

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

Document History

| Date | Course Version No. | Software Version No. | Developer / SME | Change Record Remarks |
|----------------|--------------------|----------------------|-------------------------|--|
| 31-August-2009 | 1.0 | NA | Developer | Changed into PPT with Instructor Notes |
| 10-May-2011 | | NA | Rathnajothi Perumalsamy | Revamp/Refinements |
| 10-May-2015 | 2.0 | | Vinod Pragadeesh M | Revamp/Refinements |

Course Goals and Non Goals

➤ Course Goals

- At the end of this program, participants gain an understanding of the C-Programming Language



➤ Course Non Goals

- Data structure is not included in this course

Pre-requisites

- List the Course Pre-requisites
 - None

Intended Audience

➤ Programmers



© Capgemini Engineering, Inc.
All rights reserved.

January 18, 2016 | Proprietary and Confidential | - 3 -

 Capgemini
+1 855 777 4447, +91 80 66491000

Day Wise Schedule

- **Day 1**
 - Lesson 1: C additional concepts
 - Lesson 2: Operators and Type Conversion
 - Lesson 3: Functions
- **Day 2**
 - Lesson 4: Arrays
 - Lesson 5: Pointers
 - Lesson 6: Structures (till structures and functions)
- **Day 3**
 - Lesson 6: Structures (contd)
 - Lesson 7: File Handling
 - Lesson 8: Preprocessor
 - Lesson 9: Algorithms

January 18, 2016 | Proprietary and Confidential | - 8 -



Day Wise Schedule

- Day 4
 - Lesson 1: Sorting And Searching Algorithms
- Day 5
 - Lesson 2: Linked list

January 18, 2016 | Proprietary and Confidential | > 2 *

 Capgemini
ESCALATION: 77774433447, 88888888100

Table of Contents

- **Lesson 1: Introduction to C**
 - 1.1. Escape Sequences and Format Specifiers
 - 1.2. Structure of C program
 - 1.3. First C program
- **Lesson 2: Operators and Type Conversion**
 - 2.1. Precedence and Order of Evaluation
 - 2.2. Type Conversion

Table of Contents

➤ Lesson 3: Functions

- 3.1. Storage Classes
- 3.2. Recursion
- 3.3. Variable Length Arguments

➤ Lesson 4: Arrays

- 4.1. Multi Dimension Array
- 4.2. Strings

➤ Lesson 5: Pointers

- 5.1. Addressing and Dereferencing Operators
- 5.2. Pointers to Functions
- 5.3. Pointers to Arrays
- 5.4. Pointers to Pointers
- 5.5. Dynamic Memory Allocation

Table of Contents

➤ Lesson 6: Structures

- 6.1. Structures and Arrays
- 6.2. Structures and Pointers
- 6.3. Structures and Functions
- 6.4. Unions
- 6.5. Difference Between Union and Structure
- 6.6. Typedef Statement
- 6.7. Enumerated Data Type

Table of Contents

- **Lesson 7: File Handling**
 - 7.1. Text Files and Data Files
 - 7.2. Random Access and Error Handling
- **Lesson 8: Preprocessor**
 - 8.1. Introduction to Preprocessor
 - 8.2. Macro Substitution
- **Lesson 9: Algorithms**
 - 9.1. Algorithm Analysis
 - 9.2. Comparisons of Searching and Sorting Algorithms
 - 9.3. Time vs Space Complexity

Table of Contents

- **Lesson 1: Sorting and Searching Algorithms**
 - 1.1. Sorting Algorithm
 - 1.2. Searching Algorithm
- **Lesson 2: Linked Lists**
 - 2.1. Introduction to Linked List
 - 2.2. Implementation of Linked List
 - 2.3. Dynamic Linked List: Stacks and Queues
 - 2.4. Introduction to Binary Trees

References

➤ **Student material:**

- Class Book (presentation slides with notes)



➤ **Book:**

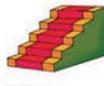
- Pointers in C; by Kanetkar Yashawant
- Test Your C; by Kanetkar Yashawant
- C : The Complete Reference; by Schildt Herbert
- Let Us C; by Kanetkar Yashawant
- Programming in ANSI C; by Balagurusamy

➤ **Web-site:**

- <http://www2.its.strath.ac.uk/courses/c/>

Next Step Courses

➤ Data Structures



January 18, 2016 | Proprietary and Confidential | + M +

 Capgemini
ESCALATION: 7774432474, 8008888195

Other Parallel Technology Areas

- Object-oriented programming language C++

C Programming

Lesson 1: Introduction to C

January 26, 2016 | Proprietary and Confidential | 1 / n



Lesson Objectives

➤ To understand the following topics:

- Evolution of C
- Escape Sequences and Format Specifiers
- Common Best Practices



1.1: Evolution of C

Stages in Evolution

- C supports development of all type of applications using one high-level language

| Language Developed | Drawbacks |
|--------------------|--|
| ALGOL 60 | - Too general and too abstract |
| CPL ↓ | - Hard to learn and difficult to implement |
| BCPL ↓ | - Less powerful and too specific |
| B ↓ | - Too specific |
| C ↓ | - Final Solution |

January 28, 2016 | Proprietary and Confidential | 34

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

1.2: Constants, Variables and Data Types

Character Data Type

- **Machine representation:**
 - character constant 1 word
 - character variable 1 byte
- **Character constants:**
 - single character enclosed in single quotes
- **For example:**

'D', '3', '?', '*'
- **Some special character constants or escape sequences:**

\n \v \r \"\n \b \| \t

January 28, 2016 | Proprietary and Confidential | - 4 -

 Capgemini
CONSULTING TECHNOLOGY SERVICES

1.2: Constants, Variables and Data Types

Character Data Type

➤ Declaration

```
char c;  
char response, answer ... ;  
char vari, var2, .., varN ;
```

➤ Character Input / Output

Functions provided - getchar() - putchar(c)
where C_value is of type char/int
char C_value ; or int C_value ;
C_value = getchar(); or C_value = getchar()
getchar() returns ASCII values from range 0 to 255

January 28, 2016 | Proprietary and Confidential | 5+

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

| 1.2: Constants, Variables and Data Types | | | |
|--|--|--|--|
| Data type | Range in environment | | Usage |
| | 16 bit | 32 bit | |
| char | -128 to 127 | -128 to 127 | A single byte capable of holding one character |
| short int | -2 ¹⁵ to 2 ¹⁵ -1 | -2 ³¹ to 2 ³¹ -1 | An integer, short range |
| int | -2 ¹⁵ to 2 ¹⁵ -1 | -2 ³¹ to 2 ³¹ -1 | An integer |
| long int | -2 ³¹ to 2 ³¹ -1 | -2 ³¹ to 2 ³¹ -1 | An integer, long range |
| float | -3.4e38 to +3.4e38 (4 bytes) | | Single-precision floating point |
| double | -1.7e308 to +1.7e308 (8 bytes) | | Double-precision floating point |
| unsigned int | 0 to 2 ¹⁶ -1 | 0 to 2 ³² -1 | Only positive integers |
| unsigned char | 0 to 255 | 0 to 255 | Only positive byte values |

January 28, 2016 | Proprietary and Confidential | + 8 +



1.3: Escape Sequences and Format Specifiers

Escape Characters

➤ **Escape characters are:**

- Non-graphic characters including white spaces
- Non-printing characters and are represented by escape sequences consisting of a backslash (\) followed by a letter

| Character | Description |
|-----------|-----------------|
| \b | Backspace |
| \n | New line |
| \a | Beep |
| \t | Tab |
| \" | " |
| \ | \ |
| \' | , |
| \r | Carriage return |

January 28, 2016 | Proprietary and Confidential | +2+

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

1.3: Escape Sequences and Format Specifiers

Format Specifiers

➤ **Format Specifiers are:**

- Formatting characters used to accept or display the value in a specific format

| Data Type | ConversionSpecifier |
|--------------------|---------------------|
| signed char | %c |
| unsigned char | %c |
| short signed int | %d |
| short unsigned int | %u |
| long signed int | %ld |
| long unsigned int | %lu |
| float | %f |
| double | %lf |
| long double | %Lf |

January 28, 2016 | Proprietary and Confidential | + 8 +

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

Lab

➤ Lab 1



January 28, 2016 | Proprietary and Confidential | + 9 +

 Capgemini
CONSULTING TECHNOLOGY SERVICES

1.4: Common Best Practices

Indentation-whitespaces

➤ **Whitespace:**

- Use vertical and horizontal whitespace generously
- Indentation and spacing should reflect the block structure of the code
- A long string of conditional operators should be split onto separate lines

January 28, 2016 | Proprietary and Confidential | + 10 +

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

Use vertical and horizontal whitespace generously.
Indentation and spacing should reflect the block structure of the code.
A long string of conditional operators should be split onto separate lines. For example:
`if (foo->next==NULL && number < limit && limit
<=SIZE && node_active(this_input)) {...`

might be better as:

```
if (foo->next == NULL  
&& number < limit && limit <= SIZE  
&& node_active(this_input)) { ...
```

1.4: Common Best Practices
Indentation-whitespaces (contd..)

➤ For example:

```
if (foo->next==NULL && number < limit && limit <=SIZE &&  
node_active(this_input)) {...  
might be better as:
```

```
if (foo->next == NULL  
&& number < limit && limit <= SIZE  
&& node_active(this_input)) { ...
```



1.4: Common Best Practices

Indentation-Loops

➤ **Loops:**

- Elaborate for loops should be split onto different lines:

```
for (curr = *varp, trail = varp;  
     curr != NULL;  
     trail = &(curr->next), curr = curr->next )  
{  
    ...  
}
```

January 28, 2016 | Proprietary and Confidential | + 12 +

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

1.4: Common Best Practices

Indentation-Expressions

➤ Complex expressions, such as those using the ternary ?: operator, are best split on to several lines, too

```
z = (x == y)
? n + f(x)
: f(y) - n;
```

January 28, 2016 | Proprietary and Confidential | +13+

 Capgemini
CONSULTING TECHNOLOGIES DEVELOPMENTS

1.4: Common Best Practices

Indentation-Comments

➤ **Comments:**

- The comments should describe what is happening, how it is being done, what parameters mean, which globals are used and any restrictions or bugs
- Avoid unnecessary comments
- comments are not checked by the compiler, there is no guarantee they are right
- Too many comments clutter code
- Nesting of comments produce unpredictable result
 - `/* / */ 2 /* */ 1` is evaluated as `2*1=2`

January 28, 2016 | Proprietary and Confidential | - 14 -

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

The comments should describe what is happening, how it is being done, what parameters mean, which globals are used and any restrictions or bugs. However, avoid unnecessary comments. If the code is clear, and uses good variable names, it should be able to explain itself well. Since comments are not checked by the compiler, there is no guarantee they are right. Comments that disagree with the code are of negative value. Too many comments clutter code.

