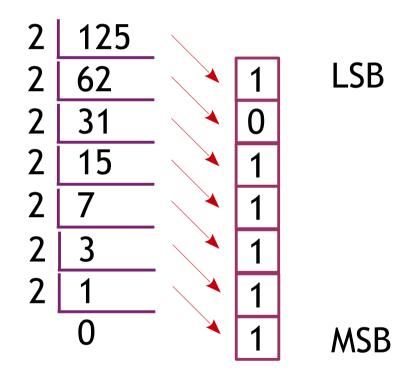
Type

Base



Number Systems - Decimal to Binary

• 125₁₀ to Binary



• So 125₁₀ is 1111101₂



Number Systems - Decimal to Octal

• 212₁₀ to Octal

• So 212₁₀ is 324₈



Number Systems - Decimal to Hexadecimal

• 472₁₀ to Hexadecimal

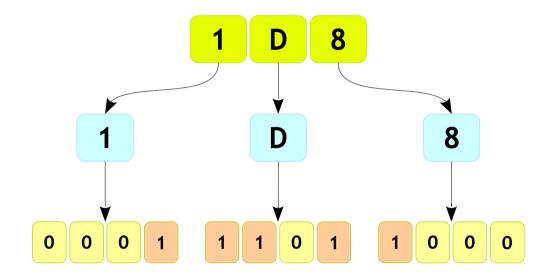
Repres	sentation	Su	bsti	tute	S												
Dec		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex		0	1	2	3	4	5	6	7	8	9	A	В	C	D	E	F

So 472₁₀ is 1D8₁₆



Number Systems - Hexadecimal to Binary

• 1D8₁₆ to Binary



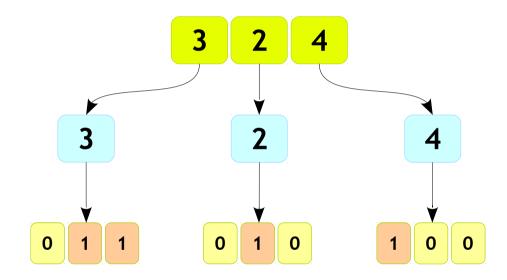
• So 1D8₁₆ is 000111011000₂ which is nothing but 111011000₂





Number Systems - Octal to Binary

• 324₈ to Binary



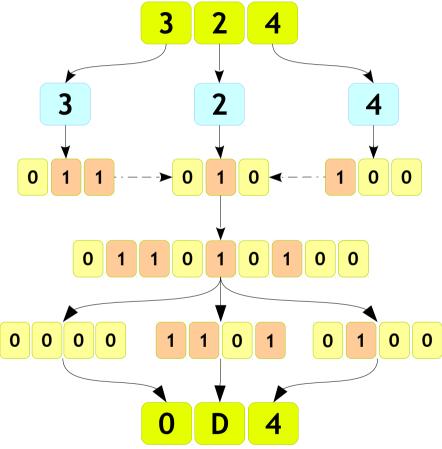
• So 324₈ is 011010100₂ which is nothing but 11010100₂





Number Systems - Octal to Hexadecimal

• 324₈ to Hexadecimal

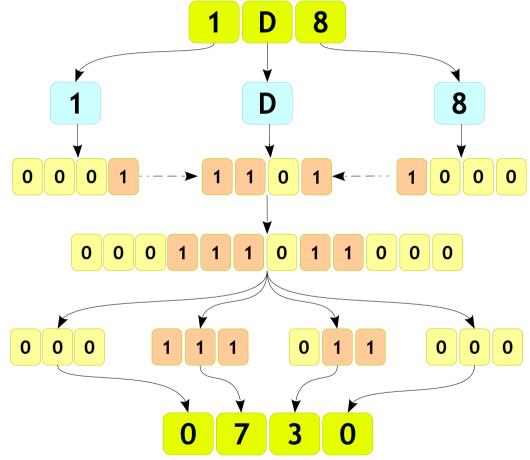


So 324₈ is 0D4₁₆ which is nothing but D4₁₆



Number Systems - Hexadecimal to Octal

• 1D8₁₆ to Octal



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



Number Systems - Binary to Decimal

• 111011000₂ to Decimal

```
1 1 1 0 1 1 0 0 0

Bit Position 8 7 6 5 4 3 2 1 0

28 27 26 25 24 23 22 21 20

256 + 128 + 64 + 0 + 16 + 8 + 0 + 0 + 0
```

So 111011000₂ is 472₁₀



Data Representation - Bit



- Literally computer understand only two states, and those are HIGH and LOW making it a binary system
- It is the nature of the build (Electronics voltage states) makes it easy to store and keep data reliable
- These states can be easily represented using 2 digits, hence making it binary
- So the words "Binary Digit" gave birth to the word "Bit"

Value	No of Bits
0	0
1	1



Data Representation - Byte

- Minimal addressing unit of a system
- One byte is equal to 8 bits generally
- Considered as a unit of memory size

Value	N	0	of	Bit	S			
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1



Data Representation - Character



- 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



Data Representation - Character



- 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

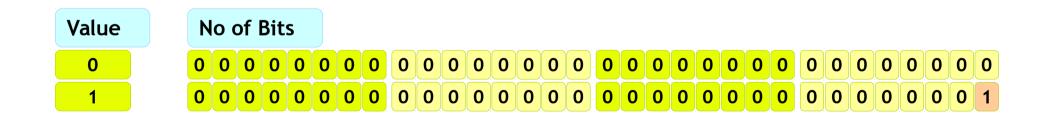
Value	N	0	of I	Bit	S			
0	0	1	1	0	0	0	0	0
A	0	1	0	0	0	0	0	1



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





Integer Number - Positive



- Integers are like whole numbers, but allow negative numbers and no fraction
- An example of 13₁₀ in 32 bit system would be

Bit	N	10	of	Bit	S																											
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



Integer Number - Negative



- Negative Integers represented with the 2's complement of the positive number
- An example of -13₁₀ in 32 bit system would be

Bit	N	0	of l	Bit	S																											
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

• Mathematically: $-k \equiv 2^n - k$



Float Point Number



- 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 and fractional parts separately. The integral part is converted with the techniques examined previously. The fractional part can be converted by multiplication.
- 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.



Float Point Number - Conversion - Example 1



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

Bit	S			E	крс	one	ent	1												Ma	nti	SSã	ì									
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	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



Float Point Number - Conversion - Example 2



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

39887.5625₁₀ = 10011011111001₂

Step 2:

Normalize: 10011011111001111.1001₂ = 1.00110111110011111001₂ × 2¹⁵

Mantissa is 001101111100111110010000 Exponent is 15 + 127 = 142 = 10001110₂ 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
Ì	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



Float Point Number - Conversion - Example 3



Convert -1313.3125 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

1313.3125₁₀ = 10100100001.0101₂

Step 2:

Normalize: $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$

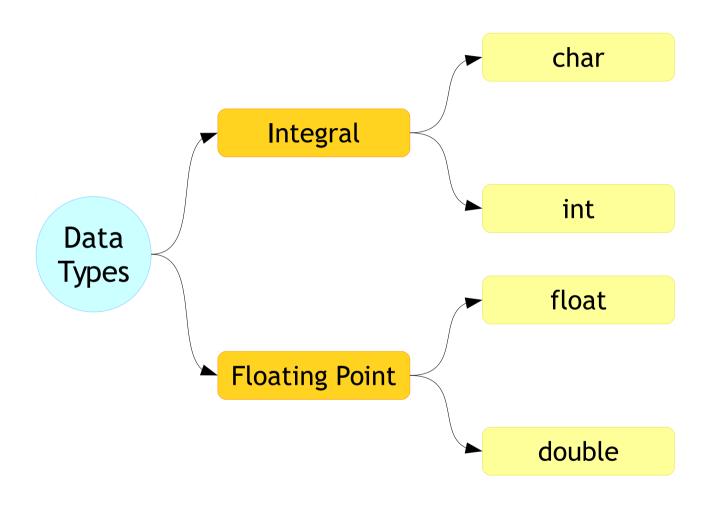
Mantissa is 01001000010101000000000 Exponent is 10 + 127 = 137 = 10001001₂ Sign bit is 1

	Bit	S		Exponent						Mantissa																							
Ì	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	1	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0



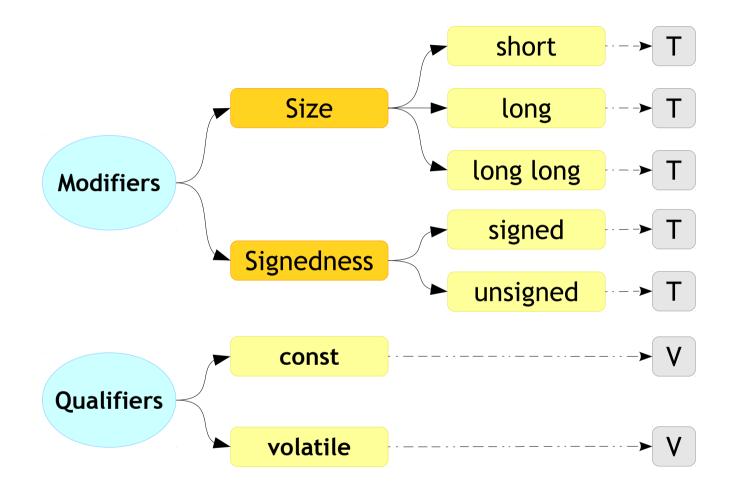
Advanced C Basic Data Types







Data Type Modifiers and Qualifiers



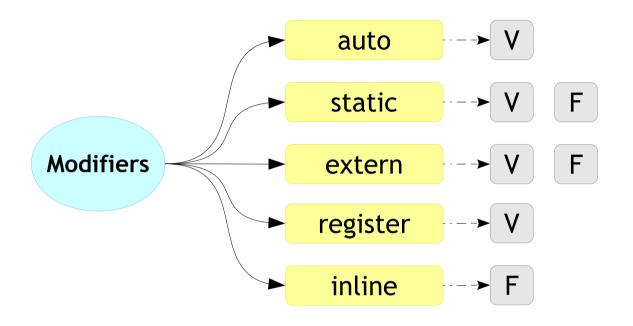
Variables

T Data Types

Functions



Data Type and Function storage modification





Variables

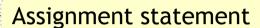
T Data Types

F Functions



Code Statements - Simple

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



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!!



