Course Booklet for Advanced C Module

$$\begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ \vdots \\ a_n + b_n \end{bmatrix}$$

Is it C or Logic?

Advanced tips, tricks & use cases for C users

By Emertxe

Version 3.0 (December 18, 2014)

Course Email: embedded.courses@emertxe.com

Contents

Preface

0.1 Course: Education goals and objectives

Advanced C course module covers the depth of C language to its core, covering all the functionalities of the language, more important from the industry perspective. By the end of the module, students are expected to have made their concepts very clear and become comfortable using the language to any level

The theory duration for this module roughly spans 3 weeks, apart from the parallel lab sessions, and assignments

Pre-requisites for this module are: BE/BTech (4th semester onwards) and Basic programming knowledge

This booklet serves as a workbook guide, as we go along through the Advanced C course module at Emertxe Information Technologies Pvt Ltd. It will provide the enough information pointers to remember, alongwith the references to detailed relevant materials. It will provide a ground for practice and for writing your personalized notes

As part of this, it would cover: Mastering the basics, Playing with all kinds of Pointers, Var Args, Godly Recursion, Functions, Files, Preprocessor Directives, and many other industry relevant topics. As an overall experience, it will take you through all the nitty-gritties of C through a well organized set of examples and exercises

The appendix will contain references and details on the topics not directly related to the module but nevertheless relevant to it. Examples include Dealing with the Compiler, Creating your own library, and Building a Project

Finally, there is a complete appendix dedicated to a set of assignments well sorted and graded, which gets reviewed after every batch

Special about this module (and this booklet):

- Concentrates on the in-depth concepts, making you more confident about you and your skills

- Well packed and graded set of assignments and project for practice
- Focus on the way industry uses C, making you more confident at interviews and getting a job
- Regular mutual feedbacks to improvise and tune the module as per the audience
- Regular updates from the industry to keep abreast with induustry
- Learn many more things, as added benefits than just the module, like:
  - Get hands-on to work on Linux
  - Become expert on various tools
  - Increase your productivity

Chapter 1

Day 1: Introduction

1.1 Expectations out of this module

Notes:

1.2 What is a language?

Simply put, a language is a stylized communication technique. The more varied vocabulary it has, the more expressive it becomes. Then, there is grammar to make meaningful communication statements out of it

Q: Which is your first language?

A:

Similarly, being more specific, a programming language is a stylized communication technique intended to be used for controlling the behaviour of a machine (often a computer), by expressing ourselves to the machine. Like the natural languages, programming languages too, have syntactic rules (to form words) and semantic rules (to form sentences), used to define meaning.

Notes:

1.3 And the types of programming languages

- Procedural: To be considered procedural, a programming language should support procedural programming by having an explicit concept of a procedure, and a syntax to define it. It should ideally support specification of argument types, local variables, recursive procedure calls and use of procedures in separately built program constructs. It may also support distinction of input and output arguments.
  The canonical example of a procedural programming language is ALGOL. A language in which the only form of procedure is a method is generally considered object-oriented rather than procedural, and will not be included in this list. This applies to Java, but not to C++.

- Object Oriented: Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

- Functional: Functional programming languages are a class of languages designed to reflect the way people think mathematically, rather than reflecting the underlying machine. Functional languages are based on the lambda calculus, a simple model of computation, and have a solid theoretical foundation that allows one to reason formally about the programs written in them. The most commonly used functional languages are Standard ML, Haskell, and ?pure? Scheme (a dia-lect of LISP), which, although they differ in many ways, share most of the properties described here.

- Logical: The program specifies a computation by giving the properties of a correct answer. Prolog and LDL are examples of declarative languages; since they emphasize the logical properties of a computation, they are often called logic programming languages. The declarative/procedural distinction is not rigid: Prolog of necessity incorporates some procedural features, for example to manage file input/output, and simple Boolean tests are common in FORTRAN and C. In the &‘&‘ideal&’&’ (I should perhaps say &‘&‘ideal to logicians&’&’) case, writing a declarative program is equivalent to defining a proof for a proposition (relationship).

- and many more

Notes:

1.4 Brief History

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

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

1.4.1 The C Standard

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

1.4.2 Important Characteristics

C is considered as a middle level language, because it possesses the qualities of both high level and low-level languages. For example, it has most of the control structures present in a higher-level language; it also has macro processor, pointers, strings as just character arrays, direct memory manipulation, ability to access hardware, weakly typed nature - such facilities resemble the assembly (low-level) language constructs.

1.4.3 What are the most important characteristics of the C language?

- C can be considered as a pragmatic language: it is language for programmers, intended to be used commercially and are designed in a way to address real problems involved with writing code. This is as opposed to academic or research languages where proper design is given credit without much consideration to pragmatic details.

- It is indented to be used by advanced programmers, for serious use, and not for novices and thus qualify less as an academic language for learning.

- Gives importance to curt code. For example, an integer is just an int, { and } for begin and end etc. Look at the following C code for string copy and recursive code for string reverse:

```
char *strcpy(char *t, char *s)
{
    while(*t++ = *s++)
        ;
}

char *strrev(char *s)
{
    return !*s ? s : strcpy(s, strncat(strrev(s+1), s,1));
}
```

It should also be noted that due to this same property, C became infamous of being a write-only language.

- It is widely available in various platforms from mainframes to palmtops and is known for its wide availability.
- It is a general-purpose language, even though it is applied and used effectively in various specific domains.
- It is a free-formatted language (and not a strongly-typed language).
- Efficiency and portability are the important considerations that influenced many major design decisions in this language. For example, for efficiency considerations, storage and evaluation of integral types as integers is done.
- Library facilities play an important role - the standard library provides the often-required facilities, for example, string manipulation, mathematical functions, I/O routines.

It is often claimed that one of C's major virtues is it is so small that every programmer understands every construct in the language.

1.5 Keywords in C

The keywords used here is as in ANSI C 1989 standard.

auto

This keyword is derived from the B language, where it is used to state that a variable is an automatic variable, i.e. the lifetime is automatically entered and destroyed as the method frame is created and destroyed. Since there are no types in B, this keyword was mandatory and used extensively. The keyword was adapted in C to explicitly say that a variable is automatic.

break

break is used to terminate the innermost looping statement or a switch statement.

case

Used in switch statements for selecting a particular case. The case should be followed by a constant integral expression.

char

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

const

It specifies the value of a field or a local variable that cannot be modified.

continue

This is used to skip the rest of the statements in a looping statement.

default

This label is used in switch statements. The statements after this label will be executed only when there is no match found in the case labels.

do

An exit-controlled (the looping condition is checked at the end of the loop) looping mechanism.

double

Variable of this type can store double-precision floating point numbers. The implementations may follow IEEE formats to represent floats and doubles, and double occupies 8-bytes in memory. In this IEEE format, in addition to the non-zero values, floating point numbers can also store:

- zero (either positive or negative zero),
- ∞ (both positive and negative),
- NaN (Not-A-Number) - in case of invalid floating point operations.

else

Used with if statements. Else part is executed when the condition in if becomes false.

enum

In C, an enumeration is a set of named constants that are internally represented as integers. They can take part in expressions as if they were of integral type. The use of enums is preferable to the use of consts or #defines for representing related set of values because the use of enums makes the code more readable and self-documenting.

extern

This indicates that a name is external to the compiler and the current translation unit.

float

float is for single precision floating point numbers. ANSI C does not specify any representation standard for these types. Still it provides a model whose characteristics are guaranteed to be present in any implementation. The standard header file ¡float.h¿ defines macros that provide information about the implementation of floating point arithmetic. With IEEE standard for floating point single precision numbers, a float type occupies 4-bytes.

for

An elegant looping statement.

goto

Used in unconditional jumps within a function (local goto).

if

Simple conditional branching statement. In C, any non-zero value is treated as a true value and will lead to execution of this statement.

int

int is to represent integers, a simple and efficient type for arithmetic operations. In C, the size of an integer is usually the word size of the processor, although the compiler is free to choose the size. However, ANSI C does not permit an integer, which is less than 16 bits.

long

short, long and int are provided to represent various sizes of possible integral types supported by the hardware. ANSI C says that the size of these types is implementation defined, but ensures that the non-decreasing order of char, short, int, and long is preserved (non-decreasing order means that the sizes are char $\leq$ short $\leq$ int $\leq$ long and not the other way).

register

You can specify a local variable as register if you think that variable may be accessed frequently. The compiler will try to put that variable in a microprocessor register if one is available; otherwise, this keyword is ignored and treated as if the variable were declared as auto. Declaring such frequently used variables to be placed in registers may gain only a small performance improvement, for example register optimizations. Programmers are usually in a better position than the compiler is to guess which variable will be used very frequently and thus can assign register storage to such variables. Modern compilers will easily find out the variables that will be frequently accessed and will place them accordingly. So, this keyword is not supported in later C based languages like Java and C#

return

To return control back from the called method.

short

See long and int.

signed

In C, both unsigned and signed are supported for integral types. The idea of separating unsigned from signed types started with the requirement of having a larger range of positive values within the same available space. This idea is valuable in programming where low-

level hardware access is required and in system programming. signed and unsigned data types are implemented in the same way. The only difference is that the interpretation of the Most Significant Bit (MSB) varies. There are many nasty problems that are encountered in C due to misuse of unsigned values. The sign or value of a type may be preserved when an unsigned type is promoted to an integer; these are referred to as sign and value preserving respectively. The choice is an implementation detail and also a source of bugs in C.

sizeof

The sizeof operator is used to obtain the size of a type or an object. It is important in writing portable code in C, since the size of a data type may differ depending on the implementation.

static

In C, the static keyword has an overloaded meaning and used in two different contexts: When used before global functions, it limits the function's scope to the current file When used inside a function, it indicates that the variable is of local scope but of static lifetime (meaning that the value remains between function calls).

struct

struct keyword provides support for aggregate types.

switch

You can look at the switch statement as a specialized version of an if-else cascade, but with limitations. You can do only equality checks and only integral type constants. Using switch instead of an if-else cascade helps in simplifying control-flow and gives you the advantage of generating much faster code. In C, case statements fall through to the next case statement when an explicit break statement is missing (which is a source of many bugs in C programs).

typedef

Typedefs do not create new types - they just add new type names. In C, they are helpful in managing complex declarations. They are also useful in abstracting the details from the users and thus help in increasing the portability of the code. It should be noted that typedefs obey scoping rules and are not textual replacements (as opposed to #defines).

union

Unions can be considered a special case of structures; the syntax for both is mostly the same and only the semantics differ. Memory is allocated such that it can accommodate the

biggest member. Unions suffer many disadvantages in practice, for example, there is no in-built mechanism to know which union member is currently used to store the value.

unsigned

See signed

void

void specifies a non-existent or empty set of values. It is used in the case of void pointers as a generic pointer type in C, and as return type of a function to specify that the function returns nothing. You cannot have objects of type void, and hence this type is sometimes called a pseudo-type.

volatile

In C, volatile tells the compiler not to do any optimizations on the variable qualified as volatile. This also indicates that the variable is asynchronous, and system or external sources may change its value. For example, suppose that you are reading from the serial port. The data may arrive and thus the field that contains the data may be modified externally without the programmers intervention. In such cases, its essential that the field be declared as volatile.

while

This provides the entry-controlled looping mechanism.

1.6 Lab Work

Complete the assignments

(Id) / Date	Assignment Topic
( ) _____	WAP to check whether a given number is odd or even and its signedness(Use nested if) eg: If input is -1, it should print -1 is negative odd number.
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	

Chapter 2

Day 2: Basics Refresher

2.1 Numbers

2.1.1 Integer storage on a computer: 2's complement

- Just invert each bit of its positive's binary value and add 1
- Mathematically: $-k \equiv 2^n - k$

Notes:

2.1.2 Binary point numbers

- $0.5_{10} = 0.1_2 = 1 * 2^{-1}$
- $4_{10} = 100_2 = 1 * 2^2$
- $5_{10} = 101_2 = 1 * 2^2 + 1 * 2^0$
- $2.25_{10} = 10.01_2 = 1 * 2^2 + 1 * 2^{-2}$

2.1.3 Floating point representation

The IEEE single precision floating point standard representation requires a 32-bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, 'S', the next 8 bits, are the exponent bits, 'E', and the final 23 bits are the fraction 'F'

The value V represented by the word may be determined as follows:

- If E=255 and F is nonzero, then V=NaN
- If E=255 and F is zero and S=1, then V=-Infinity
- If E=255 and F is zero and S=0, then V=+Infinity
- If 0<E<255, then V=$(-1)^S * 2^{(E-127)} * (1.F)$
- If E=0 and F is nonzero, then V=$(-1)^S * 2^{(-126)} * (0.F)$
- If E=0 and F is zero and S=1, then V=-0
- If E=0 and F is zero and S=0, then V=0

The IEEE double precision floating point standard representation requires a 64-bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, 'S', the next 11 bits, are the exponent bits, 'E', and the final 52 bits are the fraction 'F'

The value V represented by the word may be determined as follows:

- If E=2047 and F is nonzero, then V=NaN
- If E=2047 and F is zero and S=1, then V=-Infinity
- If E=2047 and F is zero and S=0, then V=+Infinity
- If 0<E<2047, then V=$(-1)^S * 2^{(E-1023)} * (1.F)$
- If E=0 and F is nonzero, then V=$(-1)^S * 2^{(-1022)} * (0.F)$
- If E=0 and F is zero and S=1, then V=-0
- If E=0 and F is zero and S=0, then V=0

S: 1	E: 8(f), 11(d)	F: 23(f), 52(d)

Notes:

2.2 Basic Data Types & Variabes

2.2.1 Sizes of all the basic data types in C

- 1 byte = char <= short <= int <= long <= long long
- float = 4 bytes
- double = 8 bytes
- pointer = address word, mostly same as word
- void = 1 byte

Notes:

2.3 Type, Variable & Function Qualifiers

2.3.1 What is a qualifier?

Notes:

2.3.2 The Qualifiers

- signed(T)
- unsigned(T)

Notes:
Default for char is compiler dependent. For all others, it is signed

- short(T)
- long(T)
- long long(T)

Notes:

- const(V)
- volatile(V)

Notes:
const means "Read only". Need not be constant.
volatile means "Recomputed for every access".

- static(V|F)
- extern(V|F)
- auto(V)

Notes:
Default for globals is extern. For local variables, it is auto.

- register(V)

Notes:
Request for a CPU register

- inline(F)

Notes:
Request for function call replacement by code

2.4 Statements

2.4.1 Simple Statement & Statement Terminator

Is this code valid?

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

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

2.4.2 Compound Statement & its need

Notes:

2.5 Conditional Constructs

The Conditions for loops and if/else blocks must evaluate to a boolean value, i.e. true or false.
In C, there is no seperate datatype for booleans. Instead anything with a value of zero (e.g. int i = 0)evaluates to false, and any other evaluates to true.

- Single Step
  - if (<cond>) <stmt> [else <stmt>]
  - switch (<expr>) case <val1>: <stmt(s)> ... default: <stmts>

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

Notes:

- Repetitive
  - for (<initialize>; <cond>; <post processor>;) <stmt>
  - while (<cond>) <stmt>
  - do <stmt> while (<cond>);

Notes:

2.5.1 Looping Techniques

```
repeat 10 times             for (i = 0; i < 10; i++)          i = 0
                            {                                 while (i < 10)
                                //code                        {
                            }                                     //code
                                                              }

repeat till a condition     for ( ;condition; )               while (condition)
                            {                                 {
                                //code                            //code
                            }                                 }

repeat forever              for ( ; 1; )                      while (1)
                            {                                 {
                                //code                            //code
                            }                                 }
```

for

```
int main()
{
    int x;

    for (x = 0; x < 10; x++)
    {
        printf("%d\n", x);
    }
}
```

x is set to zero, while x is less than 10 it calls printf to display the value of the variable x, and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

while

```
int main()
{
    int x = 0;            /* Dont forget to declare variables */
    while (x < 10)        /* while x is less than 10 */
    {
        printf("%d\n", x);
        x++;              /* Update x so the condition can be met eventually */
```

```
    }
}
```

do while

DO..WHILE loops are useful for things that want to loop at least once.