Symbolic constants make code easier to read. Numerical constants should generally be avoided; use the `#define` function of the C preprocessor to give constants meaningful names. Defining the value in one place (preferably a header file) also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the define. Consider using the enumeration data type as an improved way to declare variables that take on only a discrete set of values. Using enumerations also lets the compiler warn you of any misuse of an enumerated type. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

1.4: Common Best Practices

Indentation-Comments (Contd..)

- Here is a superfluous comment style:
 - i=i+1; /* Add one to i */
- It's pretty clear that the variable i is being incremented by one. And there are worse ways to do it:

```
*****  
*          *  
*      Add one to i      *  
*          *          *****  
*      i=i+1;          */
```

1.4: Common Best Practices

Constants

➤ **Constants:**

- Avoid using direct constants in execution statements
- Instead, use sizeof, or a #define or an enum for the same
- Symbolic constants make code easier to read. Numerical constants should generally be avoided
- Even simple values like 1 or 0 are often better expressed using defines like TRUE and FALSE

January 28, 2016 | Proprietary and Confidential | 108+

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

1.4: Common Best Practices

Constants

➤ **Bad Example:**

```
main()
{
    char buf[512];           /* hard coding */
    if((n=read(stdin,buf,512)>0)/* difficult to modify later */
    {...}}
```

➤ **Good Example**

```
#define BUFSIZE 512
main()
{
    char buf[BUFSIZE];       /* better */
    if((n=read(stdin,buf, BUFSIZE)>0)/* easy to modify later */
    {...}}
```

January 28, 2016 | Proprietary and Confidential | +12+



1.4: Common Best Practices
Constants (contd..)

➤ Data type of constant will be decided by compiler if its data type is not there in the declaration ,so be careful while using constants in expressions

➤ For example:

— $x = 20.25 + 10$

- 20.25,10 acting as constants in this expression but we haven't declare the data type for the two. so the compiler will decide the data type of two that is by default it will assign double data type to 20.25 and integer to 10

1.4: Common Best Practices
Constants (contd..)

- **const qualified values in initializers cannot be used for array dimensions, as in:**

- `const const int n = 5;`
- `int a[n];`
 - The `const` qualifier really means ‘read-only’; an object so qualified is a run-time object that can no be assigned to. The value of a `const` qualified object is therefore not a constant expression in the full sense of the term and cannot be used for array dimensions, case labels. When you need a true compile time constant use a preprocessor `#define`

I-4: Common Best Practices

Variables

➤ **Variable names:**

- When choosing a variable name, length is not important but clarity of expression is
- A long name can be used for a global variable which is rarely used
- For an array index used on every line of a loop need not be named any more elaborately than i
- Using "index" or "elementnumber" instead is not only more to type but also lead more mistakes

January 28, 2016 | Proprietary and Confidential | + 20 +

 Capgemini
CONSULTING TECHNOLOGIES SERVICES

When choosing a variable name, length is not important but clarity of expression is. A long name can be used for a global variable which is rarely used but an array index used on every line of a loop need not be named any more elaborately than i. Using 'index' or 'elementnumber' instead is not only more to type but also can obscure the details of the computation. With long variable names sometimes it is harder to see what is going on.

1.4: Common Best Practices Variables (Contd..)

➤ Consider:

➤ Bad Example

```
for(i=0;i<=100;i++)  
    array[i]=0;
```

➤ Good Example

```
for(elementnumber=0;  
    elementnumber<=100;elementnumber++)  
    array[elementnumber]=0;
```

January 28, 2016 | Proprietary and Confidential | - 21 -



Summary

- C supports development of all type of applications using one high-level language.
- Variable names are the names (labels) given to the memory location where different constants are stored.
- Expressions can contain constants, variable or function calls.
- Preprocessor directive statements begin with a # symbol.



Review Question

- Question 1: Character occupies _____ bytes.
- Question 2: _____ converts source code to object code.
- Question 3: String constants ends with _____ special character.
- Question 4: Object code is output of _____.

Review Question: Match the Following

| | |
|--------------------------------|--|
| 1. File inclusion directives | Links the object codes to form a single Executable Code. |
| 2. getchar() | Replace tokens in the current file. |
| 3. Macro definition directives | Embed files within the current file. |
| 4. Link Editor | Returns ASCII value range from 0 to 255 |



C Programming

Lesson 2: Operators and Type Conversion

January 16, 2016 | Proprietary and Confidential | + 1 +



Lesson Objectives

➤ To understand the following topics:

- Operators used in C
- Precedence and Order of Evaluation
- Type Conversion



2.1: Operators used in C

Relational Operators

➤ Value of the relational expression is of integer type, and is '1' if the result of comparison is true and '0' if it is false

➤ For example:

| | |
|------------|--|
| 14 > 8 | Has the value 1, as it is true |
| 34 <= 19 | Has the value 0, as it is false |
| X+y == p+q | Has the value 1(true), only if the sum of 'x' and 'y' equals the sum of 'p' and 'q'. |

January 16, 2016 | Proprietary and Confidential | - 3 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Relational Operators (contd.)

Relational operators can be applied to operands of any arithmetic type. The

result of comparison of two expressions is either true or false.

If the condition specified is satisfied, then the expression is true. If the condition specified is not satisfied, then the expression is false.

C has no special data type for logically valued quantities. The value of a relational expression is of type integer, if the result of the comparison is true and 0 if it is false.

2.1: Operators used in C

Logical Operators

➤ **Logical Operators are:**

- Used to combine two or more expressions to form a single expression
- Evaluated left to right, and evaluation stops as soon as the truth or the falsehood of the result is known

| Operator | Name | Meaning |
|----------|-------------|-------------|
| && | Logical AND | Conjunction |
| | Logical OR | Disjunction |
| ! | Logical NOT | Negation |

January 06, 2016 | Proprietary and Confidential | + 6 +

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Types of Operators: Logical Operators:

C provides three logical operators for combining expressions into logical expressions.

Table in the slide shows logical operators available in C.

The logical operators && and || are binary operators, and ! is a unary operator. The value of a logical expression is either '1' or '0', depending upon the logical values of the operands. The operands may be of any arithmetic type. The result is always an integer.

Logical AND Operator(&&):

The && operator combines two expressions into a logical expression and has the following operator formation:

expr1 && expr2

An expression of this form is evaluated by first evaluating the left operand. If its value is 0 (false), then right operand is not evaluated and the resulting value is 0 (false).

If the value of left operand is nonzero (true), the right operand gets evaluated. The resulting value is 1 (true) if the right operand has nonzero value (true) and 0 (false) otherwise.

2.1: Operators used in C

Logical Operators

- Following table shows operation of the logical && operator:

| Expr1 | Expr2 | expr && expr2 |
|----------|----------|---------------|
| 0 | 0 | 0 |
| 0 | non zero | 0 |
| non zero | zero | 0 |
| nonzero | non zero | 1 |

- Following table shows operation of the logical || operator:

| Expr1 | Expr2 | expr1 expr2 |
|----------|----------|----------------|
| non zero | non zero | 1 |
| non zero | zero | 1 |
| zero | non zero | 1 |
| zero | zero | 0 |

January 06, 2016 | Proprietary and Confidential | 5



Logical Operators (contd.)

For example:

```
int var1, var2, var3 ;
If var1 = var2 = var3 = 10 ; then logical expression
var1&&(var2+var3) evaluates 1(true), since both 'var1' and (var2+var3) evaluates
nonzero(true).
```

But in the following example:

var1&&(var2-var3) has value 0(false), since (var2-var3) is zero (false).

Logical OR Operator(||):

The logical OR operator combines two expressions into a logical expression and has following formation:

expr1 || expr2

Consider the following example:

```
int var1, var2 ;
If var1 = var2 = 10 ; then logical expression !var1 evaluates value
0(false), since 'var1' is nonzero and logical expression.
```

And logical expression !(var1-var2) evaluates value 1(true), as var1-var2 is
zero.

2.1: Operators used in C Logical Operators

➤ Following table shows operation of the logical ! operator:

| Expr | !expr |
|---------------|-------|
| nonzero(true) | 0 |
| zero (false) | 1 |

2.1: Operators used in C

Unary Increment/Decrement Operators

➤ **Unary Operators are:**

- `++` and `--`
- Used as postfix or prefix operators
- Operators whose value of expression depends on placement of the operator

Assume `count = 10` and the expression `result=count++;`
Now, 'result' has value 10, since it is postfix operator.
'result' has value 11, if it is `result=++count;`
Though after both expressions the value of `count` will be 11.

January 16, 2016 | Proprietary and Confidential | + 7 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Types of Operators: Unary Operators

Two common forms of assignments found in programs are those of increment and decrement the value of a variable by '1'. C provides two operators, increment (`++`) and decrement (`--`) for these purpose. These are unary operators that is, they require only one operand.

These operators can be used both as prefix, where the operator occurs before the operand and postfix where the operator occurs after the operand in the following manner:

`++ variable`
`variable++`
`-- variable`
`variable--`

In both prefix and postfix form, `++` adds 1 to its operand and `--` subtracts 1 from its operand.

But in prefix form, the value is incremented or decremented by 1 before it is used.

In postfix form, the value is incremented or decremented by 1 after it is used.

For example:

If value of `index` is 1, then `next = ++index`; first increments the value of `index` to 2 and then sets `next` to the current value of `index` making `next` equal to 2.