Code Statements - Compound

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



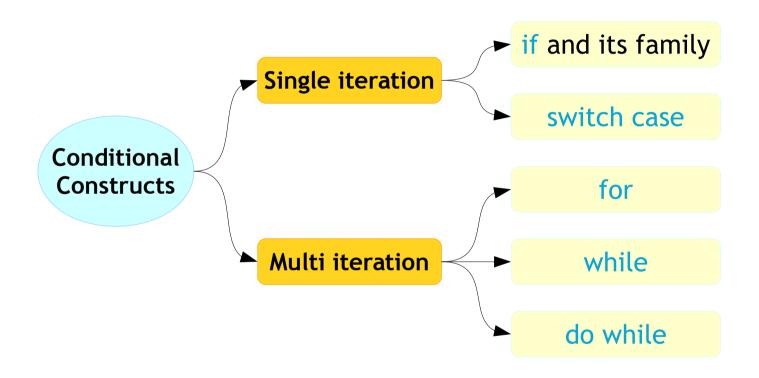
If conditional statement

Nested if statement



Advanced C Conditional Constructs





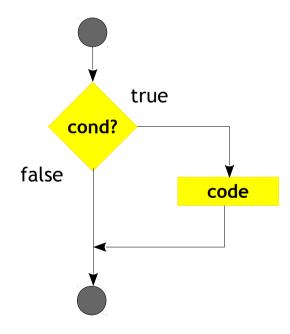


Conditional Constructs - if

Syntax

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

Flow



```
#include <stdio.h>
int main()
{
    int num1 = 2;

    if (num1 < 5)
    {
        printf("num1 < 5\n");
    }
    printf("num1 is %d\n", num1);

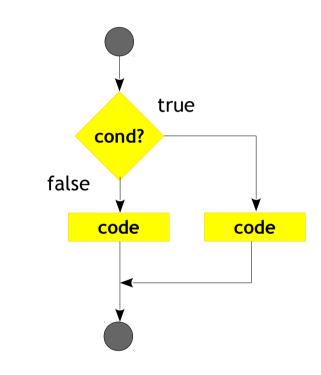
    return 0;
}</pre>
```



Conditional Constructs - if else

Syntax

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





Conditional Constructs - if else

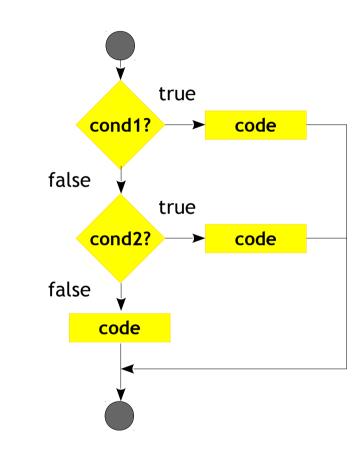
```
#include <stdio.h>
int main()
   int num1 = 10;
   if (num1 < 5)
      printf("num1 < 5\n");
   else
      printf("num1 > 5 \n");
   return 0;
```



Conditional Constructs - if else if

Syntax

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





Conditional Constructs - if else if

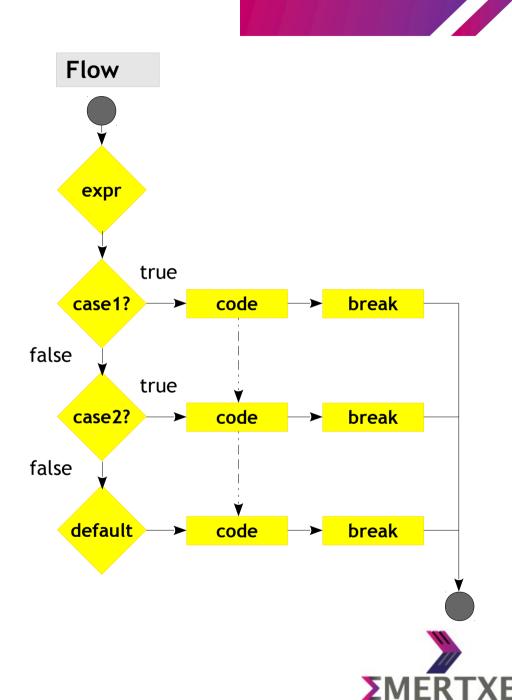
```
#include <stdio.h>
int main()
    int num1 = 10;
    if (num1 < 5)
    {
       printf("num1 < 5 \n'');
    else if (num1 > 5)
       printf("num1 > 5 \n");
    else
       printf("num1 = 5\n");
    return 0;
```



Conditional Constructs - switch

Syntax

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



Conditional Constructs - switch

```
#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;
```



Conditional Constructs - while

Syntax

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

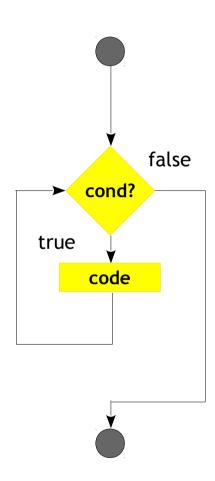
Example

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

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

   return 0;
}</pre>
```







Conditional Constructs - do while

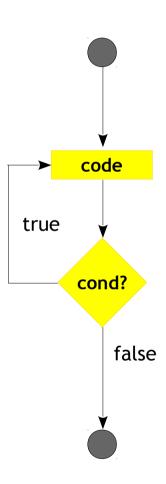
Syntax

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

Example

```
#include <stdio.h>
int main()
   int iter;
   iter = 0;
   do
       printf("Looped %d times\n", iter);
       iter++;
    } while (iter < 10);</pre>
   return 0;
```







Conditional Constructs - for

Syntax

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

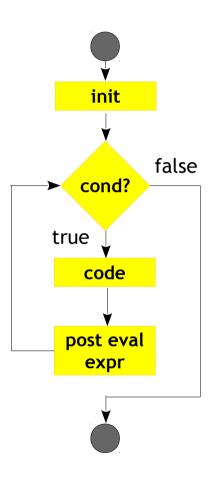
Example

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

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

   return 0;
}</pre>
```





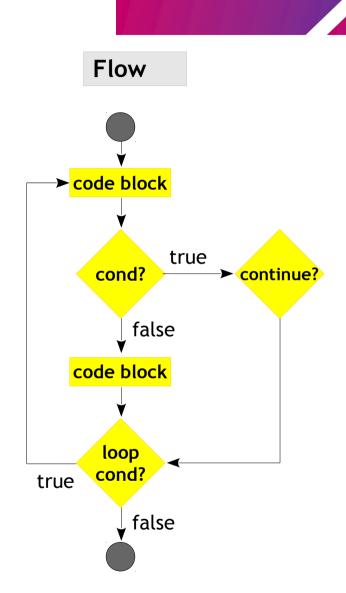


Conditional Constructs - continue

- continue is used to skip the later statements in loop encountered after it and proceed to start of loop
- Can be used in any type of loop

Syntax

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





Conditional Constructs - continue

```
#include <stdio.h>
int main()
   int iter;
   for (iter = 0; iter < 10; iter++)</pre>
       if (iter == 5)
           continue;
       printf("%d\n", iter);
   return 0;
```



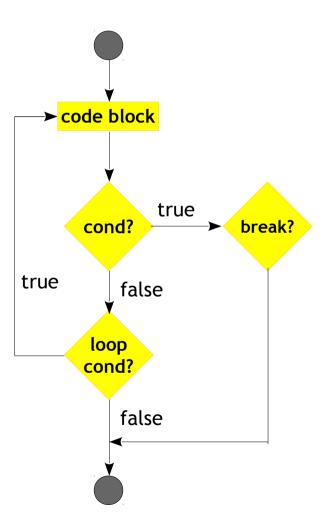
Conditional Constructs - break

- break is used to skip the later statements in loop encountered after it and break out of the loop
- Can be used in any type of loop

Syntax

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







Conditional Constructs - break

```
#include <stdio.h>
int main()
   int iter;
   for (iter = 0; iter < 10; iter++)</pre>
       if (iter == 5)
           break;
       printf("%d\n", iter);
   return 0;
```



Conditional Constructs - break

```
#include <stdio.h>
int main()
   int iter;
   for (iter = 0; iter < 10; iter++)</pre>
       if (iter == 5)
           break;
       printf("%d\n", iter);
   printf("%d\n", iter);
   return 0;
```

