C PROGRAMING

by Ketan Kore @Sunbeam Infotech

About Trainer

- Mr. Ketan Kore
  - Trainer at Sunbeam Infotech
  - M.Sc. Computer science
- Training
    - PreCAT(Training + lab)
    - DAC course(LAB support)
- Contact
  - Email: ketan.kore@sunbeaminfo.com
  - Mobile: 8805822402

Computer and Program

- What is Computer?
  - It is a machine/hardware/digital device which does various tasks for the user efficiently and effectively.
- • What is Program?
  - Set of instructions given to the machine to do specific task.

Classification of Languages

- **The low-level language** is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions.
- **The high-level language** is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.

Low level Languages

- Machine-level language
  - **The machine-level language** is a language that consists of a set of instructions that are in the binary form 0 or 1. As we know that computers can understand only machine instructions, which are in binary digits, i.e., 0 and 1, so the instructions given to the computer can be only in binary codes.
  - **Advantages:**
    - Performance is good as we are directly writing the program on machine
  - **Disdvantages:**
    - Machine dependent
    - Difficult to program
    - Error prone

- Assembly Language
  - The assembly language contains some human-readable commands Alphanumeric(Alphabet+numbers) codes, The language was introduced in 1952.
  - As we know that computers can only understand the machine-level instructions, so we require a translator that converts the assembly code into machine code. The translator used for translating the code is known as an assembler.
  - **Advantages**
    - Easier to understand and use and to locate errors
    - Easier to modify
  - **Disadvantages**
    - Machine dependent
    - Knowledge of hardware required
    - Machine level coding required

High Level Languages

- The high-level language is a programming language that allows a programmer to write the programs which are independent of a particular type of computer. The high-level languages are considered as high-level because they are closer to human languages than machine-level languages.
- A compiler is required to translate a high-level language into a low-level language.
- **Advantages:**
  - They are machine independent
  - They do not require programmer to know anything about hardware
  - They do not deal with machine level coding

- **Disadvantages:**
  - It takes additional translation times to translate the source to machine code.
  - High level programs are comparatively slower than low level programs.

C programming Language

- C programming Language is an High level Language
- C is a general-purpose programming language that is extremely popular, simple, and flexible to use.
- Machine Independent or Portable
- C language include low-level access to memory has simple set of keywords.
- C language is the most widely used language in operating systems and embedded system development today.

History

- C language was developed by Dennis Ritchie in 1972 at AT & T Bell Labs on PDP-11 machine.
- It was developed while porting UNIX from PDP-7 to PDP-11.
- Many features of C are inspired from B (Ken Thompson) and BCPL (Martin Richards).
- Initial release of C is referred as K & R C.

Standardization

- C was standardized by ANSI in 1989. This is referred as C89.
- Standardization ensures C code to remain portable.
- C standard is revised multiple times to add new features in the language.
  - C89 – First ANSI standard
  - C90 – ANSI standard adopted by ISO
  - C99 – Added few C++ features like bool, inline, etc.
  - C11 – Added multi-threading feature.
  - C17 – Few technical corrections.

Introduction

- High-level
- Compiled
- Procedural
- Block-Structured (control structures).
- Typed
- Library Functions

Features

- Data types
- Operators
- Control structures
- Functions
- Storage classes
- Pointers
- Arrays
- Strings
- Dynamic memory allocation
- Structures
- Unions
- Enums
- File IO
- Preprocessor directives

Strengths

- Low level memory access (pointers, data structures)
- Effective memory access (bitwise operators, bit-fields, unions)
- Can access OS features (functions/commands)
- Extensive library functions (math, strings, file IO, …)
- Compilers for different platforms & architectures
- Highly Readable (macros, enum, functions, …)

Applications

- System programming
  - OS development
  - Device drivers
  - System utilities
- Language development
  - Compiler development
- Achievements (tiobe.com)
  - In top-2 languages in last 40 years.
  - Language of year: 2019, 2017, 2008.

Toolchain & IDE

- Toolchain is set of tools to convert high level language program to machine level code.
  - Preprocessor
  - Compiler
  - Assembler
  - Linker
  - Debugger
  - Utilities
- Popular compiler (toolchains)
  - GCC
  - Visual Studio
- IDE – Integrated development environment
  - Visual Studio
  - Eclipse
  - VS Code (+ gcc)
  - Turbo C
  - Anjuta, KDevelop, Codeblocks, Dev C++, etc.

Compilation & Execution of C program

- Preprocessor:-
  - A preprocessor accepts the inputs in source language and produces output source program that is acceptable to the compiler.
  - Main task of preprocessor is to remove the comments and handle all the statements starting with #

- Linker:-
  - Single programmer can write small program and store it in single source code file , Hovewer Large size softwares consist of several thousands and several millions of code , To take care of this approach software developers generally follow the modular approach.
  - In modular approach software consist of mutliple source file , In this case we use the software called as linker to combine all object programs and to convert into final executable.
  - Linker is a software that takes multiple object files and fits them together to assemble into final executable

Hello World

- Source Code

```
// Hello World program
#include <stdio.h>
int main() {
    printf("Hello World\n");
    return 0;
}
```

- Commands
  - cmd> gcc hello.c
  - cmd> ./a.exe

Hello World

- printf() – library function
- stdio.h – header file
- main() – entry point function
  - void main() { … }
  - int main() { … }
  - int main(void) { … }
  - int main(int argc, char *argv[]) { … }
  - int main(int argc, char *argv[], char *envp[]) { … }
- return 0 – exit status

Tokens

- Smallest Individual unit of the program is called as token
- C program is made up of functions.

- Function is made up of statements.

- Statement contain multiple tokens.
  - Keywords
  - Data Types
  - Identifiers
  - Variables
  - Constants
  - Operators

Keywords

- Keywords are predefined words used in program, which have special meanings to the compiler.
- They are reserved words, so cannot be used as identifier.
- K & R C has 27 keywords. C89 added 5 keywords. C99 added 5 new keywords.

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

Identifiers

- Identifiers give names to variables, functions, defined types and pre-processor macros.

- Rules of Identifiers:
  - Should start with alphabet or with _ (underscore)
  - The first character of an identifier cannot be digit it should be letter (either uppercase or lowercase)
  - Can include alphabets, _ (underscore),digits
  - Case sensitive

- Examples:
  - Var_1 //Valid
  - 1_var // Not Valid
  - _var //valid
  - Var-1 // invalid
  - Basic Salary //invalid

Data Types, Variables & Constants

- C allows computations to be performed on various types of data.
  - Numerical: Whole numbers, Real numbers
  - Character: Single character, Strings

- Fixed data values are said to be constants.
  - 12, -45, 0, 2.3, 76.9, 'A', "Sunbeam", etc.

- Data is hold in memory locations identified by names called as variables.
  - Variable must be declared before its use in the program.
  - As per need, variable have some data type.

- Simple C data types are: int, double, char.
  - Data type represents amount of space assigned to the variable.
  - It also defines internal storage of the data.

printf()

- Arbitrary strings and variable values can be printed using printf() function.
  - int - %d
  - double - %lf
  - char - %c
  - float - %f

- Examples:
  - printf("Hello PreCAT @ Sunbeam");
  - printf("%d", roll_number);
  - printf("%d %lf %c", number, basic_salary, letter);
  - printf("Book price is %lf", price);

Data Types

- Data type defines storage space and format of variable.
- Primitive types
  - int
  - char
  - float
  - Double
- Type Modifiers
  It modifies the range of base type
    - Signed
    - Unsigned
    - Short
    - Long
- Integer types can be signed and unsigned
- Derived types
  - Array
  - Pointer
  - Function
- Type qualifiers
  There are used to indicate special properties
  - const and volatile
- User defined types
  - struct
  - union
  - enum
- void type – represent no value.