2.11 Operators used in C

Bitwise Operators

➤ **Bitwise Operators are**

- Applied to operands of type char, short, int and long, whether signed or unsigned
 - & bitwise AND
 - | bitwise inclusive (OR)
 - ^ bitwise exclusive (XOR)

➤ **For example:**

```
if x = 45 and y = 10, z = x | y     gives 47

x = 0 0 1 0 1 1 0 1 = 45
y = 0 0 0 0 1 0 1 0 = 10
z = x | y = 0 0 1 0 1 1 1 1 = 47
```

January 16, 2016 | Proprietary and Confidential | - 8 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Types of Operators: Bitwise Operators:

One of C's powerful features is a set of bit manipulation operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various Bitwise Operators available in C are:

- | Bitwise OR
- & Bitwise AND
- ^ Bitwise XOR

These operators can operate upon ints' and chars' but not on floats and doubles.

2.11 Operators used in C

Ternary/Conditional Operator

➤ **Ternary Operators:**

- Provide an alternate way to write the if conditional construct
- Take three arguments (Ternary operator)

➤ **Syntax:**

expression1 ? expression2 : expression3

➤ If expression1 is true (i.e. Value is non-zero), then the value returned would be expression2 otherwise the value returned would be expression3

```
int x, y;
scanf("%d", &x); /*scanf() accepts the value from the user*/
y = (x>5 ? 3 : 4);
This statement will store 3 in 'y' if 'x' is greater than 5, otherwise it will
store 4 in 'y'.
```

January 05, 2016 | Proprietary and Confidential | - 9 -

 Capgemini
EXECUTIVE TECHNOLOGY DETERMINATION

Types of Operators: Ternary Operators:

Simple condition operations can be carried out using the conditional operator (? :). The conditional operator is a ternary operator that is, it takes three arguments. It has following operator formation:

expression1 ? expression2 : expression3

Where ? and : are the two symbols that denote this operator. A conditional expression is evaluated by first evaluating expression1. If the resulting value is true, then expression2 is evaluated and the value of the expression2 becomes the result of the conditional expression. Otherwise, expression3 is evaluated and its value becomes the result.

This is used to assign one of the two values to a variable depending upon some condition.

For example:

big = x > y ? x : y ;
assigns value of x to big if x is greater than y, else assigns the value of y to big.

2.2: Precedence and Order of Evaluation Precedence and Associativity of Operators

| Operators | Associativity |
|---------------------------|---------------|
| ! ++ -- + - (unary) | right to left |
| * / % | left to right |
| + - (binary) | 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 |

January 05, 2018 | Proprietary and Confidential | - 10 -



Precedence and Associativity of Operators:

If an expression contains more than one operator and parenthesis do not explicitly state the binding of operands, it may appear that an operand may be bound to either of the operators on its two sides.

C uses a precedence and associativity rule to specify the order in which operands are bound to operators.

Precedence:

The order of priority in which the operations are performed in an expression is called Precedence.

Associativity:

Another important consideration is the order in which consecutive operations within the same precedence group are carried out. This is termed as Associativity. It can be either left to right or right to left.

Every operator in C has been assigned a precedence and an associativity. The Precedence and Associativity rule states that the operator precedence and associativity determine the order in which operands are bound to operators. If an expression contains two or more operators of equal precedence, their associativity determines their relative precedence. If the associativity is left to right, then the operator to the left in the expression has the higher precedence; if it is right to left, then the operator to the right has the higher precedence.

2.3: Type Conversion

Type Conversion

➤ When an operator has operands of different types, they are converted to a common type according to a small number of rules shown below:

| Oper1 | Oper2 | Result |
|-------|----------|----------|
| char | char | char |
| char | int | int |
| char | long int | long int |
| char | float | float |
| char | double | double |

| Oper1 | Oper2 | Result |
|-------|----------|----------|
| int | char | int |
| int | int | int |
| int | long int | long int |
| int | float | float |
| int | double | double |

| Oper1 | Oper2 | Result |
|-------|----------|--------|
| float | char | float |
| float | int | float |
| float | long int | float |
| float | float | float |
| float | double | double |

| Oper1 | Oper2 | Result |
|--------|----------|--------|
| double | char | double |
| double | int | double |
| double | long int | double |
| double | float | double |
| double | double | double |

| Oper1 | Oper2 | Result |
|----------|----------|----------|
| long int | char | long int |
| long int | int | long int |
| long int | long int | long int |
| long int | float | float |
| long int | double | double |

January 06, 2016 | Proprietary and Confidential | - 11 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Type Conversion:

An expression may contain variables and constants of different types. These expressions are evaluated using methods called as Type Conversions. Following are the different types Type Conversions.

Automatic Type Conversion:

C performs all arithmetic operations with just six data types : int, unsigned int, long int, float, double and long double. Any operand of the type char or short is implicitly converted to int before the operation. Conversion of char to int is called Automatic Unary Conversions.

For example: If an expression contains an operation between an int and a float, the int would be automatically promoted to a float.

2.3: Type Conversion

Type Casting

- Explicit type conversions can be forced in any expression, with a unary operator called a cast
- In the construction (type-name) expression the expression is converted to the named type by the conversion rules
- The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction

January 06, 2016 | Proprietary and Confidential | + 10 +

 Capgemini
EXECUTIVE TECHNOLOGY DETERMINING

Type Conversion (contd.)

Another kind of type conversion is by forcing a type conversion in a way that is different from the automatic type conversion. This is called as Type Casting.

The general form of a cast is:
(data type) expression

Consider the following example:

If int $x = 7$ and float $y = 8.5$, then $(x + y) \% 4$ cannot be evaluated. Because, $x + y$ will be converted to floating-point by automatic conversion. But $\%$ cannot be used with any operands other than int.

However, using type cast it can be evaluated as: $(int)(x+y) \% 4$, where $(x+y)$ is temporarily converted into an integer to give the remainder 3.

2.3: Type Conversion

Type Casting

```
/* Example :Type casting */
# include <stdio.h>
void main(void)
{
    float balance = 6.35 ;
    printf("Value of balance on typecasting = %d\n", (int) balance);
    printf("Value of balance = %f\n", balance);
}
```

Output :

```
Value of balance on typecasting = 6
Value of balance = 6.350000
```

January 06, 2016 | Proprietary and Confidential | +15+

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

Type Conversion (contd.)

It shows that value of balance variable does not get permanently changed as a result of typecasting. It is the value of the expression that undergoes type conversion whenever the cast appears.

Lab

- Lab 3
- Lab 4



January 06, 2016 | Proprietary and Confidential | - 14 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

The screenshot shows a presentation slide with the following details:

- Title:** Operators
- Section:** 3.4: Operators
- Content:**
 - **What does `a++++b` mean?**
 - The only meaningful way to parse this is:
 - `a++ + ++ b`
 - However, broken down as:
 - `a++ ++ + b`
 - This is syntactically invalid: it is equivalent to:
 - `((a++)++) + b`
- Footers:**
 - January 16, 2016
 - Proprietary and Confidential
 - 95 •
- Logo:** Capgemini Executive Technology Resources

However, the result of `a++` is not an lvalue. Hence it is not acceptable as an operand of `++`. Thus the rules for resolving lexical ambiguity make it impossible to resolve this example in a way that is syntactically meaningful. In practice, of course, the prudent thing to do is to avoid construction like this unless you are absolutely certain what they mean. Of course, adding whitespace helps the compiler to understand the intent of the statement. However, it is preferable (from a code maintenance perspective) to split this construct into more than one line:

`++b;` `(a++) + b;`

2.4: Operators

Operators

- **Avoid to make use of compound assignment operator**
 - count++ /* count value evaluated in one cycle */
 - count = count + 1 /* count value evaluated in two cycle */
- **So from performance point of view use count++ instead of count=count+1**
- **Don't change the value of variable in conditional expression**

January 08, 2013 | Proprietary and Confidential | - 16 -

 Capgemini
EXECUTIVE TECHNOLOGY DETERMINING

3.4: Operators

Operators

| | |
|--|---|
| Instead of doing this | Do this |
| <pre>void main() {int marks[]={78,12,63}; int index=0; if(marks[index++]==2) printf("Marks=%d",marks); else printf("bad"); /*will go to else part*/}</pre> | <pre>void main() { int marks[]={78,12,63}; int index=0,discriminent; discriminent=index+1; If(marks[discriminent]==2) printf("Marks=%d",marks); /*will go to if part*/ else printf("bad");}</pre> |

January 16, 2016 | Proprietary and Confidential | - 17 -

 Capgemini
EXECUTIVE TECHNOLOGY DETERMINING

But at the same time be careful while using it with conditional statements means don't change the value of variable in conditional expression. As each time value of index is going to be incremented in first e.g. whereas second one will not change the value of the variable in conditional expression

2.4: Operators

Operators

➤ **++ and --**

- When the increment or decrement operator is used on a variable in a statement, that variable should not appear more than once in the statement because order of evaluation is compiler-dependent
- Do not write code that assumes an order, or that functions as desired on one machine but does not have a clearly defined behavior: `int index = 0, marks[5];`
- `marks[index] = index++;` /* assign to marks[0]? or marks[1]? */

January 08, 2013 | Proprietary and Confidential | + 18 +

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

2.4: Operators
Operators

➤ Never rely on sizeof() to determine the size of an array passed as an argument

```
finding(char grade[10])
{
    int capacity = sizeof(grade)
    printf("%d\n",capacity);
}
Output:4
```

January 06, 2016 | Proprietary and Confidential | - 10 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

A Function accepts array argument as a pointer (to the first element), regardless of whether it is passed as (char *grade) or (char grade[]). So sizeof(grade) returns the size of a pointer variable which is 4 bytes.