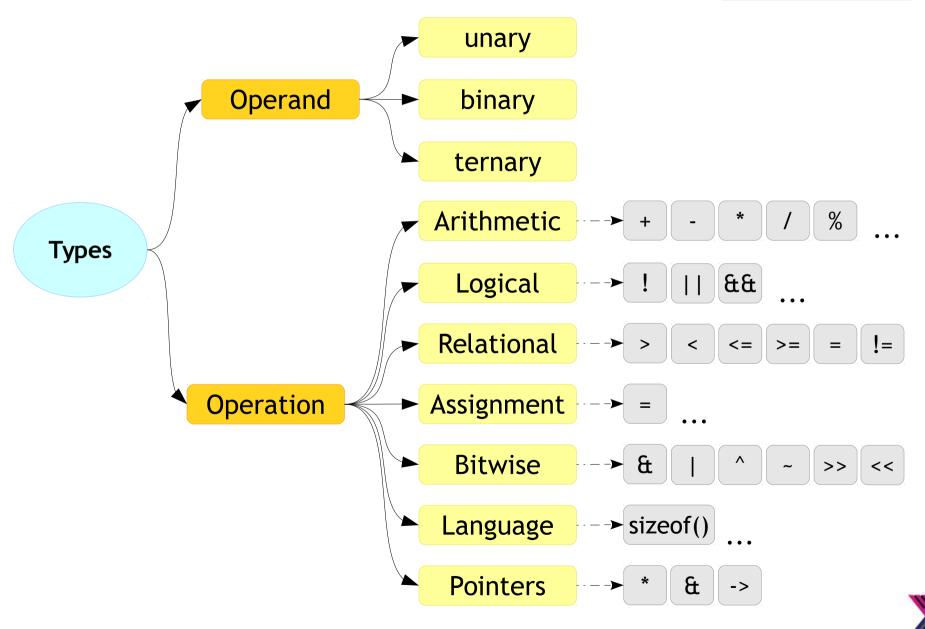


Operators

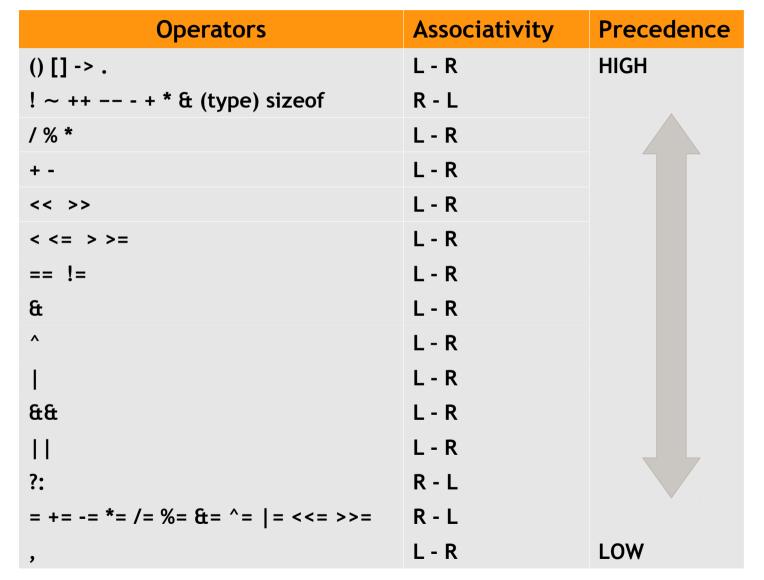
- iler to perform specific
- Symbols that instructs the compiler to perform specific arithmetic or logical operation on operands
- All C operators do 2 things
 - Operates on its Operands
 - Returns a value



Operators



Operators - Precedence and Associativity





Operators - Arithmetic



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

Example

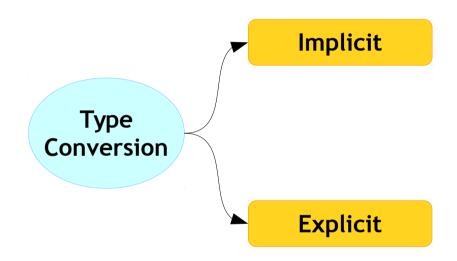
```
#include <stdio.h>
int main()
{
   int num1 = 0, num2 = 0;
   printf("sum is %d\n", num1++ + ++num2);
   return 0;
}
```

What will be the output?



Advanced C Type Conversion







Advanced C Type Conversion Hierarchy



long double double float unsigned long long signed long long unsigned long signed long unsigned int signed int unsigned char signed short unsigned char signed char



Type Conversion - Implicit



- Automatic Unary conversions
 - All operands of type char & short are converted to int before any operations
- Automatic Binary conversions
 - If one operand is of LOWER RANK data type & other is of HIGHER RANK data type then LOWER RANK will be converted to HIGHER RANK while evaluating the expression.
- Example
 - If one operand is int & other is float then, int is converted to float.



Type Conversion - Implicit



- Type conversions in assignments
 - The type of right hand side operand is converted to type of left hand side operand in assignment statements.
 - If type of operand on right hand side is LOWER RANK data type & left hand side is of HIGHER RANK data type then LOWER RANK will be promoted to HIGHER RANK while assigning the value.
- If type of operand on right hand side is HIGHER RANK data type & left hand side is of LOWER RANK data type then HIGHER RANK will be demoted to LOWER RANK while assigning the value.
- Example
 - Fractional part will be truncated during conversion of float to int.



Type Conversion - Explicit (Type Casting)

Syntax

```
(data type) expression
```

```
#include <stdio.h>
int main()
{
  int num1 = 5, num2 = 3;
  float num3 = (float) num1 / num2;
  printf("nun3 is %f\n", num3);
  return 0;
}
```



Operators - Logical



Operator	Description	Associativity
!	Logical NOT	R to L
88	Logical AND	L to R
11	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?



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?



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;
}
```

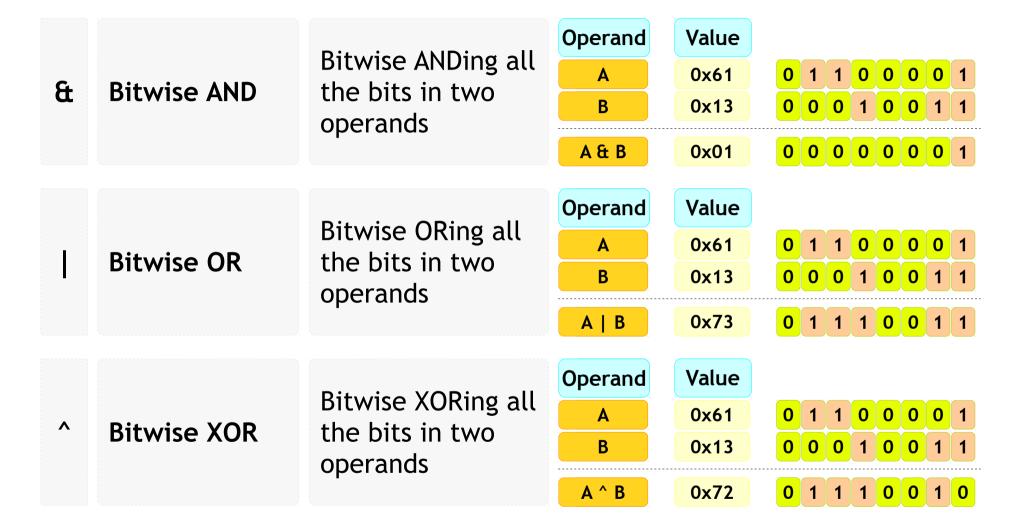
```
#include <stdio.h>
int main()
{
    float num1 = 1;

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

    return 0;
}
```

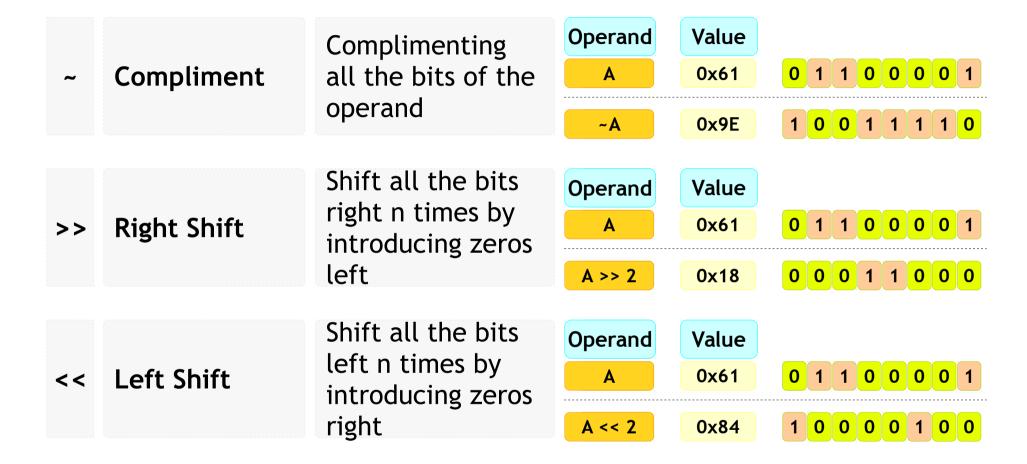


Operators - Bitwise





Operators - Bitwise





Operators - Bitwise

```
#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;
```



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;
}
```

```
#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;
}
```



Operators - Language - sizeof()

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





Operators - Ternary

Syntax

Condition ? Expression 1 : Expression 2;

Example

```
#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;
```

```
#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;
}
```



Operators - Circuit Logical



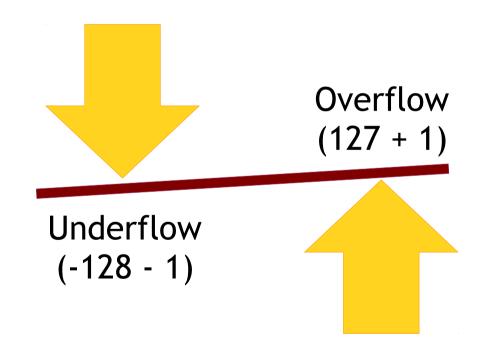
- These operators are similar to the & and | operators that are applied to Boolean type
- Have the ability to "short circuit" a calculation if the result is definitely known, this can improve efficiency
 - AND operator &&
 - If one operand is false, the result is false.
 - OR operator ||
 - If one operand is true, the result is true.