FORMAT SPECIFIERS

- char - %c
- int - %d,%i
- float - %f
- double - %lf
- long int - %ld
- short int - %hd
- unsigned long - %hu
- unsigned short - %hu
- string type - %s
- Pointer type - %p

Data Types

- **char**
  - **signed char (-128 to 127)**
  - **unsigned char (0 to 255)**
- int / long (32-bit)
  - signed int (-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647)
  - unsigned int (0 to 65,535 or 0 to 4,294,967,295)
- short int
  - signed short (-32,768 to 32,767)
  - unsigned short(0 to 65,535)
- long long / long (64-bit)
  - signed long (-9223372036854775808 to 9223372036854775807)
  - unsinged long(0 to 18446744073709551615)
- float: ± 3.4E +/- 38
- double: ± 1.7E +/- 308

Data Types

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	- 9,223,372,036,854,775,808- 9,223,372,036,854,775,807
long long	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807	8	9,223,372,036,854,775,808- 9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

Constants

- Integer Constants
  - If integer contants starts with 0 it is assumed to be in octal number system
  - If integer constant starts with 0x or 0X then it is assumed to be in hexadecimal number system 0x23 or 0X23 is valid which means 23 in hexadecimal number system is equivalent to 19 in decimal
  - If integer constant has suffix character l,L,u,U,f and F , If integer constant is terminated with l,L then it is assumed to Long , if it is terminated with u or U then it is assumed to unsigned int

    23l is a long integer , 23u is a unsigned integer , f and F can be used with floating point not with integer

- Float constant
  - Floating point constant by default assumed to be of double eg:23.45 is a double
  - Floating point constant if suffixed with f or F is considered to be of type float instead of double 23.45 is double and 23.45f or 23.45F is float

printf() and scanf()

- #include <stdio.h> -- function declaration
- scanf()
  - Used to input values from user.
  - Same format specifiers as of printf().
  - Do not use any char other than format specifiers in format string.
  - To skip a char from input use %*c.

ASCII Chart

0		1	☺	2	☻	3	♥	4	♦	5	♣	6	♠	7	•	8	◘	9	○	10	◙	11	♂	12	♀
13	♪	14	♫	15	☼	16	►	17	◄	18	↕	19	‼	20	¶	21	§	22	▬	23	↨	24	↑	25	↓
26	→	27	←	28	∟	29	↔	30	▲	31	▼	32		33	!	34	"	35	#	36	$	37	%	38	&
39	'	40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/	48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7	56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?	64	@
65	A	66	B	67	C	68	D	69	E	70	F	71	G	72	H	73	I	74	J	75	K	76	L	77	M
78	N	79	O	80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W	88	X	89	Y	90	Z
91	[	92	\	93	]	94	^	95	_	96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o	112	p	113	q	114	r	115	s	116	t
117	u	118	v	119	w	120	x	121	y	122	z	123	{	124	¦	125	}	126	~	127	⌂	128	Ç	129	ü
130	é	131	â	132	ä	133	à	134	å	135	ç	136	ê	137	ë	138	è	139	ï	140	î	141	ì	142	Ä
143	Å	144	É	145	æ	146	Æ	147	ô	148	ö	149	ò	150	û	151	ù	152	ÿ	153	Ö	154	Ü	155	¢
156	£	157	¥	158	₧	159	ƒ	160	á	161	í	162	ó	163	ú	164	ñ	165	Ñ	166	ª	167	º	168	¿
169	⌐	170	¬	171	½	172	¼	173	¡	174	«	175	»	176	░	177	▒	178	▓	179	│	180	┤	181	╡
182	╢	183	╖	184	╕	185	╣	186	║	187	╗	188	╝	189	╜	190	╛	191	┐	192	└	193	┴	194	┬
195	├	196	─	197	┼	198	╞	199	╟	200	╚	201	╔	202	╩	203	╦	204	╠	205	═	206	╬	207	╧
208	╨	209	╤	210	╥	211	╙	212	╘	213	╒	214	╓	215	╫	216	╪	217	┘	218	┌	219	█	220	▄
221	▌	222	▐	223	▀	224	α	225	ß	226	Γ	227	π	228	Σ	229	σ	230	µ	231	τ	232	Φ	233	Θ
234	Ω	235	δ	236	∞	237	φ	238	ϵ	239	∩	240	≡	241	±	242	≥	243	≤	244	⌠	245	⌡	246	÷
247	≈	248	°	249	∙	250	·	251	√	252	ⁿ	253	²	254	■	255									

Escape sequences

- \’ - Single quotation mark — prints ’
- \” - Prints Double quotation mark — prints”
- \\ - Backslash Character — Prints\
- \a - Alert — Alerts by Generating beep
- \b - Backspace — Movers cursor one postion to the left of its current position
- \f - Form Feed — Moves cursor to the beginning of next page
- \n - New line — Moves cursor to the beginning of next line
- \r - Carraige return — Moves the cursor to the beginning of current line
- \t - Horizontal tab — Moves the cursor to next horizontal tab stop
- \v - Vertical tab — Vertical tab

Operators

- An expression in C is made up of operands , for eg a= 2 + 3 is a meaningful expression which involves 3 operands 2 , 3 , a and 2 operators i.e =,+ , Thus expression is sequence of Operators and operands
- Precedence of Operators:-Each Operator in C has a precedence associated with it , In a compound expression operator involved are of different precedence so operator with highest priority is evaluated first .
- Associativity : In compound expression when several operator are of same precedence operators are evaluated according to there associativity either left to right or right to left.

Classification of operators

- Unary Operators – Unary Operator operates on only one operand for example in the expression -3 – is a unary minus operator examples of unary operator are &,sizeof,!(logical negation),~(bitwise negation),++(increment),--(decrement) operator
- Binary Operators – Binary operator operates on 2 operands for example expression 2-3, - acts as a binary minus operator as it operates on 2 operands 2 and 3 for exampe *,/,<<(left shift),>>(Right shift),Logical And(&), Bitwise And(&)
- Ternary Operator – A ternary operator operates on three operands for example Condtional operator (?:) is the only ternary operator in C

- Classification Based on operator

  Based on there role operators are classified as
  - Arithmetic Operators
  - Relational Operators
  - Logical Operators
  - Bitwise Operators
  - Assignment Operators

Operators Precedence and Associativity

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Arithmetic operators

- Arithmetic operators work with all primitive data types i.e. int, float, char, double.
- Precedence of * & / is higher than + & -.

- % operator doesn't work with float and double type.
- % operator follows sign of numerator

- If two operands are of different types, the lower type is promoted temporarily for computation.

- char and short are promoted is promoted temporarily for computation.

- Char types are treated as integers (ASCII values) for calculation.

- If result exceed range of data type (overflow), then it rollback.

Short-hand operators

- Short-hand operators will change value in variable.
- +=, -=, …
  - num+=2;
  - num=+2;
  - num-=2;
  - num=-2;

- Pre-increment/decrement
  - x = ++a;
  - y = --b;

- Post-increment/decrement
  - x = a++;
  - y = b--;

Comma, Relational and logical operators

- Comma operator
  - evaluate to right-most value.
  - have lowest precedence.