2.5: Type Casting Type Casting

➤ Data truncation & Data widening caused due to type conversion & type casting

➤ Truncation

- Casting floating-point values to integers may produce useless values
- The conversion is made simply by truncating the fractional part of the number
- For example, the floating-point value 3.712 is converted to the integer 3, and the floating-point value -504.2 is converted to -504

2.5: Type Casting

Type Casting

➤ Here are some more examples:

```
void main(void)
{
    float rate =3.789;
    printf("%d %d", (int)rate, (unsigned int)rate, (char)rate);

}
```

Output:3 3 3

January 18, 2018 | Proprietary and Confidential | - 21 -

 Capgemini
EXECUTIVE TECHNOLOGY RESEARCH

2.5: Type Casting

Type Casting

➤ Widening

- Casting an integer to a larger size is fairly straightforward. The value remains the same, but the storage area is widened
- The compiler preserves the sign of the original value by filling the new leftmost bits with ones if the value is negative, or with zeros if the value is positive

January 06, 2016 | Proprietary and Confidential | - 22 -

 Capgemini
EXECUTIVE TECHNOLOGY CONSULTING

2.5: Type Casting Type Casting

- When it converts to an unsigned integer, the value is always positive, so the new bits are always filled with zeros

```
char i = 37  
(short) i => 0037  
(int) i => 00000037
```

```
void main(void)  
{  
    char letter = 37;  
    printf("%d %d", (short)letter, (int)letter);  
}  
Output: 37 37
```

Common Best Practices in C Programming

January 16, 2021

Proprietary and Confidential

• 34 •



2.6: Common Best Practices

Decision Control Statements-If

➤ **Test for true or false:**

- Do not default the test for non-zero, that is:
`if (sample() != FAIL)`
is better than
`if (sample())`
even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0.

January 08, 2013 | Proprietary and Confidential | + 35 +

 Capgemini
EXECUTIVE TECHNOLOGY CONSULTING

The first one is not a particularly good example. It is saying, did sample() succeed, that is, return a non-zero (= logically true) value. However, logic like “did it not fail” instead of “did it succeed” is confusing. I prefer the following:

```
if (sample() == FAIL)
```

Then add a block for failure plus a block for success after the else rather than use if (sample() != FAIL). I do use if (sample()) more than I should! However, what the authors of the webpage are saying is that because C does not have TRUE/FALSE explicitly (yes I know C99 adds it...), all the if () treat 0 as false but non-zero as true. So where sample() returns 0 or non-zero, it is easier to test not = 0 to indicate success rather than to test against a non zero true value. I always use #DEFINE or constants for TRUE/FALSE. Usually TRUE = 1, FALSE = 0.

Decision Control Statements-If

➤ **Do not check a boolean value for equality with 1 (TRUE, YES, etc).**

- Instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

➤ **Bad Example:**

```
if (func() == TRUE) {...
```

➤ **Good Example:**

```
if (func() != FALSE)
```

January 06, 2016 | Proprietary and Confidential | + 26 +



Do not check a boolean value for equality with 1 (TRUE, YES, etc). Instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

if (func() == TRUE) {...
is better written as:
if (func() != FALSE)

Decision Control Statements-If

➤ Good practice to write if(`0==count`) instead of (`count==0`).

- It is a trick to guard against the common error of writing if (`count=0`). If you are in the habit of writing the constant before the `==`, the compiler will complain if you accidentally type if (`0 = count`).

Decision Control Statements-Else

Dangling else:

- Stay away from “dangling else” problem unless you know what you are doing:

```
if (level == 1)
if (level == 2)
    printf("***\n");
else
    printf("##\n");
```

- The rule is that an else attaches to the nearest if. When in doubt, or if there is a potential for ambiguity, add curly braces to illuminate the block structure of the code

The rule is that an else attaches to the nearest if. When in doubt, or if there is a potential for ambiguity, add curly braces to illuminate the block structure of the code.

Decision Control Statements-Switch

Fall-through in switch:

- The break statement causes an immediate exit from the switch. Cases serve just as labels. Hence, after the code for one case is done, execution falls through to the next unless you take explicit action to escape. Break and return are the most common ways to leave a switch.

```
switch (expr){  
    case ABC:  
    case DEF:  
        statement;  
        break;  
    case UVW:
```

Decision Control Statements-Switch

```
statement;      /*FALLTHROUGH*/  
case XYZ:  
    statement;  
    break;  
}
```

- While the last break is technically unnecessary, the consistency of its use prevents a fall-through error if another case is later added after the last one
- Implies that normally each case must end with a break to prevent falling through to the next
- The default case, if used, should always be last and does not require a final break statement if it is last

While the last break is technically unnecessary, the consistency of its use prevents a fall-through error if another case is later added after the last one. The default case, if used, should always be last and does not require a final break statement if it is last.

Control Loops

➤ Null statement:

- The null body of a for or while loop should be alone on a line and commented so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++);  
/* VOID */
```

Control Loops-while

➤ **Do not let yourself believe you see what is not there.**

- Look at the following example:
`while (choice == 't' || choice = ' ' || choice == '\n')
choice = getc(file);`

- The statement in the while clause appears at first glance to be valid C

- The use of the assignment operator, rather than the comparison operator, results in syntactically incorrect code

Control Loops-while (Contd..)

- The precedence of = is lowest of any operator so it would have to be interpreted this way (parentheses added for clarity):

```
while ((choice == '\t' || choice) = (' ' || choice == '\n'))  
choice = getc(file);
```

- The clause on the left side of the assignment operator is:

(choice == '\t' || choice)

which does not result in an lvalue. If c contains the tab character, the result is “true” and no further evaluation is performed, and “true” cannot stand on the left-hand side of an assignment

The clause on the left side of the assignment operator is:

(choice == '\t' || choice)

It does not result in an lvalue. Suppose choice contains the tab character, then the result is “true” and no further evaluation is performed, and “true” cannot stand on the left-hand side of an assignment.

Lab

➤ Lab 2



Summary

- An operator is a symbol, which represents a particular operation that can be performed on some data
- The logical operators && (AND), || (OR) allow two or more expressions to be combined to form a single expression
- The sizeof operator returns the number of bytes the operand occupies in memory
- Explicit type conversions can be forced in any expression, with a unary operator called a cast



Review Question

➤ Complete the following:

1. By default real number is treated as _____.
2. The sizeof operator returns _____.
3. The operator used for type casting is _____.



Review Question: Match the Following

1. !

2. A

3. !=

4. ||

1. Operand

1. Relational Operator

1. Logical Operator

1. Logical Operator



C Programming

Lesson 3: Functions

January 26, 2016 | Proprietary and Confidential | 1 of 1



Lesson Objectives

➤ To understand the following topics:

- Storage Classes
- Recursion



Module Coverage

➤ In this module we will cover:

- Functions
 - Scope Of Variables
 - Storage Classes
 - Nesting of Functions
 - Recursion

4.1: Function Arguments

Call by Reference

- Instead of passing the value of a variable, the memory address (Reference) of the variable is passed to the function using Pointers. It is termed as Call by Reference

January 05, 2016 | Proprietary and Confidential | - 4 -



Function Arguments: Call by Reference:

Instead of passing the value of a variable, we can pass the memory address of the variable to the function. It is termed as Call by Reference. We will discuss call by reference when we come to pointers.

4.1: Function Arguments

Scope of Variables

- The part of the program within which variable/constant can be accessed is called as its scope
- By default, the scope of a variable is local to the function in which it is defined
- Local variables can only be accessed in the function in which they are defined
- A variable defined outside any function is called as External variable
- Scope of an external variable will be the whole program, and hence such variables are referred to as Global variables

January 18, 2016 | Proprietary and Confidential | - 5 -



Function Arguments: Scope of Variables:

The part of the program within which a variable/constant can be accessed is called as its scope.

By default the scope of a variable is local to the function in which it is defined.

Local variables can only be accessed in the function in which they are defined; they are unknown to other functions in the same program.

The default scope of variable can be changed by setting up variables that are available across function boundaries. If a variable is defined outside any function at the same level as function definitions is called as External variable.

Scope of an external variable is the rest of the source file starting from its definition.

Scope of an external variable defined before any function definition, will be the whole program, and hence such variables are sometimes referred to as Global variables.

4.2: Storage Classes

Storage Classes

- **Storage Classes:**
 - Determine the lifetime of the storage associated with the variable.
 - When not defined for a variable, the compiler will assume a storage class depending on the context in which the variable is used
- **A variable's storage class gives the following information:**
 - Where the variable would be stored
 - What will be the default initial value
 - What is the scope of the variable
 - What is the life of the variable that is, how long would the variable exist
- **Following four types of storage classes are provided in C:**
 - Automatic Storage Class
 - Static Storage Class
 - Register Storage Class
 - External Storage Class

January 18, 2016 | Proprietary and Confidential | - 6 -

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Storage Classes:

All variables have a data type, they also have a 'Storage class'.

The storage class determines the lifetime of the storage associated with the variable.

If we don't specify the storage class of a variable in its declaration, the compiler will assume a storage class dependent on the context in which the variable is used.

From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variables' value is stored.

Basically, there are two types of locations in a computer where such a value is kept. They are "Memory" and "CPU Registers".

It is variable's storage class, which determines in which of these two locations the value is stored.

4.2: Storage Classes

Automatic Storage Class

- A variable is said to be automatic, if it is allocated storage upon entry to a segment of code, and the storage is reallocated upon exit from this segment
- Features of a variable with an automatic storage class are:

| | |
|-----------------------|--|
| Storage | Memory |
| Default initial value | Garbage Value |
| Scope | Local to the block, in which it is defined |
| Life | Till the control remains within the block, in which it is defined. |

- General format to declare automatic variable is: **auto data-type variable-name;**
- By default, any variable declared in a function is of the automatic storage class;

January 06, 2016 | Proprietary and Confidential | - 2 -

 Capgemini
CONSULTING TECHNOLOGIES OUTSOURCING

Storage Classes: Automatic Storage Class:

A variable is said to be automatic, if it is allocated storage upon entry to a segment of code, and the storage is reallocated upon exit from this segment.

Features of a variable with an automatic storage class are as follows:

| | |
|-----------------------|--|
| Storage | Memory |
| Default initial value | - Garbage value |
| Scope | - Local to the block, in which it is defined |
| Life | - Till the control remains within the block, in which it is defined. |

Automatic Storage Class (contd...)

- They are initialized to garbage value at run-time

```
int calculate(int rollno)
{
    int mark, result;
}
are equivalent to
int calculate(int rollno)
{
    auto int mark, result;
}
```

January 08, 2006 | Proprietary and Confidential | - 8 -



Automatic Storage Class (contd..):

A variable is specified to be automatic by prefixing its type declaration with the storage class specifier `auto` in the following manner

`auto data-type variable-name;`

By default, any variable declared in a function is of the automatic storage class. They are automatically initialised at run-time.

Thus the declarations of the variables `mark` and `result` in