Over and Underflow



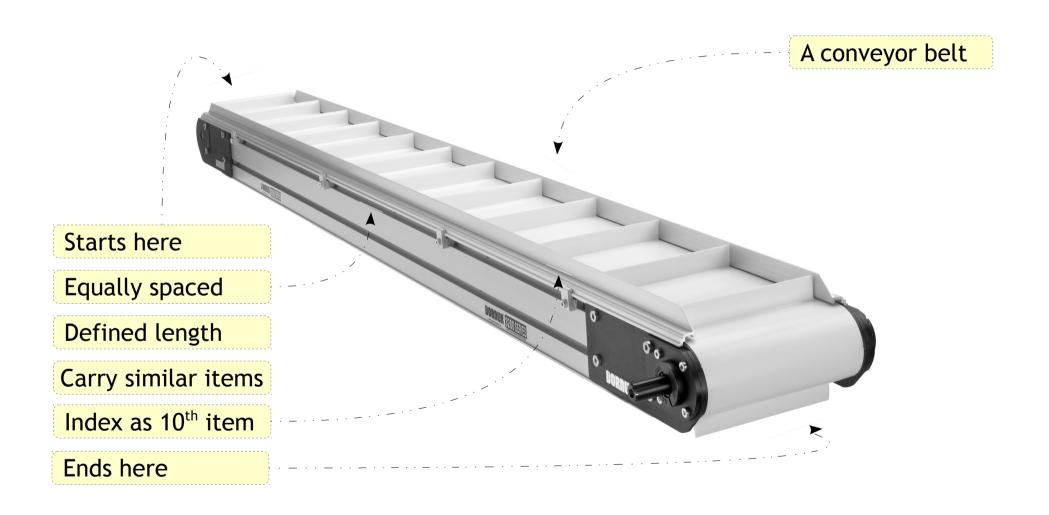
- 8-bit Integral types can hold certain ranges of values
- So what happens when we try to traverse this boundary?





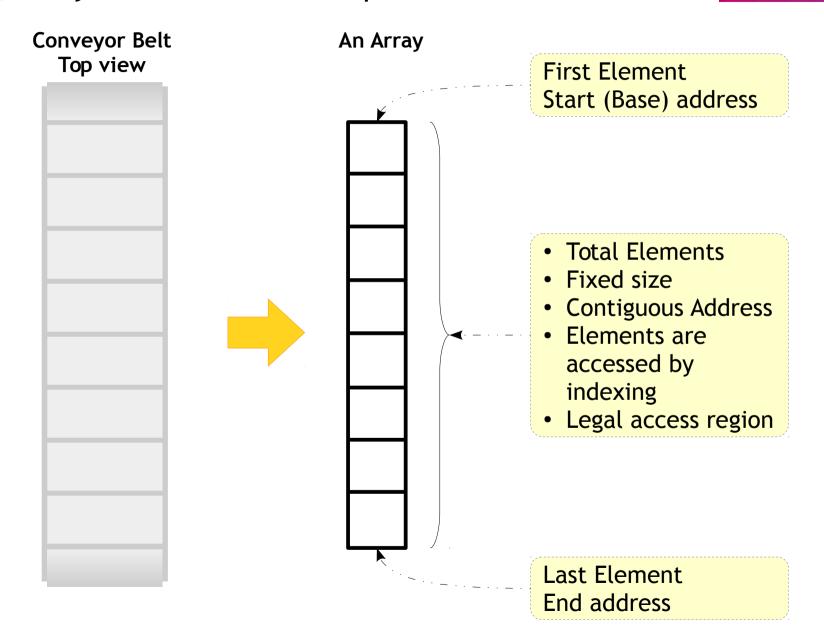
Arrays - Know the Concept







Arrays - Know the Concept



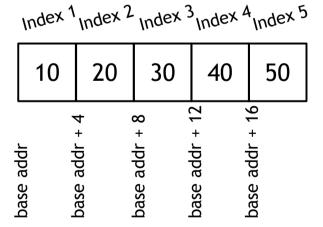


Arrays

Syntax

```
data_type name[SIZE];
Where SIZE is number of elements
The size for the array would be SIZE * <size of data_type>
```

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





Arrays - Point to be noted



- An array is a collection of data of same data type.
- Addresses are sequential
- First element with lowest address and the last element with highest address
- Indexing starts from 0 and should end at array SIZE 1. Example say array[5] will have to be indexed from 0 to 4
- Any access beyond the boundaries would be illegal access Example, You cannot access array[-1] or array[SIZE]



Arrays - Why?

```
#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;
}</pre>
```



Arrays - Reading

```
#include <stdio.h>
int main()
   int array[5] = \{1, 2, 3, 4, 5\};
   int index;
   intex = 0;
   do
       printf("Index %d has Element %d\n", index, array[index]);
       index++;
    } while (index < 5);</pre>
   return 0;
```



Arrays - Storing

```
#include <stdio.h>
int main()
{
   int array[5];
   int index;

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

   return 0;
}</pre>
```



Arrays - Initializing

```
#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;
```



Arrays - Copying

Can we copy 2 arrays? If yes how?

```
#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;
```





Arrays - Copying



- 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



Arrays - DIY



- W.A.P to find the average of elements stored in a array.
 - Element should 5 and of integer type
 - Elements should be accepted from the user
 - Average should be double
- W.A.P to find the largest array element
- W.A.P program to compare two arrays element by element, print its comparison result as TRUE or FALSE.
 - Arrays could be of different sizes but integer type
 - Both array elements should be accepted from the user



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)"

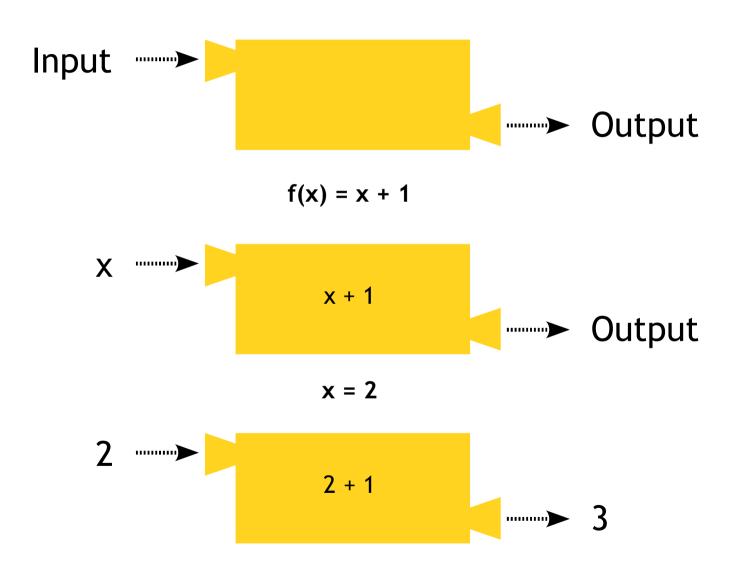
Source: Google

- In programing languages it can something which performs a specific service
- Generally it can have 3 parameters like
 - Input
 - Operation
 - Output



Functions - What?