- Relational and logical operators result in 0 or 1.
  - 0 – indicate false condition
  - 1 – indicate true condition
- Relational operators
  - <, >, <=, >=, ==, !=
- Logical operators
  - &&, ||, !

Logical operators

- Logical operators
  - &&, ||, !

P	Q	P && Q	P || Q	!P
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

  - Logical operators operate according to the truth table given above
  - Logical AND and Logical OR operator guarantee left to right evaluation
  - Logical NOT OperatorIt is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.

Bit-wise operators

The C language provides six operators for bit manipulation they operate on the individual bits of the operands . The Bitwise operators available in C are

- Bitwise AND &

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND operators on the individual bits of the operand according to the truth table shown above

Example : - 10 & 5

0000 1010 -> Binary of 10

0000 0101 -> Binary of 5

0000 0000 → O/P is 0

- Bitwise OR

x	y	x | y
1	1	1
1	0	1
0	1	1
0	0	0

Bitwise OR operators on the individual bits of the operand according to the truth table shown above

Example : - 10 | 5

0000 1010 -> Binary of 10

0000 0101 -> Binary of 5

0000 1111 → O/P is 15

- Bitwise XOR ^

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise XOR operators on the individual bits of the operand according to the truth table shown above

Example : - 10 | 5

0000 1010 -> Binary of 10

0000 0101 -> Binary of 5

0000 1111 → O/P is 15

- Bitwise NOT ~

  Bitwise NOT operator results in one’s compliment of its operand

NOT "~"	
INPUT	OUTPUT
0	1
1	0

- Left shift << and Right shift >>
  - 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

- Left shift operator : - num << n = num * 2 raise to n
  - **5 << 2 = 5 * 2 to the power 2**

    5 * 4 = 20
- Right Shift operator :- num >> n = num / 2 raise to n
  - **9 >> 1 = 9 / 2 to the power n**

    = 4

Twisters

- If precedence of two operators in an expression is same, their associativity is considered to decide their binding with operands.
- Data type conversions and ranges should be considered while doing arithmetic operations.
- sizeof() is compile time operator. Expressions within sizeof are not executed at runtime.
- Relational and logical operators always result in 0 or 1.
- In logical AND, if first condition is false, second condition is not evaluated. Result is false.
- In logical OR, if first condition is true, second condition is not evaluated. Result is true.
- Increment/Decrement operators in arithmetic expressions are compiler dependent.

Control Statements

- Decision or Selection
  - if-else
  - switch-case
- Iteration (loop)
  - for
  - while
  - do-while
- Jump
  - break
  - continue
  - goto
  - return

Selection Statement

- If statement
- If-else statement
- Switch statement

- If statement
  - General form of if statement is

```
If(expression)  // if header
    statement // if body
```

  - If the if controlling expression is evaluated to true the statement constituting if body is executed
  - There should be no semicolon at the end of if header
- If-else
  - Most of the problems require one set of action to be performed if particular condition is true and another set of action to be performed if condtion is false
  - C language provides an if-else statement
  - General form is

```
If(expression)
    statement 1;
else
    statement 2;
```

- Nested if Statement
  - If the body of if statement contains another if statement then we say that if are nested and the statement is known as nested if statement
    - If(expression)

```
{
        statement

        ---------------
          if statement
        ------------------
}
```

- Nested if- else statement
  - In nested if-else statement the if body or else body of an if-else statement contains another if statement or if else statement

if-else statement

```
if (condition) {
        statement 1;
        statement 2;
}

if (condition) {
        statement 1;
        statement 2;
}
else {
        statement 3;
        statement 4;
}
```

```
if (condition)
        statement 1;

if (condition)
        statement 1;
else
        statement 2;
```

- Condition is any expression – using relational, logical or other operators.
  - 0 – false condition
  - 1 – true condition

Ternary/conditional operator

```
if (condition) {
        // execute if condition is true
}
else {
        // execute if condition is false
}
```

- if-else can be nested within each other.

```
condition ? expression1 : expression2
```

- If condition is true, expression1 is executed; otherwise expression2 is executed.
- Ternary operators can also be nested.
- expression1 & expression2 must be expressions (not statement).
  - expression – evaluate to some value.
  - statement – C statement ends with ;

Switch statement

- .A switch statement is used to control Complex branching operations , When there are many Conditions it becomes too difficult to use if and if-else
- In such case switch provides easy and organized way to select among multiple operations
- General form of switch statement is
    - Switch (expression)

      statement
- Switch selection expression must be of integral type
- Case labeled constituting the body of switch should be unique i.e. no two case labels should evaluate to same value

switch-case

```
switch (expression) {
    case const-expr1:
        statement(s);
        break;
    case const-expr2:
        statement(s);
        break;
    …
    default:
        statement(s);
        break;
}
```

- Switch-case is used to select one of the several paths to execute depending on value of int expression.
- case constants cannot be duplicated.
- break statement skips remaining statements and continues execution at the end of switch closing brace.
- If break is missing, statements under sub-sequent case continue to execute.
- default case is optional and it is executed only if int expression is not matching with any of the case constant.
- Sequence of cases and default case doesn't matter.

Loops

- 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 for terminating condition is referred as ‘loop variable’.

while loop

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

- Syntax:

```
initialization;
while(condition) {
        statement1;
        statement2;
        modification;
}
```

for loop

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

```
for(initialization; condition; modification) {
        statement1;
        statement2;
}
```

do-while loop

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

```
do {
        statement1;
        statement2;
} while(condition);
```

- do-while is exit control loop.
- while & for are entry control loops.
- do-while is executed at least once.

Infinite loop

- If loop condition is always true, program never terminates.

```
while(1) {
        …
}

for( ; ; ) {
        …
}

do {
        …
} while(1);
```

break/continue

- break statement
  - Used to early exit from loop, or to exit an infinite loop
  - Takes control out of current loop and continues execution of statements after the loop.
  - Statements after break are skipped.

- continue statement
  - Used to continue next iteration of the loop.
  - Statements after continue are skipped (for current iteration).

- break is used with loop/switch case.
- continue used with only loop.

- In case of nested loops, break/continue affects current loop only (not outer).

goto statement

- Jumps to statement label, must be within same function as the goto.
  - Statement label is an identifier followed by a colon (:)
  - Unstructured control statement
  - Used rarely (less readable)
  - Advised to use only for forward jump
- Best use is to exit from deeply nested loops.

- Syntax:

```
goto label_name;
..
..
label_name: C-statements
```

enum

- enum is user defined data type.
- Used to improve readability of C program (int constants, switch-case constants).
- `enum color { RED, GREEN, BLUE, WHITE, YELLOW };`
- `enum color c1 = BLUE;`
- enum constant values by default start from 0 and assigned sequentially.
- Programmer may choose to modify enum constant to any +ve, 0 or –ve value.
- Enum constants can be duplicated.
- `enum color { RED=-2, GREEN, BLUE, WHITE, YELLOW=0 };`
- Internally enum is integer, so size of enum = size of int.
- The enum constants are replaced by int values.

typedef

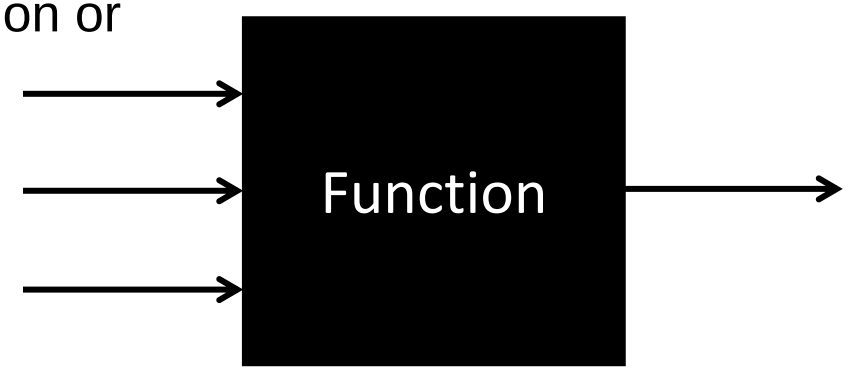
- typedef is used to create alias for any data-type.
- These aliases are helpful to
  - increase readability of the code.
  - port same code across multiple architecture/platforms.
  - simplify complex declarations.