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

    x = 0;
    do
    {
        /* "Hello, world!" is printed at least one time
           even though the condition is false */
        printf( "Hello, world!\n" );
    } while ( x != 0 );
}
```

Keep in mind that you must include a trailing semi-colon after the while in the above example.

2.6 Declaration vs Definition

Notes: Declaration is an announcement and can be done 1 or more times Definition is an actual execution and should be done exactly once

2.7 Syntax and Semantics

Notes: Syntax is the Punctuation and Spellings of a Language Semantics is the Vocabulary and Grammar of the Language

2.8 Operators

All C operators do 2 things:

- Operates on its Operands
- Returns a value

Notes:

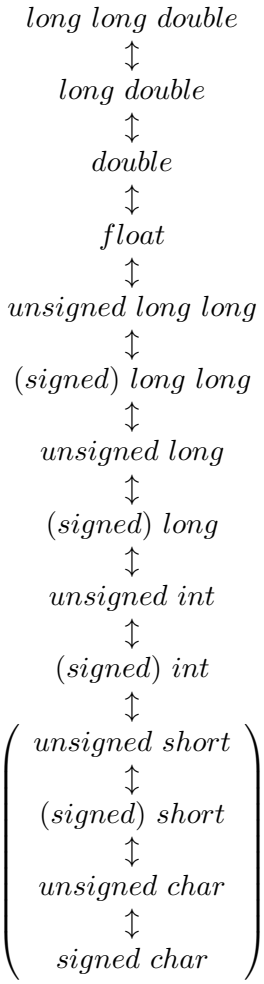
2.9 Operator Types

Based on:

- Number of operands
  - Unary
  - Binary
  - Ternary
- Operation
  - Arithmetic
  - Logical
  - Relational
  - Assignment
  - Bitwise
  - Language
  - Pointers
  - Miscellaneous

Notes:

2.10 Type promotion hierarchy



Notes:

2.11 Operation based Operators

Let's stride through examples:

2.11.1 Arithmetic

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

Notes:
Direction of expression evaluation vs Associativity

2.11.2 Logical

```
int main()
{
    int a = 1, b = 0;
    if (++a || ++b)
        printf("In first if a = %d, b = %d", a, b);

    a = 1, b = 0;

    if (b++ && ++a)
        printf("In second if a = %d, b = %d", a, b);
    else
        printf("In second if a = %d, b = %d", a, b);
    return 0;
}
```

Notes:

DeMorgan's Law trick

```
int main()
{
    int a, b = 0, c;

    if ((!(a < b) && !(b < c)) == (!((a < b) || (b < c))))
        printf("Hey it works");
    else
        printf("No! it did'nt!!!");
}
```

Notes:

2.11.3 Relational

```
int main()
{
    float f = 0.7;

    if(f == 0.7)
        printf("Equal");
    else
        printf("Not Equal");
}
```

Notes:

2.11.4 Assignment

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

    a += b += c += d += e += f;
}
```

Notes:

```
int main()
{
    int x = 0;

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

Notes:

2.11.5 Bitwise

```
int bitcount(unsigned char x)
{
    int count;

    for (count = 0; x != 0; x >>= 1);
        if (x & 01)
            count++;
    return count;
}
```

Notes:

2.11.6 Language

sizeof

Try this:

```
int main()
{
    int x = 5;

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

Notes:
3 reasons for why sizeof is not a function: 1) Any type of operands, 2) Type as an operand, 3) No brackets needed across operands

```
int main()
{
    int i = 0;
    int j = sizeof(++i);

    printf("%d:%d\n", i, j);
    /* Assume sizeof int is 4 bytes */
}
```

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

```
int main()
{
    int i;
    int array[5] = {0, 2, 4, 1, 3};

    for(i = -1; i < sizeof(array) / sizeof(int) - 1; i++)
        printf("%x\n", array[i + 1]);
}
```

Notes:

(type)

This operator is the complete follower of the type promotion table Notes:

2.11.7 Misc

Ternary Operator

What will the following expression evaluate to?
1 ? 5 : 4 ? 3 : 6
Notes:

Comma Operator

```
int main()
{
    int i = 0, j = 0;

    j = i++ ? i++ : ++i;
    j = i++ ? i++, ++j : ++j, i++;
}
```

Notes:

[ ] - Commutative like +

Notes:

```
int main()
{
    char a[] = "Hello World";
    int i;

    for (i = 0; i < sizeof(a) - 1; i++)
        printf("%c", i[a]);
}
```

Arrays

- Array Indexing
  Array index starts from zero. This idea is exploited in case of strings and arrays by having close relation with pointers in C. Technically, it is necessary for arrays to start with index 0 as the array name can be treated as the base address (a pointer). The following demonstrates the array-pointer equality:

```
int arr[10];
int firstElement = *(arr);
/* *(arr) is equivalent to *(arr+0) */
int fifthElement = *(arr+5)
/* had the index started from 1, this expression will access the */
/* sixth element and not the fifth element*/
```

- Array out of bounds
- Multi-dimension(Array of arrays)

```
typedef int IntArray30[30];
IntArray30 a;
```

Notes:

()

Is sizeof((x, y)) syntactically correct?
Notes:

. - Is it Commutative ?

Notes: We'll discuss on these during user-defined types discussion

2.11.8 Pointers

&, *, ->
Notes:
We'll discuss on these during pointers discussion

2.12 Precedence & Associativity of Operators

Operators	Associativity
() [] -> .	left to right
! ~ ++ −− + - * & (type) sizeof	right to left
/ % *	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Unary +, -, and * have higher precedence than the binary forms
Notes:

2.13 Non Operating Operators

Notes: Operator symbols used as not operators, e.g int a = 5; f(a, 3);

2.14 Embedded Care with Operators

2.14.1 Short Circuit Evaluation

There exists a short circuit evaluation for the logical operators && and ||. C introduced this tradition and is followed by the other three languages. The expression is evaluated only so long as to determine the truth-value of the expression (also referred to as truth-value context)

2.14.2 Operator Equivalence

The use of bitwise and logical operators seems to be equivalent in many cases. For example:

```
int main()
{
    int i = 1, j = 0;

    if ((i & j) == (i && j))
    {
        printf("Both seems be equivalent\n");
        printf("i = %d, j = %d", i, j);
    }
}
```

The difference lies in shortcut evaluation. See this:

```
int main()
{
    int i = 1, j = 0;
    if (i || (j = 1))
    {
        printf("But appearances can be deceptive \n");
        printf("i = %d, j = %d", i, j);
    }
}
```

Had you used a bit-wise operator, the whole expression would be evaluated with j set to one. Due to the short circuit evaluation, we get different answers.

2.14.3 Underflows and Overflows

In integral expressions, if the results cross the limits (say INT_MIN or INT_MAX), underflow or overflow occurs, as the case may be and the value rotates back according to the 2's complement representation of integers

In case of floating point numbers, it may lead to floating point exceptions (say SIGFPE). Now-a-days, most of the implementations provide support to IEEE 754 floating-point standard that supports the use of -INF, +INF (INF stands for infinity), and NaN. So, when an overflow occurs, it goes to either infinity or NaN, as the case may be

Notes:

2.15 Practice - 1

Read n and check prime number or not.

2.15.1 Prerequisite

1. Looping Concepts
2. Arithmetical Operators

2.15.2 Objective

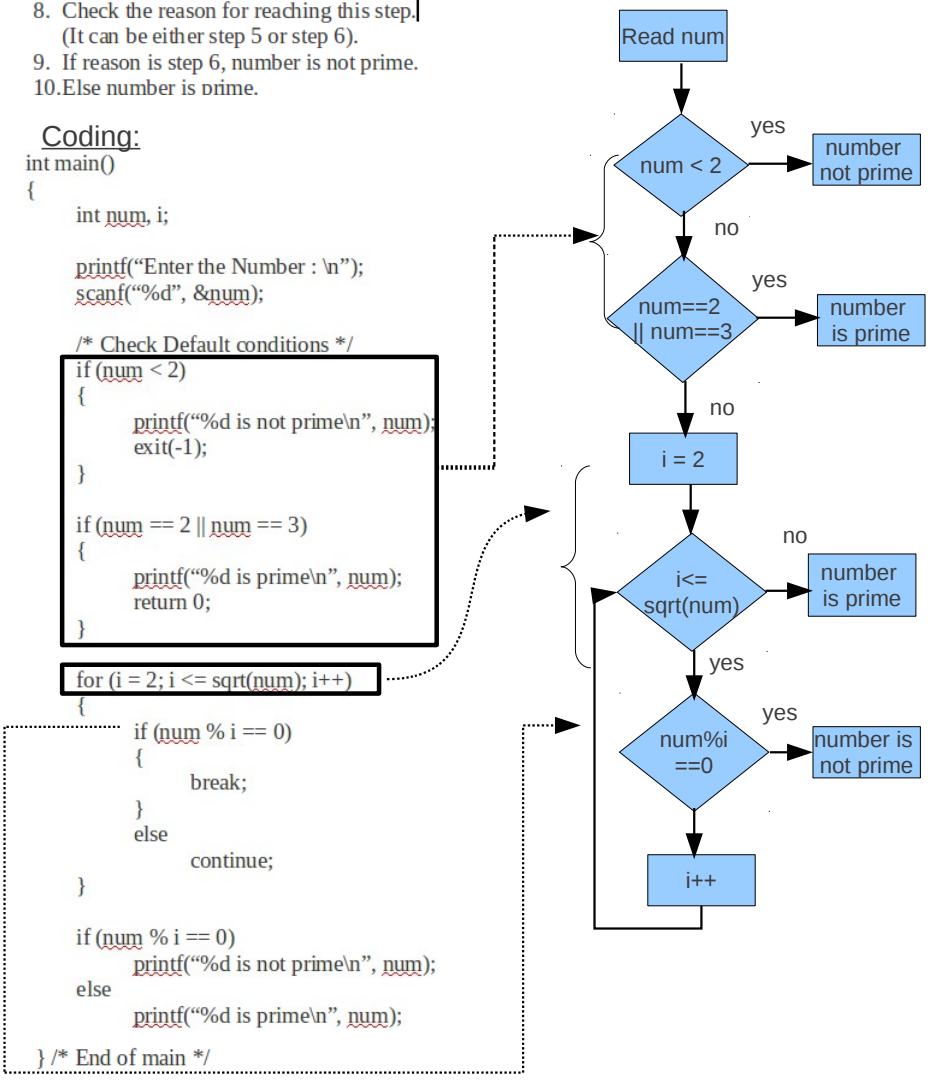
1. Understanding if else conditions
2. Understanding concept of continous looping

2.15.3 Algorithm Design

1. Read the number from the user.
2. Check the default conditions.
3. Assign i variable with a value 2.
4. Check i less than or equal to sqrt of num.
5. If not, go to step 8.
6. Else divide num and i, and if remainder is zero, go to step 8.
7. Else increment i value. Continue step step 4.
8. Check the reason for reaching this step.
   (It can be either step 5 or step 6).
9. If reason is step 6, number is not prime.
10. Else number is prime.

Coding:

```
int main()
{
        int num, i;

        printf("Enter the Number : \n");
        scanf("%d", &num);

        /* Check Default conditions */
        if (num < 2)
        {
                printf("%d is not prime\n", num);
                exit(-1);
        }

        if (num == 2 || num == 3)
        {
                printf("%d is prime\n", num);
                return 0;
        }

        for (i = 2; i <= sqrt(num); i++)
        {
                if (num % i == 0)
                {
                        break;
                }
                else
                        continue;
        }

        if (num % i == 0)
                printf("%d is not prime\n", num);
        else
                printf("%d is prime\n", num);

} /* End of main */
```



2.15.4 Dry Run

2.15.5 Practical Implementation

1. Encryption
2. Psuedo random Generator
3. Computer hash tables

2.16 Practice - 2

Read n and print the greatest fibonacci no $\leq$ n

2.16.1 Prerequisite

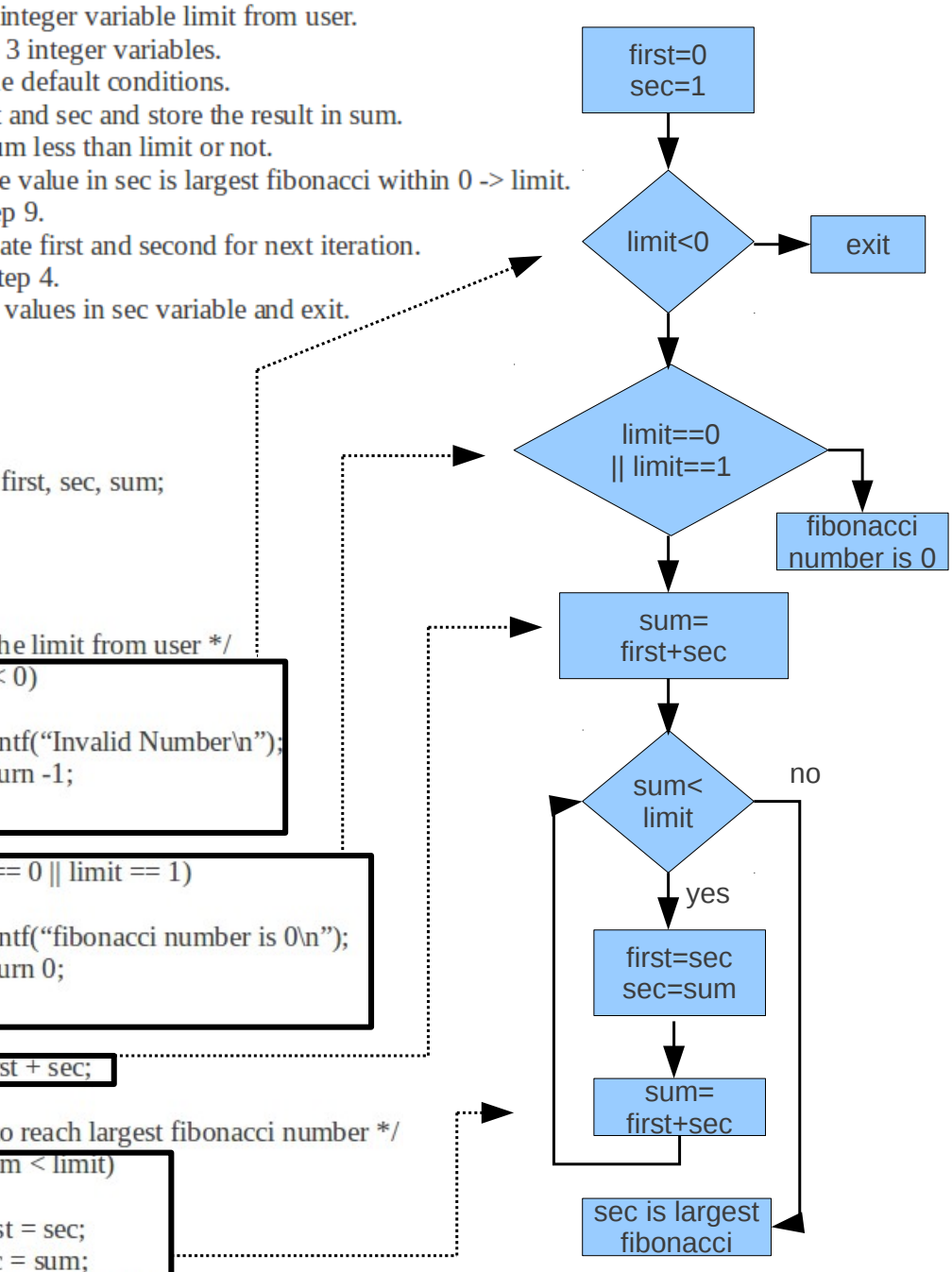
1. Looping Concepts
2. Arithmetical Operators

2.16.2 Objective

1. Understanding if else condition
2. Understanding continous looping

2.16.3 Algorithm Design

1. Read an integer variable limit from user.
2. Initialise 3 integer variables.
3. Check the default conditions.
4. Add first and sec and store the result in sum.
5. Check sum less than limit or not.
6. If not, the value in sec is largest fibonacci within 0 -> limit. Go to step 9.
7. Else update first and second for next iteration.
8. Repeat step 4.
9. Print the values in sec variable and exit.




```
int main()
{
        int limit, first, sec, sum;

        first = 0;
        sec = 1;

        /* Read the limit from user */
        if (limit < 0)
        {
                printf("Invalid Number\n");
                return -1;
        }

        if (limit == 0 || limit == 1)
        {
                printf("fibonacci number is 0\n");
                return 0;
        }

        sum = first + sec;

        /* Loop to reach largest fibonacci number */
        while (sum < limit)
        {
                first = sec;
                sec = sum;
                sum = first + sec;
        }

        printf("largest fibonacci number within
                        %d is %d\n", limit, sec);

        return 0;
} /* End of main */
```

2.16.4 Dry run

2.16.5 Practical Implementation

1. Fibonacci numbers are used by some pseudorandom number generators.
2. Fibonacci numbers are used in a polyphase version of the merge sort algorithm.