Functions - How to write



Syntax

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

```
Return data type as int

First parameter with int type

Second parameter with int type

Second parameter with int type
```



Functions - How to write



$$y = x + 1$$

Example

```
int foo(int x)
{
  int ret = 0;
  ret = x + 1;
  return ret;
}
```

Return from function



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



Functions - A complete look

```
#include <stdio.h>
int main() <</pre>
                                           The main function
                                           The function call
    int num1 = 10, num2 = 20;
    int sum = 0;
                                           Actual arguments
    sum = add numbers(num1, num2);
    printf("Sum is %d\n", sum);
                                           Return type
    return 0;
                                           Formal arguments
int add_numbers(int num1, int num2)
    int sum = 0;
                                           Function
    sum = num1 + num2;
                                           Return from function
   return sum;
```



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



Functions - DIY



Write a function to calculate square a number

$$-y=x*x$$

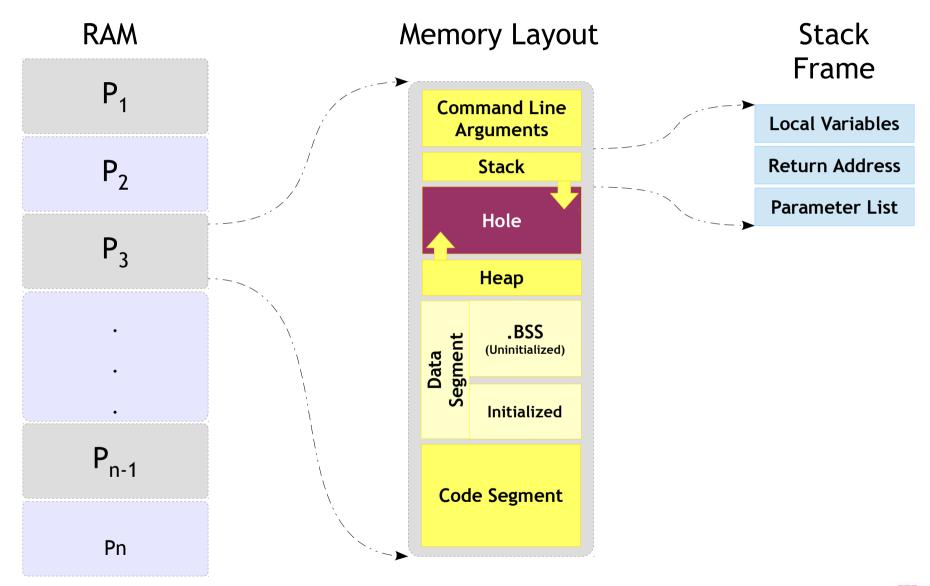
 Write a function to convert temperature given in degree Fahrenheit to degree Celsius

$$- C = (F - 32) / 1.8$$

 Write a program to check if a given number is even or odd. Function should return TRUE or FALSE



Function and the Stack





Functions - Parameter Passing Types



Pass by Value

- This method copies the actual This method copies the address of 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.

Pass by reference

- an argument into the formal parameter.
- Inside the function, the address is access the actual used to argument used in the call. This means that changes made to the parameter affect the argument.



Functions - Pass by Value

```
#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;
int add numbers(int num1, int num2)
    int sum = 0;
    sum = num1 + num2;
    return sum;
```



Functions - Pass by Value

```
#include <stdio.h>
void modify(int num1);
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;
void modify(int num1)
    num1 = num1 + 1;
```



Functions - Pass by Value





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



Functions - Pass by Reference

```
#include <stdio.h>
void modify(int *num ptr);
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;
void modify(int *num ptr)
    *num ptr = *num ptr + 1;
```

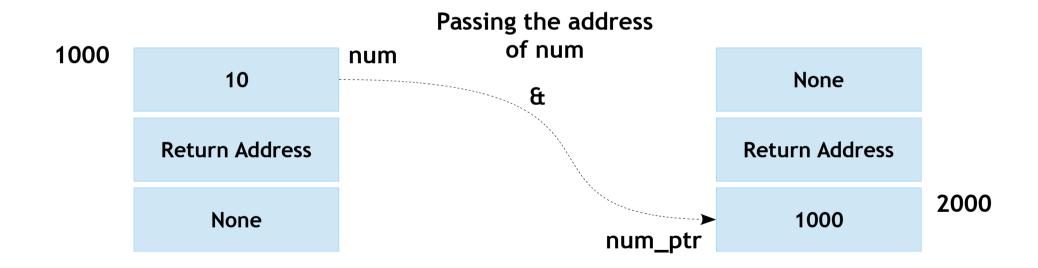


Functions - Pass by Reference



main function's stack frame

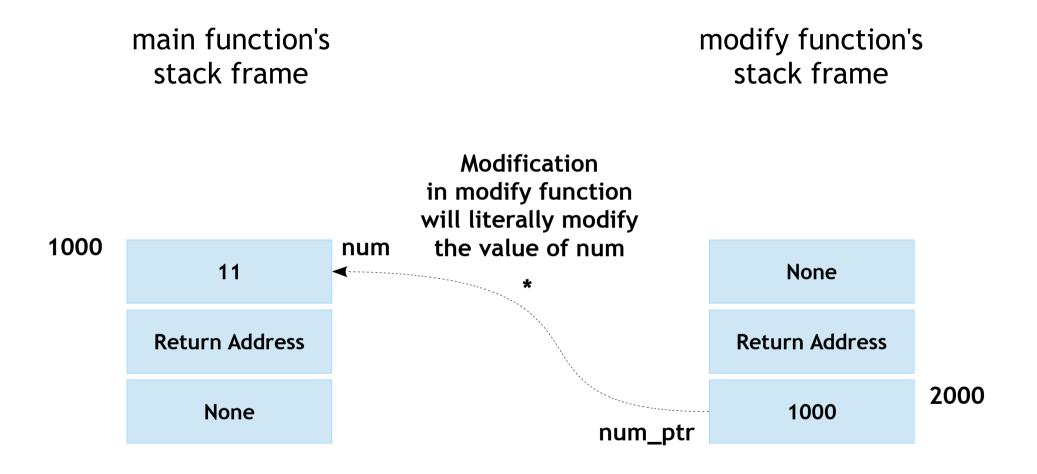
modify function's stack frame





Functions - Pass by Reference







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.



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



Functions - Passing Array



- 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



Functions - Passing Array

```
#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++)</pre>
         printf("Index %d has Element %d\n", iter, array[iter]);
```



Functions - Passing Array

```
#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++)</pre>
         printf("Index %d has Element %d\n", iter, *array);
         array++;
```



Functions - Passing Array

```
#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;
void print array(int *array, int size)
    int iter;
    for (iter = 0; iter < size; iter++)</pre>
        printf("Index %d has Element %d\n", iter, *array++);
```



Functions - Returning Array

```
#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;
}
```

```
int *modify_array(int *array, int size)
{
    int iter;
    for (iter = 0; iter < size; iter++)
    {
        *(array + iter) += 10;
    }
    return array;
}</pre>
```

```
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]);
      }
}</pre>
```



Functions - Returning Array

```
#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]);
   }
}</pre>
```



Functions - DIY



- 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



Functions





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



Functions - Recursive - Example

```
#include <stdio.h>
/* Factorial of 3 numbers */
int factorial(int number)
    if (number <= 1)</pre>
        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;
```



Functions - Variadic



- 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, ...);
```



Functions - Variadic - Definition & Usage

- 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.



Functions - Variadic - Argument access macros



- Descriptions of the macros used to retrieve variable arguments
- These macros are defined in the header file stdarg.h

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



Functions - Variadic - Example

```
#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;
}
```

```
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++)</pre>
       /* Extract args */
       sum += va arg(ap, int);
    }
    /* Cleanup */
   va end(ap);
    return sum;
```



Chapter 4 Pointers and Strings

Pointers - Jargon

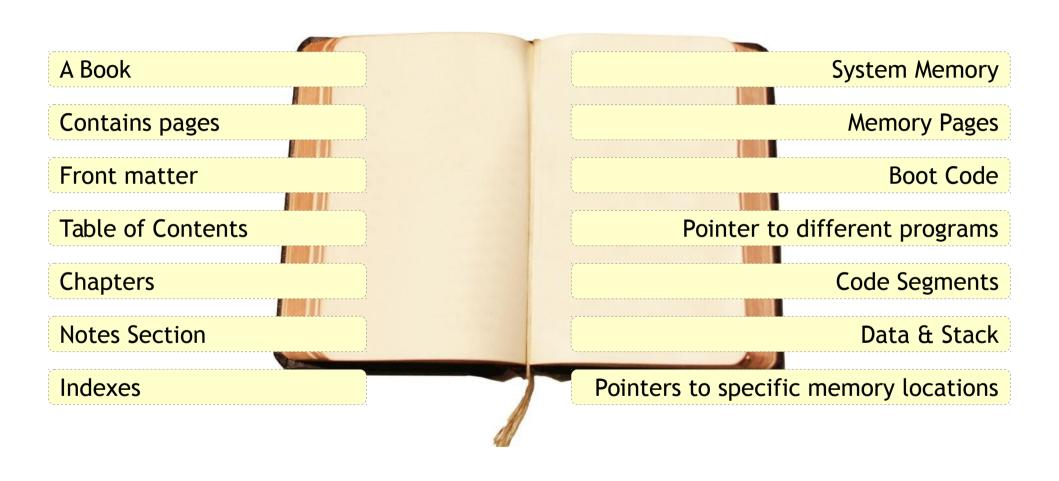


- What's a Jargon?
 - Jargon may refer to terminology used in a certain profession, such as computer jargon, or it may refer to any nonsensical language that is not understood by most people.
 - Speech or writing having unusual or pretentious vocabulary, convoluted phrasing, and vague meaning.
- Pointer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



Pointers - Analogy with Book







Pointers - Computers



- 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



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



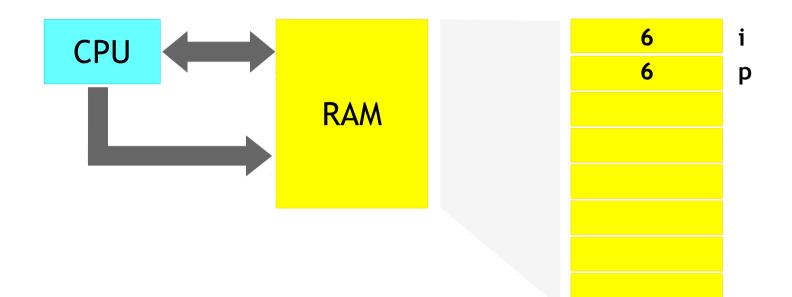
Pointers - The 7 Rules

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



Pointers - The 7 Rules - Rule 1







Pointers - The 7 Rules - Rule 1



- 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)

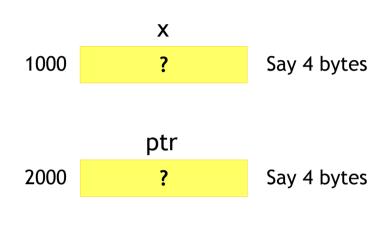


Pointers - Rule 1 in detail