```
int calculate(int rollno)
{
    int mark, result;
    ----
}
are equivalent to
int calculate(int rollno)
{
    auto int mark, result;
    ----
}
```

and declare `mark` and `result` to be automatic variables of type integer.

An automatic variable may be initialised at the time of its declaration by following its name with an equal sign and an expression.

4.2: Storage Classes

Static Storage Class

- A variable is said to be static, if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates
- Variables declared outside all blocks at the same level as function definitions are always static
- Features of a variable with a static storage class are:

| | |
|-----------------------|--|
| Storage | Memory |
| Default initial value | Zero |
| Scope | Local to the block, in which it is defined |
| Life | Value of the variable persists between different function calls. |

- General format for declaring static variable is:
- `static data-type variable-name;`

January 06, 2016 Proprietary and Confidential - 9 -

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Storage Classes: Static Storage Class:

A variable is said to be static, if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates.

Variables declared outside all blocks at the same level as function definitions are always static.

4.2: Storage Classes Register Storage Class

- In cases when faster computation is required, variables can be placed in the CPU's internal registers, as accessing internal registers take much less time than accessing memory
- Therefore, if a variable is used at many places in a program it is better to declare its storage class as register
- Only character and integer variables are supported
- But, the compiler will decide whether a variable is to be stored as a register variable or automatic

January 18, 2016 | Proprietary and Confidential | > 10 |



Storage Classes: Register Storage Class:

In cases when faster computation is required, variables can be placed in the CPU's internal registers, as accessing internal registers takes much less time than accessing memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as Registers.
Only character and integer variables are supported.

Register Storage Class (contd...)

- Features of a variable with a register storage class are:

| | |
|-----------------------|--|
| Storage | CPU registers |
| Default initial value | Garbage Value |
| Scope | Local to the block, in which it is defined |
| Life | Till the control remains within the block, in which it is defined. |

- General format of declaring register variable is:
`register data-type variable-name;`

January 05, 2016 | Proprietary and Confidential | - 11 -



A variable can be specified to be register by prefixing its type declaration with the storage class specifier register in the following manner:

`register data-type variable-name;`

But, the compiler will decide whether a variable is to be stored as a register variable or automatic.

For Example:

```
main()
{
    register int length=10;
    printf("%d",length);
}
```

Output:

10

4.2: Storage Classes

External Storage Class

- If the declared variable is needed in another file, or in the same file but at a point earlier than that at which it has been defined, it must be declared of storage class external
- Features of a variable with an external storage class are:

| | |
|-----------------------|---|
| Storage | Memory |
| Default initial value | Zero |
| Scope | Global |
| Life | As long as the program's execution doesn't come to an end |
- An extern data type variable declaration is of the form:
`extern date-type variable-name;`
- Definition of an external variable, specified without the keyword `extern`, causes the storage to be allocated, and also serves as the declaration for the rest of that source file

January 06, 2016 Proprietary and Confidential | 13 | Capgemini INNOVATION TECHNOLOGY CONSULTING

Storage Classes: External Storage Class:

If the declared variable is needed in another file, or in the same file but at a point earlier than that at which it has been defined, it must be declared of storage class external.

Features of a variable with an external storage class are as follows

| | |
|-----------------------|--|
| Storage | Memory |
| Default initial value | Zero |
| Scope | Global |
| Life | As long as the program's execution doesn't come to an end. |

A variable has to be declared with the keyword `extern` before it can be used.

External Storage Class: Example

Following example shows the definition, declaration and use of external variables

| main.c | compute.c |
|--|---|
| #include <stdio.h> int calculate(void); /* Declaration of Theory_Mark & Practical Mark */ int Theory_Mark,Practical_Mark; int main(void) {printf("Enter Marks"); scanf("%d%d",&Theory_Mark,&Practical_Mark); printf("Module Mark=%d",calculate()); return 0; } | #include <stdio.h> /* Extern declaration of Theory_Mark & * Practical_Mark variables. * No new variables are created */ extern int Theory_Mark,Practical_Mark; int output(void); int result=0; int calculate() {result=Theory_Mark+Practical_Mark; return(result); } |

January 05, 2016 | Proprietary and Confidential | - 13 -



External Storage Class (contd..):

Example in the slide above shows the definition, declaration and use of external variables.

The program consists of two modules main.c and compute.c.

4.2: Storage Classes
Which to use and When

➤ **Tips for usage of different storage classes in different situations:**

- Static storage class is used only if you want the value of a variable to persist between different function calls. Typical application of this storage is recursive functions
- Register storage class is used only those variables which are being used very often in the program. Typical application of this storage is loop counters, which get used a number of times in a program since the access is faster
- Extern storage class is used only those variables which are being used by almost all the functions in the program. This would avoid unnecessary passing of these variables as arguments when making a function call. If you don't have any of the express needs mentioned above, then use the auto storage class

January 06, 2016 | Proprietary and Confidential | 14 |

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Storage Classes: Which to use When:

Following are some tips for the usage of different storage classes in different situations:

1. Static storage class is used only if you want the value of a variable to persist between different function calls. Typical application of this storage is recursive functions.
 2. Register storage class is used only those variables which are being used very often in the program. Typical application of this storage is loop counters, which get used a number of times in a program since the access is faster.
 3. Extern storage class is used only those variables which are being used by almost all the functions in the program.
- This would avoid unnecessary passing of these variables as arguments when making a function call.
- If you don't have any of the express needs mentioned above, then use the auto storage class.

4.3: Recursion

Recursion

- Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied
- The process is used for repetitive computations in which each action is stated in terms of a previous result
- Functions may be defined recursively; that is, a function may directly or indirectly call itself in the course of execution
- If the call to a function occurs inside the function itself, the recursion is said to be direct
- If a function calls another function, which in turn makes a call to the first one, the recursion is said to be indirect
- The chain of calls may be more involved; there may be several intermediate calls before the original function is called back

January 18, 2016 | Proprietary and Confidential | - 19 -

 Capgemini
CONSULTING TECHNOLOGY INNOVATION

Recursion:

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. Many iterative problems can be written in this form.

Functions may be defined recursively; that is, a function may directly or indirectly call itself in the course of execution,

If the call to a function occurs inside the function itself, the recursion is said to be direct.

Recursion (contd...)

➤ For example:

```
long int factorial(int fact_num)
{
    if((fact_num == 0)||(fact_num == 1))
        return 1;
    else
        return(fact_num * factorial(fact_num-1));
}
```

January 08, 2016 | Proprietary and Confidential | 16 -



Recursion (contd..):

If a function calls another function, which in turn makes a call to the first one, the recursion is said to be indirect.

The chain of calls may be more involved; there may be several intermediate calls before the original function is called back.

Recursion: Example

Code Snippet

```
/*To print fibonacci series of first 10 numbers using recursion */
#include <stdio.h>
int fibonacci();
int fibonacci(int index_value)
{
    if(index_value==1||index_value==2)
        return 1;
    else
        return(fibonacci(index_value-1)+fibonacci(index_value-2));
}
```



Recursion (contd..):

The function fibo() is called recursively to get the next number in Fibonacci series. If the value of cur_fibo_value is 1 or 2, the recursive function will not be called and fibo() function return 1.

Recursion: Example (Contd..)

```
void main(void)
{
    int index_value;
    for(index_value=1; index_value <=10; index_value++)
        printf("%d\n",fibonacci(index_value));
}
```

January 05, 2016 | Proprietary and Confidential | - 18 -



Recursion (contd..):

The compiler uses one such data structure called stack for implementing normal as well as recursive function calls.

Common Best Practices in C Programming

January 26, 2016

Proprietary and Confidential



4-4: Common Best Practices

Functions

- Function names should be meaningful
- Function names should reflect what they do and what they return
 - Functions are used in expressions, often in an if clause, so they need to read appropriately. For example:
 - if (checksize(x))
 - The above expression is unhelpful because it does not tell us whether checksize returns true on error or non-error
 - if (validsize(x))
 - Instead the above expression makes the point clear
 - By making function names verbs and following other naming conventions programs can be read more naturally
 - Good example:
`check_for_errors()`
 - Bad example:
`error_check()`

January 26, 2016 | Proprietary and Confidential | - 30 -



Functions (contd...)

- **scanf** should never be used in serious applications. Its error detection is inadequate. Look at the example below:

```
int main(void)
{ int i; float f;
printf("Enter an integer and a
float: ");
scanf("%d %f", &i, &f);
printf("I read %d and %f\n", i, f);
return 0;
}
```

Test run

Enter an integer and a float: 182 52.38
Output: 182 and 52.380001

Another TEST run

Enter an integer and a float: 6713247896
4.4
Output: -1876686696 and 4.400000



Functions (contd...)

➤ getchar() - macro or function:

- The following program copies its input to its output:

```
#include <stdio.h>
int main(void) {
register int empno;
while ((empno = getchar()) != EOF)
putchar(empno); }
```

- Removing the #include statement from the program would cause it to fail to compile because EOF would then be undefined



Functions (contd...)

- We can rewrite the program in the following way:

```
#define EOF -1
int main(void)
{
    register int empno;
    while ((empno = getchar()) != EOF)
        putchar(empno);
}
```

- This will work on many systems but on some it will run much more slowly

January 18, 2016 | Proprietary and Confidential | + 23 +



Since function calls usually take a long time, getchar is often implemented as a macro. This macro is defined in stdio.h, so when #include <stdio.h> is removed, the compiler does not know what getchar is. On some systems it assumes that getchar is a function that returns an int.

In reality, many C implementations have a getchar function in their libraries, partly to safeguard against such lapses. Thus in situations where #include <stdio.h> is missing the compiler uses the function version of getchar. Overhead of function call makes the program slower. The same argument applies to putchar.

Functions (contd...)

- Avoid Defining functions before main as the reader have to search for main, which is the place to start when analyzing code and it gets annoying
- Hence it reduces the readability