2.17 Quiz

1. (1 point) What is the output of the following code ?

```
int func()
{
    const int initial = 1;
    static int count = initial;
    return ++count;
}
```

2. (1 point) How many distinct values can you represent with a sequence of 3 bits?

3. (1 point) How many bits does a single hexadecimal digit(eg F) usually represent?

4. (1 point) sizeof(int [0])

5. (1 point) What is the output of the following code?

```
int main()
{
    char ch1 = 5, ch2 = 250;
    printf("%u, %u, %u\n", 5 - 250, (char)5 - (char)250, ch1 - ch2);
}
```

6. What is the return type of sizeof.

7. Explain the difference between compile time and run time operators with examples.

8. What is type promotion.

2.18 Lab Work

Run all the related templates and understand them.Complete all the assignments in 3days

(Id) / Date	Assignment Topic
( ) _____	Read n and Print the greatest fibonacci no $\leq$ n Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Read n and check for its perfectness Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Print all ASCII characters Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( )	Try out examples relating mod and div with with negative numbers on your own Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Print sizes of all basic data types of C Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Sieve of Eratosthenes Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Read 3 nos a, r, n. Generate AP, GP, HP Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Print Hello world in X formation Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	List 5 different compiler dependent independent expressions Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( )	Read n and generate fibonacci nos. $\leq$ n Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Read n and n nos. of ints and print the median of those Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Given a number between 1 to 365 (incl), find which day is it Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Verify type promotion table using sizeof Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Write a bitwise program to check a number is even or odd Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Print bits of signed & unsigned types and check for 2's complement Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

Chapter 3

Day 3: Functions

3.1 Why functions?

A C function prototype just specifies the interface and hence it can be separated from the actual implementation. This helps us in having inter-dependant translation units arranged in separate files. The actual implementation can be in a file and only the prototype can be used in the current file for using it. It is tedious to find out the prototypes of the functions used and provide it in every file where the function is used. Further, when the function's prototype is modified, the change has to be reflected to all the files that uses it. The header files avoid this problem and they generally contain the declarations of functions and other types. The #include directive takes the responsibility of replacing the header files in the source files.

Notes:

3.2 Parameters, Arguments and Return Values

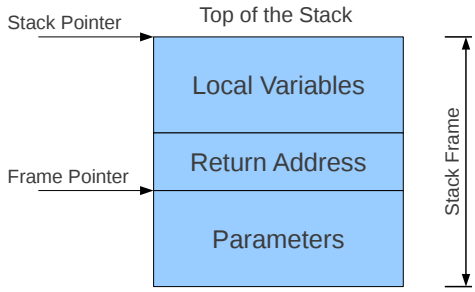
Parameters are also commonly referred to as arguments, though arguments are more properly thought of as the actual values or references assigned to the parameter variables when the subroutine is called at runtime. When discussing code that is calling into a subroutine, any values or references passed into the subroutine are the arguments, and the place in the code where these values or references are given is the parameter list. When discussing the code inside the subroutine definition, the variables in the subroutine's parameter list are the parameters, while the values of the parameters at runtime are the arguments.

Many programmers use parameter and argument interchangeably, depending on context to distinguish the meaning. In practice, distinguishing between the two terms is usually unnecessary in order to use them correctly or communicate their use to other programmers

```
int add(int a, int b);          /*declaration*/
int main()
{
    int i, j, sum;

    sum = add(i, j);            /*function call*/
    printf("\nSum of %d and %d is %d\n", i, j, sum);
    return 0;
}

int add(int a, int b)            /*definition*/
{
    int val_to_return;

    val_to_return = a + b;
    return val_to_return;
}
```

Notes:

3.2.1 Function and the Stack



Notes:

3.3 Procedures vs Functions

Notes: Procedure is a function which returns nothing, i.e. void

3.4 Various parameter passing mechanisms

- Call by value
- Call by reference / address / location / variable /
- Copy-restore / Copy-in copy-out / Value-result
- Call by name

Notes:

3.4.1 C's only parameter passing mechanism: Pass-by-value

C supports only one argument passing mechanism: pass-by-value. When the pointers are copied passed by value, it looks similar to pass-by-address or pass-by-reference, but it is only pass-by-value only. Macros in C are the way that looks like pass-by-name, but it is error-prone (see the chapter "Preprocessor" for more details).

As opposed to the functions where we pass the value as arguments, in macros, we pass the names to the arguments. For example:

```
#define arrSize(array) (sizeof(array) / sizeof(array[0]))
void foo()
{
    int iArr[10];
    char cArr[5][15];
    printf("Size of iArr = %d\n", arrSize(iArr));
    printf("Size of iArr = %d\n", arrSize(cArr));
}
```

Remember, you cannot obtain this functionality with the other type of argument passing mechanisms. However, this mechanism will never check for types and is error-prone.

Coming to pass-by-value, in functions, we can pass the value of the variable alone and a copy is saved in the calling function. This leaves the original variable unaffected by the changes inside the function. This fact is exploited much in the recursive functions:

```
int is_even(int num)
{
    if (num & 1)
        return 0;
    else
        return 1;
}
int main()
{
    ret = iseven(14);
    ret ? printf("%d is even\n", ret): printf("%d is odd\n", ret);
    return 0;
}
```

Passing Pointers: Consider an example, where we have to apply a filter to a JPEG image. We can do it by passing the original image as an argument and the filtered image would be the return value:

```
JPEGImage apply_filter(JPEGImage image)
{
    /* apply filter */
    return image;
}
```

This apply_filter function has several disadvantages. When we have a local variable, we allocate space in the function stack and to initialize/destroy the whole object. This is inefficient in terms of space and time. Consider the situation where the JPEGImage would take 50 KB of memory and initializing it would take 0.5 seconds. This could be right scenario where we can make use of pass-by-address.

In pass-by-address, the calling function passes the address of the argument and the called function makes uses of the address to manipulate it. The same function could be written as:

```
void apply_filter(JPEGImage *image)
{
    /* apply filter */
}
```

So, only the address is copied into the stack. This is very advantageous and efficient in places where the object is very to copy the object.

DIU: Which kind of parameter passing method is mentioned below ?

```
void twotimes(int x)
{
    printf ("\nFUN:Argument before change took place %d",x);
    x *= 2;
    printf ("\nFUN:Argument after tempering with it %d",x);
}
int main(void)
{
    int number=10;
    printf ("\nMain:Number before function-call %d",number);
    twotimes(number);
    printf("\nMain:Number after function-call is %d",number);
}
```

3.5 Ignoring Function's Return Value

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

```
int i = get_an_integer();
/*
 * fetch the integer in i
 * or
 */
get_an_integer();
/* ignore the return value */
```

You may think that, the programmer could have forgotten to fetch the value and it would be very helpful if the compiler warns him or produces the error telling to capture the return value. However, this can be very helpful in many cases when you need to call the method only for its behavior and not for the result it may return back. So it is convenient for the programmers that they need not fetch the value (sometimes by creating a local variable just to fetch it) and ignore it later. For example, many C programmers have used the printf function in C without knowing that it returns the number of characters printed; or the scanf function that returns the number of items successfully scanned.
Notes:

DIU: Pass an integer array to a function and display the contents in the array.

3.6 Returning an array from a function

Explain the different errors and bugs in the following codes.

```
int n, arr[10], *top;

int *top2(int n, int array[])
{
    int biggest, second_biggest;
    int big2[2];

    /* Do calculation here */

    big2[0] = biggest;
    big2[1] = second_biggest;
    return big2;
}
int main()
{
    /* Read n & n elements in arr */
    top = top2(n, arr);
    printf("%d:%d\n", top[0], top[1]);
    return 0;
}
```

Notes:

```
int main()
{
    printf("%s\n", fun());
    return 0;
}
void fun()
{
    char buff[] = "Hello World";
    return buff;
}
```

Notes:

3.7 main & its arguments

Notes: In C, main is considered special and is much different from other user-defined functions. The parameters with which main can be declared are restricted. The first argument has the number of arguments (conventionally called as argc), second holds the string arrays having actual arguments (argv) and third one is the optional parameter which holds the string array (the array has a null terminated string at the end) that has list of environment variables (argp). The main() function is considered to be declared implicitly by the compiler, but defined by the user. The parameters to main() are passed from command line instead of through a function call. main() also considered to do implicit return of 0 if no explicit return is provided. Though the compiler might allow the user to define the return type of main as void or other types, the main is treated as having return type int.

The prototype for the main() is

```
int main(void);
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char *envp[]);
```

- argc - no. of command line arguments passed to the main from the shell prompt.
- argv - is an array of pointers to the list of arguments(all are charecter strings)
- envp - is an array of pointer to the environment variable

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    int i;
    char **env = envp;

    printf("\n The count is %d:", argc);
    for (i = 0; i < argc; i++)
        printf("%s \n", argv[i]);
    while (*env)
        printf("%s \n", *env++);

    return 0;
}
```

./a.out April 17 2012

3.7.1 3 ways of taking input

- Through user interface

- Through command line arguments
- Through environment variables

3.8 Function Type

The functions can be considered to be a type in C: you can apply operators *, & to functions as if they are variables, and function definitions reserve a space (in code area of the program), function calls can participate in expressions as if they are variables and in that case, the type of the function is its return type.

```
/* function definition*/
int foo() { return 0; }
/* the function pointer can hold the function*/
int (*fp)() = foo;
/* using & for function is optional to take the address of the function*/
fp = &foo;
/*the value of i is 0; the type of the expression 10 * foo() is int*/
int i = 10 * fp();
```

Notes:

```
typedef int F(void);
```

3.9 Variable argument functions

- The header: #include <stdarg.h>
- The type: va_list ap;
- The macros: va_start(ap, last) va_arg(ap, type) va_end(ap)

Example:

```
double calc_mean(int num, ...)
{
    va_list ap;
    double val;
    int i;

    va_start(ap, num);
    val = 0;
    for (i = 0; i < num; i++)
    {
        val += va_arg(ap, double);
    }
    va_end(ap);
    return (val / num);
}
```

Notes:

3.10 Practice - 1

Write a function (my_ispunct) to check whether a character is punctuation or not. It should check whether the character is 'any printable character which is NOT a space or an alphanumeric character'

3.10.1 Prerequisite

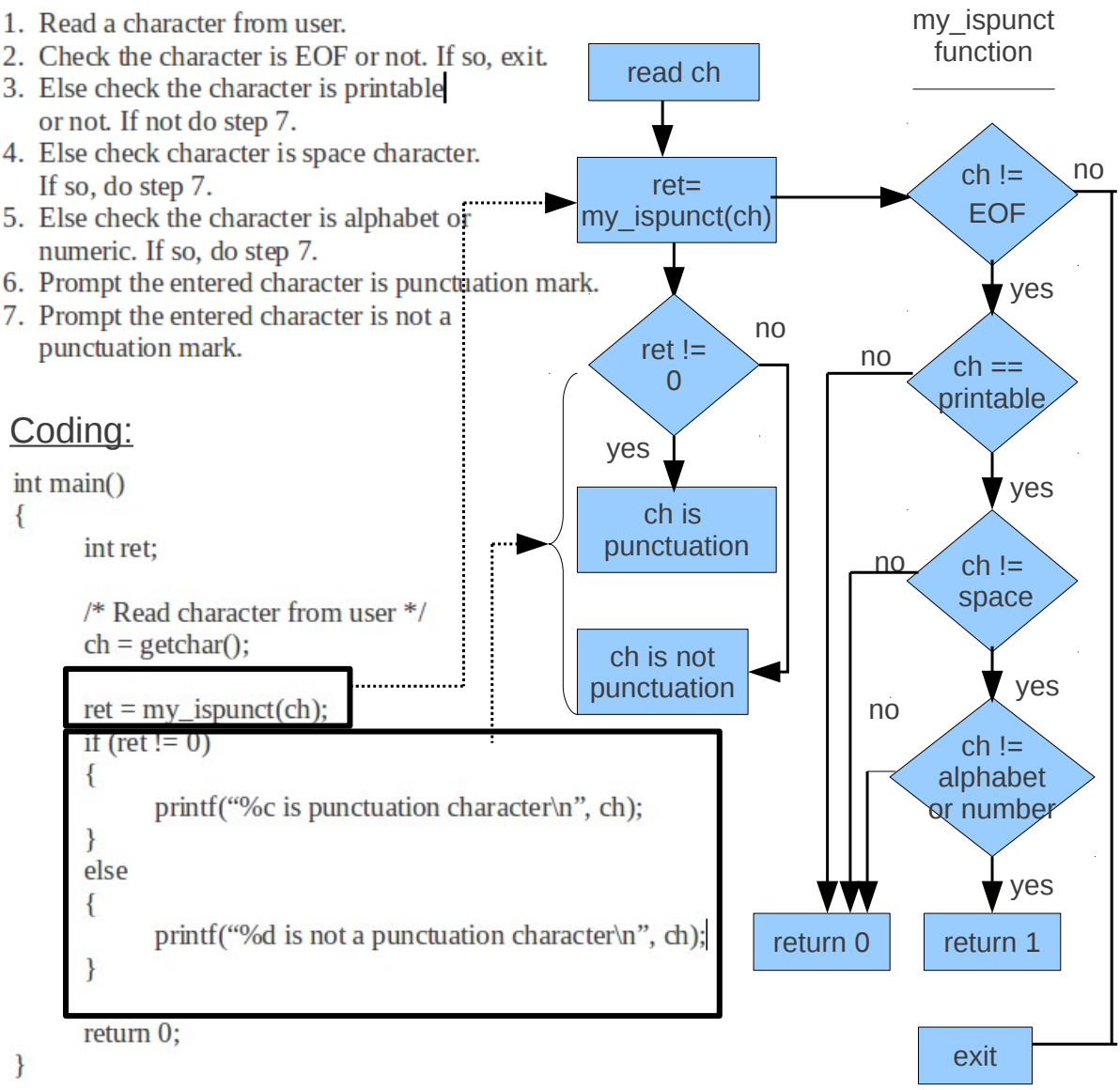
1. Function Concepts
2. getchar usage
3. ascii table
4. logical operators

3.10.2 Objective

1. Implementing and understanding user defined functions.
2. Understanding modular approach of programming.

3.10.3 Algorithm Design

1. Read a character from user.
2. Check the character is EOF or not. If so, exit.
3. Else check the character is printable or not. If not do step 7.
4. Else check character is space character. If so, do step 7.
5. Else check the character is alphabet or numeric. If so, do step 7.
6. Prompt the entered character is punctuation mark.
7. Prompt the entered character is not a punctuation mark.



Coding:

```
int main()
{
        int ret;

        /* Read character from user */
        ch = getchar();

        ret = my_ispunct(ch);
        if (ret != 0)
        {
                printf("%c is punctuation character\n", ch);
        }
        else
        {
                printf("%d is not a punctuation character\n", ch);
        }

        return 0;
}

int my_ispunct(int ch)
{
        if (ch != EOF && isprint(ch) && !isalnum(ch) && !isspace(ch))
        {
                /* Entered character is punctuation character */
                return 1;
        }
        else
        {
                /* Not a punctuation character */
                return 0;
        }
}
```

3.10.4 Dry run

3.10.5 Practical Implementation

1. C standard function.
2. Used for parsing the strings.

3.11 Practice - 2

Write a function for Swapping two integers by using pass by reference method.

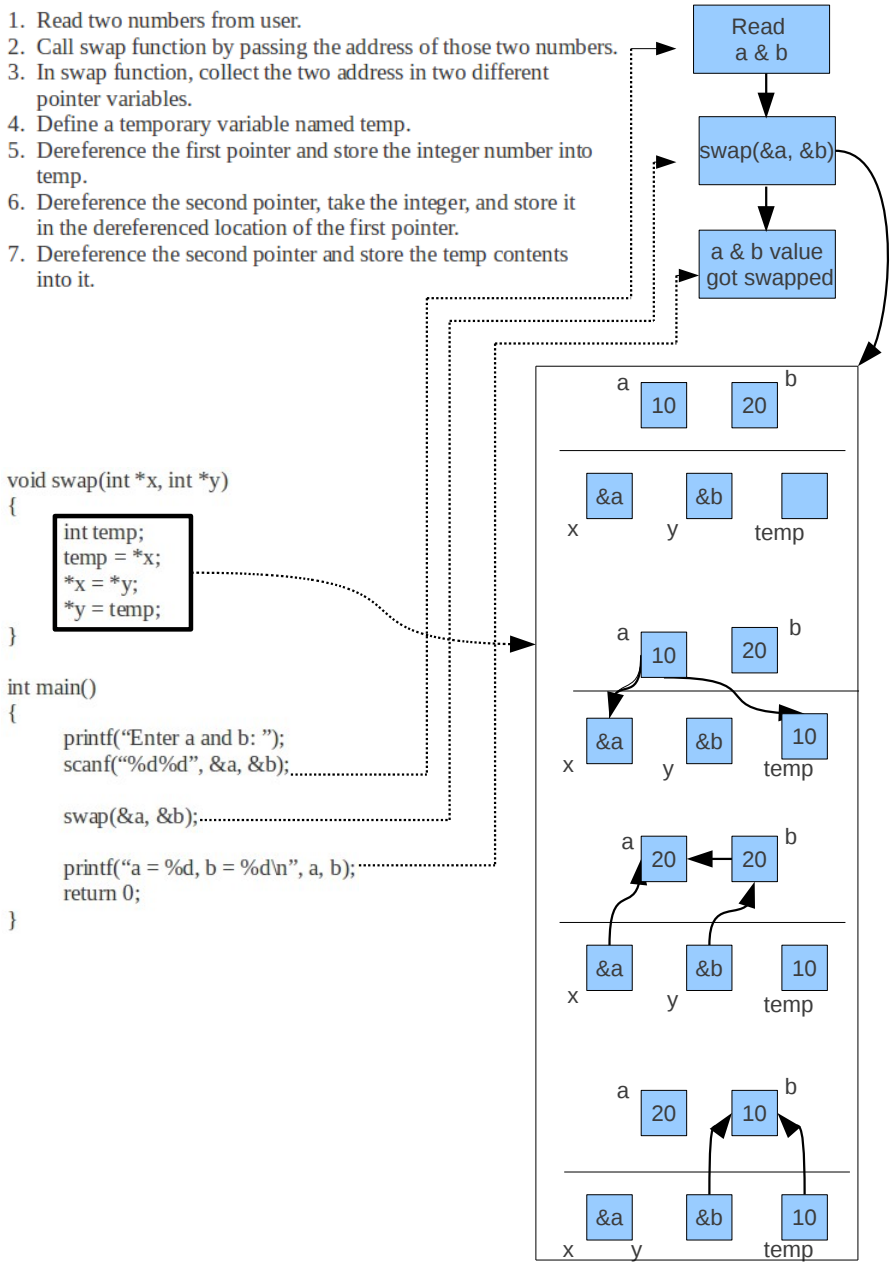
3.11.1 Prerequisite

1. Function Concepts
2. Pass by reference method

3.11.2 Objective

1. Implementing and understanding user defined functions.
2. Implementing and understanding pass by reference method.

3.11.3 Algorithm Design



3.11.4 Dry run

3.11.5 Practical Implementation

1. Swapping the contents in two memory locations (can be just integer or an address or a big structure).

3.12 Quiz

Short Answer

```
1. What are the difference between formal arguments and actual
arguments in a function ?

2. Are parameter names mandatory in function prototype ? Give an example ?

3. What information does function prototype carry ? Give an example ?
```

3.13 Lab Work

Run all the related templates and understand them. Complete all the assignments in 3 days

(Id) / Date	Assignment Topic
( ) _____	Implement your own ctype library. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

( )
_____
	Implement below mentioned bitwise functions,
	int get_nbits(int num, int n);
	int set_nbits(int num, int n, int val);
	int get_nbits_from_pos(int num, int n, int pos);
	int set_nbits_from_pos(int num, int n, int pos, int val);
	int toggle_bits_from_pos(int num, int n, int pos);
	int print_bits(unsigned int num, int n);
	
	Prerequisites:
	
	Algorithm:
	
	Dry Run:
	
	Objective:
	
	Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Read int i, Read 0 ≤ a ≤ b ≤ 31. Read an int n, put the (b-a+1)lsb's of n into i[b:a] Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Read an int, & a no. n. Circular right & left shift the int by n. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Create a library file named libbitwise.c and include all bitwise functions in it. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Write a C program to swap two variables by using pass by reference method. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Average n number by taking input in 3 different ways Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Recursive factorial without using any other function than main Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Write the functions for post and pre increment, passing int pointer as their parameter Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Fibonacci with using its recursive relation Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	

Chapter 4

Standard Input / Output

4.1 printf & scanf

4.1.1 The first parameter, format string

Notes:

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

int main()
{
    int a = 0;
    char str[10];

    scanf("%d%s", &a, str);
    printf(str);
    printf("%d%s", a, str);
}
```

4.1.2 Pointers with printf and scanf

Which of the following is a valid input parameter to scanf?
◇ &i, &a[i], a + i, &c, &str[i], str + i
◇ Is &str ≡ &str[0] ≡ str?
What's the output:
◇ printf("%c", *("C is an Ocean" +5));
◇ printf("%c", *(&5["C is an Ocean"] -1));
Notes:

4.1.3 Return values

printf returns the number of characters printed.
scanf returns the number of items successfully read.
Notes:

4.1.4 Eating whitespaces by %d, %f, ...

scanf eats white spaces to be able to read any integer or real equivalent number.
Notes:

4.2 % character conversion table

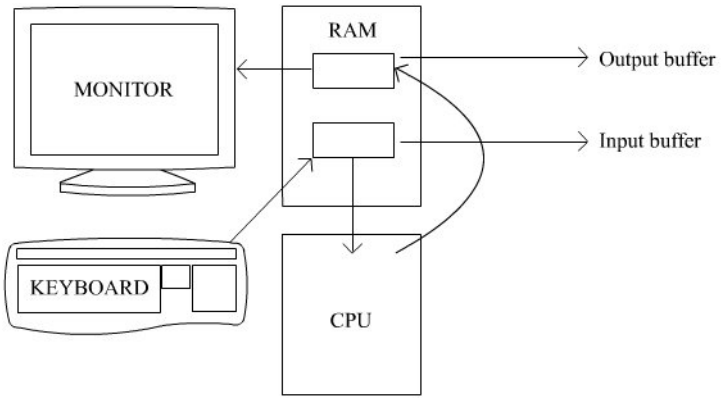
Format	Prints argument as
\c	Escape sequences that are converted into the characters they represent: \ (backslash), a (alert), b (backspace), f (formfeed), n (newline), r (carriage return), t (horizontal tab), v (vertical tab) any other character not mentioned above (causes printf to ignore any remaining characters in format) 0nnn, where nnn is a 1, 2 or 3-digit octal number to be converted to a byte with the corresponding octal value
%c	the character referred to by the least significant 8 bits of the numeric value of argument; truncates argument to the nearest integer
%d	a decimal integer (0 if no arguments are available); truncates argument to the nearest integer
%e	a floating point number (0.0 if no arguments are available); the format used is [-]d.ddde[+—-]dd where the number of digits after the decimal point is controlled by the value of precision
%E	a floating point number (0.0 if no arguments are available); the format used is [-]d.dddE[+—-]dd where the number of digits after the decimal point is controlled by the value of precision
%f	a floating point number (0.0 if no arguments are available); the format used is [-]ddd.ddd where the number of digits after the decimal point is controlled by the value of precision
%g	a floating point number (0.0 if no arguments are available); the format used is the most compact result from applying the 'e' and 'f' conversions
%G	a floating point number (0.0 if no arguments are available); the format used is the most compact result from applying the 'E' and 'f' conversions
%lf	a double (printf only)
%h	a short integer
%i	a decimal integer (0 if no arguments are available); truncates argument to the nearest integer
%l	a long integer
%L	a long long integer
%ll	a long long integer
%o	an unsigned octal number (0 if no arguments are available); truncates argument to the nearest integer
%p	a pointer
%s	a string
%u	an unsigned decimal integer (0 if no arguments are available); truncates argument to the nearest integer
%x	an unsigned hexadecimal number displayed in lowercase display (0 if no arguments are available); truncates argument to the nearest integer
%X	an unsigned hexadecimal number displayed in uppercase display (0 if no arguments are available); truncates argument to the nearest integer
%%	prints a %; no argument is converted
%*	(scanf) ignore the value read; (printf) replace by the value of its corresponding parameter
%[ ]	pattern matching read (scanf only)

4.3 Analyze the following loop

```
int main()
{
    char ch;

    scanf("%[^,]", &ch);
    while (ch != 'e')
    {
        switch (ch)
        {
            case 'a':
                break;
            case 'b':
                break;
            default:
                break;
        }
        printf("%c\n", ch);
        scanf("%[^,]", &ch);
    }
    /*
     * Input given is:
     * a,b,c,d,e,f,g,h
     */
}
```

Find out the Bug in the above code.

Notes:



Notes:

- Five reasons for a flush
  - Buffer full
  - fflush
  - `\n`
  - Normal program termination
  - Read

Notes:

4.4 Practice - 1

Read the strings entered by the user and print the largest line among that.

4.4.1 Prerequisite

1. Array Concepts
2. fgets usage

4.4.2 Objective

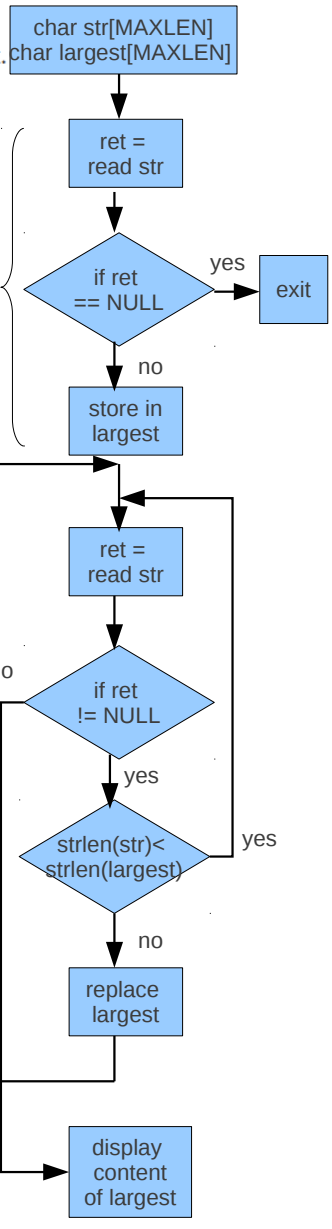
1. Understanding and Implementing fgets function.
2. Understanding simple array concepts.

4.4.3 Algorithm Design

1. Create an array of MAXLEN length, named largest.
2. Read the strings from user till EOF.
3. While reading first string, store it into the array largest.
4. For subsequent reads, compare the read string with the string stored in largest array.
5. If the the read string is bigger then the one in array, then replace the string in array by the read one.
6. If user hits EOF, print the string inside largest.

Coding:

```
#define MAXLEN        40

int main()
{
    char str[MAXLEN];
    char largest[MAXLEN];

    if (fgets(str, MAXLEN, stdin) != NULL)
    {
        strcpy(largest, str);
    }
    else
    {
        return 1;
    }
    while (1)
    {

        /* Read a string and check EOF or not */
        if (fgets(str, MAXLEN, stdin) != NULL)
        {

            if (strlen(str) > strlen(largest))
            {
                strcpy(largest, str);
            }
            else
                continue;
        }
        else
        {
            /* Loop terminates */
            break;
        }
    }

    printf(" The largest string entered is %s\n", largest);
    return 0;
}
```



4.4.4 Dry run

4.4.5 Practical Implementation

1. Collecting the biggest line from database.

4.5 Quiz

Short Answers

1. What is the output for the following code ?

```
printf("Hello World" + 4);
```

2. What would be the output of the following C program ?

```
#include <stdio.h>
int main()
{
    int i=43;
    printf("%d",printf("%d",printf("%d",i)));
    return 0;
}
```

3. Why must x be preceded by & inside scanf ?

4. Why getchar return value is of type int ?

4.6 Lab Work

Complete all the templates and asignments by 2 days

(Id) / Date	Assignment Topic
( ) _____	Write a program to count number of characters, words and lines, entered through stdin. Prerequisites: Algorithm: Dry Run: Objective:
( ) _____	Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank. Prerequisites: Algorithm: Dry Run: Objective:
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	

Chapter 5

Day 4: Input / Output - Files

5.1 What is a file?

File as a sequence of bytes
Notes:

5.2 Why files?

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

Notes:

5.3 The functions for file operations

- fopen - Open. FILE *fopen(const char *path, const char *mode);
- fwrite - Write → Read Modify Write. size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
- fread - Read. size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
- fclose - Close. int fclose(FILE *fp);
- fseek - Seek. int fseek(FILE *stream, long offset, int whence);
- feof - End of File check. int feof(FILE *stream);

Notes:

5.3.1 fgetc()

Notes:

5.3.2 Modes the file can be opened

The various modes in which a file can be opened using fopen:

r: Open text file for reading. The stream is positioned at the beginning of the file.

r+: Open for reading and writing. The stream is positioned at the beginning of the file.

w: Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+: Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a: Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+: Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

Notes:

```
int main(int argc, char *argv[])
{
    FILE *fp;
    int ch;

    if (argc == 1)
    {
        printf("Usage: %s <file_to_cat>\n");
        return -1;
    }

    fp = fopen(argv[1], "r");
    if (fp == NULL)
    {
        perror("fopen");
        return -1;
    }
    while ((ch = fgetc(fp)) != EOF)
    {
        fputc(ch, stdout);
    }
    return 0;
}
```

DIU: Try the above program by using fgets and fputs.

5.4 Practice - 1

Reverse a file.