```
#include <stdio.h>
int main()
{
   int x;
   int *ptr;

   x = 5;
   ptr = 5;

   return 0;
}
```



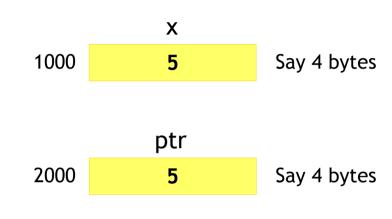


Pointers - Rule 1 in detail

```
#include <stdio.h>
int main()
{
   int x;
   int *ptr;

   x = 5;
   ptr = 5;

   return 0;
}
```

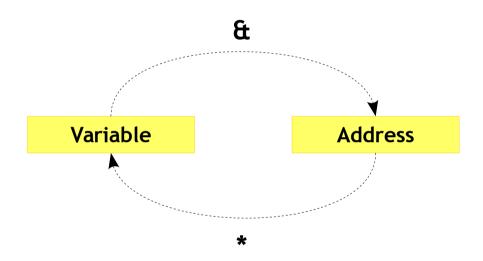


- So pointer is an integer
- But remember the "They may not be of same size"



Pointers - The 7 Rules - Rule 2

• Rule: "Referencing and Dereferencing"





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



Pointers - Rule 2 in detail

```
#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



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?

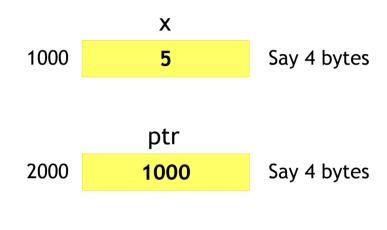


Pointers - Rule 2 in detail

```
#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



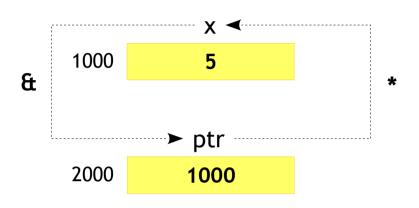
Pointers - Rule 2 in detail

Example

```
#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"



Pointers - Rule 2 in detail

```
#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;
}
```



Pointers - Rule 2 in detail

```
#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;
}
```



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);

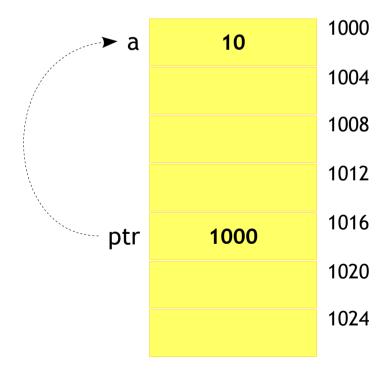
   return 0;
}
```

By compiling and executing the above code we can conclude



Pointers - The 7 Rules - Rule 3

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





Pointers - The 7 Rules - Rule 4



- 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"



Pointers - Rule 4 in detail

• The question is, does address has a type?

```
#include <stdio.h>
int main()
{
   int num = 1234;
   char ch;
   return 0;
}
```

```
num
1000 1234 4 bytes

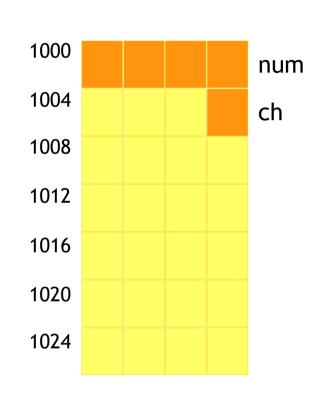
ch
1004 ? 1 bytes
```

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



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?

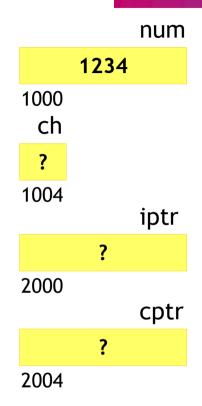




Pointers - Rule 4 in detail

Example

```
#include <stdio.h>
int main()
{
   int num = 1234;
   char ch;
   int *iptr;
   char *cptr;
   return 0;
}
```



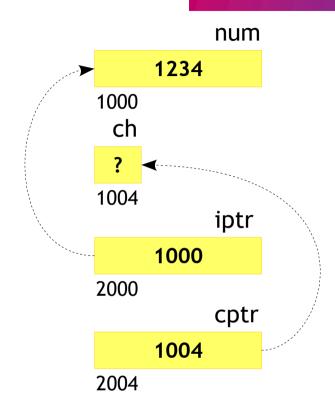
- Lets consider the above examples to understand it
- Say we have a integer and a character pointer



Pointers - Rule 4 in detail

Example

```
#include <stdio.h>
int main()
{
   int num = 1234;
   char ch;
   int *iptr = &num;
   char *cptr = &ch;
   return 0;
}
```

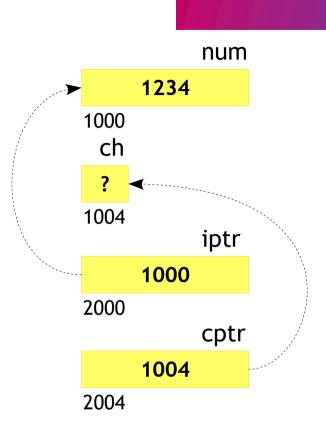


- Lets consider the above examples to understand it
- Say we have a integer and a character pointer



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

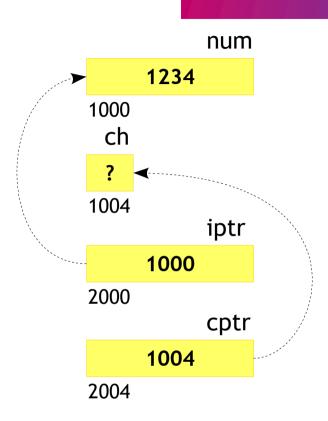




Pointers - Rule 4 in detail

- 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





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 Higher Memory Address
 - Big Endian MSB in Lower Memory Address



Pointers - Rule 4 in detail - Endianness



• 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

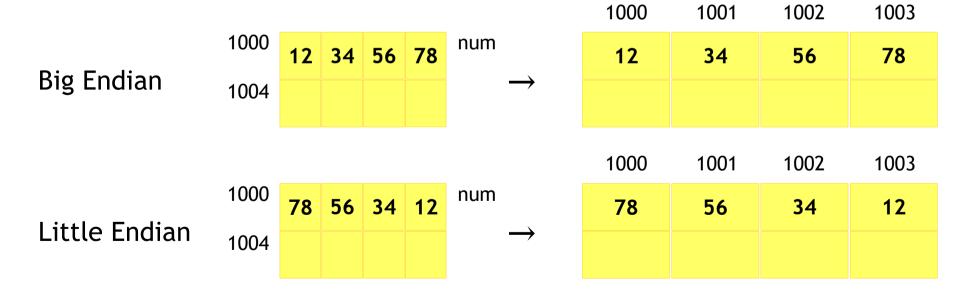


Pointers - Rule 4 in detail - Endianness

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





Pointers - Rule 4 in detail - Endianness

- OK Fine. What now? How is it going 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;

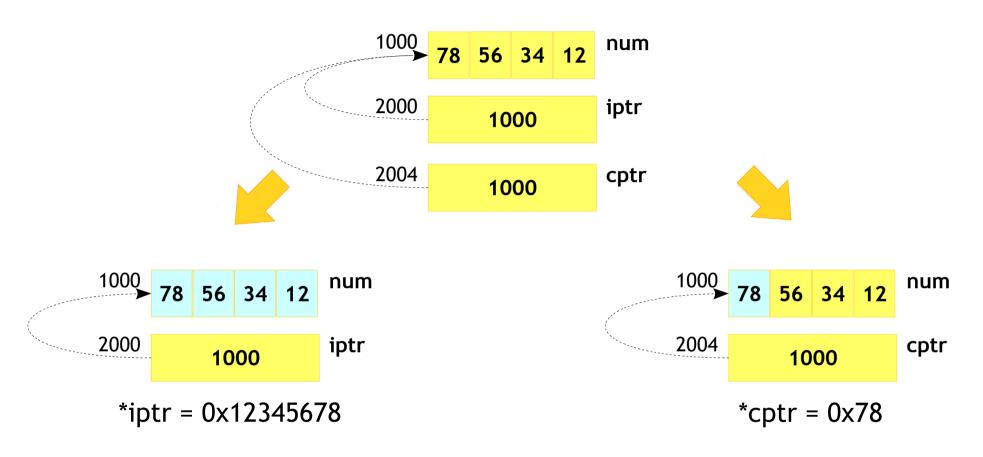
   iptr = &num;
   cptr = &num;
}
```

- 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



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



Pointers - The 7 Rules - Rule 5

Example

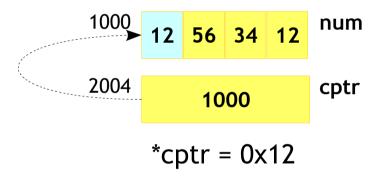
```
#include <stdio.h>
int main()
{
   int num = 0x12345678;
   char ch;

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

   *cptr = 0x12;

   return 0;
}
```

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



 So *iptr would contain 0x12345612 now!!