```
int tax_calculation();
int main()
{ tax_calculation();
  return 13;
  int tax_calculation()
  { // ... ten pages worth of code. }
```

- If a program is making use of multiple functions then it is a good practice to keep those functions in separate file
- It increase the performance and will be easy to debug

Summary

In this lesson, you have learnt:

- Functions break large complicated computing tasks into smaller and simpler ones .
- The functions written by the user, are termed as "User Defined Functions".
- If a variable is defined outside any function at the same level as function definitions is called as External variable.



January 06, 2016 | Proprietary and Confidential | - 15 -



Summary:

Any function can be called from any other function. Even main() can be called from other functions.

Review Question

➤ **Complete the following statement:**

- Function declaration is also called as _____.



➤ **True or False?**

1. If return type of function is not specified, it defaults to int. (True / False)
2. A C program consists of one or more functions with one main() function. (True / False)

Review Question: Match the Following

1. Automatic Storage class
2. Static storage class
3. Register storage class
4. External storage class

1. Value of the variable persists between different function calls.
2. Till the control remains within the block, in which it is defined.
3. As long as the program's execution doesn't come to an end.
4. Is typically specified for heavily used variables enhancing performance by minimizing access time.



January 18, 2016 | Proprietary and Confidential | - 27 -



Lab Session

➤ Lab 3



January 26, 2016 Proprietary and Confidential | > 18 |

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

C Programming

Lesson 4: Arrays



January 05, 2016 | Proprietary and Confidential | 1 / 1

Lesson Objectives

➤ **To understand the following topics:**

- Single-Dimension Array
- Two-Dimension Array
- Strings
- Common Best Practices



S.1: Arrays

Memory Representation Of Two Dimensional Array

- A two-dimensional array $a[i][j]$ can be visualized as a table or a matrix of i rows and j columns
- All the elements in a row are placed in contiguous memory locations

| | col 1 | col 2 | — | col colid-1 | col colid |
|----------------|-----------------|-----------------|---|-----------------------|-----------------------|
| row 1 | Emp[0][0] | Emp[0][1] | — | Emp[0][colid-2] | Emp[0][colid-1] |
| row 2 | Emp[1][0] | Emp[1][1] | — | Emp[1][colid-2] | Emp[1][colid-1] |
| row rowid-1 | Emp[rowid-2][0] | Emp[rowid-2][1] | — | Emp[-2][colid-2] | Emp[rowid-2][colid-1] |
| row rowid | Emp[rowid-1][0] | Emp[rowid-1][1] | — | Emp[rowid-1][colid-2] | Emp[rowid-1][colid-1] |

January 05, 2016 | Proprietary and Confidential | - 9 -

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Multi-dimension arrays: Memory representation:
A two-dimensional array $\text{Emp}[\text{rowid}][\text{colid}]$ can be visualised as a table of matrix of rowid rows and colid columns as below:
All the elements in a row are placed in contiguous memory locations.

Two-Dimension Array Processing

- Processing of two-dimensional array is same as that of single dimensional arrays.
- The common way to process a two-dimension array is by means of nested loops.

Multi-dimension arrays: Array Processing:

Processing of two-dimensional array is same as that of single dimensional arrays.
The common way to process a two dimension array is by means of nested loops.

Two-Dimensional Arrays: Sample Program

```
/*students in a test by accepting roll number and marks of each student */
#include <stdio.h>
void main(void){
    int index;
    float sum=0;
    int student[25][2];
    for(index=0; index<25; index++) {
        printf("Enter Roll no and marks : ");
        scanf("%d%d", &student[index][0], &student[index][1]);
        sum += student[index][1];
    }
    /* Roll no will get stored in students[index][0] and marks in student[index][1]
     */
    printf("\n Average marks : %.2f\n", sum/25);
}
```



January 05, 2016 | Proprietary and Confidential | - 5 -

Two-dimension array: Sample program:

This program accepts the roll number and marks.

Marks and roll numbers are stored in columns in the array.

Average of all the marks is calculated by summing just one column student[i][1];

Two-Dimensional Array of Characters

```
char namelist[3][10] = {  
    "akshay",  
    "parag",  
    "raman"  
};
```

Declares an Array of 3 strings of 10 characters.

- The memory representation of the above array is

| | | | | | | | | | | |
|------|---|---|---|---|---|----|----|--|--|--|
| 1001 | a | k | s | h | a | y | \0 | | | |
| 1011 | p | a | r | a | g | \0 | | | | |
| 1021 | r | a | m | a | n | \0 | | | | |

- Even though 10 bytes are reserved for storing the name 'akshay', it occupies only 7 bytes. Thus, 3 bytes go waste.



January 05, 2016 Proprietary and Confidential - 6 -

Two-Dimension Array Characters:

```
char namelist[3][10] = {  
    "akshay",  
    "parag",  
    "raman"  
};
```

Declares an Array of 3 strings of 10 characters

Even though 10 bytes are reserved for storing the name 'akshay', it occupies only 7 bytes. Thus 3 bytes go waste.

Built-in Library Functions

| Function | Description |
|-----------------------|--|
| strlen (string) | Finds the length of a string. |
| strlwr(string) | Converts a string to lowercase. |
| strupr (string) | Converts a string to uppercase. |
| strcat (str1,str2) | Appends one string at the end of another. |
| strncat (str1,str2,n) | Appends first n character of a string at the end of another. |
| strcpy (str1,str2) | Copies a string into another. |
| strncpy(str1,str2,n) | Copies first n character of one string into another. |
| strcmp(str1,str2) | Compares two strings. |
| strncmp(str1,str2,n) | Compares first n characters of two strings. |

January 08, 2016 | Proprietary and Confidential | - 7 -



Built-in Library Functions:

The header file string.h provides useful set of string functions. These functions help in manipulating strings. To use these functions, the header file string.h must be included in the program with the statement:

The strlen() function returns an integer value which denotes the length of the string passed. The length of a string is the number of characters present in it, excluding the terminating null character.

Strlwr Converts a string to lowercase

Strupr Converts a string to uppercase

strcat() function

The strcat() function concatenates two strings, i.e. it appends one string at the end of another. The function accepts two strings as parameters and stores the contents of the second string at the end of the first.

The strcpy() function copies one string to another.

This function accepts two strings as parameters and copies the second string character by character into the first one, upto and including the null character of the second string.

The function strcmp() compares two strings.

This function is useful while writing program for constructing and searching strings. The function accepts two strings as parameters and returns an integer value, depending upon the relative order of the two strings, as follows:

Less than 0 If the first string is less than the second

Equal to 0 If both are identical

Greater than 0 If the first string is greater than the second

5.2: Common Best Practices

Strings

- The size of the array should be one byte more than the actual space occupied by the string since the compiler appends a null character at the end of the string.
- Example:
 - `char city[9] = "NEW YORK";`
 - Length of the character buffer shall be expected as length of the string+1.

January 05, 2016 | Proprietary and Confidential | - 8 -

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

The reason that city had to be 9 elements long is that the string NEW YORK contains 8 characters and one element space is provided for the null terminator.

Strings

- strcpy function is not null terminated so always add null character after doing string manipulation

- Example

```
char source[MAX] = "123456789"; char destination[MAX] = "abcdefg";
strcpy(destination,source); // Output:destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'
strcpy(destination,source); // Output:destination is originally = 'abcdefg'
After strcpy, destination becomes '12345fg'
/*the result will not be null-terminated. */
```



January 05, 2016 | Proprietary and Confidential | - 9 -

The strcpy function is not null terminated so always add null character after doing string manipulation.

Example

```
char source[MAX] = "123456789"; char destination[MAX] = "abcdefg";
strcpy(destination,source); // Output:destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'
strcpy(destination,source); // Output:destination is originally = 'abcdefg'
After strcpy, destination becomes '12345fg'
/*the result will not be null-terminated. */
```

In the previous example of strcpy, if the following line of code is added, then it will give the desired output.

```
destination1[index-1]= '\0';
Adding null after manipulation .
Output: destination1 is originally='abcdefg'
After strcpy, destination1 becomes '12345'
/*the result is made to be null-terminated. */
```

Refer the following example:

```
#include<stdio.h>
#define MAX 10
void main(void)
{
    char source[MAX] = "123456789";
    char source1[MAX] = "123456789";
    char destination[MAX] = "abcdefg";
    char destination1[MAX] = "abcdefg";
    char *return_string;
    int index = 5;

    /* This is how strcpy works */
    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("after strcpy, dest becomes '%s'\n\n", destination);

    /* This is how strncpy works */
    printf("destination1 is originally = '%s'\n", destination1 );
    return_string = strncpy( destination1, source1, index );
    printf("After strncpy, destination1 becomes '%s'\n", destination1 );
}
```

Output: destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'

Output: destination1 is originally = 'abcdefg'
After strncpy, destination1 becomes '12345fg'

Now add the null terminating statement in previous program:

```
#include<stdio.h>
#define MAX 10
void main(void)
{
    char source[MAX] = "123456789";
    char source1[MAX] = "123456789";
    char destination[MAX] = "abcdefg";
    char destination1[MAX] = "abcdefg";
    char *return_string;
    int index = 5;

    /* This is how strcpy works */
    printf("destination is originally = '%s'\n", destination);
    return_string = strcpy(destination, source);
    printf("after strcpy, dest becomes '%s'\n\n", destination);

    /* This is how strncpy works */
    printf("destination1 is originally = '%s'\n", destination1 );
    return_string = strncpy( destination1, source1, index );
    destination1[index-1] = '\0';
    printf("After strncpy, destination1 becomes '%s'\n", destination1
    );
}
```

Output: Output:destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'

Output:destination1 is originally = 'abcdefg'
After strcpy, destination1 becomes '12345'

Strings

- Characters and string are very different so strings function will not work for character.
 - `strcat(string, '|')` //not valid
 - `strcat(string, "|")` //valid



Arrays

➤ **Cannot use const values in array dimensions, as in:**

```
const int size = 5;  
int mark[size];
```

- The const qualifier really means ‘read-only’; an object so qualified is a run-time object that cannot be assigned to.
- The value of a const qualified object is therefore not a constant expression in the full sense of the term and cannot be used for array dimensions.
- When you need a true compile time constant, use a preprocessor #define.

```
const int size = 5;  
int mark[size];
```

const qualifier really means ‘read-only’. An object so qualified is a run-time object as the value of size is going to be evaluated at run time. Whereas size of array will be evaluated at compile time itself.

So if you try to do the same, it will give the following error: “cannot allocate an array of constant size 0”

Arrays

- An extern array defined in one file and used in another.
 - file1.c
 - int array[] = {1, 3, 5};
 - file2.c
 - extern int array[];
- The sizeof operator will not work on extern array in file2.c. An extern array of unspecified size is an incomplete type. You cannot apply sizeof to it because sizeof operates at compile time and is not able to learn the size of an array that is defined in another file.

January 05, 2016 | Proprietary and Confidential | + 14 +



An extern array of unspecified size is an incomplete type; you can not apply sizeof to it because sizeof operates at compile time and is not able to learn the size of an array that is defined in another file.

For example:

```
File1.c
#include<stdio.h>

/*
Extern declaration array.
No new variables are created
*/
extern int mark[];
```

```
int arrayszie()
{
    int length;
    length=sizeof(mark);
    return(length);
}
```

```
File2.c
#include<stdio.h>
#include "exterarray.c"

int arrayszie(void);

mark[] = {71,82,93};

void main()
{
    int index;
    for(index=0;index<3;index++)
    {
        printf("%d ",mark[index]);
    }
    printf("Result_firstfile= %d\n",sizeof(mark));
    printf("Result_Secondfile= %d\n",arraysize());
}

}
```

Output:

```
718293 result_firstfile = 12.
Result_secondfile = 0.
```

We can go for following option:

Declare a companion variable, containing the size of the array, defined and initialized in same source file where the array is defined.

```
file1.c
int array[] = {1, 2, 3};
int arraysz = sizeof(array);
file2.C
extern int array[];
extern int arraysz;
```

Lab Session

➤ Lab 4



Hands On

January 05, 2016 | Proprietary and Confidential | + 16 +

 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Summary

➤ **In this lesson, you have learnt:**

- Arrays, whose elements are specified by one subscript, are called one-dimensional arrays.
- The array size may be omitted during declaration.
- Multidimensional Arrays are defined in much the same manner as single dimensional arrays, except that a separate pair of square brackets is required for each subscript.



Review Questions

➤ **Complete the following statement:**

- Arrays, whose elements are specified by one subscript, are called as _____.

➤ **True or false?**

- The location of an array is the location of its last element.
 - True/False
- The type of array is the data type of its elements.
 - True/False



 Capgemini
CONSULTING TECHNOLOGY OUTSOURCING

Review – Match the Following

| | |
|-------------------------|---|
| 1. <code>strncat</code> | 1. Converts a string to lowercase |
| 2. <code>strlen</code> | 2. Compares two strings |
| 3. <code>strcmp</code> | 3. Finds the length of a string |
| 4. <code>strlwr</code> | 4. Append first n character of a string at the end of another |



C Programming

Lesson 5: Pointers

January 18, 2016 | Proprietary and Confidential | 1 / 1



Table Of Contents

➤ Pointers

- Pointers to Functions
- Pointers to Arrays
- Pointers to Pointers
- Dynamic Memory Allocation

Lesson Objectives

➤ In this lesson, you will learn about:

- The method in which pointer arithmetic and pointer is used with functions
- The method in which the pointers are used to process one-dimensional and two-dimensional arrays



Lesson Coverage

➤ In this lesson, you will learn the following topics:

- Pointers and Functions:
 - Call by Value, Call by Reference
 - Pointers to Functions: Declaration, Invoking function using Pointer
 - Functions returning Pointers
- Pointer and Array:
 - Pointers and multidimensional (Two dimensional) arrays
 - Array of pointers (int, string)

January 18, 2016 | Proprietary and Confidential | + 4 +



Lesson Coverage:

The significance of pointers in C is the flexibility it offers in the programming. Pointers enable us to achieve parameter passing by reference, deal concisely and effectively either arrays, represent complex data structures, and work with dynamically allocated memory.

Although, a lot of programming can be done without the use of pointers, their usage enhances the capability of the language to manipulate data. Pointers are also used for accessing array elements, passing arrays and strings to functions, creating data structures such as linked lists, trees, graphs, and so on.

Lesson Coverage

➤ In this lesson, you will learn the following topics (contd.):

- Pointers to Pointers
- Dynamic Memory Allocation:
 - * List of Pointer Declarations examples
 - * Command Line Arguments

6.1: Pointers and Functions

Concept of Pointers and Functions

- A function can take a pointer to any data type, as argument and can return a pointer to any data type
 - For example, consider the following function definition.

```
double *maxp(double *xp, double *yp)
{
    return (*xp >= *yp ? xp : yp);
}
```

- It specifies that the function maxp() returns a pointer to a double variable, and expects two arguments, both of which are pointers to double variables.

Concept of Pointers and Functions

- The function de-references the two argument pointers to get the values of the corresponding variables, and returns the pointer to the variable that has the larger of the two values .
- Thus given that, double xp=1.5, yp=2.5, *mp;
the statement mp = maxp(&xp, &yp);
makes mp point to v.

6.1: Pointers to Functions

Concept of Pointers to Functions

- Functions have addresses just like data items.
- A pointer to a function can be defined as the address of the code executed when the function is called.
- A function's address is the starting address of the machine language code of the function stored in the memory.
- Pointers to functions are used in the following tasks:
 - writing memory resident programs
 - writing viruses or vaccines to remove the viruses

January 18, 2016 | Proprietary and Confidential | - 8 -

 Capgemini
INNOVATING THROUGH DIVERSITY

Pointers to Functions:

Address of a Function:

The address of a function can be obtained by only specifying the name of the function without the trailing parentheses.

For example, ash() is a function already defined, then ash is the address of the function ash().

Pointer to Functions – Declaration

- The declaration of a Pointer to a Function requires the function's return type and the function's argument list to be specified along with the pointer variable.
- The general syntax for declaring a pointer to a function is:

```
return-type (*pointer variable)(argument_list);
```

- Thus, the declaration `int (*fp)(int i, int j);` declares fp to be a variable of type “pointer to a function that takes two integer arguments and returns an integer as its value”.

January 18, 2016 | Proprietary and Confidential | 9 |



Pointers to Functions:

The declaration of a pointer to a function requires the function's return type and the function's argument list to be specified along with the pointer variable.

The general syntax for declaring a pointer to a function is as follows:

```
return type (*pointer variable)(function's argument list);
```

Thus, the declaration `int (*fp)(int i,int j);` declares fp to be a variable of type “pointer to a function that takes two integer arguments and return an integer as its value.”

The identifiers i and j are written for descriptive purposes only.

The preceding declaration can, therefore also be written as `int (*fp)(int,int);`

Thus, the following declarations hold true:

| | |
|-------------------------------|---|
| <code>int i(void);</code> | : declares i to be a function with no parameters that return an int. |
| <code>int* pi(void);</code> | : declares pi to be a function with no parameters that returns a pointer to an int. |
| <code>int (*ip)(void);</code> | : declares ip to be a pointer to a function that returns an integer value and takes no arguments. |

Pointer to Functions – Example

- Example: Let us see an example for Pointer to Functions:

```
#include<stdio.h>
int stud_detail(int rollno)
{
    return rollno;
}
float mark_detail(float mark)
{
    return mark;
}
```

January 18, 2016 | Proprietary and Confidential | 10 |



Note:

After declaring the function prototypes and two pointers s_ptr and m_ptr to the functions; s_ptr is assigned the address of function stud_detail and m_ptr is assigned the address of function mark_detail.

Pointer to Functions – Example

➤ Example contd. :

```
void main()
{
    int (*s_ptr)(int); /*declaring pointer to function*/
    float (*m_ptr)(float); /*declaring pointer to function*/
    int rollno=5;
    float mark=1.5;
    s_ptr=stud_detail;
    m_ptr=mark_detail;
    printf("RollNo=%d Mark=%f\n",s_ptr(rollno),m_ptr(mark));
}
```

Note:

In the above example, the pointer to function is declare in main() function as int(*s_ptr)(int) and float(*m_ptr)(float).

Demo on Invoking Functions

- Pointer_to_fun.c and
pointer_to_function.c demo



Invoking Functions - Example

- Example: Let us see an example on Invoking Functions:

```
/* Example : Invoking function using pointers */
#include <stdio.h>
void main(void)
{
    unsigned int fact(int);
    unsigned int ft,(*ptr)(int);
    int num;
    ptr=fact; /* assigning address of fact() to ptr */

    printf("Enter integer whose factorial is to be found:");
    scanf("%d",&num);
    ft=ptr(num); /* call to function fact using pointer ptr */
    printf("Factorial of %d is %u \n",num,ft);
}
```

January 18, 2016 | Proprietary and Confidential | + 19 +



Note:

In the pointer declaration to functions, the pointer variable along with the operator (*) plays the role of the function name. Hence, while invoking function by using pointers, the function name is replaced by the pointer variable.

Invoking Functions – Example

➤ Example contd.:

```
unsigned int fact(int num)
{
    unsigned int index,ans;
    if (num == 0)
        return(1);
    else
    {
        for(index=num, ans=1; index>1 ; ans *= index--);
        return(ans);
    }
}
```

Output:
Enter integer whose factorial is to be found : 8
Factorial of 8 is 40320

6.2: Functions returning Pointers

Concept of Functions returning Pointers

- A function can return a pointer like any other data type.
- However, it has to be explicitly mentioned in the calling function and in the function declaration.
- While retaining pointers, return the pointer to global variables or static or dynamically allocated address.
- Do not return any addresses of local variables because it does not exist after the function call.

January 18, 2016 | Proprietary and Confidential | 15 / 2



Functions returning Pointers:

We have already learnt that a function returns an int, a double, or any other data type. Similarly, it can return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration.

While retaining pointers, return the pointer to global variables or static or dynamically allocated address. Do not return any addresses of local variables because stop to exit after the function call.

Functions returning Pointers - Example

Example:

```
/* Example: Function returning pointers */
/* Program to accept two numbers and find
greater number */
#include <stdio.h>
void main(void)
{
    int a,b,*c;
    int* check(int*,int*);
    printf("Enter two numbers : ");
    scanf("%d%d",&a,&b);
    c=check(&a,&b);
    printf("\n Greater numbers : %d",*c);
}
```

check function
takes two integers
pointers as
arguments and
returns a pointer to
integer

January 18, 2016 | Proprietary and Confidential | 16 |



Functions returning Pointers:

The address of integers being passed to check() are collected in p and q. Then in the next statement the conditional operators test the value of *p and *q and return either the address stored in p or the address stored in q. This address gets collected in c in main().

Functions returning Pointers - Example

➤ Example (contd.):

```
int* check(int *p,int *q)
{
    if(*p >= *q)
        return(p);
    else
        return(q);
}
```

6.3: Pointers and Arrays

Concept of Pointers and Arrays

- Arrays are internally stored as pointers.
- A pointer can efficiently access the elements of an array.
 - Example:

```
#include <stdio.h>
void main(void)
{
    int ar[5]={10, 20, 30, 40, 50};
    int i, *ptr;
    ptr = &ar[0]; /* same as ptr = ar */
    for(i=0; i<5; i++)
    {
        printf("%d-%d\n", ptr, *ptr);
        ptr++;
    }
}
```

increments the
pointer to point to
the next element
and not to the next
memory location

January 18, 2016 | Proprietary and Confidential | + 18 +



Pointers and Arrays:

This program prints the array elements and their addresses using a pointer `*ptr`.

`*ptr` has the value of the first element of the array.

Array elements are printed by incrementing the pointer.

Note: The pointer is of type `int` therefore it will be incremented by 4 bytes.

Concept of Pointers and Arrays

– Example (contd.):

```
}
```

Output:

```
5000-10
5004-20
5008-30
5012-40
5016-50
```

Concept of Pointers and Arrays

➤ $\text{ar} \equiv \&\text{ar}[0] \equiv 5000$

- Hence, $*\text{ar} \equiv *(\&\text{ar}[0])$ i.e. $*\text{ar} \equiv \text{ar}[0]$ or $*(\text{ar}+0) \equiv \text{ar}[0]$
- To make the above statement more general, we can write:
 - + $(\text{ar}+i) \equiv \text{ar}[i];$
 - where, $i=0,1,2,3,\dots$

| | | | | | |
|----|-------|-------|-------|-------|-------|
| ar | 5000 | 5004 | 5008 | 5012 | 5016 |
| | ar[0] | ar[1] | ar[2] | ar[3] | ar[4] |

January 18, 2016 | Proprietary and Confidential | + 30 +



Pointers and Arrays:

An integer pointer, `ptr` is explicitly declared and assigned the starting address. The memory representation of above declared array `ar` is shown below: (assuming an integer takes 4 bytes of storage)

Recall that an array name is really a pointer to the first element in that array. Therefore, address of the first array element can be expressed as either `&ar[0]` or simply `ar`.

i.e. `ar=&ar[0]=5000`

Concept of Pointers and Arrays

- Hence any array element can be accessed using pointer notation, and vice versa. It will be clear from following table:

| char c[10], int i; | |
|--------------------|------------------|
| Array Notation | Pointer Notation |
| &c[0] | c |
| c[i] | *(c+i) |
| &c[i] | c+i |

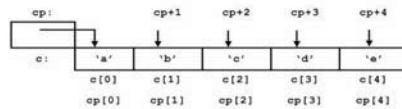
Concept of Pointers and Arrays

For example, given that

```
char c[5] = {'a', 'b', 'c', 'd', 'e'};  
char *cp; //
```

and

```
cp = c;
```



and

| | | | |
|-------------------|------------------------|--------------------|-------------------------|
| <code>c[0]</code> | <code>&c[0]</code> | <code>*cp</code> | <code>&cp[0]</code> |
| <code>c[1]</code> | <code>&c[1]</code> | <code>*cp+1</code> | <code>&cp[1]</code> |
| <code>c[2]</code> | <code>&c[2]</code> | <code>*cp+2</code> | <code>&cp[2]</code> |
| <code>c[3]</code> | <code>&c[3]</code> | <code>*cp+3</code> | <code>&cp[3]</code> |
| <code>c[4]</code> | <code>&c[4]</code> | <code>*cp+4</code> | <code>&cp[4]</code> |

Concept of Pointers and Arrays

➤ Using this concept, we can write the program as shown below:

```
# include <stdio.h>
void main(void)
{
    int ar[5]={10,20,30,40,50};
    int i;
    for(i=0;i<5;i++)
        printf("%d-%d\n",*(ar+i),*(ar+i));
}
```

Pointers and Arrays:

The main difference between an array and a pointer is that an array name is a constant, (a constant pointer to be more specific). On the other hand, a pointer is a variable.

Concept of Pointers and Arrays

- C does not allow to assign an address to an array.
- For example:
 - Consider the array: ar=&a;
 - It is invalid.
- The main difference between an array and a pointer is that an array name is a constant, (a constant pointer to be more specific), whereas a pointer is a variable.

6.4: Pointers and Multidimensional Arrays

Concept of Pointers & Multidimensional Arrays

➤ A Multi-dimensional array in C is really a one-dimensional array, whose elements are themselves arrays, and is stored such that the last subscript varies most rapidly. The name of the multi-dimensional array is, therefore, a pointer to the first array. Thus, the following declaration holds true:

— int matrix[3][5];

January 18, 2016 | Proprietary and Confidential | + 25 +

 Capgemini
INNOVATING. TRANSFORMING. DELIVERING.

Pointers and Multidimensional Arrays:

A Multi-dimensional array in C is really a one-dimensional array, whose elements are themselves arrays, and is stored such that the last subscript varies most rapidly. The name of the multi-dimensional array is, therefore, a pointer to the first array.

For example: Consider the following declaration:

int matrix[3][5];

It specifies that the array matrix consists of three elements, each of which is an array of five integer elements, and that the name matrix is a pointer to the first row of the matrix.

Instead of using subscripts, an element in a multi-dimensional array can be referenced using an equivalent pointer expression.

For example: The element matrix[i][j] can be referenced by using the pointer expression:

((matrix+i)+j)Matrix is a pointer to the first row;

matrix + i is a pointer to the ith row;

*(matrix + i) is the ith row which is converted into a pointer to the first element in the ith row;

*(matrix + i)+j is the pointer to the jth element in ith row;

((matrix + i)+j) is matrix[i][j], the jth element in ith row;

Concept of Pointers & Multidimensional Arrays

- For example: The element `matrix[i][j]` can be referenced using the pointer expression `*(*(matrix+i)+j)`

- `Matrix` is a pointer to the first row;
- `matrix + i` is a pointer to the `i`th row;
- `*(matrix + i)` is the `i`th row which is converted into a pointer to the first element in the `i`th row;

Concept of Pointers & Multidimensional Arrays

- A two-dimensional array is a collection of one-dimensional array as a pointer to a group of contiguous one-dimensional arrays.
- So a two-dimensional array declaration can be written as the following data-type
 - `(*array)[expression 2];`
 - * where,
 - data-type refers to the data type of the array,
 - *array is the pointer to array and
 - expression2 indicate the max. number of elements in a row
- Note: The parentheses that surround the array name must be present.

Pointers and Multidimensional Arrays:

Since a one-dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), it is reasonable to expect that a multidimensional array can also be represented with an equivalent pointer notation. This is indeed the case.

For example: A two-dimensional array, is actually a collection of one-dimensional array as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

data-type `(*array)[expression 2];`
 rather than

data-type `array[expression 1] [expression 2];`

In these declarations, data-type refers to the data type of the array, array is the corresponding array name, and expression1, expression2....., expression are positively-valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Notice the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present.

Without them we will be defining an array of pointers rather than a pointer to a group of arrays, since these particular symbols (that is, the square brackets and the asterisk) will normally be evaluated right-to-left. We will say more about this in the next section.

Concept of Pointers & Multidimensional Arrays

➤ Example:

- Suppose x is a two-dimensional integer array having 10 rows and 20 columns, then it can be declared as follows:
 - * int (*x)[20];
 - Here, x is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays.
 - Thus, x points to the first 20-element array, which is actually the first row (row 0) of the original two-dimensional array.
 - Similarly, (x + 1) points to the second 20-element array, which is the second row (row 1) of the original two-dimensional array, and so on.

January 18, 2016 | Proprietary and Confidential | 18 |



Pointers and Multidimensional Arrays:

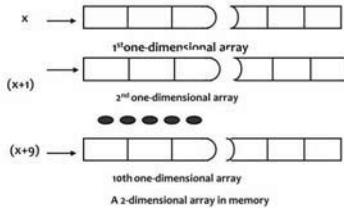
Example:

Suppose that x is a two-dimensional integer array having 10 rows and 20 columns, we can declare x as:

int (*x)[20];
rather than
int x[10][20];

In the first declaration, x is defined to be a pointer to a group of contiguous, one-dimensional, 20-element integer arrays. Thus, x points to the first 20-element array, which is actually the first row (row 0) of the original two-dimensional array.

Concept of Pointers & Multidimensional Arrays



- Example: Suppose that x is a two-dimensional integer array having 10 rows and 20 columns, as declared in the previous example.

Then the item in row 2, column 5 can be accessed by writing either
 $x[2][5]$ or $*(*x + 2) + 5$

January 18, 2016 | Proprietary and Confidential | 29 |



Pointers and Multidimensional Arrays:

Similar to the example in the slide, $(x + 1)$ points to the second 20-element array, which is the second row (row 1) of the original two-dimensional array, and so on. The second form requires some explanation. First, note that $(x + 2)$ is a pointer to row 3. Therefore, the object of this pointer, $*x + 2$, refers to the entire row. Since row 3 is a one-dimensional array, $*x + 2$ is actually a pointer to the first element in row 3. We now add 5 to this pointer. Hence, $(*x + 2) + 5$ is a pointer to element 5 (the sixth element) in row 3. The object of this pointer, $*(*x + 2) + 5$, therefore refers to the item in column 5 to row 3, which is $x[2][5]$.

6.5: Array of Pointers

Concept of Array of Pointers

- An array of pointers is collection of addresses.
- Addresses can be address of “isolated variables” or addresses of “array elements” or any other addresses.
 - For example: Consider the following declaration.

`char *day[7];`

- It defines day to be an array consisting of seven character pointers.

January 18, 2016 | