C Programming

C Programming Subject Contents

- C development environment
- Introduction to C
- Control flow
- Functions
- Storage class
- Pointer
- Type qualifiers
- Preprocessor Directives
- Array
- Dynamic memory allocation
- Structure & union
- Enum
- File IO
- Function pointer
- Bitwise operator and Advanced features

* Every day 3 Hours Session.
* Saturday and Sunday Weekly OFF.

Introduction to C

Origin and History

- Created in the 1970's on PDP-11 under UNIX by Dennis Ritchie at AT&T Bell Labs
- Many ideas came from type less languages BCPL and B.
- ANSI standard emerged in 1988-1989
- It is quite compact as it is a One man language.
- C Programming = B Programming Language (Ken Thompson) + BCPL - Basic Combine Programming Language (Martin Richards).

Introduction to C

- **C is High level language.**
  - C combines the features of both high-level and low-level languages.
  - It can be used for low-level programming, such as scripting for drivers and kernels and it also supports functions of high-level programming languages, such as scripting for software applications etc.
- **It is procedural / structural programming language**
  - a complex program to be broken into simpler programs called functions
- **It is portable.**
- **Small Instruction set and extensive library functions.**
- Fundamental types are characters, integers and floating point numbers
- Hierarchy of derived types in pointers, arrays, structures and unions
- Provides control-flow constructs for well structured programs
- Functions allow complex problems to be broken down into smaller, more manageable pieces
- Highly extensible - power of language easily extended by data types, macros and functions provided by user libraries and header files.

VS Code IDE (Integrated Development Environment)

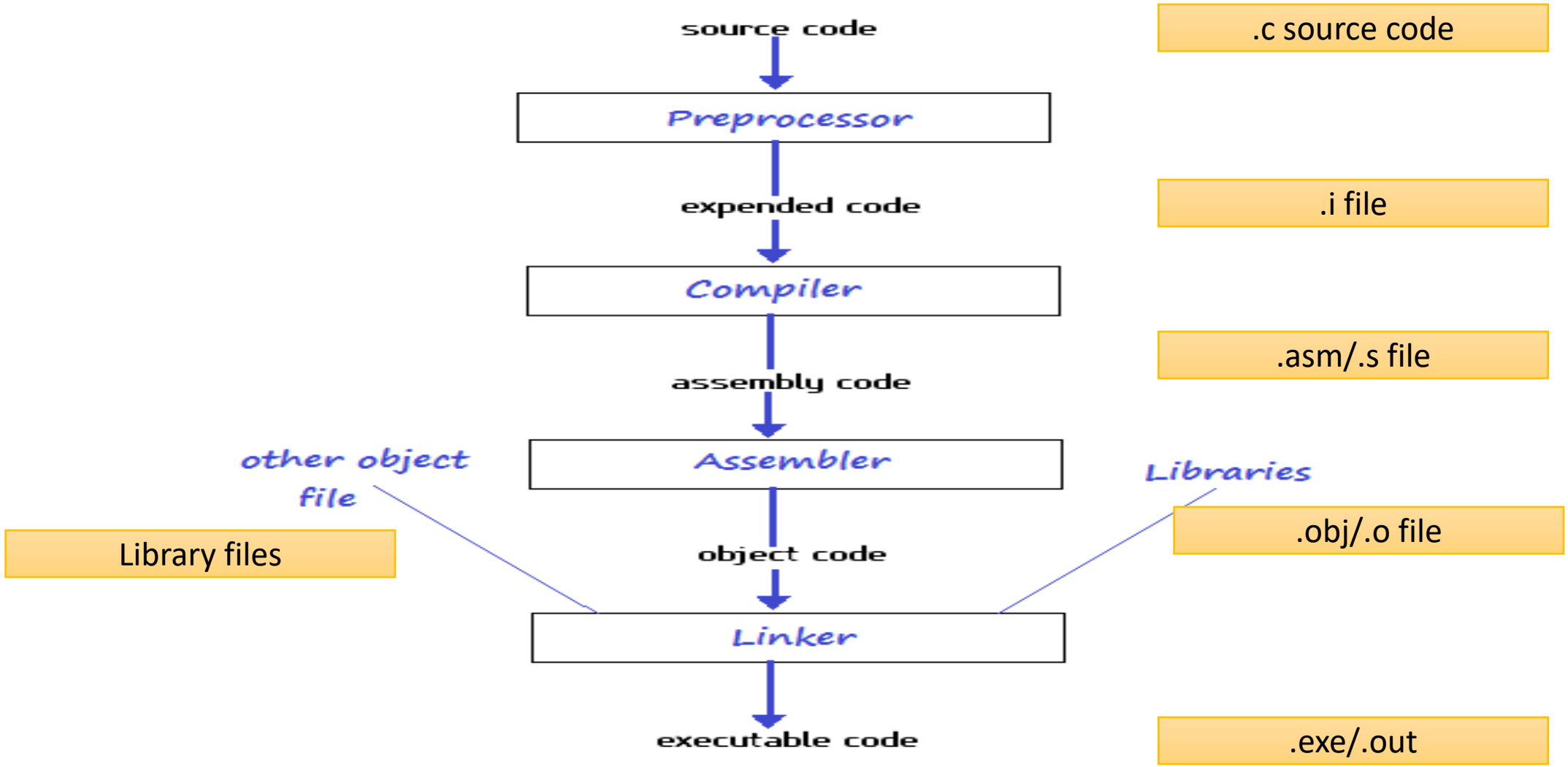
- It is an application software which is a collection of tools/programs required for faster s/w development.

- It contains tools like
  - source code editor
  - build tools
  - preprocessor
  - Compiler
  - Linker
  - Assembler
  - dubugger etc...
- E.g Other editors like netbeans, ms visual studio etc

Compiler

- It is an application program which converts high level programming langauge code (c source code) into the low level programming language code (assembly code).
- e.g. GCC - GNU Compiler's Collection,
- TC - Turbo Compiler -- Borlands

Program Execution Phases



C Tokens

- Keywords
- Identifiers
- Variables
- Data Types
- Character Set
- Constants
- Operators

Keywords

- Keywords are predefined, reserved words used in programming that have special meanings to the compiler.
- Keywords are part of the syntax and they cannot be used as an identifier.

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

Identifiers

- Identifiers give names to variables, functions, defined types and preprocessor macros
- Formed from letters, digits and underscores
- First character must be letter or underscore (but names beginning with underscore normally reserved for internal compiler use)
- Case-sensitve (upper and lower case letters are different)
- Rules of Identifiers:
  - Should start with alphabet or with _ (underscore)
  - Can include alphabets,_ (underscore),digits
  - E.g :
    - Var_1 //Valid
    - 1_var // Not Valid
    - _var //valid
    - Var-1 // invalid

Declaring Variables

- Must be declared before first use
- Format of statement declaring variable(s) is

  modifier type varname1;
- Initial value may be specified when variable is declared (otherwise initial value may be unpredictable)

  int num = 10;
- For local variables, declaration statements must be placed at the beginning of block or function
- For global variables, declaration statements must be placed before first function definition

Data Types

- Data types in c refer to an extensive system used for declaring variables or functions of different types.
- The type of a variable determines how much space it occupies in storage.