Pointers - The 7 Rules - Rule 5

- So as a summary the type to the pointer does not say its type, but the type of the 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;
}
```



Pointers - The 7 Rules - Rule 5

Pointer Arithmetic

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



Pointers - The 5 in detail



- Before proceeding further let us understand an array interpretation
 - Original Big Variable
 - Constant Pointer to the 1st Small Variable in the Big Variable
 - When first interpretation fails than second interpretation comes to picture.
 - The following are the case when first interpretation fails:
 - When we pass array variable as function argument
 - When we assign a array variable to pointer variable



Pointers - The 5 in detail

Example

```
#include <stdio.h>
int main()
{
   int array[5] = {1, 2, 3, 4, 5};
   int *ptr = &array;

   return 0;
}
```

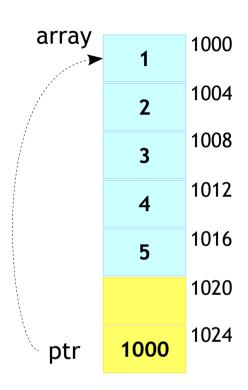
So,

Address of array = 1000

Base address = 1000

 $&array[0] = 1 \rightarrow 1000$

 $&array[1] = 2 \rightarrow 1004$





Pointers - The 5 in detail

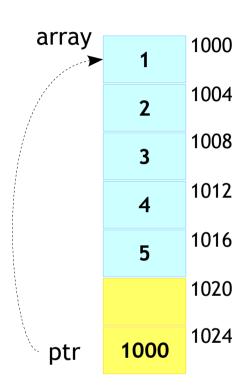
Example

```
#include <stdio.h>
int main()
{
   int array[5] = {1, 2, 3, 4, 5};
   int *ptr = &array;

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

   return 0;
}
```

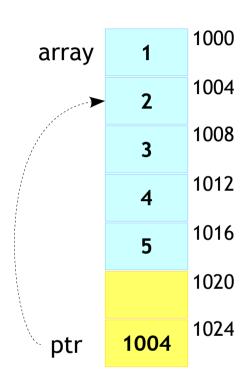
- This code should print 1 as output since its points to the base address
- Now, what should happen if we do ptr = ptr + 1;





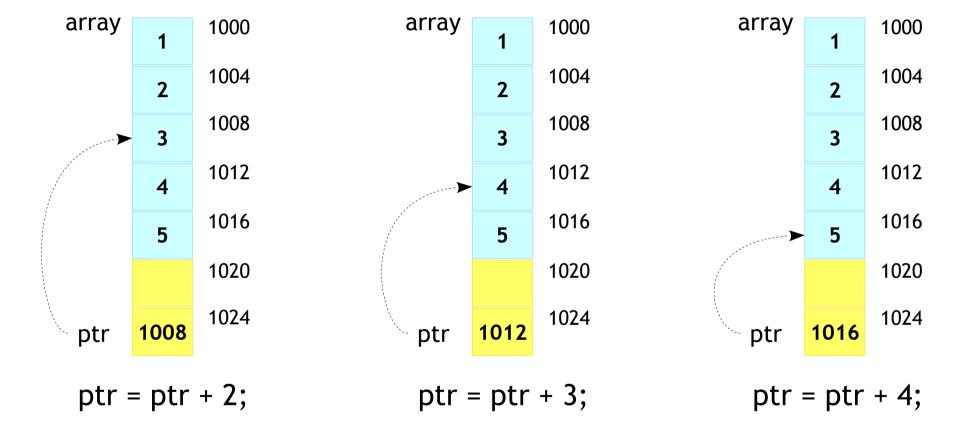
Pointers - The 5 in detail

- ptr = ptr + 1;
- The above line can be discribed as follows
- ptr = ptr + 1 * sizeof(data type)
- In this example we have a integer array, so
- ptr = ptr + 1 * sizeof(int)
 = ptr + 1 * 4
 = ptr + 4
- Here ptr = 1000 so
 = 1000 + 4
 = 1004





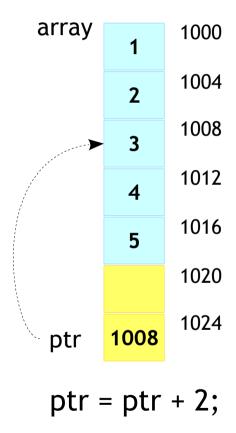
Pointers - The 5 in detail

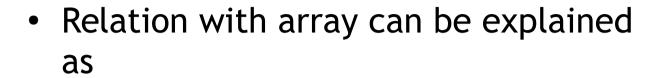


Why does the compiler does this?. Just for convenience



Pointers - The 5 in detail







Pointers - The 5 in detail



So to access a array element using a pointer would be

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

This can be written as following too!!

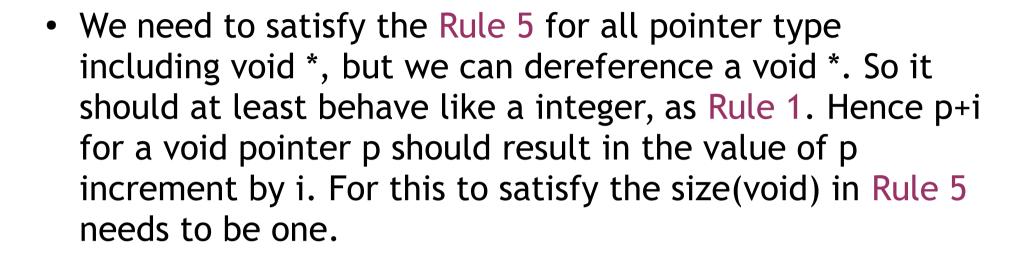
$$array[i] \rightarrow *(array + i)$$

Which results to

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



Pointers - The 7 Rules - Rule 5 - Size of void





Pointers - The 7 Rules - Rule 6



- Terminating Linked Lists
- Indicating Failure by malloc, ...
- 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
- Rule: "Pointer value of NULL or Zero = Null Addr = Null Pointer = Pointing to Nothing"





Pointers - The 7 Rules - Rule 7



- Rule: "Static Allocation vs Dynamic Allocation"
 - Named vs Unnamed Allocation = Named/Unnamed Houses
 - Managed by Compiler vs User
 - Done internally by Compiler vs Using malloc/free
- Dynamic corresponding of a 1-D Static Array
 - char c[10];
 - char *c;

```
c = (char *)(malloc(10 * sizeof(char))); // Must do
```

Use either as *(c + i) or c[i]

free(c); // Must do when done using the malloc'ed memory



S____s - Fill in the blanks please;)





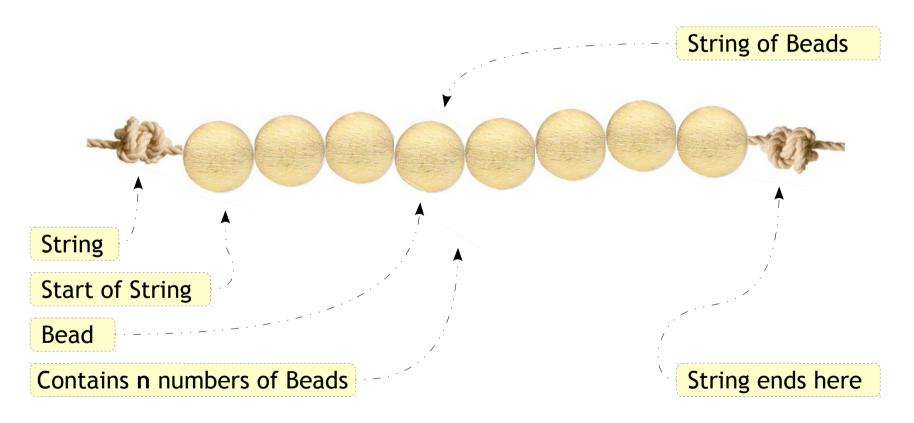


Advanced C Strings



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

Source: Google





Advanced C Strings



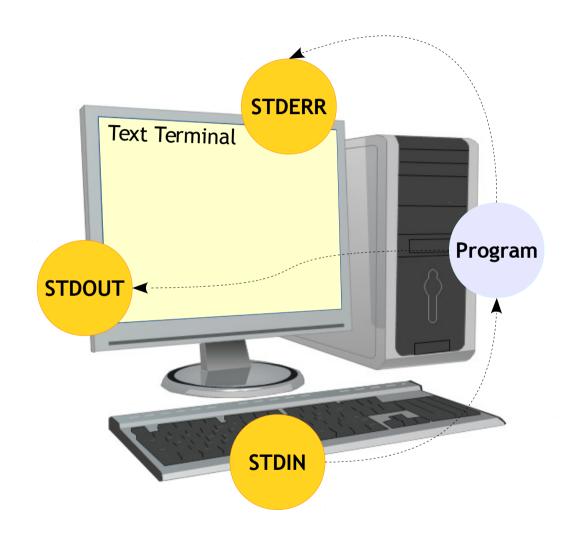
- Contiguous sequence of characters
- Easily stores ASCII and its extensions
- End of the string is marked with a special character, the null character '\0'
- '\0' is implicit in strings enclosed with ""
- Example

"You know, now this is what a string is!"



Chapter 4 Standard Input / Output

Standard I/O





Standard I/O - The File Descriptors



- OS uses 3 file descriptors to provide standard input output functionalities
 - $-0 \rightarrow stdin$
 - $-1 \rightarrow stdout$
 - $-2 \rightarrow stderr$
- 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



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. The stderr could be used while doing diagnostics. I.e if we want to separate out the error messages from normal messages!!
- So how the C language help us in the standard IO?.



Standard I/O - The header file



You need refer input/output library function

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

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



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 getche(), 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



Standard I/O - Unformatted (Basic)

Example

```
#include <stdio.h>
#include <ctype.h>
int main()
   int ch;
   for ( ; (ch = getchar()) != EOF; )
       putchar(toupper(ch));
   puts("EOF Received");
   return 0;
```



Standard I/O - Formatted



- 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



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("string", [variables]);
where string arguments can be
%[flags][width][.precision]format_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



Standard I/O - printf()



Prototype

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

What is this!?



Standard I/O - printf()



Prototype

- 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", );
```



Standard I/O - printf()



```
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



Standard I/O - printf()

Example

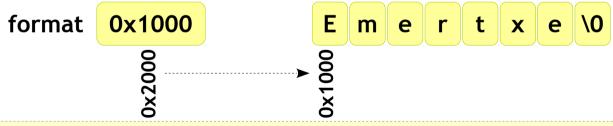
```
#include <stdio.h>
int main()
{
    char *a = "Emertxe";
    printf(a);
    return 0;
}
```

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?

So a pointer hold a address, can be drawn as



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



Standard I/O - printf()

Format Specifier	Example	Expected Output
%c	printf("%c", 'A')	A
%d	printf("%d", 10)	10
%0	printf("%o", 8)	10
%x %X	printf("%x %X %x", 0xA, 0xA, 10)	аАа
%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
%g %G	printf("%g %G", 1.21, 1.2)	1.21 1.2



Standard I/O - printf() - Example

```
#include <stdio.h>
int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string[] = "String Test";

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

    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 string instead of stdout
- Formats the arguments in arg1, arg2, etc., according to format
- string must be big enough to receive the result



Standard I/O - sprintf() - 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",numl , ch, num2, string1);
    printf("%s", string2);

    return 0;
}
```



Standard I/O - Formatted Input - scanf()

Prototype

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

- Reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments.
- Each of the other argument must be a pointer



Standard I/O - Formatted Input - scanf()

- 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.



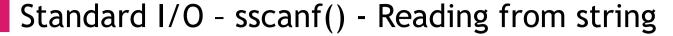
Standard I/O - scanf() - Example

```
#include <stdio.h>
int main()
{
   int num1;
   char ch;
   float num2;
   char string1[];

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

   return 0;
}
```







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

```
#include <stdio.h>
int main()
{
   int age;
   int array_1[10];
   int array_2[10];

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

   return 0;
}
```



Chapter 5 File Input / Output

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



File Input / Output



- 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



File Input / Output - File Pointer

- 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)
```

```
#include <stdio.h>
int main()
{
    FILE *fp;
    fp = open("test.txt", "r");
    fclose(fp);
    return 0;
}
```



File Input / Output - Functions

```
#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;
```



File Input / Output - Functions

Prototype

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

```
#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;
```



File Input / Output - Functions

Prototype

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

```
#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;
```



Chapter 5 Storage Classes

Memory Segments



Linux OS

User Space

Kernel Space 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



Memory Segments

Linux OS

User Space

Kernel Space **User Space**

 P_1

 P_2

 P_3

•

•

 P_{n-1}

 P_n

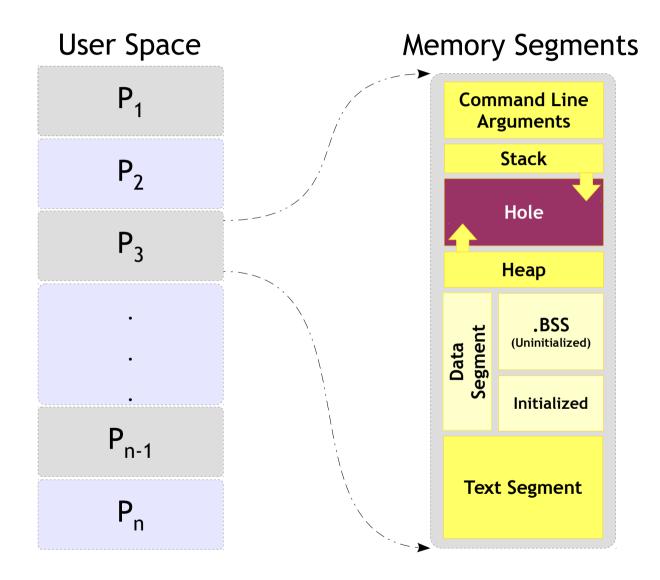
The User space contains many processes

Every process will be scheduled by the kernel

Each process will have its memory layout discussed in next slide



Memory Segments



The memory segment of a program contains four major areas.

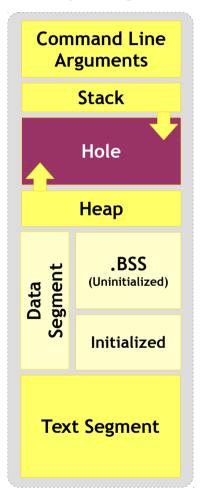
- Text Segment
- Stack
- Data Segment
- Heap



Memory Segments - Text Segment



Memory Segments



Also referred as Code Segment

Holds one of the section of program in object file or memory

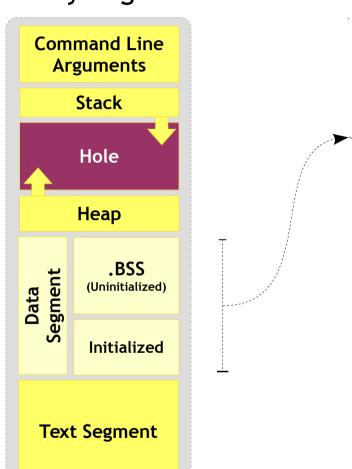
In memory, this is place below the heap or stack to prevent getting over written

Is a read only section and size is fixed



Memory Segments - Data Segment

Memory Segments



Contains 2 sections as initialized and uninitialized data segments

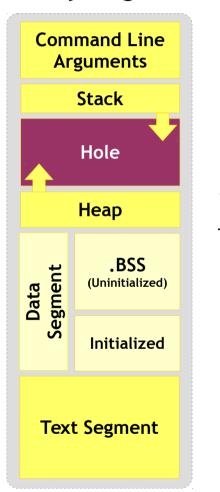
Initialized section is generally called as Data Segment

Uninitialized section is referred as BSS (Block Started by Symbol) usually filled with 0s



Memory Segments - Data Segment

Memory Segments



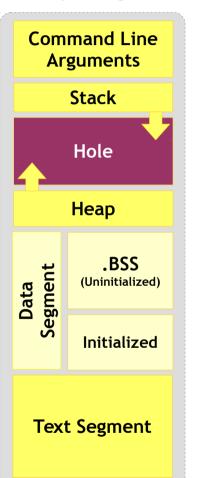
Dynamic memory allocation takes place here

Begins at the end of BSS and grows upward from there



Memory Segments - Stack Segment

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"



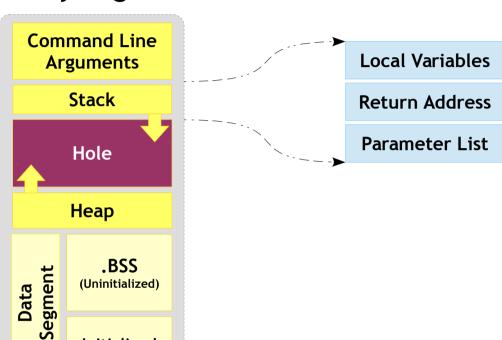
Initialized

Text Segment

Memory Segments - Stack Segment

Memory Segments

Stack Frame



A stack frame contain at least of a return address



Memory Segments - Stack Frame

```
#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 n1, int n2)
    int s = 0;
    s = n1 + n2;
    return s;
```

Stack Frame

num1 = 10
num2 = 20
sum = 0

Return Address to the caller

s = 0

Return Address to the main()

n1 = 10
n2 = 20



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()



Memory Segments - Runtime



- Text Segment: The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions
- Initialized Data Segment: This segment contains global variables which are initialized by the programmer
- Uninitialized Data Segment: Also named "BSS" (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute



Memory Segments - Runtime



- 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



Storage Classes

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



Storage Classes

```
#include <stdio.h>
int global 1;
int globa1_2 = 10;
static int global 3;
static int globa1_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++)</pre>
       /* 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



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



Storage Classes - extern

file1.c

```
#include <stdio.h>
int num;
int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
        func_2();
    }

    return 0;
}
```

file2.c

```
#include <stdio.h>
extern int num;
int func_1()
{
    printf("num is %d from file2\n", num);
    return 0;
}
```

file3.c

```
#include <stdio.h>
extern int num;
int func_2()
{
    printf("num is %d from file3\n", num);
    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



Thank You