5.4.1 Prerequisite

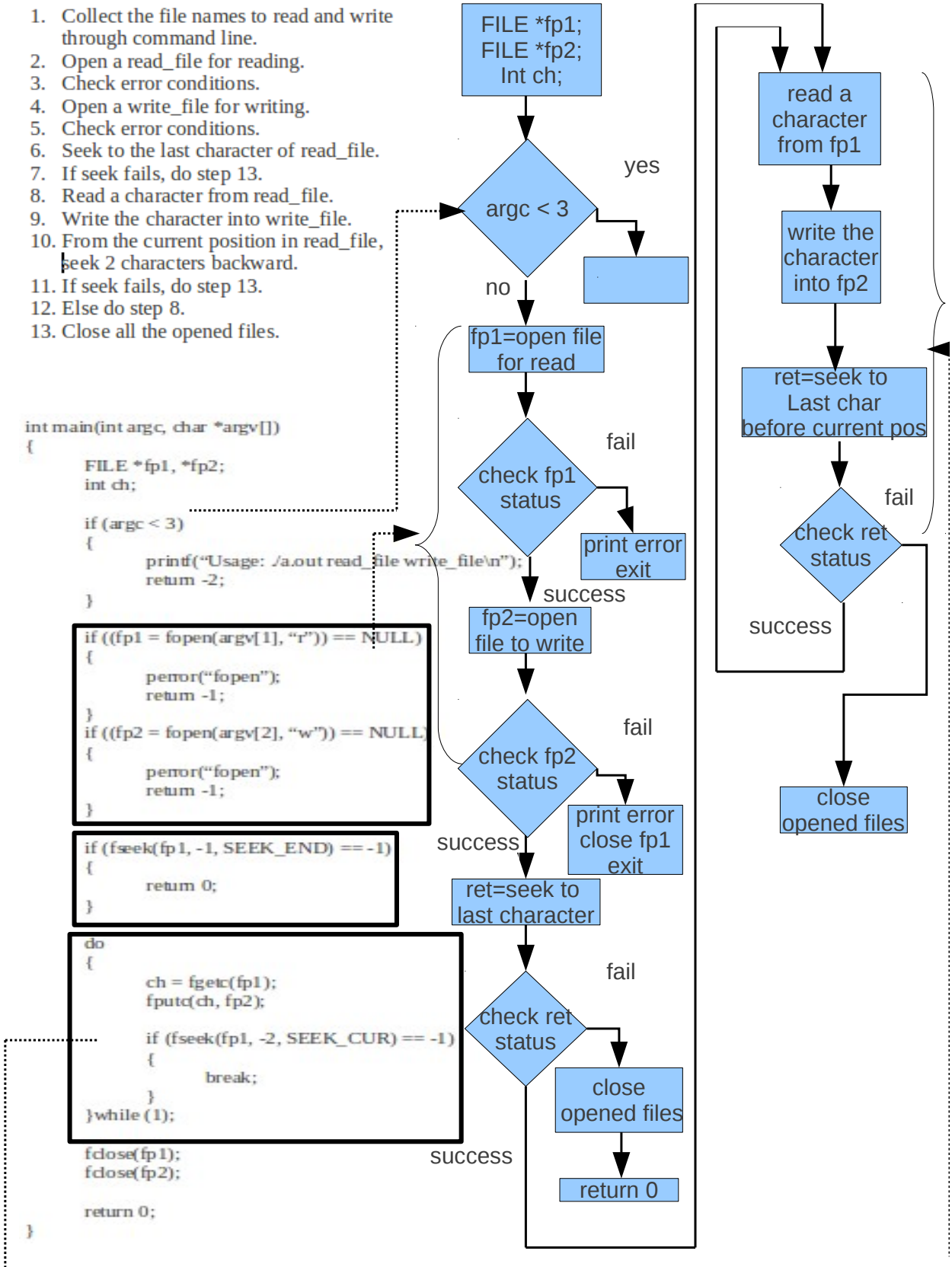
1. Familiarity in File related functions

5.4.2 Objective

1. Using file related functions like fopen, fseek, fgetc, fputc, flose.

5.4.3 Algorithm Design

1. Collect the file names to read and write through command line.
2. Open a read_file for reading.
3. Check error conditions.
4. Open a write_file for writing.
5. Check error conditions.
6. Seek to the last character of read_file.
7. If seek fails, do step 13.
8. Read a character from read_file.
9. Write the character into write_file.
10. From the current position in read_file, seek 2 characters backward.
11. If seek fails, do step 13.
12. Else do step 8.
13. Close all the opened files.



5.4.4 Dry run

5.4.5 Practical Implementation

1. Usage of fseek will help to do Random access of data among the databases.

5.5 Quiz

```
1. What is the result of calling fopen continously without fclose ?

2. Why fgets is preferred over gets ?

3. While I am writing buffers to the file and parallely reading them,
   I am not getting the buffer which I wrote. What can be problem.
   How can I solve it ?

4. What will be the output ?
   #include <stdio.h>
   #include <unistd.h>
   int main()
   {
       while(1)
       {
           fprintf(stdout,"hello-out");
           fprintf(stderr,"hello-err");
           sleep(1);
       }
       return 0;
   }
```

5.6 Lab Work

Complete all assignments in 2 days with progress on the project

(Id) / Date	Assignment Topic
( ) _____	Read input till eof and print no of words read. Prerequisites: Algorithm: Dry Run: Objective:
( ) _____	Copy one file into another Prerequisites: Algorithm: Dry Run: Objective:
( ) _____	Concatenate two files, as ./your_cat file1 file2 [filenew] and like the command 'cat' Prerequisites: Algorithm: Dry Run: Objective:
( ) _____	
( ) _____	
( ) _____	
( ) _____	
( ) _____	

Chapter 6

Day - 5 & 6: Strings & Pointers

6.1 Strings

C strings are low-level in nature (and bug-prone). C does not support strings as a data type - it is just an array of characters. But it has library support for strings (as in strstr, strcmp etc.) in literals. This leads to the close relationship between arrays and strings in C.

```
if(sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
    printf("WoW\n");
else
    printf("Huh\n");
```

Notes:

6.2 Initializing a string

```
char a[5] = "Hello", b[] = "Hi", c[5] = "Hi";
char str1[] = "Hi", str2[] = {'H', 'i'};
printf("%s:%s:%s", a, b, c);
printf("%s:%s", str1, str2);
```

Notes:

6.3 Sizes of

```
char *str1 = "Hi";
char str2[] = "Hi";
printf("%d", sizeof(str1));
printf("%d", sizeof(str2));
```

Notes:

6.4 String Manipulations

Strings are character arrays and an array name refers to an address - a pointer constant. So, there is a close relationship between arrays, string and pointers. Exploiting this makes string manipulation a very efficient one. For example:

```
int my_strlen(const char *s)
{
    char *t = s;
    while (*t++)
        ;
    return t-s-1;
}
```

Notes:

Another way of writing the above code is

```
int my_strlen(const char *s)
{
    int i = 0;
    while (s[i] != '\0')
    {
        i++;
    }
    return i - 1;
}
```

DIU: Implement strcpy function.

On the other hand, this also leads to lots of inconsistencies and problems. Consider:

```
char s1[10] = "string";
char s2[10];
s2 = "string";
/* error! only initialization is possible and not the */
/* assignment - because array assignment is not possible */

char *s3 = "string";
char *s4;
s4 = "string";
/* s3 is a pointer, so both initialization */
/*    and assignment are possible */

s1[0] = 'S';
/* O.K. s1 now contains the string "String" */

s3[0] = 'S';
/* It points to a string literal - it is read only */
/* So, undefined behavior */

/* assume that sizeof pointer is 4 bytes */
printf(" \%d  \%d  \%d ", sizeof(s1), sizeof(s3), sizeof("string"));
/* sizeof the array, pointer and string literal respectively.*/

printf(" \%d  \%d  \%d ", strlen(s1), strlen(s3), strlen("string"));
```

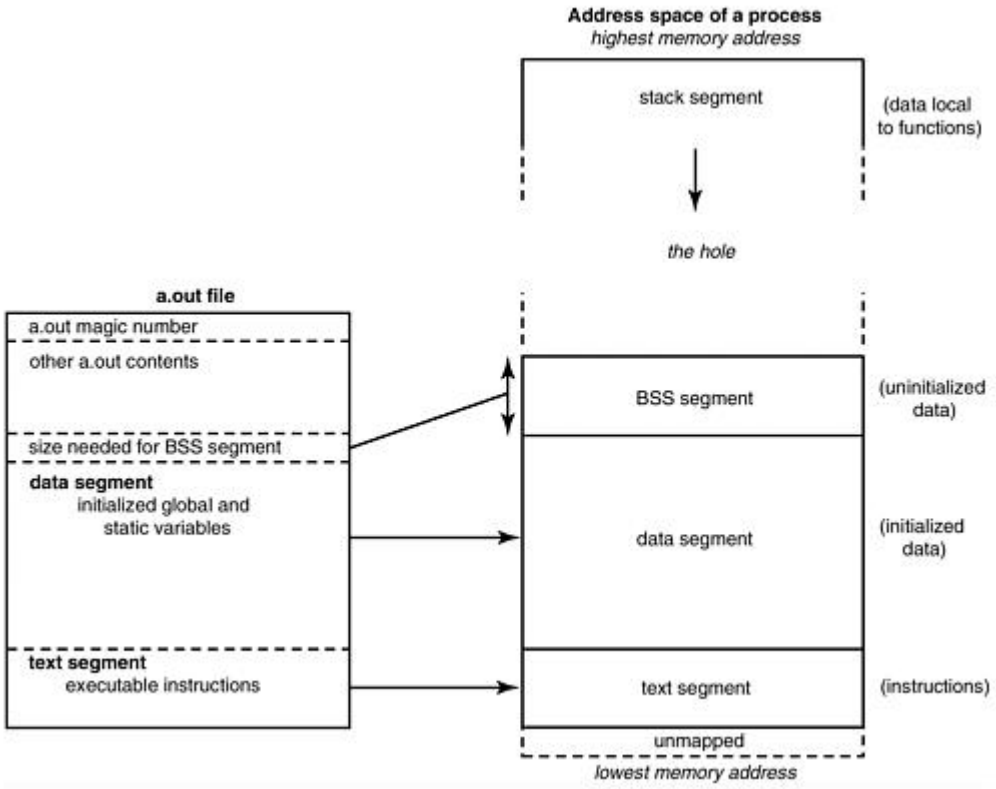
Notes:

In C, the strings are always NULL terminated. This is often convenient but also bug prone. Consider the example of strcpy() - inside the function there is no way to check neither the argument is properly null terminated nor the target is capable enough to hold the source. In both cases, it leads to undefined behavior because it reads/writes past the array bounds.

The serious disadvantage of this representation is to know the length of the string, traversal has to be made until the end of the string. Except such inconveniences, the C string implementation works out better in terms of efficiency and ease of implementation.

Notes:

6.5 The program segments



List the segments of a program:

- Code
- Data
- Stack
- Heap

Notes:

BSS: Block Started by Symbol

Example: Modifying the constant string

```
char *a = "Hello World";
main()
{
    a[1] = 'i';
    printf("%s", a);
}
```

Notes:

6.5.1 Shared Strings

In C, string constants (string literals) are shared as they cannot be modified. Consider the following code:

```
char *s1 = "string";
char *s2 = "string";
s1[0] = 'S';
printf("%s", s2);
/* may print 'String' !!! */
```

Since string literals are immutable in C, the compiler is free store both the string literals in a single location and so can assign the same to both s1 and s2. Such sharing of string is an optimization technique done by the compiler to save space.

It is easy to check if the two string literals are shared or not in your compiler/platform.

```
if(s1 == s2)
{
    printf("Yes. shared strings");
}

/* or check it directly with the string constants */
if("string" == "string")
{
    printf("Yes. shared strings");
}
```

This checking work on the common sense, that no two different string constants can be stored in the same location.

Notes:

6.6 Why pointers?

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

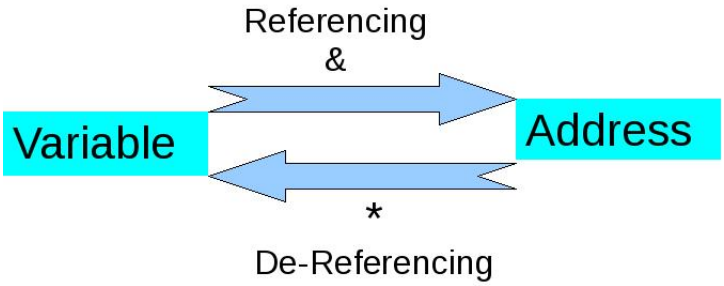
Notes:

6.7 Pointers & the 7 rules

6.7.1 Rule #1: Pointer as a integer variable

Notes:

6.7.2 Rule #2: Referencing & Dereferencing



Notes:

6.7.3 Rule #3: Type of a pointer

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

- All pointers are of same size
- Pointers are defined indirectly

6.7.4 Rule #4: Value of a Pointer

Pointing means Containing, i.e.,
Pointer pointing to a variable ≡ Pointer contains the address of the variable
Notes:

6.7.5 Rule #5: NULL pointer

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

Segmentation Fault

```
int main()
{
    int a[5], i;
    printf("Enter even numbers\n");
    for(i = 100; i <= 1000; i++)
        scanf("%d", &a[i]);
    printf("Entered even nos. are\n");
    for(i = 1; i <= 1000; i++)
        printf("%d", a[i]);
    return 0;
}
```

Notes:

Bus Error

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

Notes:

6.7.6 Array Interpretations

Two Interpretations:
⋆ Original Variable
⋆ Constant Pointer (Compiler only)

Rule: When (to interpret array variable as) what?
First variable, then pointer, then warning, then error
Notes:

6.7.7 Rule #6: Arithmetic Operations with Pointers & Arrays

value(p + i) ≡ value(p) + value(i) * sizeof(*p)
Notes:

Array → Collection of variables vs Constant pointer variable

short sa[10];
&sa → Address of the array variable
sa[0] → First element
&sa[0] → Address of the first array element
sa → Constant pointer variable
Notes:

Arrays vs Pointers

◇ Commutative use
◇ (a + i) ≡ i + a ≡ &a[i] ≡ &i[a]
◇ *(a + i) ≡ *(i + a) ≡ a[i] ≡ i[a]
◇ constant vs variable
Notes:

6.7.8 Rule #7: Static & Dynamic Allocation

- Static Allocation ≡ Named Allocation - Compiler's responsibility to manage it - Done internally by compiler, when variables are defined
- Dynamic Allocation ≡ Unnamed Allocation - User's responsibility to manage it - Done using malloc & free

Notes:

Named & Unnamed allocations

Analogy & Notes:
♢ Named and Unnamed allocation
♢ Analogy with houses with and without names
♢ Removing all pointers from unnamed location

Differences at program segment level

♢ Defining variables (data & stack segmant) vs Getting & giving it from the heap segmant using malloc & free
♢ int x, int *xp, *ip;
xp = &x;
ip = (int*)(malloc(sizeof(int)));
Notes:

Dynamic Memory Allocation

In C functions for dynamic memory allocation functions are declared in the header file <stdlib.h>. In some implementations, it might also be provided in <alloc.h> or <malloc.h>.

malloc

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

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

calloc

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

The calloc function returns the memory (all initialized to zero) so may be handy to you if you want to make sure that the memory is properly initialized. calloc can be considered as to be internally implemented using malloc (for allocating the memory dynamically) and later initialize the memory block (with the function, say, memset()) to initialize it to zero.

realloc

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

/* allocate similar to malloc */
ptr = realloc(ptr,100);
/* extend */
ptr = realloc(ptr, 250);
/* shrink */
ptr = realloc(ptr,100);
/* release  similar to free */
ptr = realloc(ptr,0);
```

The function realloc has the following capabilities

1. to allocate some memory (if p is null, and size is non-zero, then it is same as malloc(size)),
2. to extend the size of an existing dynamically allocated block (if size is bigger than the existing size of the block pointed by p),
3. to shrink the size of an existing dynamically allocated block (if size is smaller than the existing size of the block pointed by p),
4. to release memory (if size is 0 and p is not NULL then it acts like free(p)).

free

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

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

1. if free() is called on a null pointer, nothing happens.
2. if free() is called on pointer pointing to block other than the one allocated by dynamic allocation, it will lead to undefined behavior.
3. if free() is called with invalid argument that may collapse the memory management mechanism.
4. if free() is not called on the dynamically allocated memory block after its use, it will lead to memory leaks.

Differences between a pointer and an array

- Variable pointer vs Constant pointer
- sizeof a pointer and an array
- Initialization to point to correct location vs Correctly pointing
- char *str = “str”; vs char str[] = “str”;

Notes:

6.8 Static vs Dynamical Allocation of 2-D arrays

Multi-dimensional arrays can be implemented in two ways: rectangular and ragged. C supports both of them (and you can mix them also). When we do static allocation, we follow rectangular array. When we pass command line arguments we are using ragged array and we can implement ragged arrays with pointers. Let us consider an example of two-dimensional array:

Note that ragged arrays need extra memory locations for storing the pointers.

- Both dimensions static (Rectangular)

```
/* rectanguar array*/
int rec [10][10];
/* takes totally 10 * 10 * sizeof(int) bytes */
```

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

```
int *ra[10]; /* ragged and rectangular array  */
int i;
for(int i = 0; i < 10; i++)
    ra[i] = (int*) malloc( 10 * sizeof(int));
/* total memory used
10 * sizeof(int *) + 10 * 10 * sizeof(int) bytes */
```

Notes:

Notes:

- Both dimensions dynamic (Ragged)

```
/* ragged array */
int **arr;
/* takes 2 bytes for arr */
int i;
arr = (int **) malloc (10 * sizeof(int*));
/* takes 10 * sizeof(int*) for first level of indirection */
for(i = 0; i < 10; i++)
    arr[i] = (int*) malloc( 10 * sizeof(int));
/* total memory used */
/* 1 * sizeof(int **) + 10 * sizeof(int *)  */
/* + 10 * 10 * sizeof(int) bytes */
```

Notes:

Notes: So ragged arrays usually take more memory than static arrays. Although it seems that it takes more memory it is not the case always. In a multidimensional array, if all the elements are of same size, and we know them at compile time then rectangular arrays are advantageous. But this is an ideal case. In real world programming, in many cases we don't know it. Consider a requirement of storing an array of strings of variable lengths. Here the memory can be efficiently utilized with ragged arrays and is advantageous than rectangular arrays.

Notes:

- Defining different ways
- Pointer interpretable
- When to use what

6.8.1 Various equivalences in 2-D arrays

- a[i][j] ≡ *(a[i] + j) ≡ *(*(a + i) + j) ≡ (*(a + i))[j] ≡ j[a[i]] ≡ j[i[a]] ≡ j[*(a + i)]

Notes:

6.8.2 2-D arrays using a single level pointer

Notes:

6.9 Function Pointers

6.9.1 Why Function Pointers?

- "Call back functions" provide the solution to following requirements:
- Chunk of code that can be called independently and is standalone
- Independent code that can be used to iterate over a collection of objects
- Event management which is essentially asynchronous where there may be several objects that may be interested in "Listening" such an event
- "Registering" a piece of code and calling it later when required.

6.9.2 Function Name - The Second Interpretation

C provides function pointers that are pretty low-level, efficient and direct way of providing support to callback functions.
Notes:
Only Rule #1 to Rule #4 are applicable for function pointers

6.9.3 Theory & Examples

A pointer stores an address. The address is not limited to variables and objects - it can be of functions also. The code for the function is available in the memory. The starting address is stored in a function pointer. In other words, a function pointer is best thought of as an address, usually in a code/text segment, where that function's executable code is stored. Consider one such example, the bsearch function in the standard header file ¡stdlib.h¿:

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

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

Another example in C standard library for exploiting the use of function pointers is atexit, whose prototype is given as:

```
int atexit(int (*)(void));
```

You can register the functions to be called when the program exits. Consider:

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

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

Output choices:
+ Inside main
+ About to quit
+ Exiting the program

On the normal termination of the program, (even if the program is terminated by calling exit function) the functions registered with atexit will be called.

Thus, function pointers serve as a good example to show the low-level nature of C. The problem with this low-level nature is that it is not secure and type-safe. It is easy to violate

the basic rule that the return type and arguments must be identical for the function pointers and the function assigned:

```
void  (*functionPtr)(int, float );
void foo(int i, float f)
{
    return 0;
}
functionPtr = foo;
/* No problem, the arguments and return type matches */
extern bar();
functionPtr = bar;
/* There is no way by which the compiler can verify that the */
/* arguments and return types matches correctly */
```

If the arguments and return type does not match, that may lead to undefined behavior. No doubt, function pointer is a very powerful feature in C but is prone to misuse and is sometimes unsafe because the programmers can easily make mistakes, as it is not type-safe.
Notes:

6.10 Practice - 1

Write a function read_int to read an integer

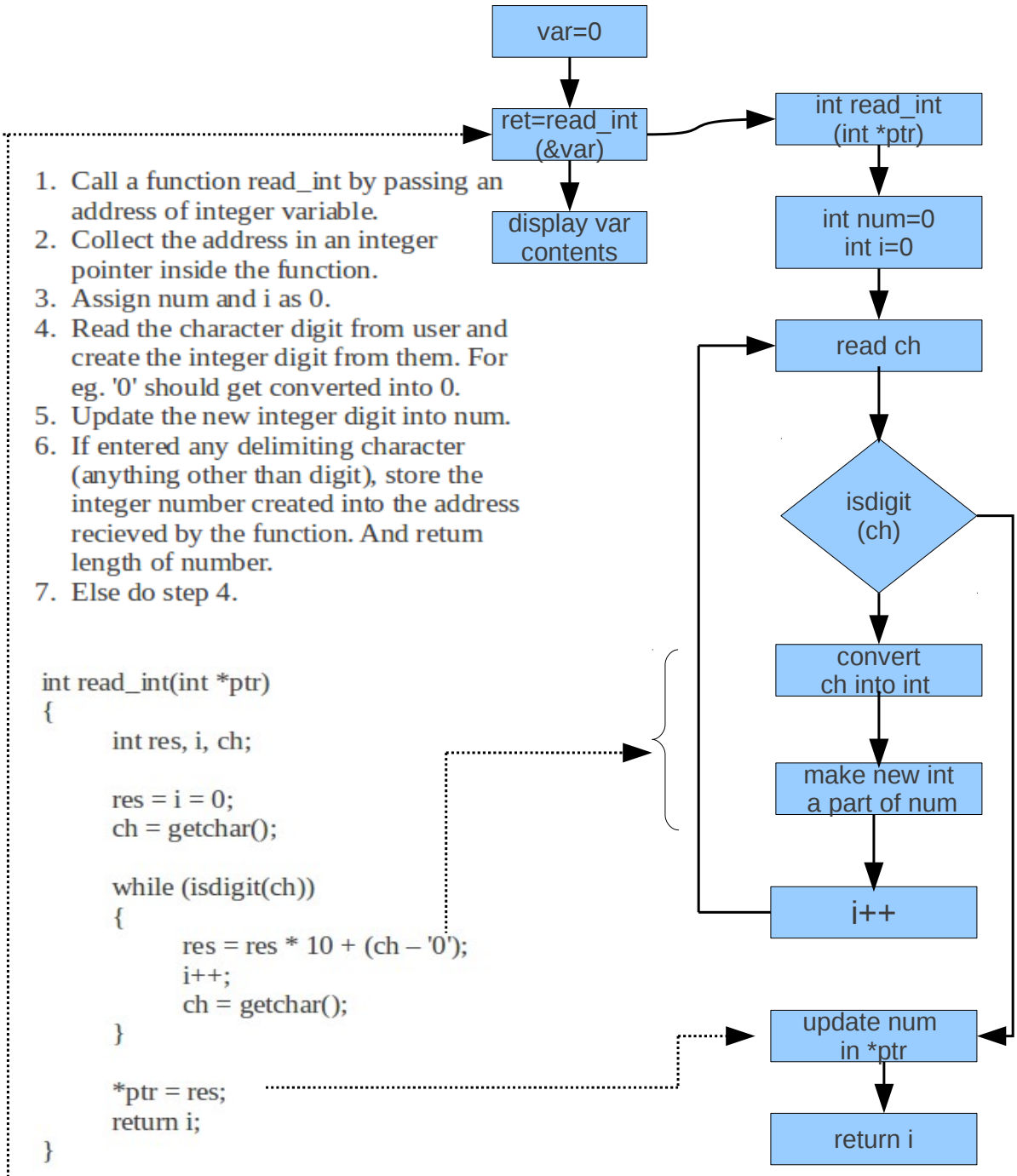
6.10.1 Prerequisite

1. Basic pointer concept
2. getchar usage

6.10.2 Objective

1. Implementing pass by reference method.
2. Converting character digits into integer digits.

6.10.3 Algorithm Design



1. Call a function read_int by passing an address of integer variable.
2. Collect the address in an integer pointer inside the function.
3. Assign num and i as 0.
4. Read the character digit from user and create the integer digit from them. For eg. '0' should get converted into 0.
5. Update the new integer digit into num.
6. If entered any delimiting character (anything other than digit), store the integer number created into the address recieved by the function. And return length of number.
7. Else do step 4.

```
int read_int(int *ptr)
{
        int res, i, ch;

        res = i = 0;
        ch = getchar();

        while (isdigit(ch))
        {
                res = res * 10 + (ch – '0');
                i++;
                ch = getchar();
        }

        *ptr = res;
        return i;
}

int main()
{
        int num, len;

        len = read_int(&num);

        printf("num = %d,  len = %d\n", num, len);
}
```

6.10.4 Dry run

6.10.5 Practical Implementation

1. Can be used for extracting the numbers stored in a string data.

Notes:

6.11 Practice - 2

Write an alternative version of squeeze(s1,s2) that deletes each character in s1 that matches any character in the string s2.

6.11.1 Prerequisite

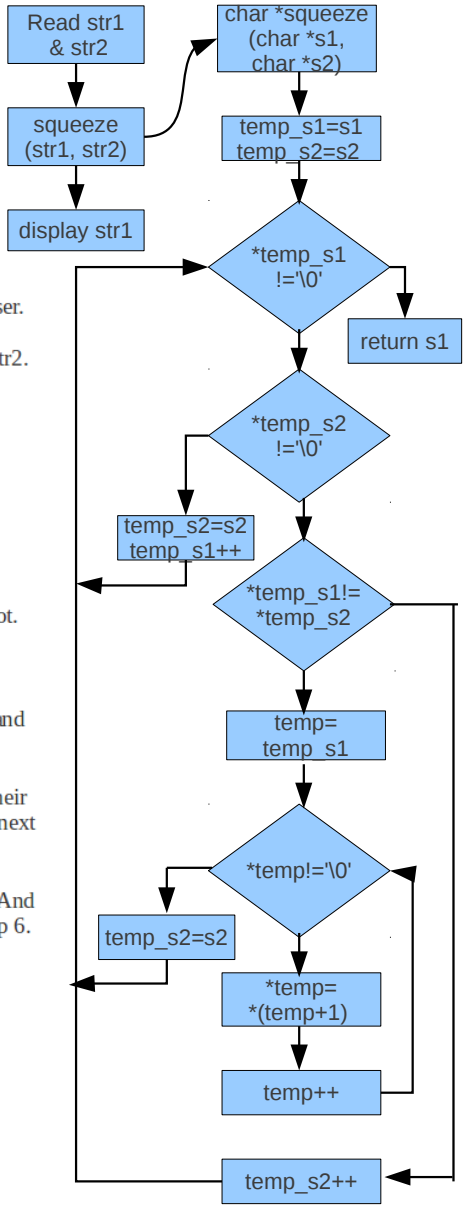
1. Pointer Concepts.

6.11.2 Objective

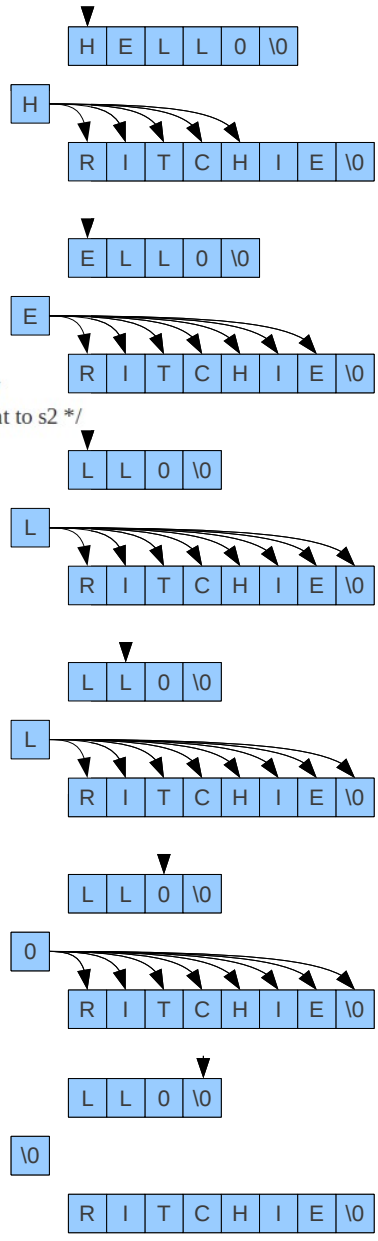
1. Revising float and double bits representation.
2. Understanding pointer type casting concept.

6.11.3 Algorithm Design

1. Recieve two strings in str1 and str2 from user.
2. Call squeeze function by passing str1 and str2.
3. Recieve them as s1 and s2.
4. Assign temp_s1 to s1
5. Assign temp_s2 to s2
6. Check *temp_s1 is nul or not.
7. If not, Check *temp_s2 is equal to nul or not.
8. If not, compare *temp_s1 and *temp_S2.
9. If both are not equal, increment *temp_s2 and repeat step 6.
10. Else run a loop to override *temp_s1 and their subsequent location values till nul by their next location values. Do step 6.
11. (From step 7) Update temp_s2 to point s2. And increment temp_s1 to next location. Do step 6.
12. Return s1 to calling function.
13. Display the changes in s1.



```
char *squeeze(char *s1, char *s2)
{
        char *temp_s1 = s1;
        char *temp_s2 = s2;

        while (*temp_s1 != '\0')
        {
                if (*temp_s2 != '\0')
                {
                        if (*temp_s1 == * temp_s2)
                        {
                                /* Run a loop to update temp_s1 */
                                /* After loop make temp_s2 to point to s2 */
                        }
                        else
                        {
                                /* increment temp_s2 */
                        }
                }
                else
                {
                        /* Assign temp_s2 to point s2 */
                        /* Increment temp_s1 */
                }
        }
        return s1;
}


int main()
{
        char str1[MAX_LEN];
        char str2[MAX_LEN];

        /* Read str1 and str2 from user */
        squeeze(str1, str2);

        /* Display and see the change in str1 */

}
```



6.11.4 Dry run

6.11.5 Practical Implementation

1. Store informations inside floating point numbers.

6.12 Quiz

Short Answer

1. What is the result of

```
int main()
{
    while (1)
    {
        malloc(100);
    }
    return 0;
}
```

2. What will happen if I am not freeing the dynamically allocated memory ?

3. Pointer which holds an address which is out of scope is termed as _____ pointer.

4. What is wrong in this code.

```
int main()
{
    int *ip;
    ip = malloc(20);
    for (i = 0; i < 20; i++)
    {
        scanf("%d", &ip[i]);
    }
    return 0;
}
```

5. How can I use calloc to allocate memory for 10 integers, initialized with zeroes?
   How can I achieve it without using calloc ?

6.
```
int main()
{
    char *ptr;

    get_name(); // modify the function prototype so that inside ptr,
                   I need an address which points to a name.
}
```

7. What is tunable array ? How can I use it ?

```
8. What are the legal and illegal things allowed in pointers ?

9. typedef int int_10[10];
   int_10 arr;

   What is the size of arr ?
   What is the size of arr[0] ?
   What is the size of int_10[0] ?
   What is the size of int_10[2] ?

10. int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    char *pa;

    pa = arr;
    *++pa = 0x01;
    printf("arr[0] = %X , arr[1] = %X\n", arr[0], arr[1]);

11. How can we achieve genericity feature in C language ?
    Illustrate with an example.
```

True or False

```
1. Dangling pointer errors can be solved with the help of NULL pointers.
(T / F)
```

6.13 Lab Work

Complete all the assignments in 7 days

(Id) / Date	Assignment Topic
( )	Write a function read_int to read an integer Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Print bits of float & double. Check IEEE std. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Generate a n*n magic square (n is odd +ve no.) Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( )	Take 8 consecutive bytes in memory. Provide a menu to manipulate or display the value of variable of type t (input) & indexed at i (i/p) Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Implement i) int getword(char *word) ii) int itoa(int n, char *s) iii) atoi(const char *s) Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Variance calculation with and without static/dynamic arrays Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( )	Read n & n floats in a float array 'store'. Print the values in sorted order without modifying or copying 'store' Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Read n for the number of fragments (≤ 32). Read k_i as the number of integer elements (≤ 32) in each fragment. Read all the elements of all the fragments. Sort each fragment and store their average as $(k_i + 1)$th element. Sort the fragments based on their average value and print all. Re-sort the fragments based on their median and print all Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Read n & n person names of (i) maxlen 32, and (ii) avg len a. Sort and print the names Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Implement string functions strstr, strtok, strcmp, memmove Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Read a string and print it in reverse without storing in an array(try recursive and non-recursive methods). Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Write an alternative version of squeeze(s1,s2) that deletes each character in s1 that matches any character in the string s2 Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Check for your Endianess of your processor Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Implement binary search for int, double, string, rational Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Implement calc_mean for all types Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( )	Implement myprintf() Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Implement myscanf() Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( )	Tower of Hanoi recursively Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:

(Id) / Date	Assignment Topic
( ) _____	Take n & k ($1 \leq k \leq 10$) as i/p. Generate consecutive NRPS of len n using k distinct char($0 \leq k \leq 9$) Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Generate 0-n combination of a n-length string recursively and non-recursively Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Matrix - Transverse, Inverse, Determinent Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	
( ) _____	

Chapter 7

Day 7: Preprocessing

7.1 What is preprocessing & When is it done?

Preprocessor is a powerful tool with raw power. Preprocessor is often provided as a separate tool with the C compilers. After preprocessing the preprocessed output is sent to the C compiler for compilation.

The main functionalities of the preprocessor are: file inclusion (#include), conditional compilation (#ifdefs) and textual replacement (#defines). Preprocessor is also responsible for removing the comments from source code (// and /* */), processing the line continuation character (\ character), processing escape sequences (characters such as \ n), processing of the trigraph sequences (such as ′?? <′character for {) etc.

Notes:

7.2 Built-in Defines

- `__FILE__` : Represents the current source file name in which it appears.
- `__LINE__` : Represents the current line number during the preprocessing in the source file.
- `__DATE__` : Represents the current date during the preprocessing.
- `__TIME__` : Represents the current time at that point of preprocessing.
- `__STDC__` : This constant is defined to be true if the compiler conforms to ANSI/ISO C standard.
- `__func__` : Represents the current function name in which it appears.

Example:

```
printf("Error in %s @ %d on %s @ %s \n",
    __FILE__, __LINE__, __DATE__, __TIME__);
```

Notes:

7.3 The preprocessor directives

The C tradition is to have function declarations and type declarations in the header files and the function definitions in the source files. The preprocessor does textual replacement of the header file in the source file with the #include directive (while expanding necessary macros, doing conditional compilation etc as necessary).

7.3.1 #include

Q: What is the difference between <> and " " with respect to #include? A:
Notes:

Header vs Source File

- All Declarations
- All Definitions except Typedefs
- Typedefs
- Defines
- Inline functions

Notes:

7.3.2 #ifdef, #ifndef, #else, #endif

Notes: In many cases, we need to have more than one version of the program and depending on the target platform, we may wish to compile accordingly. Conditional compilation supported by the preprocessor makes this easier. In any non-trivial implementations, it is usual to support debug and release versions and preprocessor comes handy. You can use conditional compilation to support that. An another common usage is to avoid multiple inclusion of headers.

7.3.3 #define, #undef

Using #define directive is not limited to the conditional compilation alone - it is used to provide symbolic constants as well as the function like macro expansion. The directives for conditional compilation are: #if, #else, #elif, #endif, #ifdef, #ifndef, #endif.
Q: Whats the difference between a macro and a define
A:

#define Constants

Prior to ANSI C where const was not available, programmers just used the # define directive to have the functionality of the constants. The #define is not type safe and can be redefined again, defeating the whole purpose. Use consts and enums (for set of related values) instead of #defines:

```
int main()
{
    #define MYCONSTANT 100
    enum { enum_constant = 100; };
    const int int_constant = 100;
    printf("%d %d %d", MYCONSTANT enum_constant, int_constant);
}
```

Note that, with #undef directive, you an undefined a previously defined preprocessor constant or macro. OK, one more example:
Will the following tiny code work!!!

```
#include <stdlib.h>
int main()
{
    char *msg = "Hello World";
    printf("%s", msg);
    #include <stdio.h>
}
#include <stdio.h>
```

#define Macros

The # define directive can be used for function like macro definitions. What the main differences between the macros and functions? Consider the following macro definition:

```
#define max_macro(a, b) (a > b) ? a : b

inline int max_function (int a, int b)
{
    return ((a > b) ? a : b);
}
```

Notes:

7.3.4 #if, #else, #elif, #endif

```
#include <stdio.h>

#if (DEBUG == 1)
#define ERR_PRINT(args...) printf("%d:%s", __LINE__, \
    __FILE__); printf(args)
#define WARN_PRINT(args...)
#define DBG_PRINT(args...)
#elif(DEBUG == 2)
#define ERR_PRINT(args...) printf("%d:%s", __LINE__, \
    __FILE__); printf(args)
#define WARN_PRINT(args...) printf("%d:%s", __LINE__, \
    __FILE__); printf(args)
#define DBG_PRINT(args...)
#elif(DEBUG == 3)
#define ERR_PRINT(args...) printf("%d:\%s", __LINE__, \
    __FILE__); printf(args)
#define WARN_PRINT(args...) printf("%d:\%s", __LINE__, \
    __FILE__); printf(args)
#define DBG_PRINT(args...) printf("%d:\%s", __LINE__, \
    __FILE__); printf(args)
#else
#define ERR_PRINT(args...)
#define WARN_PRINT(args...)
#define DBG_PRINT(args...)
#endif

main()
{
    ERR_PRINT("This is error %d\n", 3);
    DBG_PRINT("This is info\n");
    WARN_PRINT("This is warning %d\n", 1);
    return 0;
}
```

Notes:

```
# if defined (DEBUG)
/* do the checking */
# endif
```

7.3.5 #error, #line

C provides other constructs also like #error to pass the error messages to the compiler and #line to change the line number to be kept track while issuing such error messages.
Notes:

7.3.6 #pragma

The directive #pragma indicates that a particular feature is implementation dependent. If a compiler encounters an unknown option in # pragma, it is free to ignore it.
Notes:
These are compiler/assembler/linker dependent flags. Will be discussed in detail during embedded modules' discussion

7.3.7 #,

The preprocessor supports two operators useful for macro expansion: stringization("#") and concatenation ("##") operations.

```
#define STRINGIZE(string) #string
#define CONCATENATE(string1,string1) string1##string2
/* sample use: */
CONCATENATE(dou, ble) d;
/* this is same as declaring d as:*/
/* double d; */
printf("The int keyword is %s", STRINGIZE(int));
/* prints: the int keyword is int */
```

One more:

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

int main()
{
    int CAT(x, 0);

    CAT(x, 0) = 5;
    printf(STRFY(CAT(Hello, World)));
    print(CAT(x, 0));
}
```

Notes:

7.4 Macro know-hows

1. Macros are not type-checked
   Example:

```
int k = max_macro(i, j);
/* works with int */
float max_float = max_macro(10.0, 20.0);
/* also works with float constants */
int k = max_function(i, j);
/* works with int */
float max_float = max_function (10.0, 20.0);
/* does not work - you can pass only integral values */
```

2. Macros have side effects during textual replacement whereas functions does not have that since textual replacement is not done but a function call is made
   Example:

```
int k = max_macro (i++, j);
/* we are in trouble as i++ is evaluated twice */
/* int k = (i++ > j) ? i++ : j */
int k = max_function (i++, j);
/* no problem, as it is not expanded, but a call to max_function is made */
```

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

4. The function evaluates its arguments. A macro does textual replacement of its arguments, so it does not evaluate its arguments

5. A function gets generated as a code and hence has an address (so you can use it for storing it in a function pointer). A preprocessor macro gets replaced with text replacement, so a macro becomes part of code in which it is used (it does not have an address by itself like a function, so you cannot use it for storing it in a function pointer).

```
typedef int (*fp)(int, int);
fp = max_function;
int k = fp(10, 20); /* ok, calls max_function */
fp = max_macro; /* error; unknown identifier max_macro */
int k = fp(10, 20);
```

7.5 Practice - 1

Define a macro SIZEOF(x), where x is a variable, without using sizof operator.

7.5.1 Prerequisite

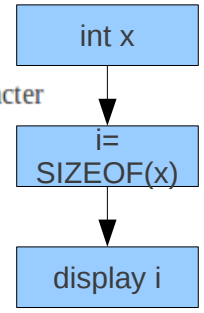
1. Knowledge in Macros
2. Pointer concepts

7.5.2 Objective

1. Understanding usage of macros with arguments.
2. Pointer type casting.

7.5.3 Algorithm Design

1. Declare a variable of any type.
2. Pass the variable into a macro named SIZEOF.
3. In SIZEOF, typecast the variables address and variables next address into character addresses and take their differences.
4. The difference will tell the count of number of bytes between the locations.
5. The count is nothing but the sizeof the variable.
6. Display the count.




```
#define SIZEOF(var)          (char *)(var + 1) – (char *)var

int main()
{
        int x, size;

        size = SIZEOF(x);
        printf("%d\n", x);
}
```

7.5.4 Dry run

7.5.5 Practical Implementation

1. While counting the distance between two memory locations.

7.6 Quiz

Short Answers

1. Observe the result of gcc -E option for the following code.

```
int main()
{
    int i = EOF;
    char *ptr = NULL;
}
```

7.7 Lab Work

Try all the preprocessor templates and complete the assignments in a day

(Id) / Date	Assignment Topic
( ) _____	Define a macro SIZEOF(x), where x is a variable, without using sizeof operator. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Define a macro swap(t,x,y) that interchanges two arguments of type t. Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	
( ) _____	
( ) _____	

Chapter 8

Day 8: User Defined Types

8.1 Why structures & Why unions?

8.1.1 User-Defined Types

Arrays, enums, structs, unions are the building blocks for the users to build their own type, as they need them. They can be built using primitive or aggregate types. Arrays have been already covered. In this chapter enums, structs and unions are covered
Notes:

8.2 Various ways of defining a user-defined type

Combination of struct, union and arrays with typedef

```
typedef int AgeType;
typedef float SalaryType;

struct _Tag
{
    char name[100];
    AgeType age;
    SalaryType salary;
};

typedef struct _Tag
{
    char name[100];
    AgeType age;
    SalaryType salary;
} Tag;

typedef struct
{
    char name[100];
    AgeType age;
    SalaryType salary;
} Tag;

typedef Tag DB[100];
```

Notes:

8.3 Unions

C unions are of invaluable use in system programming, network programming and other related areas. For example, screen coordinates in video memory can be addressed in DOS based machines as pair of bytes in text mode. The first byte is used to represent the attribute describing the character followed by the actual character itself. A programmer may wish to use the pair of bytes as one unit or access as individual elements:

```
/* This is an implementation dependent code */
union VideoMemEntry
{
    struct
    {
        unsigned char attr;
        unsigned char value;
    }entry;
    short attrValue;    /* assume sizeof(short)==2 bytes */
}screen[25][80];        /* in a 25 * 80 text mode monitior */

screen[0][0].entry.attr = BOLD;
/* set the attribute of the character to BOLD */
screen[0][0].entry.value = 65;
/* the character to be displayed is ASCII character 'a' */
/* access them as individual bytes */
 screen[0][0].attrValue = (BOLD<<8) + 65;
/* or set them together */
```

But unions in these languages suffer many problems. For example, it is easy to violate type-safety as it is easy to access wrong union members mistakenly and you have to keep track of the currently used member explicitly.
Notes:

8.4 Size of

Guess the size of the following structures!!!

```
struct example1
{
    char a[2];
    int x;
    char b[2];
};
sizeof(struct example1) = 12 (Assuming sizeof(int) == 4)

struct example2
{
    char a[2];
    char b[2];
    int x;
};
sizeof(struct example2) = 8 (Assuming sizeof(int) == 4)
```

8.4.1 Why Padding?

- Ease of operation for compilers - Word aligned vs No padding
- Compiler dependent

Notes:

8.5 Initializing Structures

Notes:

8.6 Zero sized array

```
typedef struct
{
    int n;
    double val[0];
} Elements;
Elements *elements;
int i, n;
read(n);
elements = (Elements *)(malloc(sizeof(Elements) + n * sizeof(double)));
for(i = 0; i < n; i++)
    read(elements->val[i]);
```

Notes:

8.7 Enumeration

In C, an enumeration specifies a set of named integral values and its members can be explicitly initialized:

```
enum color {black = 0, white = 1};

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

There is no constraint about the values, they can be in any order and a same value can be repeated several times. Enumerations become part of the enclosing namespace and do not have a namespace of its own. Thus, it pollutes the surrounding namespace with an annoying problem that the enumerators must be distinct. For example:

```
enum color {black, white, orange, red, blue};
enum fruits {apple, orange, banana};
/* error: orange redefined */
int black;
/* error: black redefined */
```

Sometimes it is useful to have unnamed enums:

```
enum {abort, retry, fail} response;
```

The enums are internally treated as integral values and this gives an added advantage: they can take part in expressions as integers. For example:

```
enum color {red = 1, green = 2, blue = 4};
int yellow = red + green;
/* now yellow = 3 */
int white = red + green + blue;
/* white = 7 */
```

Enumeration is useful in places where we need a closed set of named values (instead of having unrelated constant variables or preprocessor constants).
Notes:

8.8 Bit fields

31:17	16:14	13	12:1	0

Field names: code(31:17), reset(16:14), enable(13), flags(12:1), priority(0)

8.8.1 Bit operations

- Setting a field:
- Resetting a field:
- Extracting the value of a field:
- Putting a value into a field:

Notes:

8.8.2 How with bit fields?

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

Notes:

8.8.3 Why bit fields?

Ease of Usage
Ex: control.enable = 1;
Notes:

8.8.4 Ease vs Efficiency & Portability

Notes:

8.8.5 Size considerations

Case Study I:

- Increase enable to 2 bits
- Increase enable to 3 bits

Case Study II:

- Replace unsigned int to unsigned long
- Replace unsigned int to unsigned short

8.8.6 Bit Padding

Notes: Analogous to bytes and fields

8.9 Practice - 1

8.9.1 Prerequisite

1. Structure usage.

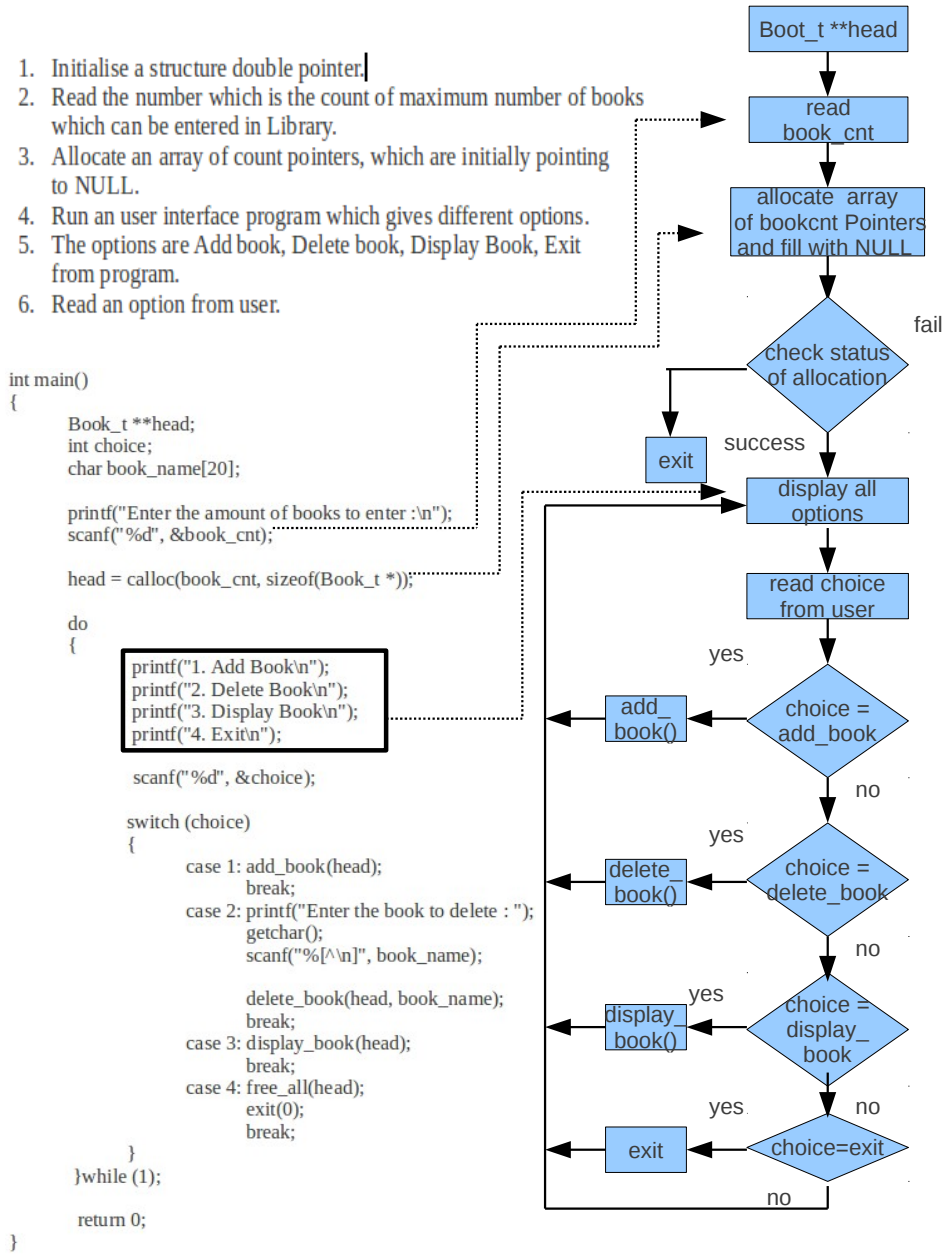
8.9.2 Objective

1. Implementing data base managing concepts.
2. Structure pointer concepts.

8.9.3 Algorithm Design

1. Initialise a structure double pointer.
2. Read the number which is the count of maximum number of books which can be entered in Library.
3. Allocate an array of count pointers, which are initially pointing to NULL.
4. Run an user interface program which gives different options.
5. The options are Add book, Delete book, Display Book, Exit from program.
6. Read an option from user.

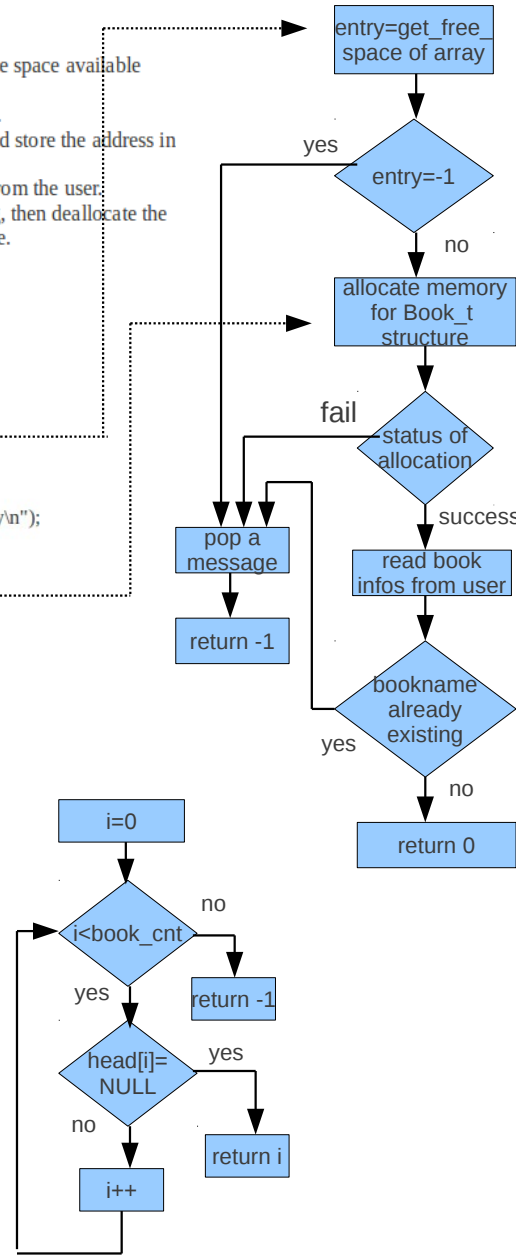
```
int main()
{
        Book_t **head;
        int choice;
        char book_name[20];

        printf("Enter the amount of books to enter :\n");
        scanf("%d", &book_cnt);

        head = calloc(book_cnt, sizeof(Book_t *));

        do
        {
                printf("1. Add Book\n");
                printf("2. Delete Book\n");
                printf("3. Display Book\n");
                printf("4. Exit\n");

                scanf("%d", &choice);

                switch (choice)
                {
                        case 1: add_book(head);
                                break;
                        case 2: printf("Enter the book to delete : ");
                                getchar();
                                scanf("%[^\n]", book_name);

                                delete_book(head, book_name);
                                break;
                        case 3: display_book(head);
                                break;
                        case 4: free_all(head);
                                exit(0);
                                break;
                }
        }while (1);

        return 0;
}
```



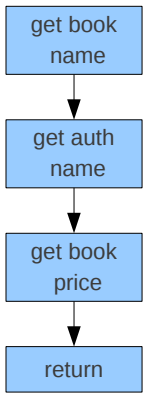
7. If user inputs Add book option, check the free space available in array.
8. If no free space pop a message and do step 4.
9. Else allocate memory for Book structure. And store the address in the free space in array.
10. Fill in the fields of Book structure, reading from the user.
11. If user entered book name is already existing, then deallocate the allocated memory for booklet. Pop a message.
12. Continue step 4.

```
int add_book(Book_t **head)
{
        Book_t *temp;
        int entry;

        entry = get_free_space(head);
        if (entry == -1)
        {
                printf("No more space in Library\n");
                return entry;
        }

        temp = calloc(1, sizeof(Book_t));
        if (temp == NULL)
        {
                printf("Allocation fail\n");
                return -1;
        }

        head[entry] = temp;

        read_book(head[entry]);
        return 0;
}
```

```
int get_free_space(Book_t **head)
{
        int i;
        for (i = 0; i < book_cnt; i++)
        {
                if (!head[i])
                return i;
        }
        return -1;
}
```

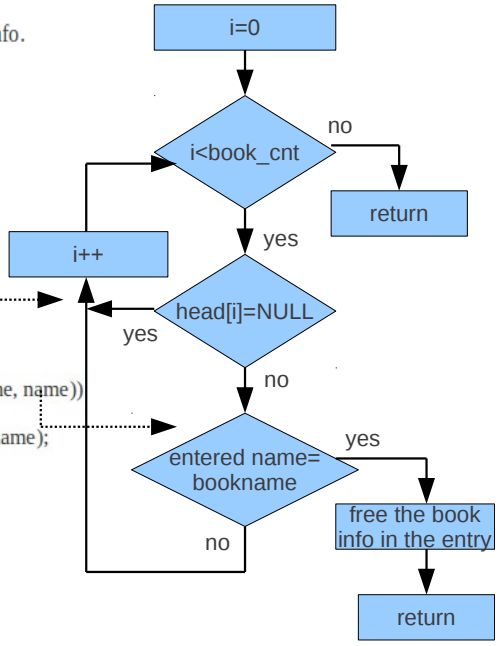


```
void read_book(Book_t *loc)
{
        printf("Enter Book name : ");
        getchar();
        scanf("%[^\n]", loc -> book_name);
        printf("Enter Author name : ");
        getchar();
        scanf("%[^\n]", loc -> auth_name);
        printf("Enter the price : ");
        scanf("%f", &loc -> price);
}
```



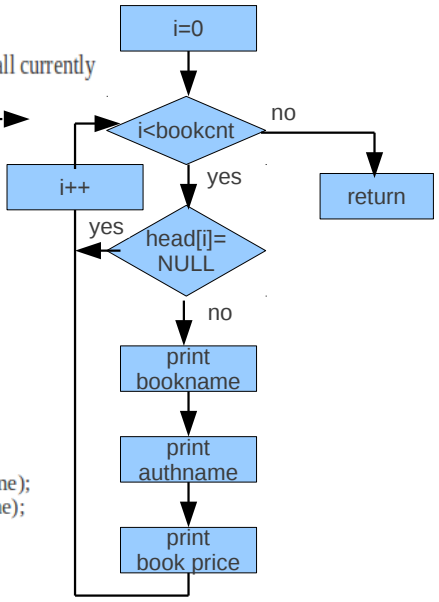
13. (step 6) If user inputs Delete book, read the book name from user.
14. Search the book among the structures.
15. If no book by the name, then pop a message.
16. Else find the book and Deallocate the Book info.
17. Continue step 4.

```
void delete_book(Book_t **head, char *name)
{
        int  i;

        for (i = 0; i < book_cnt; i++)
        {
                if (head[i])
                {
                        if (!strcmp(head[i]->book_name, name))
                        {
                                printf("freeing %s\n", name);
                                free(head[i]);
                                head[i] = NULL;
                                break;
                        }
                }
        }
}
```



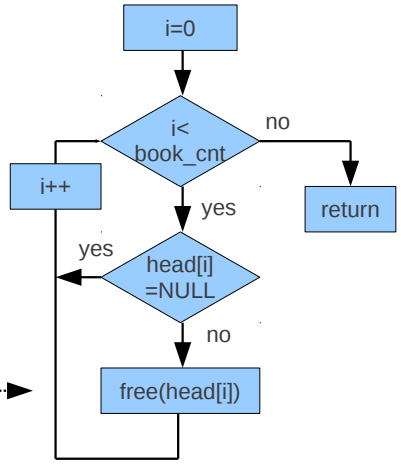
18. (step 4) If user inputs Display Book, informations for all currently entered books are displayed.
19. Continue step 4.

```
void display_book(Book_t **head)
{
        int i = 0;

        for (i = 0; i < book_cnt; i++)
        {
                if (head[i])
                {
                        printf("%s\n", head[i] -> book_name);
                        printf("%s\n", head[i] -> auth_name);
                        printf("%f\n", head[i] -> price);
                }
        }
}
```



20. If user enters Exit, deallocate all the allocated book space. And exit from the program.

```
void free_all(Book_t **head)
{
        int i = 0;
        for (i = 0; i < book_cnt; i++)
        {
                if (head[i])
                {
                        free(head[i]);
                }
        }
}
```



8.9.4 Dry run

8.9.5 Practical Implementation

1. Data base management

8.10 Quiz

Short Answer

1. What is bit padding ? How can I achieve CPU speed optmisation by bit padding ?
2. How can I avoid bit padding ?
3. What are the limitations of enum while comparing with Macros & const variables ?

8.11 Lab Work

Run all the related templates and understand them. Implement the mini project assigned to you

(Id) / Date	Assignment Topic
( ) _____	Create a family tree with struct containing name, age, sex, marital status, child count, and zero-sized array of void pointers. In case, the member is married, he/she will have at least 1 element in the array with the first void pointer pointing to its spouse's structure, and the remaining to its children's structures. Provide function for the following operations for a given person name: 1) Display his/her info; 2) Display the complete family tree starting from him/her; 3) Marry with another given person (with validity checks); 4) Display all the children for requested age group & sex; 5) Add a newly born child; 6) Preserve the tree Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	Read a float. Print the demoted integer value without assigning to int Prerequisites: Algorithm: Dry Run: Objective: Practical Implementation:
( ) _____	
( ) _____	

Chapter 9

Day 9: Interview Preparation

9.1 Complicated Nested Definitions

- char **argv
  Notes:

- char (*daytab)[13]
  Notes:

- char *daytab[13]
  Notes:

- void *comp()
  Notes:

- void (*comp)()
  Notes:

- double (comp())[10]
  Notes:

- char (*(*x())[])()
  Notes:

- char (*(*x[3])())[5]
  Notes:

Note: Innermost operation defines the final type of the variable

Appendix A

Assignment Guidelines

The following highlights common deficiencies which lead to loss of marks in Programming assignments. Review this sheet before turningin each Assignementt to make sure that the it is complete in all respects.

A.1 Quality of the Source Code

A.1.1 Variable Names

- Use variable names with a clear meaning in the context of the program whenever possible.

A.1.2 Indentation and Format

- Include adequate white-space in the program to improve readability. Insert blank lines to group sections of code. Use indentation to improve readability of control flow. Avoid confusing use of opening/closing braces.

A.1.3 Internal Comments

- Main program comments should describe overall purpose of the program. You should have a comment at the beginning of each source file describing what that file contains/does. Function comments should describe their purpose and other pertinent information, if any.

- Compound statements (control flow) should be commented. Finally, see that commenting is not overdone and redundant.

A.1.4 Modularity in Design

- Avoid accomplishing too many tasks in one function; use a separate module (Split your code into multiple logical functions). Also, avoid too many lines of code in a single module; create more modules. Design should facilitate individual module testing.

Use automatic/local variables instead of external variables whenever possible. Use separate header files and implementation files for unrelated functions.

A.2 Program Performance

A.2.1 Correctness of Output

- Ensure that all outputs are correct. Incorrect outputs can lead to substantial loss in grade

A.2.2 Ease of Use

- The program should facilitate repeated use when used interactively and should allow easy exit. Requests for interactive input from the user should be clear. Incorrect user inputs should be captured and explained. Outputs should be well-formatted.

Appendix B

Grading of Programming Assignments

- Total points per assignment = 10
- Points for timely/early submission = 1
- The source code is out of 3 points. The distribution of points is as follows:
  - (a) The existence of the code itself (1 pts)
  - (b) Proper indentation of the code and comments (1 pts)
  - (c) Proper naming of the functions, variables + Modularity + (1 pts)
- You get 4 points if the program does exactly what it is supposed to do.
- Two (2) points are reserved for the ease of use, the type of user interface, the ability to handle various user input errors, or any extra features that your program might have.