- **typedef existing-data-type data-type-alias;**

- Examples:
  - typedef char int8_t;
  - typedef unsigned char uint8_t;
- typedef unsigned int size_t; // declared in C library.

Functions

- C program is made up of one or more functions.
- C program contains at least one function i.e. main() function.
  - Execution of C program begins from main.
  - It returns exit status to the system.
- Programs are divided into multiple logical parts called as function or sub-routine.
- Advantages
  - Reusability
  - Readability
  - Maintainability



- Function is set of instructions, that takes zero or more inputs (arguments) and return result (optional).
- Function is a black box.

Functions

- Each function has
  - Declaration
  - Definition
  - Call
- A function can be called one or more times.
- Arguments
  - Arguments passed to function → Actual arguments
  - Arguments collected in function → Formal arguments
  - Formal arguments must match with actual arguments

Examples:

1. addition()
2. print_line()
3. factorial()
4. combination()

Functions

- Function Declaration
  - Informs compiler about function name, argument types and return type.
  - Usually written at the beginning of program (source file).
  - Can also be written at start of calling function).
  - Examples:
    - `float divide(int x, int y);`
    - `int fun2(int, int);`
    - `int fun3();`
    - `double fun4(void);`
    - `void fun5(double);`
  - Declaration statements are not executed at runtime.
- Function Definition
  - Implementation of function.
  - Function is set of C statements.
  - It process inputs (arguments) and produce output (return value).

```
float divide(int a, int b) {
        return (float)a/b;
}
```

  - Function can return max one value.
  - Function cannot be defined in another function.
- Function Call
  - Typically function is called from other function one or more times.

Function execution

- When a function is called, function activation record/stack frame is created on stack of current process.
- When function is completed, function activation record is destroyed.
- Function activation record contains:
  - Local variables
  - Formal arguments
  - Return address
- Upon completion, next instruction after function call continue to execute.

Function types

- User defined functions
  - Declared by programmer
  - Defined by programmer
  - Called by programmer
- Library (pre-defined) functions
  - Declared in standard header files e.g. stdio.h, string.h, math.h, …
  - Defined in standard libraries e.g. libc.so, libm.so, …
  - Called by programmer
- main()
  - Entry point function – code perspective
  - User defined
  - System declared
  - int main(void) {…}
  - int main(int argc, char *argv[]) {…}

Storage class

	Storage	Initial value	Life	Scope
auto / local	Stack	Garbage	Block	Block
register	CPU register	Garbage	Block	Block
static	Data section	Zero	Program	Limited
extern / global	Data section	Zero	Program	Program

- Each running process have following sections:
  - Text
  - Data
  - Heap
  - Stack
- Storage class decides
  - Storage (section)
  - Life (existence)
  - Scope (visibility)
- Accessing variable outside the scope raise compiler error.

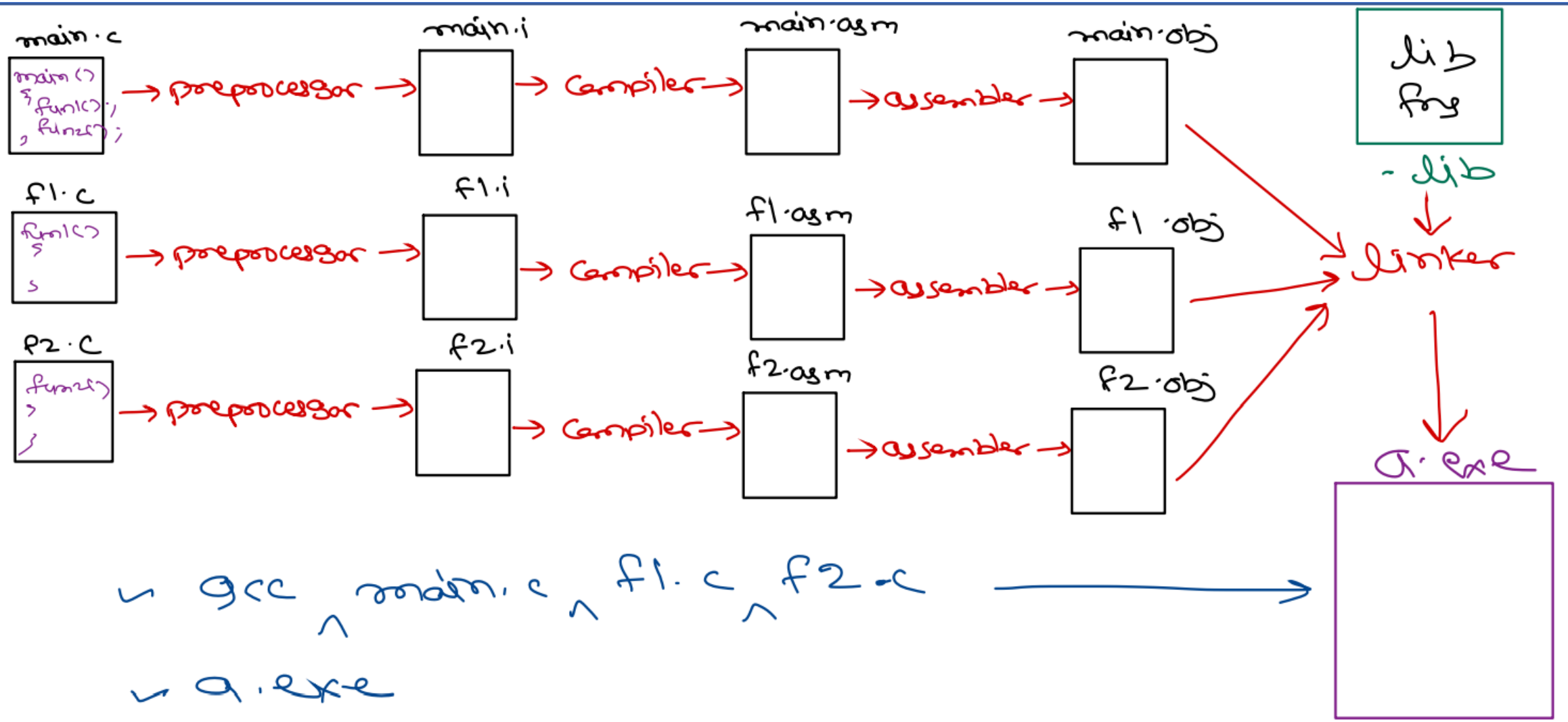
Storage class

- Local variables declared inside the function.
  - Created when function is called and destroyed when function is completed.
- Global variables declared outside the function.
  - Available through out the execution of program.
  - Declared using extern keyword, if not declared within scope.
- Static variables are same as global with limited scope.
  - If declared within block, limited to block scope.
  - If declared outside function, limited to file scope.
- Register is similar to local storage class, but stored in CPU register for faster access.
  - register keyword is request to the system, which will be accepted if CPU register is available.

Recursion

- Function calling itself is called as recursive function.
- To write recursive function consider
  - Explain process/formula in terms of itself
  - Decide the end/terminating condition
- Examples:
  - $n! = n * (n-1)!$ $0! = 1$
  - $x^y = X * x^{y-1}$ $x^0 = 1$
  - $T_n = T_{n-1} + T_{n-2}$ $T_1 = T_2 = 1$
  - factors(n) = 1^{st} prime factor of n * factors(n)

- Advantages
  1. Simplified Readable programs (Divide and conquer Problems)
- Disadvantages
  - Needs More Space (FAR on stack)
  - Needs More Time (FAR Created)


main.c
main()
{
fun1();
fun2();
}
preprocessor
main.i
compiler
main.asm
assembler
main.obj
lib
fns
-lib
f1.c
fun1()
{
}
preprocessor
f1.i
compiler
f1.asm
assembler
f1.obj
linker
f2.c
fun2()
{
}
preprocessor
f2.i
compiler
f2.asm
assembler
f2.obj
a.exe
gcc main.c f1.c f2.c
a.exe

Recursion execution

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int fact(int n) {
  int r;
  if(n==0)
    return 1;
  r = n * fact(n-1);
  return r;
}
```

```
int main() {
   int res;
   res = fact(5);
   printf("%d", res);
   return 0;
}
```

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

0! = 1

Function arguments

```
void sumpro(int a, int b, int ps, int pp) {
    ps = a + b;
    pp = a * b;
}
int main() {
    int x = 12, y = 4, s, p;
    sumpro(x, y, s, p);
    printf("%d %d", s, p);
    return 0;
}
```

Globally declared

Function arguments

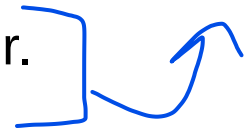
```
void sumpro(int a, int b, int ps, int pp) {
    ps = a + b;
    pp = a * b;
}
int main() {
    int x = 12, y = 4, s, p;
    sumpro(x, y, s, p);
    printf("%d %d", s, p);
    return 0;
}
```

Passing arguments: Call by value vs Call by address/reference

- Call by value
  - Formal argument is of same type as of actual argument.
  - Actual argument is copied into formal argument.
  - Any change in formal argument does not reflect in actual argument.
  - Creating copy of argument need more space as well as time (for bigger types).
  - Most of data types can be passed by value – primitive & user defined types.
- Call by address
  - Formal argument is of pointer type (of actual argument type).
  - Address of actual argument is collected in formal argument.
  - Actual argument can be modified using formal argument.
  - To collect address only need pointer. Pointer size is same irrespective of data type.
  - Array and Functions can be passed by address only.

Pointer - Introduction

- Pointer is a variable that stores address of some memory location.
- Internally it is unsigned integer (it is memory address).
- In C, pointer is a special data type.
- It is not compatible with unsigned int.
- Pointer is derived data type (based on primitive data type). means to store address of int/char we need pointer of int/char datatype.
  - To store address of int, we have int pointer.
  - To store address of char, we have char pointer, …
- Size of pointer variable is always same, irrespective of its data type (as it stores only the address).



Pointer - Syntax

- Pointer syntax:
  - Declaration:
    - double *p;
  - Initialization:
    - p = &d;
  - Dereferencing:
    - printf("%lf\n", *p);
- Reference operator - &
  - Also called as direction operator.
  - Read as “address of”.
- Dereference operator - *
  - Also called as indirection operator.
  - Read as “value at”.

Pointer - Syntax

```
int main() {
  double a = 1.2;
  double *p = &a;
  double **pp = &p;
  printf("%lf\n", a);
  printf("%lf\n", *p);
  printf("%lf\n", **pp);
  return 0;
}
```

- Pointer to pointer stores address of some pointer variable.
- Level of indirection: Number of dereference operator to retrieve value.

Pointer - Scale factor

- Size of data type of pointer is known as Scale factor.
- Scale factor defines number of bytes to be read/written while dereferencing the pointer.
- Scale factor of different pointers
  - Pointer to primitive types: char*, short*, int*, long*, float*, double*
  - Pointer to pointer: char**, short**, int**, long**, float**, double**, void**
  - Pointer to struct/union.
  - Pointer to enum.

Pointer arithmetic

- Scale factor plays significant role in pointer arithmetic.
- n locations ahead from current location
  - ptr + n = ptr + n * scale factor of ptr
- n locations behind from current location
  - ptr - n = ptr - n * scale factor of ptr
- number of locations in between
  - ptr1 – ptr2 = (ptr1 – ptr2) / scale factor of ptr1

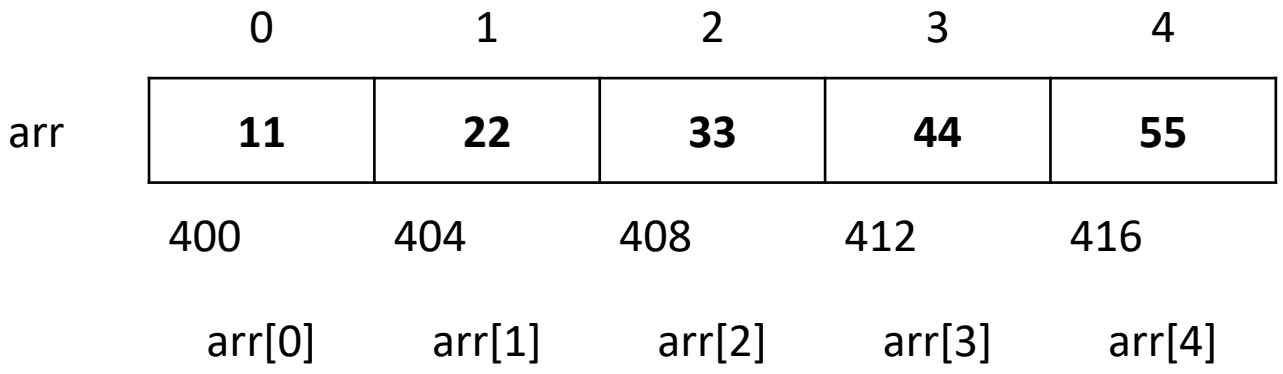
Pointer arithmetic

- When pointer is incremented or decremented by 1, it changes by the scale factor.
- When integer 'n' is added or subtracted from a pointer, it changes by n * scale factor.
- Multiplication or division of any integer with pointer is not allowed.
- Addition, multiplication and division of two pointers is not allowed.
- Subtraction of two pointers gives number of locations in between. It is useful in arrays.

Arrays

- Array is collection of similar data elements in contiguous memory locations.
- Elements of array share the same name i.e. name of the array.
- They are identified by unique index/subscript. Index range from 0 to n-1.
- Array indexing starts from 0.
- Checking array bounds is responsibility of programmer (not of compiler).
- Size of array is fixed (it cannot be grow/shrink at runtime).

```
int main() {
    int i, arr[5] = {11, 22, 33, 44, 55};
    for(i=0; i<5; i++)
        printf("%d\n", arr[i]);
    return 0;
}
```



Arrays

- If array is initialized partially at its point of declaration rest of elements are initialized to zero.

- If array is initialized partially at its point of declaration, giving array size is optional. It will be inferred from number of elements in initializer list.

- The array name is treated as address of 0^{th} element in any runtime expression.

- Pointer to array is pointer to 0^{th} element of the array.

Pointer to array

```
int main() {
    int i, arr[5] = { 11, 22, 33 };
    int *ptr = arr;
    for(i=0; i < 5; i++) {
        printf("%d %d %d %d\n",
            arr[i], *(arr+i), *(i+arr), i[arr]);
        printf("%d %d %d %d\n",
            ptr[i], *(ptr+i), *(i+ptr), i[ptr]);
    }
    return 0;
}
```

Passing array to function

- Array can be passed to function by address only.
- To collect it in formal argument, array or pointer notation can be used.
  - void print_array(int arr[]);
  - void print_array(int *arr);
- Since it is pass by reference, any changes done in array within called function will be visible in calling function.
- You should not return address of local array from the function, because local variables will be destroyed when function returns.

```
#include <stdio.h>
int main() {
    int arr[5] = { 11, 22, 33, 44, 55 };
    print_array(arr, 5);
    return 0;
}
void print_array(int arr[], int n) {
    int i;
    for(i=0; i<n; i++)
        printf("%d\n", arr[i]);
}
```

Type qualifier – const

- const keyword inform compiler that the variable is not intended to be modified.
- Compiler do not allow using any operator on the variable which may modify it e.g. ++, --, =, +=, -=, etc.

- Note that const variables may be modified indirectly using pointers. Compiler only check source code (and do not monitor runtime execution).

Constant pointers

- int a = 10;
- const int *ptr = &a;

- int const *ptr = &a;

- int * const ptr = &a;

- int * ptr const = &a;

- const int * const ptr = &a;
- const int * const ptr = &a;

String

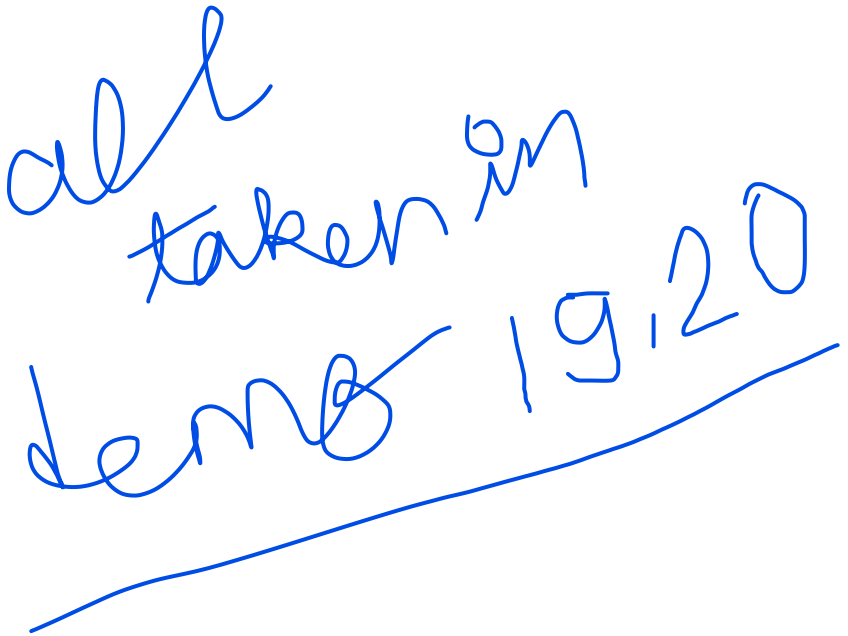
- String is character array terminated with '\0' character.
  - '\0' is character with ASCII value = 0.
- Example :

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

- String input/output
  - char str[20];
  - scanf("%s",str); /*Input*/
  - printf("%s",str); /*Output*/
  - gets(str); /*Input*/
  - puts(str); /*Output*/
  - scanf("%[^\n]", str); // scan whole line

String functions

- C library have many string functions.
- They are declared in string.h
  - strlen() – size_t strlen(const char *s);
  - strcpy() – char* strcpy(char *dest, const char *src);
  - strcat() – char* strcat(char *dest, const char *src);
  - strcmp() – int strcmp(const char *s1, const char *s2);
  - strchr() – char* strcat(const char *s, int ch);
  - strstr() – char* strstr(const char *s1, const char *s2);
  - strrev() – char* strrev(char *s);

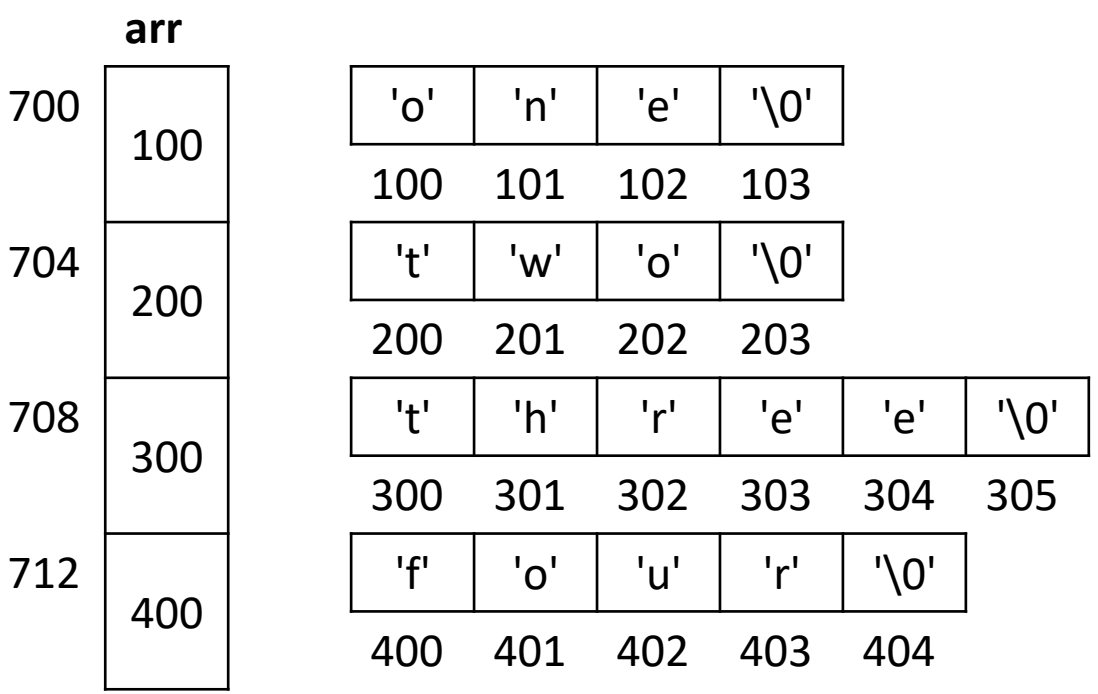


NULL pointer

- If pointer is uninitialized, it will hold garbage address (local pointer variables).
- Accessing such pointer may produce unexpected results. Such pointers are sometimes referred as wild pointers.
- C defined a symbolic const NULL, that expands to (void*)0.
- It is good practice to keep well known address in pointer (instead of garbage).
- NULL is typically used to initialize pointer and/or assign once pointer is no more in use.
- Many C functions return NULL to represent failure.
  - strchr(), strstr(), malloc(), fopen(), etc.

Array of pointers

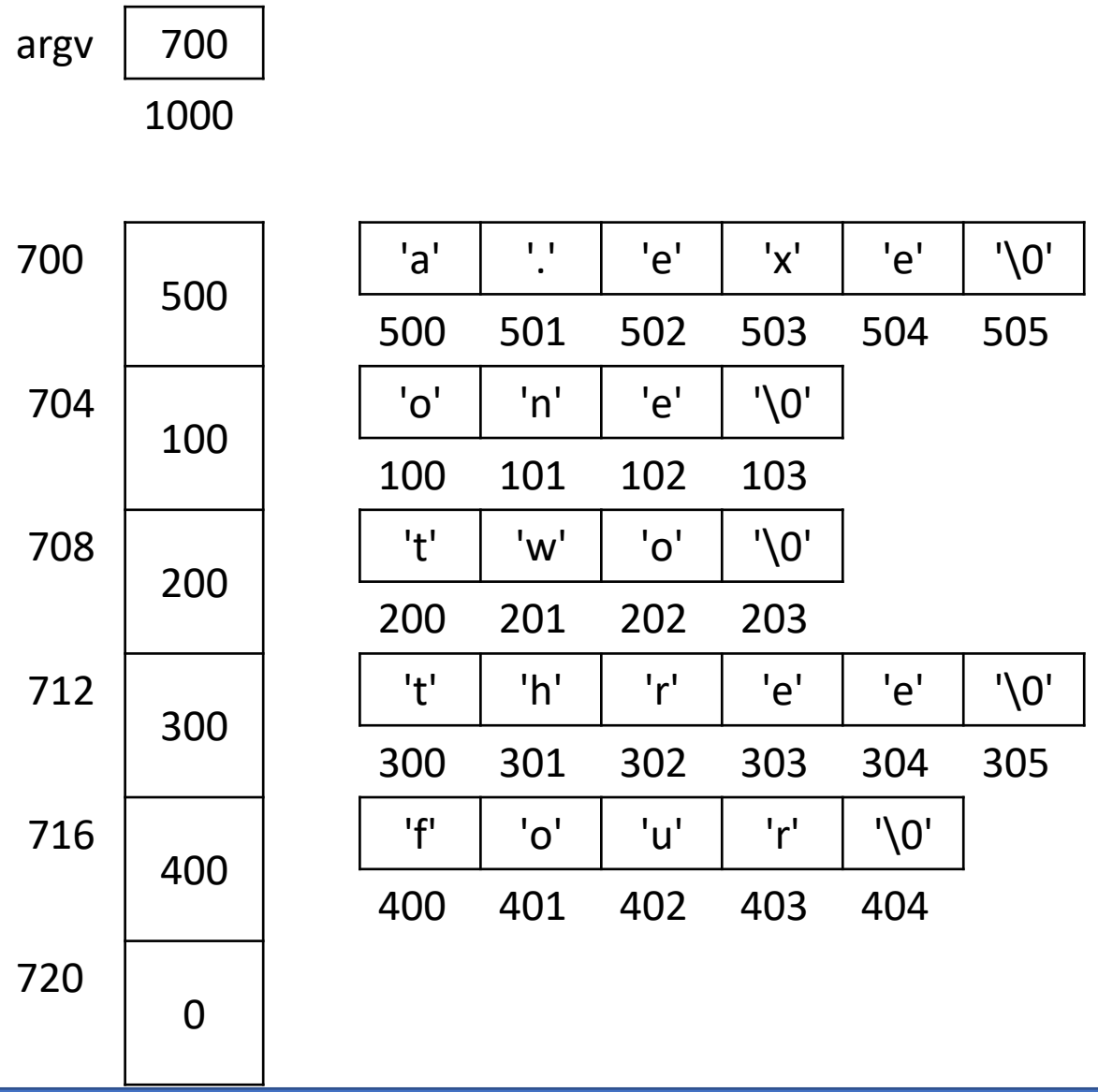
```
char *arr[] = { "one", "two", "three", "four" };
for(i = 0; i < 4; i++)
        puts(arr[i]);
```



Command line arguments

- Command line arguments are information passed to the program while executing it on command line.
- cmd> a.exe one two three four

```
int main(int argc, char *argv[]) {
        int i;
        for(i=0; i < argc; i++)
                puts(argv[i]);
        return 0;
}
```



2-D array

- Logically 2-D array represents m x n matrix i.e. m rows and n columns.
  - int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };

- Array declaration:
  - int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };
  - int arr[3][4] = { {1, 2 }, {10}, {11, 22, 33 } };
  - int arr[3][4] = { 1, 2, 10, 11, 22, 33 };
  - int arr[ ][4] = { 1, 2, 10, 11, 22, 33 };

	0	1	2	3
0	**1**	**2**	**3**	**4**
1	**10**	**20**	**30**	**40**
2	**11**	**22**	**33**	**44**

2-D array

- 2-D array is collection of 1-D arrays in contiguous memory locations.
  - Each element is 1-D array.
- int arr[3][4] = { {1, 2, 3, 4}, {10, 20, 30, 40}, {11, 22, 33, 44} };

	0				1				2			
	0	1	2	3	0	1	2	3	0	1	2	4
arr	**1**	**2**	**3**	**4**	**10**	**20**	**30**	**40**	**11**	**22**	**33**	**44**
	400	404	408	412	416	420	424	428	432	436	440	444
	400				416				432			

Passing 2-D array to Functions

- 2-D array is passed to function by address.
- It can be collected in formal argument using array notation or pointer notation.
- While using array notation, giving number of rows is optional. Even though mentioned, will be ignored by compiler.

void pointer

- Void pointer is generic pointer it can hold address of any data type (without casting).
- Scale factor of void* is not defined, so cannot perform pointer arithmetic.
- To retrieve value of the variable need type-casting.
- void* is used to implement generic algorithms.

Dynamic memory allocation

- Dynamic memory allocation allow allocation of memory at runtime as per requirement.
- This memory is allocated at runtime on Heap section of process.
- Library functions used for Dynamic memory allocation are
  - malloc() – allocated memory contains garbage values.
  - calloc() – allocated memory contains zero values.
  - realloc() – allocated memory block can be resized (grow or shrink).
- All these function returns base address of allocated block as void*.
- If function fails, it returns NULL pointer.

Memory leakage

- If memory is allocated dynamically, but not released is said to be "memory leakage".
  - Such memory is not used by OS or any other application as well, so it is wasted.
  - In modern OS, leaked memory gets auto released when program is terminated.
  - However for long running programs (like web-servers) this memory is not freed.
  - More memory leakage reduce available memory size in the system, and thus slow down whole system.

- In Linux, valgrind tool can be used to detect memory leakage.

```
int main() {
    int *p = (int*) malloc(20);
    int a = 10;
    // ...
    p = &a; // here addr of allocated block is lost, so this memory can never be freed.
    // this is memory leakage
    // ...
    return 0;
}
```

Dangling pointer

- Pointer keeping address of memory that is not valid for the application, is said to be "dangling pointer".
- Any read/write operation on this may abort the application. In Linux it is referred as "Segmentation Fault".
- Examples of dangling pointers
  - After releasing dynamically allocated memory, pointer still keeping the old address.
  - Uninitialized (local) pointer
  - Pointer holding address of local variable returned from the function.
- It is advised to assign NULL to the pointer instead of keeping it dangling.

```
int main() {
    int *p = (int*) malloc(20);
    // ...
    free(p); // now p become dangling
    // ...
    return 0;
}
```

Preprocessor Directives

- Preprocessor is part of C programming toolchain/SDK.
  - Removes comments from the source code.
  - Expand source code by processing all statements starting with #.
  - Executed before compiler
- All statements starting with # are called as preprocessor directives.
  - Header file include
    - #include
  - Symbolic constants & Macros
    - #define
  - Conditional compilation
    - #if, #else, #elif, #endif
    - #ifdef #ifndef
  - Miscellaneous
    - #pragma, #error

#include

- #include includes header files (.h) in the source code (.c).
- #include <file.h>
  - Find file in standard include directory.
  - If not found, raise error.
- #include "file.h"
  - File file in current source directory.
  - If not found, find file in standard include directory.
  - If not found, raise error.

#define (Symbolic constants)

- Used to define symbolic constants.
  - #define PI 3.142
  - #define SIZE 10
- Predefined constants
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__
- Symbolic constants and macros are available from there declaration till the end of file. Their scope is not limited to the function.

#define (Macro)

- Used to define macros (with or without arguments)
  - #define ADD(a, b) (a + b)
  - #define SQUARE(x) ((x) * (x))
  - #define SWAP(a,b,type) { type t = a; a = b; b = t; }
- Macros are replaced with macro expansion by preprocessor directly.
  - May raise logical/compiler errors if not used parenthesis properly.
- Stringizing operator (#)
  - Converts given argument into string.
  - #define PRINT(var) printf(#var " = %d", var)
- Token pasting operator (##)
  - Combines argument(s) of macro with some symbol.
  - #define VAR(a,b) a##b

#define

- Functions
  - Function have declaration, definition and call.
  - Functions are called at runtime by creating FAR on stack.
  - Functions are type-safe.
  - Functions may be recursive.
  - Functions called multiple times doesn't increase code size.
  - Functions execute slower.
  - For bigger reusable code snippets, functions are preferred.
- Macros
  - Macro definition contain macro arguments and expansion.
  - Macros are replaced blindly by the processor before compilation
  - Macros are not type-safe.
  - Macros cannot be recursive.
  - Macros (multi-line) called multiple times increase code size.
  - Macros execute faster.
  - For smaller code snippets/formulas, macros are preferred.

Conditional compilation

- As preprocessing is done before compilation, it can be used to control the source code to be made available for compilation process.
- The condition should be evaluated at preprocessing time (constant values).
- Conditional compilation directives
  - #if, #elif, #else, #endif
  - #ifdef, #ifndef
  - #undef

```
#define VER 1
int main() {
    #ifndef  VER
        #error "VER not defined"
    #endif
    #if  VER == 1
         printf("This is Version 1.\n");
    #elif  VER == 2
         printf("This is Version 2.\n");
    #else
        printf("This is 3+ Version.\n");
     #endif
     return 0;
}
```

Structure

- Structure is a user-defined data type.
- Structure stores logically related (similar or non-similar) elements in contiguous memory location.
- Structure members can be accessed using "." operator via struct variable.
- Structure members can be accessed using "->" operator via struct pointer.
- Size of struct = Sum of sizes of struct members.
- If struct variable initialized partially at its point of declaration, remaining elements are initialized to zero.

```
// struct data-type declaration (global or local)
struct emp {
    int empno;
    char ename[20];
    double sal;
};
// struct variable declaration
struct emp e1 = {11, "John", 20000.0};
// print struct members
printf("%d%s%lf", e1.empno, e1.ename, e1.sal);
```

Struct – User defined data-type

- int a = 10;
- printf("%d", a);
- scanf("%d", &a);
- int *p = &a;
- printf("%d", *p);

- fun1(a);
- void fun1(int x) { … }
- fun2(&a);
- void fun1(int *p) { … }
- int arr[3] = {11, 22, 33};
- for(int i=0; i<3; i++)
  printf("%d", arr[i]);

- struct emp e = { 11, "John" };
- printf("%d, %s", e.empno, e.ename);
- scanf("%d%s", &e.empno, &e.ename);
- struct emp *p = &e;
- printf("%d, %s", p->empno, p->ename);
- printf("%d, %s", (*p).empno, (*p).ename);
- fun1(e);
- void fun1(struct emp x) { … }
- fun1(&e);
- void fun2(struct emp *p) { … }
- struct emp arr[3] = { {…}, {…}, {…} };
- for(int i=0; i<3; i++)
  printf("%d", arr[i].empno);

Struct

- A variable of a struct can be member of another struct.
- This can be done with nested struct declaration.

```
struct emp {
    int empno;
    char ename[20];
    double sal;
    struct {
        int day, month, year;
    }join;
};
```

```
struct date {
    int day, month, year;
};
struct emp {
    int empno;
    char ename[20];
    double sal;
    struct date join;
};
struct emp e = { 11, "John", 2000.0, {1, 1, 2000} };
printf("%d %s %d-%d-%d\n", e.empno, e.ename, e.join.day, e.join.month, e.join.year);
```

Struct padding

- For efficient access compiler may add hidden bytes into the struct called as "struct padding" or "slack bytes".
- On x86 architecture compiler add slack bytes to make struct size multiple of 4 bytes (word size).
- These slack bytes not meant to be accessed by the program.
- Programmer may choose to turn off this feature by using #pragma.
  - #pragma pack(1)

```
struct test {
    int a;
    char b;
};
printf("%u\n", sizeof(struct test));


#pragma pack(1)
struct test {
    int a;
    char b;
};
printf("%u\n", sizeof(struct test));
```

Bit Fields

- 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.
- Bit-fields can be signed or unsigned.
  - Signed bit-field, MSB represent size + or -.
  - Unsigned bit-field, all bits store data.
- Limitations of bit-fields
  - Cannot take address of bit-field (&)
  - Cannot create array of bit-fields.
  - Cannot store floating point values.

```
struct student {
    char name[20];
    unsigned int age: 7;
    unsigned int roll: 6;
};
struct student s1 = { "Ram", 10, 21 };
printf("%s, %d, %d", s1.name, s1.age, s1.roll);
```

Union

- Union is user defined data-type.
- Like struct it is collection of similar or non-similar data elements.
- All members of union share same memory space i.e. modification of an member can affect others too.
- Size of union = Size of largest element
- When union is initialized at declaration, the first member is initialized.
- Application:
  - System programming: to simulate register sharing in the hardware.
  - Application programming: to use single member of union as per requirement.

File IO

- File is collection of data and information on storage device.
- Each file have data (contents) and metadata (information).
- File IO can enable read/write file data.
- File Input Output
  - Low Level File IO
    - Explicit Buffer Management. Use File Handle.
  - High Level File IO
    - Auto Buffer Management. Use File Pointer.
    - Formatted (Text) IO
      - fprintf(), fscanf()
    - Unformatted (Text) IO
      - fgetc(), fputc(), fgets(), fputs()
    - Binary File IO
      - fread(), fwrite()

High Level File IO

- File must be opened before read/write operation and closed after operation is completed.
- FILE * fp = fopen("filepath", "mode"); – to open the file
  - File open modes:
    - w: open file for write. If exists truncate. If not exists create.
    - r: open file for read. If not exists, function fails.
    - a: open file for append (write at the end). If not exists create.
    - w+: Same as "w" + read operation.
    - r+: Same as "r" + write operation.
    - a+: Same as "a" + append (write at the end) operation.
  - Return FILE* when opened successfully, otherwise return NULL.
- fclose(fp);
  - Close file and release resources.

File IO

- Character IO
  - fgetc()
  - fputc()
- String (Line) IO
  - fgets()
  - fputs()
- Formatted IO
  - fscanf()
  - fprintf()
- Binary (record) IO
  - fread()
  - fwrite()
- File position
  - fseek()
  - ftell()

Thank you!

Ketan Kore <ketan.kore@sunbeaminfo.com>