Basic data types (primitive)

1.character(char)

2.Integer(int)

3.Single precision floating point (float)

4.Double precision floating point(double)

5.void

Derived data types : Derived from basic data types

1.Array type

2.Pointer type

3.Function type

User defined data types

1.Structure

2.Union

3.Enumeration

Format Specifier

%c	converts data into character
%d	converts data into signed int
%f	converts data into float(6 precisions)
%s	converts data into string
%ld	converts data into long int
%u	converts data into unsigned format
%x	converts data into hexadecimal format
%o	converts data into octal format
%e	converts data into exponential format

printf

- printf is a standard library function used to write formatted output to the standard output device (usually the screen)
- Requires header file *stdio.h*
- Format printf(*formatstring [, value] ...*);
- Should have one format specifier for each value listed, and types should match
- Example - printf("Value = %.2f",temp);

scanf

- scanf is a standard library function used to receive formatted input from the standard input device (usually the screen)
- Requires header file *stdio.h*
- Format scanf(*formatstring [, value] ...*);
- Should have one format specifier for each address passed, and types should match
- Example - scanf("%f",&temp);

Type modifiers

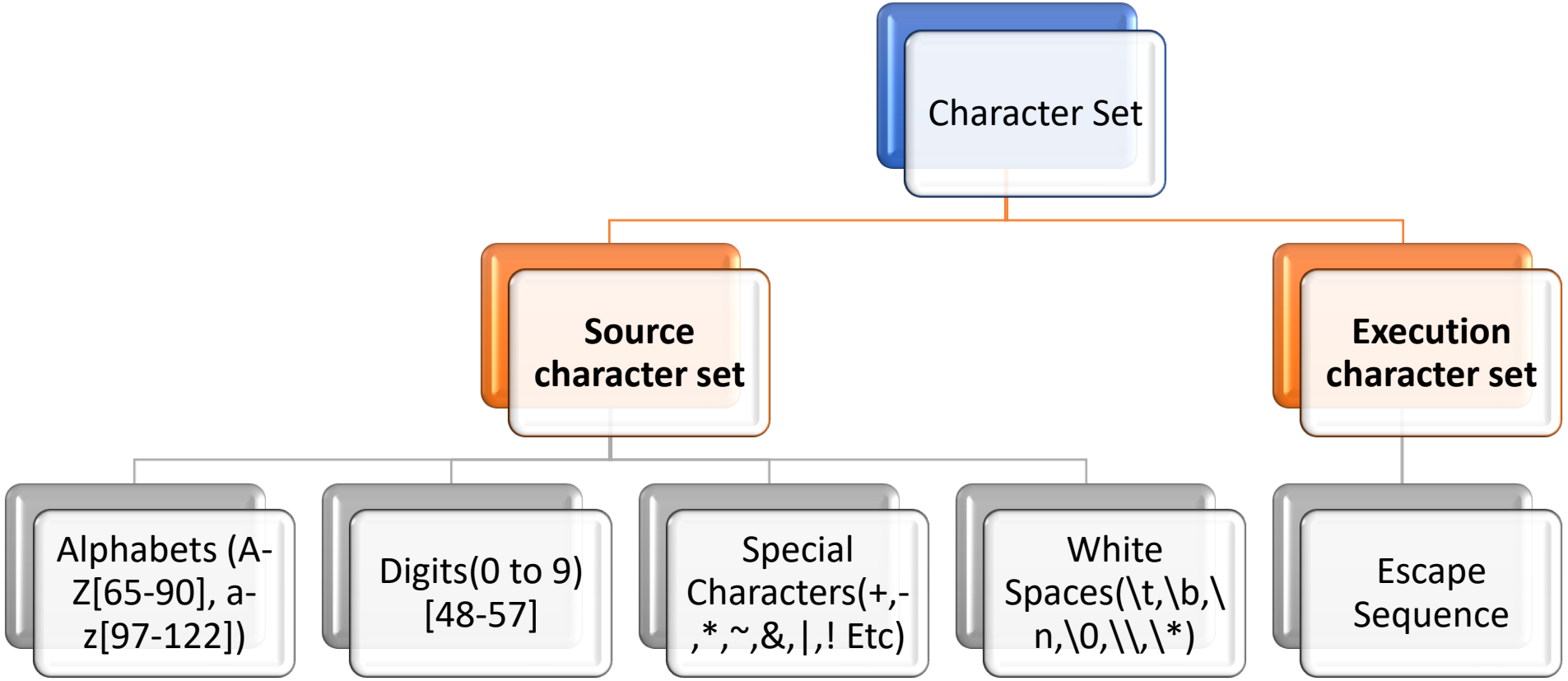
- char (-128 to 127 or 0 to 255)
- unsingned char (0 to 255)
- signed char (-128 to 127) // by defualt each char variable is signed char

- int (-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647)
- unsinged int (0 to 65,535 or 0 to 4,294,967,295)
- short int(-32,768 to 32,767)
- unsingned short(0 to 65,535)

- long (-9223372036854775808 to 9223372036854775807)
- unsinged long(0 to 18446744073709551615)

C Character Set

- The character set is the fundamental raw material of any language and they are used to represent information.
- These characters can be combined to form variables. C uses constants, variables, operators, keywords and expressions as building blocks to form a basic C program.



Escape Sequence

- \n adds new line
- \t adds horizontal tabspace
- \r returns carriage to the beginning of line
- \b moves cursor one character back

(Note: Its effect cant be visualized on eclipse , it can be visualized when we use printer)

- \f formfeed
- \v vertical tab
- \a adds alert

- \\ considers \as a backslash character
- * escape the actual meaning of next character

```
# Escape Sequnce
```

Operators

- Arithmetic Operators (+ , - , * , / , %)
- Assignment & shorthand Operators (= , += , -= , *= , /= , %= , &= , |= , ^= , ~= , <<= , >>=
- Relational Operators (< , <= , > , >= , !=)
- Logical Operators (&&, ||, !)
- Conditional Operator (? :)
- Bitwise Operators (& , | , ^ , ~ , << , >>)
- Unary Operators (+ , - , ++, -- , * , & , ->)
- Special Operator (, comma operator , sizeof() , [ ])

Operators precedence chart

Symbol	Meaning	Associativity
( )	function call	left to right
[ ]	array element	
->	pointer to structure member	
.	structure or union member	

Operators precedence chart

Symbol	Meaning	Associativity
+ -	unary plus or minus	right to left
++ --	pre increment / decrement	
! ~	logical & bitwise NOT	
*	indirection	
&	address of	
Sizeof	sizeof	
(type)	typecast	

Operators precedence chart

Symbol	Meaning	Associativity
* / %	multiplication, division, modulus	left to right
+ -	addition, subtraction	
>> <<	right shift, left shift	
< > <= >=	relational inequalities	
== !=	relational equal, not equal	

Operators precedence chart

Symbol	Meaning	Associativity
&	bitwise AND	left to right
^	bitwise exclusive OR	
|	bitwise OR	
&&	logical AND	
||	logical OR	
? :	conditional	right to left

Operators precedence chart

Symbol	Meaning	Associativity
= += -= *= /= %= <<= >>= &= ^= |=	Assignment & shorthand	right to left
,	comma	left to right

Sunbeam Infotech

Conditional operator

- Evaluates to either of two values based on a logical expression
- *condition* ? s*tmt1* : *stmt2*

  stmt1 is executed *if condition* is true

  else stmt2 is executed.

  e.g. x = **(a > b) ? a : b**; /*max of a or b*/
- Actually an expression, not statement, so it evaluates to certain value & can be used as

  putchar(**(Hide = 'Y') ? '*' : ch**);

Operator ++ / --

- Post Increment/Decrement

x++ => use old value of *x*, then increment *x*

y-- => use old value of *y*, then decrement *y*

- Pre Increment/Decrement

++*x* => increment *x*, then use new value

--*y* => decrement *y*, then use new value

- Operators increment or decrement value of variable by 1.

Bitwise Operator (&, | , ! , ~)

- Bitwise operators fall into two categories:
  - binary bitwise operators : Binary operators take two arguments
  - Unary bitwise operators. : Unary operators only take one.

AND (&)

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x & y	1	0	0	0

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001
  ________
  00001000  = 8 (In decimal)
```

OR (|)

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x | y	1	1	1	0

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  ________
  00011101  = 29 (In decimal)
```

XOR (^)

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
y	1	0	1	0
z = x ^ y	0	1	1	0

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
^ 00011001
  ________
  00010101  = 21 (In decimal)
```

Complement (~)

Variable	b_3	b_2	b_1	b_0
x	1	1	0	0
z = ~x	0	0	1	1

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
~ 00100011
  ________
  11011100  = 220 (In decimal)
```

Bitwise Shift Operator

- The bitshift operators take two arguments, and looks like:
  - x << n : shifts the value of x left by n bits
  - x >> n : shifts the value of x right by n bits

Operator <<

Let's look at an example. Suppose **x** is a char and contains the following 8 bits.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

If we shift left by 2 bits, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	1	0	0

That means that as you shift left, the bits on the high end (to the left) fall off, and 0 is shifted in from the right end.

Operator >>

Let's look at an example of this. Suppose x looks like before:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	0	0	0	1	1	1

Let's shift right by 3 bits. If the sign bit is shifted in, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	1	1	1	1	0	0	0

If 0's are shifted in, the result is:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	1	0	0	0

Program Constructs / Control Statements in C

Program Constructs / Control Statement in C

- **C statements for decision/selection**
  - if...else
  - switch...case
- **C statements for iteration (looping)**
  - for
  - while
  - do...while
- **C statements for jump**
  - break / continue
  - return
  - goto

if statement

```
if ( expression )
{
        stmt1_1
        stmt1_2…
}
```

```
if ( expression ) {
        stmt1_1
        stmt1_2…  }
else {
        stmt2_1
        stmt2_2… }
```

- Expression ➔ condition
- true ➔ expression evaluates to nonzero value
- false ➔ expression evaluates to zero value
- Executes *stmt1* if expression is true (not zero) When else is present executes *stmt2* if expression is false (zero)
- *expression* may be built using relational or mathematical operators
- { } braces are optional and used to group statements into block
- More than one conditions may be checked simultaneously using Logical operators

switch statement

```
switch( expression )
  {
      case constant-expr1:      stmt ...
                                        break;
      case constant-expr2:      stmt ...
                                        break; . . .
      default:                          stmt …
   }
```

- Expression must be integer one
- Used to select one of several paths to execute based on the value of an expression
- *break* statement skips remaining statements in the switch structure and continues execution at the statement following the closing brace.
- default case is optional and executed if result of expression don't match with any of the case constants.

Switch case

1. default clause is optional.
2. case has be followed by integer constant
3. It is necessary to use break statement as a last statement of each case.
4. Break is one of jump statement. It helps to move execution control forcefully out of switch/loop.
5. if we do not include break statement as a last statement in switch execution, the control is given to all next cases even though they are not satisfied.
6. Duplicate case is not allowed
7. Sequence of cases do not matter.

Loop

- Control statements used for repeating a set of instructions number of times is called as “LOOP”.
- Every loop has
  - Initialization statement
  - Terminating condition
  - Modification statement(Increment/Decrement)
  - Body of loop
- The variable that is used in terminating condition is called as ‘loop variable’.

while statement

- Used to repeat a statement (or block) while an expression is true (not zero)
- Syntax

```
initialization;
while ( condition ) {
        stmt1;
        stmt2;
        modification;
}
```

While Example:

```
1.  void main( )
2.  {
3.      int cnt = 1;                    /*Initialization*/
4.      while ( cnt <=10 ) /*condition*/
5.      {
6.              printf("%d\n",cnt);
7.              cnt++;   /*modification*/
8.      }
9.  }
```

for statement

- **Syntax :**

```
for (initialization;condition;modification)
{
        stmt1;
        stmt2;
}
```

```
/*example*/
for(cnt=1 ; cnt<=10 ; cnt++)
{
        printf("%d\n",cnt);
}
```

1. Evaluates *initialization expression*, if present
2. Evaluates *condition expression*, if present
3. If *condition* is true (not zero) or omitted, executes *stmt* once and proceed to next step; *otherwise exits from loop*
4. Evaluates *modification;*
5. *Jump to step 2*

do...while statement

- Used to repeat a statement (or block) while an expression is true (not zero)
- do {

  stmt..

  initialization / modification;

  } while (*condition*);
- Similar to *while* except *stmt* always executes at least once - test for repeating done after each time *stmt* is executed

Infinite loop

- while(1){

  …

  }
- for(j=0; ;j++) {

  …

  }

break statement

- It is a jump statement which can be used in switch / loop.
- break statements helps to move execution control forcefully out of switch / loop
- In *for*, *while*, and *do...while* loops, *break* exits from the lowest-level loop in which it occurs
- Used for early exit from loop, or for exiting a *infinite* loop
- Jump statements :
  - break - can be used inside switch or loop which helps to move execution control forcefully out of loop/switch
  - return - can be used inside function helps to move execution control forcefully back to calling function
  - continue: can be used only inside loop. Helps to move execution control forcefully to next iteration. Whenever continue is encountered all c statements below continue are skipped from evaluation in current iteration from.
  - goto: can be used to mark statements as label and jump to specific label definition with goto we can jump to label definition which is in scope of current function.

continue statement

- The *continue* statement cause immediate loop iteration, skipping the rest of the loop statements
- Jumps to the loop test when used with *while* or *do*
- Jumps to loop increment, then test when used with *for*
- Does not exit the loop (unless next loop test is false)

typedef

- Helps to give alias/another name to existing data type.

  Syntax:

  typedef <existing data type> <alias/another name> ;

  Example:

```
typedef   int whole_number;
typedef int number;                    // typedef <existing data type> <alias/another name> ;
number n1,n2;
```

enum

- enumerated data type.
- enum helps to define user type data.
- It can include members of integer type only.
- Enum is a collection of enumarated fields,s where every field represents integer constant
- In Collection if we do not assign any enumarated field by default first member of collection receives default value as zero
- Every member in enum receives step 1 value ahead of previous one.
- Enum is used to improve readability of source code.

  Syntax:

  enum <tag> {[<fields>] } ;

  enum colors{RED,BLUE=5,GREEN};

  enum colors clr;
- To find size of enum, we use sizeof(enum_name)

Functions / User Defined Functions

- It is a set of instructions written to gather as a block to complete specific functionality.
- Function can be reused.
- It is a subprogram written to reduce complexity of source code
- Function may or may not return value.
- Function may or may not take argument
- Function can return only one value at time
- Function is building block of good top-down, structured code function as a "black box“
- **Writing function helps to**
  - improve readability of source code
  - helps to reuse code
  - reduces complexity
- **Types of Functions**
  - Library Functions
  - User Defined Functions

User Defined Functions

- **Function declaration / Prototype / Function Signature**

  <return type> <functionName> ([<arg type>...]);

- **Function Definition**

  <return type> < functionName > ([<arg type> <identifier>...])

  {

  }

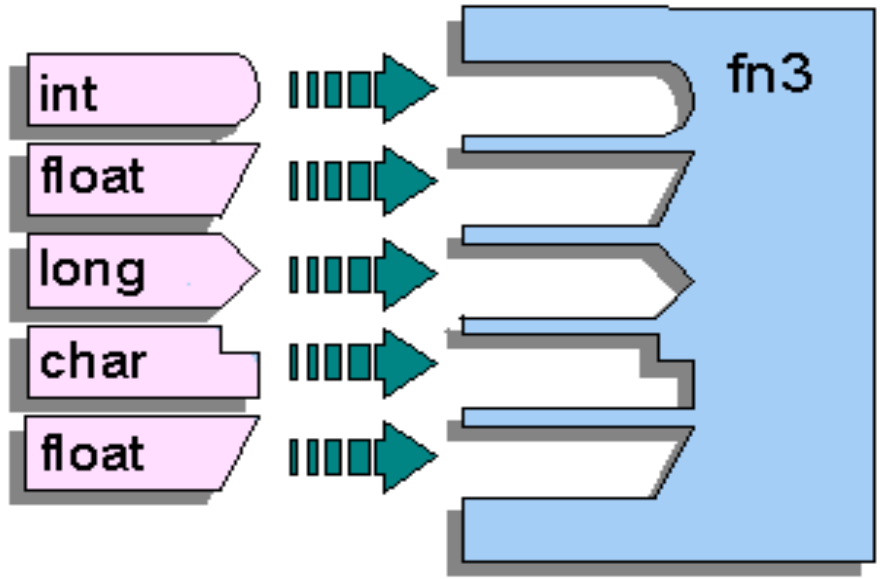
- **Function Call**

  <location> = < functionName >(<arg value/address>);

- **We can pass arguments to function using**
  - pass by value
  - pass by address

Function Argument Must Match

```
int fn3( int, float, long, char, float );
...
   z = fn3( i, 5, 20, ch, z );
...
```




```
int fn3(int a, float b, long c, char d, float e)
{
   int w;
   ...
   return( w );
}
```

- Type and number of arguments should match the function declaration
- Return statement specifies return value, if any, and returns control to the point from which the function was invoked
- Function's return value can be ignored (statement with no assignment)
- Function cannot be defined within a function (can't nest definitions)

Storage Classes

- auto
- register
- static
- extern or global

Variable scope and lifespan

- Scope refers to where in a program a particular variable is known.
- Lifespan refers to how long a variable retains its storage
- Global variables and static variables retain their storage through program execution
- Local variables (auto) are allocated off the stack and retain their storage only while the block in which they are declared is executing

Storage Classes

Variable	Keyword	Default Initial	Scope	Life	Memory From
Local	auto	garbage	block	block	stack
Register	register	garbage	block	block	CPU register
Static	static	zero	block	program	data section
Global	extern	zero	block	program	data section

Static

- Can be declared within function
- It is necessary to initialize static variables at the time of declaration else it violets the rule of static
- Static variable is to be initialized with constant value only.
- Static variables helps to retain state of particular variable through multiple calls of same function.
- static variables are initialized only once on first invocation of particular function in which is declared.

Register

- when register storage class is used we try to request register to identify with some name.
- There is no guarantee that our register request will be entertained.
- As number of registers availability is very less our request may be rejected and it will be automatically converted in auto type. Needs more time and slows down performance
- If request is entertained then programmer can enjoy speed/performance of application.
- we can not apply address of operator on registers.
- We can not request registers other than local scope.
- Use of register in global section is not allowed

- Syntax to declare a register variable:

```
register int regvar;
```

extern

- extern keyword extends the visibility of the C variables and C functions.
- We write **extern** keyword before a variable to tell the compiler that this variable is declared somewhere else. Basically, by writing extern keyword before any variable tells us that this variable is a global variable declared in some other program file.
- A global variable is a variable which is declared outside of all the functions. It can be accessed throughout the program and we can change its value anytime within any function through out the program.
- While declaring a global variable, some space in memory gets allocated to it like all other variables. We can assign a value to it in the same program file in which it is declared.
- **Extern** is used if we want to use it or assign it a value in any other program file.

Recursion

- It is a algorithm in which the one functionality which is called at last will complete its job first (LIFO)
- It is necessary to mention terminating condition else causes runtime error - stack overflow
- Recursion helps to call block of statement repeatedly
- In case of recursion function is called within it's own definition
- Function calling itself is called as *recursive function.*
- For recursive function terminating condition must be given.
- If terminating condition is not given properly, then *stack overflow* occurs.

Loop VS Recursion

Recursion

- uses more memory
- Follows LIFO
- more time consumption
- if no terminating condition is mentioned causes to runtime error stack overflow state

Loop

- less memory utilization
- FIFO
- less time consumption
- if no terminating condition is mentioned causes to infinite loop

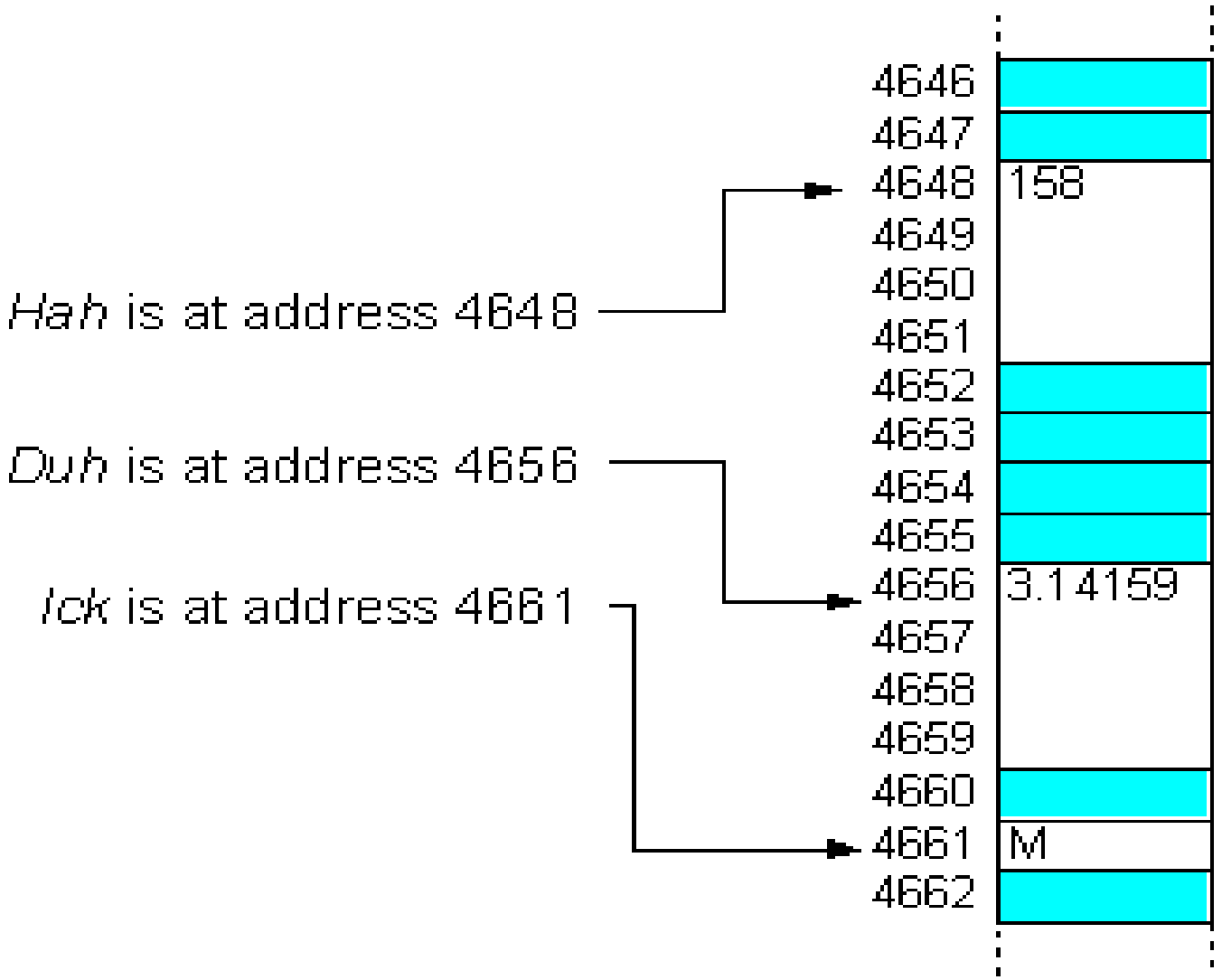
Why Pointers?

- To return more than one value from a function
- To pass arrays &structure more conveniently to functions.
- Concise and efficient array access
- Access dynamic memory
- To create complex data structures
- To communicate information about memory.

Compiler uses memory to access Variables....

Machine instructions that compiler generates use the address to access memory, not the variable name.

```
float Duh = 3.142;
long Hah = 158;
char Ick = 'M';
```



What is Pointer?

- It is known as derived data type.
- A Pointer is a variable that stores address of a memory location.
- Using pointer concept we can create a variable to store address.
- We should always store a valid/in alive state address in pointer.

Operator & and *

- & → called as direction or reference or address of operator and used to get address of any variable

- * → called as indirection or deference or value at operator and used to get value at the address stored in a pointer

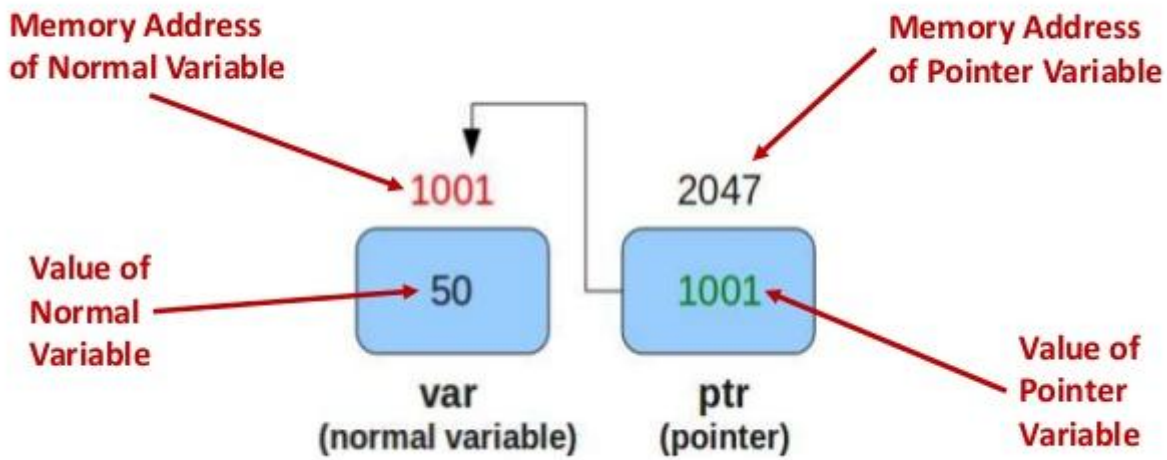
When we use sizeof() operator for pointer variable, it gives output as 8 bytes incase of 64bit compiler and 4 bytes incase of 32 nit compiler.

Reason : Address through system is always given in unsigned int format. Hence size of any pointer will be equals to unsigned int of respective compiler

Example of Pointer

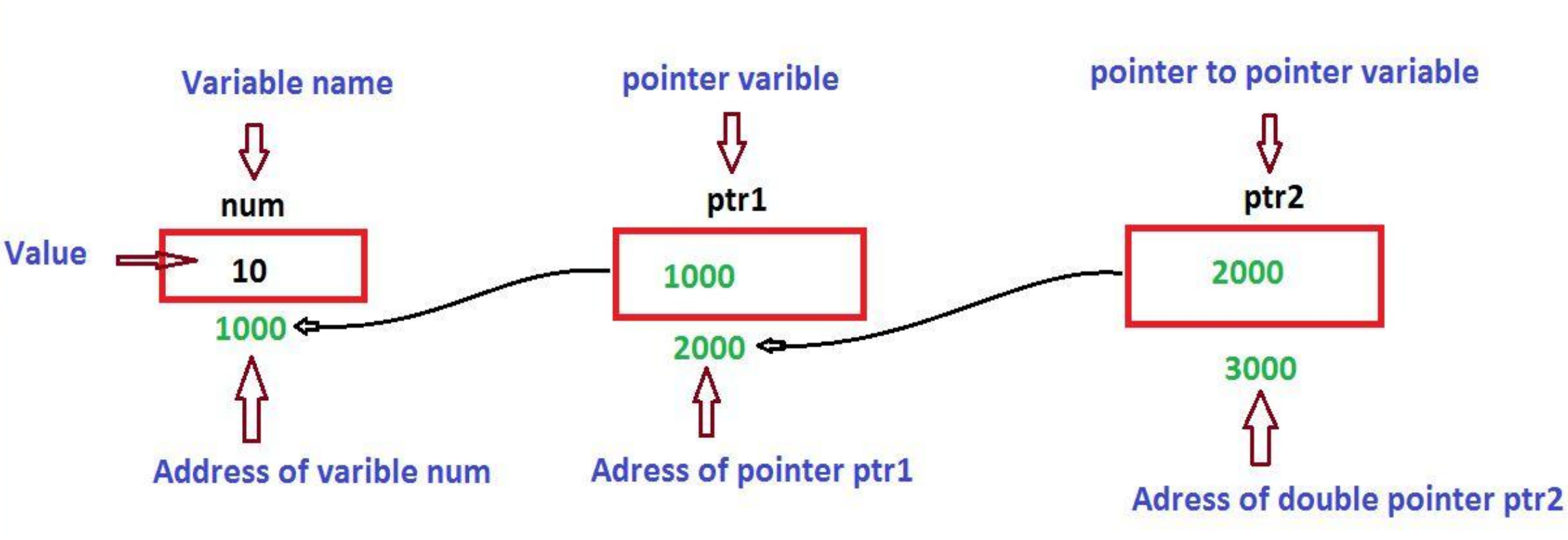
```
void main() {
    int var = 50;
    int *ptr;          /*Declaration*/
    ptr = &var;        /*Initialization*/
    printf("%d",*ptr);/*(50) accessing value*/
    *ptr = 20;                  /*setting value*/

    printf("%d",var);/*(20)*/
}
```



Level of Indirection

- We can declare n indirection level of pointers as shown in diagram.



Void pointer and null pointer

- void * is a pointer which is designed to hold address of any location. It is a generic pointer.
- Example :
  - char *cptr=100 //assumes address is of character whose scale factor is 1byte
  - *cptr derefers 1 byte from given base address 100

- Examle:
  - int *ptr=100; //assumes address is of integer whose scale factor is 4 bytes
  - *ptr derefers 4 bytes from given base address 100

- void pointer is not aware of scale factor i.e. how many bytes to be dereferred from given base address. Hence if we want to receive value at void pointer it is necessary always typecast prior its to use.

Pointer and ++

- A pointer in c is an address, which is a numeric value.
- Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -.
- Icrementing a pointer Variable Depends Upon data type of the Pointer variable.
- new value = current address + i * size_of(data type)

Three Rules should be used to increment/decrement pointer :

1. Address+1=Address
2. Address++ = Address
3. ++Address = Address

Void pointer and null pointer

- void * is a pointer which is designed to hold address of any location. It is a generic pointer.
- Example :
  - char *cptr=100 //assumes address is of character whose scale factor is 1byte
  - *cptr derefers 1 byte from given base address 100

- Examle:
  - int *ptr=100; //assumes address is of integer whose scale factor is 4 bytes
  - *ptr derefers 4 bytes from given base address 100

- void pointer is not aware of scale factor i.e. how many bytes to be dereferred from given base address. Hence if we want to receive value at void pointer it is necessary always typecast prior its to use.

Pointer and ++

- A pointer in c is an address, which is a numeric value.
- Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -.
- Icrementing a pointer Variable Depends Upon data type of the Pointer variable.
- new value = current address + i * size_of(data type)

Three Rules should be used to increment/decrement pointer :

1. Address+1=Address
2. Address++ = Address
3. ++Address = Address

Pointer to function

- Pointer as a function parameter is used to hold addresses of arguments passed during function call.
- This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

Functions returning Pointer variables

A function can also return a pointer to the calling function.
In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function end

Array

- Array is collection of similar data elements in contiguous memory locations.
- It is a collection of similar type elements.
- Elements of array share the same name i.e. name of the array.
- And they are identified by unique index in the array called as subscript.
- Array indexing starts from 0.
- Very first element of array receives by default index as 0 and last (nth position) element receives index as n-1
- Checking array bounds should be done by programmer.
- We can implement array statically or dynamically.
- If it is static implementation of array memory once given can't be shrinked or grown at runtime.
- In case of static implementation base address of array is always locked by compiler.

1D Array Example

```
void main()
{
        int arr[5] = { 1, 2, 3, 4, 5 };
        int j;
        for(j=0;j<5;j++)
                printf("\n%d",arr[j]);
        printf("size = %d",sizeof(arr));
}
```

- &arr[0] = 0x100
- &arr = 0x100
- arr = 0x100
- *arr = 1

Points to be noted

- If array is initialized partially at the time of declaration then elements which are not initialized are set with default value 0(ZERO).
- It is necessary to specify last dimension in case of all dimension of array only in case of 1D array we can skip last dimension providing members of array are initialized at the time of declaration.
- Array bounds is job of programmer.
- When we request array a location is identified with array name stores always first element address of first element because of which array interchangeability with pointer is possible

Passing Array to Function

- Arrays are passed to function by address.
- The address of starting element of array (Base address) is passed to the function.
- The address is collected in a pointer.
- We can use pointer as well as array notation to access array elements in the function.
- In a function definition, a formal parameter that is declared as an array is actually a pointer.
- When an array is passed, its base address is passed call-by-value.
- The array elements themselves are not copied.
- As a notational convenience, the compiler allows array bracket notation to be used in declaring pointers as parameters.

Pointer Arithmetic

- When pointer is incremented or decremented by 1, it changes by the size of data type to which it is pointing (scale factor)
- When integer ‘n’ is added or subtracted from a pointer, it changes by ‘n’ times size of data type to which pointer is pointing.
- Multiplication or division of any integer with pointer is meaning less.
- Addition, multiplication and division of two pointers is meaning less.
- Subtraction of two pointers have meaning only in case of arrays.

Array and Pointer

- An array name represents the address of the beginning of the array in memory.
- When an array is declared, the compiler must allocate a base address and a sufficient amount of storage (RAM) to contain all the elements of the array.
- The base address of the array is the initial location in memory where the array is stored.
- It is the address of the first element (index 0) of the array.

- Example:
  - int arr[5] ;

  - arr[i] is internally resolved as *(arr+i)
  - arr[i] == *(arr+i)
  - *(i+arr) == i[arr]

  - int *ptr=arr;
  - *(ptr+i) == ptr[i]
  - *(i+ptr) == i[ptr]

Accessing array Elements using Pointer

```
int main(void) {
    int arr[5], *ptr, j;
    ptr = arr ;
    printf("Enter 5 elements..");
    for(j=0;j<5;j++)          {
            scanf("%d", ptr);
            ptr ++;
    }
    ptr = arr;
    for(j=0;j<5; j++, ptr++)
            printf("\n%d",*ptr);
}
```

Pointer and Constant

1. Show that if we do not declare variable as constant then we can modify it.
2. Demo of const variables are assigned values at the time of declaration.
3. Declare one pointer variable and try to modify value of constant (float *fptr; fptr=&PI;)
4. const float PI=3.14;

   const float *fptr;

   fptr=&PI; / /*fptr=40.5 // not allowed as *fptr is constant

   float fval=48.23; fptr=&fval; print(fval,*fptr)

   Conclusion is : value at fptr where it is pointing to is constant where as fptr is not constant, initially fptr is pointing to int const later it points to char const.
5. const float * const fptr=&PI;

   we cant do following:

   -change the value inside pointer

   -change the address location of ptr

   Conclusion : value at fptr where it is pointing to is constant where as fptr is also constant

String

- Character array and String
  - Character Array : It is collection of characters
  - String:It is collection of characters where always last element is NULL.
- Example :

```
char arr[5] = “abcde”;
int j;
for(j=0; j<5; j++)
        printf(“%c”,arr[ j ]);
```

- Accepting String as a input

```
char str[20];
scanf(“%s”,str);            /*Input*/
printf(“%s”,str);           /*Output*/

char str[20];
gets(str);                  /*Input*/
puts(str);                  /*Output*/
```

String Scan Sets

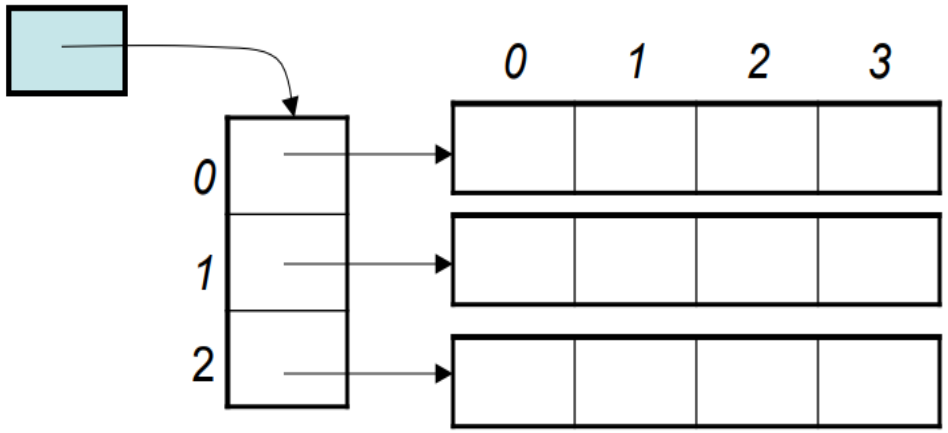
- %s
- %[^\n]s //scan upto \n (single line)
- %[^.]s // scan upto . (multiple line)
- %[0-9]s // scan upto digits
- %[^0-9]s // scan upto aplhabets
- %[A-Z]s // scan upto capital
- %[^a-z]s // scan upto capital
- %[^A-Z]s // scan upto small letter
- %[a-z]s // scan upto small letter

String Functions

- String operations are done by library functions.
- The functions are declared in “string.h”
- The functions are –
  - strlen
  - strcpy
  - strcat
  - strcmp
  - Strrev
  - Strstr
  - strchr

2D Array

- Arrays that we have consider up to now are onedimensional arrays, a single line of elements.
- Often data come naturally in the form of a table, e.g., spreadsheet, which need a two-dimensional array.
- Two-dimensional (2D) arrays are indexed by two subscripts, one for the row and one for the column.
- Example: int a[3][5];
  - Logically it may be viewed as a two-dimensional collection of data, three rows and five columns, each location is of type int.



	0	1	2	3	4
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

A 2D array is a 1D array of (references to) 1D arrays.

2D Array Declaration

- **Valid Declarations :**

1. int mat[2][2]={{1,1},{1,2},{2,1},{2,2}}; //allowed
2. int mat1[ROW][COL]={{1,1},{1,2},{2,1},{2,2}}; //allowed
3. int mat3[][COL]={{1,1},{1,2},{2,1},{2,2}}; // allowed
4. int mat4[2][2];

- **Invalid Declarations :**

1. int mat[][]={{1,1},{1,2},{2,1},{2,2}};// not allowed
2. int mat2[ROW][]={{1,1},{1,2},{2,1},{2,2}}; //not allowed

Pointer Arithmetic

- Increment operator when used with a pointer variable returns next address pointed by the pointer.
- The next address returned is the sum of current pointed address and size of pointer data type.
- Decrement operator returns the previous address pointed by the pointer.
- The returned address is the difference of current pointed address and size of pointer data type.
- Array can be interchangeably used with pointer that is called as Pointer arithmetic.

Command Line Argument

- The C language provides a method to pass parameters to the main() function.
- This is typically accomplished by specifying arguments on the operating system command line.
- The prototype for main() looks like:
  - int main(int argc, char *argv[]) { … }
  - The first parameter is the number of items on the command line (int argc).
  - argc always retains the count of arguments passed to main
  - Each argument on the command line is separated by one or more spaces, and the operating system places each argument directly into its own null-terminated string.
    - Note: The name of the program is counted and is the first value.
    - Note: Values are defined by lists of characters separated by whitespace.
  - The second parameter passed to main() is an array of pointers to the character strings containing each argument (char *argv[]).
  - Argv catches actual arguments passed at command prompt to main function
    - Note: The array has a length defined by the number_of_args parameter.
- If we add **char **env** an argument to main it will display list of environment variables.
- Environment variables are used for information about your home directory, terminal type, and so on; you can define additional variables for other purposes. The set of all environment variables that have values is collectively known as the *environment*.

Standard main() prototypes

- int main();
- int main(int argc, char *argv[]);
- int main(int argc,char*argv[],char*env[]);
- argc represents number of arguments passed to program when it is executed from command line.
- argv represents argument vector or argument values.
- envp represents system information.

Pre-processor Directives

- Preprocessor is small application helps to generate intermediate code.
- It process all c statements starting with # which is called as preprocessor directives.
- Preprocessor directives are the commands given preprocessor about processing source code before compilation.
- These are categorized as
  - Macro substitution (#define)
  - File inclusion (#include)
  - Conditional compilation (#if, #else, …)
  - Others (#pragma, #, ##)
- Examples: #define
- can be used for defining symbolic constant or macro
- #define <symbol> <replacable text>
- #undef #if #else #endif #pragma

Macro substitution

- #define PI 3.14
- #define SQR(x) (x)*(x)
- #define SWAP(x,y,type) {type temp; temp = x ;x=y; y=temp;}

File inclusion

- #include <stdio.h>
- #include “myheader_file.h”

Conditional compilation

- #if
- #else
- #elif
- #endif
- #ifdef
- #ifndef
- Example:

```
#define PI 3.14 ,
#if defined (PI)
    printf("DEFINED");
#else
        printf("NOT DEFINED");
#endif
```

Other Directives

- #error
- #line
- #pragma
- Stringizing operator #
- Token pasting operator ##
- Example:
  - #define STRDISP(x) {printf("\n %s ",#x);}
  - #define STR_GET_DATA(x) {printf("\n %s ",#x+4);}

Predefined constants

- __LINE__
- __FILE__
- __TIME__
- __DATE__

- INT_MAX
- CHAR_MAX

Structure

- It helps to collect members/fieldsof similar or disimilar type.
- It can be used to define data type.
- In case of structure every member receives memory separately.(i.e Each element has its own storage)0
- Structure is collection of non-similar data elements in contiguous memory locations.
- Structure is user defined data type.
- Members of structures can be accessed using "." operator with structure variable.
- Members of structures can be accessed using "→" operator with pointer to structure variable.
- Syntax of structure

```
struct <tagname>
{
        <data type> <identifier> ;
        ....
};
```

Structure Declaration

```
/*Structure declarations are generally done before main i.e. global declaration. We can also do it in a
   function.*/
struct student {
    int roll_no;
    char name[10];
    float avg;
};

/*initializing  structure variable at its declaration*/
void main() {
    struct student s1={1,"Akshita",80};
    printf("size=%d",sizeof(s1));
    printf("roll=%,name=%s,avg=%f",s1.roll, s1.name, s1.avg);
}
```

Nested structures

- One can define a structure which in turn can contain another structure as one of its members.
- Example:

```
typedef struct
{   int dd;  int mm;    int yy;  }DATE;
typedef struct
{   int rollno;
    int marks;
    struct
    {           char fname[10];
                char mname[10];
                char lname[10];
    }name;
    DATE dob;
}STUDENT;
```

Arrays of structure

- C does not limit a programmer to storing simple data types inside an array. User defined structures too can be elements of an array.
- <keyword_struct> <struct_name> <struct_variablename>[sizeof_elements];
- struct student s[10];

Passing Structure to a Function by Value and by Reference

- A structure can be passed to any function from main function or from any sub function.
- Structure definition will be available within the function only.
- Example

```
struct student
{ int rollno; int age; };
 void display(student s);  // passing structure by value in function argument
//display(st); // calling function

 void show(student *s); // passing structure by reference in function argument
//show(&st); // calling function
```

Structures and Pointers

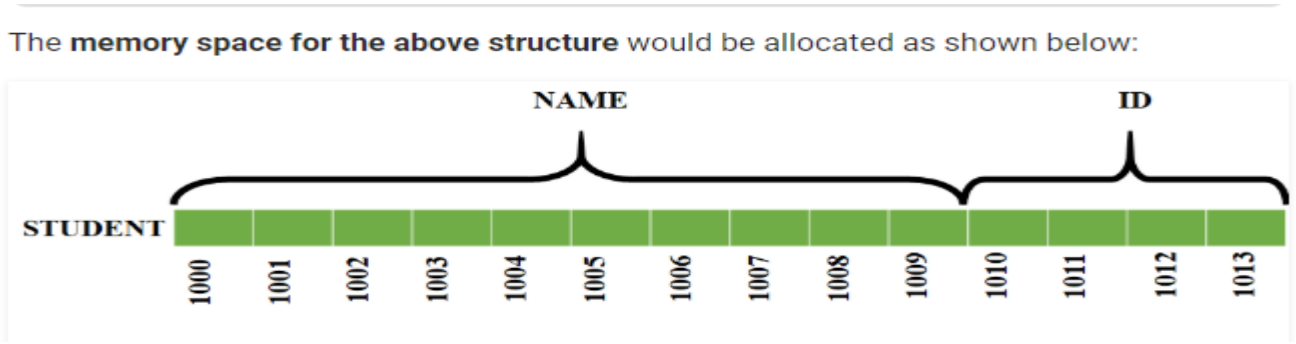
- Just like a variable, you can declare a pointer pointing to a structure and assign the beginning address of a structure to it.

Pointer to Structure

```
void main()
 {
        struct student s1={10,”Akshita”,78.67};
        struct student *ptr = &s1;
        printf(“size=%d”,sizeof(ptr));
        printf(“roll=%d,nm=%s,avg=%f”,ptr→roll,     ptr→name, ptr→avg);
}
```

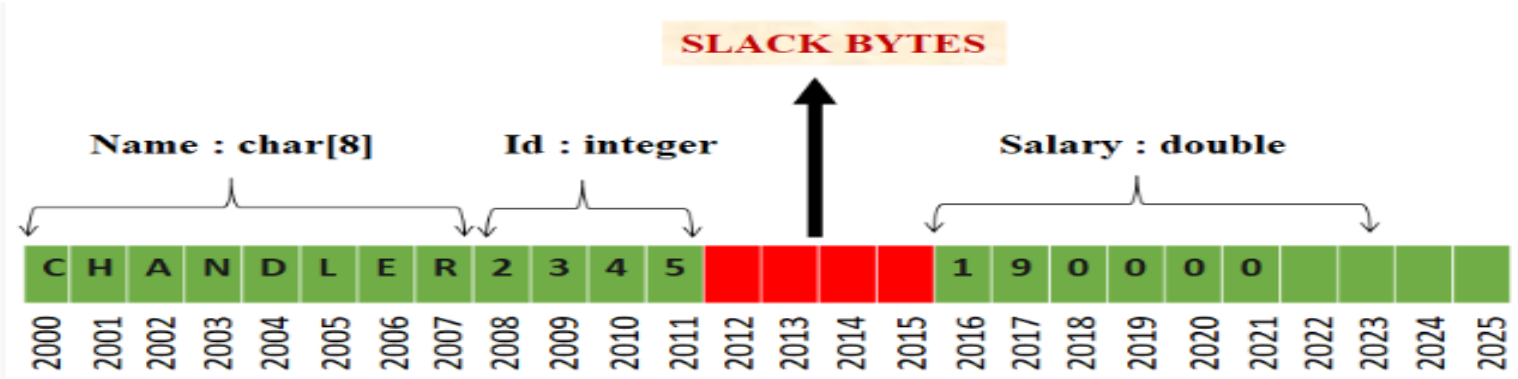
Slack Bytes

- Structures are used to store the data belonging to different data types under the same variable name.
- struct STUDENT { char[10] name; int id; };



•Here we see that there is no empty spaces in between the members of the structure. But in some cases **empty spaces** occur between the members of the structure in and these are known as ***slack bytes***.

```
Struct EMPLOYEE
{
char name[8];
int id;
double salary;
}
```



Bit field

- In programming terminology, *a bit field is a data structure that allows the programmer to allocate memory to structures and unions in bits in order to utilize computer memory in an efficient manner*.
- Example:

```
typedef struct
{
    char name[20];
    int rn:5;
    int marks:4;
}STUDENT;
```

Union

- A **union** is a special data type available in C that allows to store different data types in the same memory location.
- We can define a union with many members, but only one member can contain a value at any given time.

- **Syntax Union Declaration**
- union [union tag]
- { member definition;
- member definition;
- ... member definition;
- } [one or more union variables];

File Functions

- Fopen() --- helps to load file in memory file can be loaded with different modes

- Fclose() --- helpsto unload file
- ftell() -- provides current file pointer position
- fseek () -- helps to reposition filepointer

  using three constants

  1. SEEK_SET 0
  2. SEEK_CUR 1
  3. SEEK_END 2

- **Types of files**
  1. Text File
  2. Binary File

- **Text File I/O**
  - fgetc / fputc : helps to read/write a character / byte to file
  - gets / fputs : helps to read/write buffer size data to file
  - fscanf/ fprintf : helps to read/write of different type
- **Binary File I/O**
  - fread / fwrite : helps to read/write block of byte in binary format to file

- **Modes:**

text		**binary**	
r	or	rb	helps to read file only
w	or	wb	helps to only write
a	or	ab	helps to always write data at end
a+	or	ab+	helps to always write data at end as well read
r+	or	rb+	helps to read/write to file
w+	or	wb+	helps to write and read from file

Various file operations like ftell, fseek(SEEK_SET, SEEK_CUR,SEEK_END)

- Steps:

  1. open a file in read mode and check for file availability

  2. use various options to know cursor position.

- ftell(fptr) by using options like

  a) ftell(fptr) %u

  b) ch=fgetc(fptr); print(ftell(fptr) %u ch %c)

  c) fseek(fptr,10,SEEK_SET); ch=fgetc(fptr); print(ftell(fptr) %u ch %c)

  d) fseek(fptr,-5,SEEK_CUR); ch=fgetc(fptr); print(ftell(fptr) %u ch %c)

  e) fseek(fptr,10,SEEK_END); ch=fgetc(fptr); print(ftell(fptr) %u ch %c)

Thank You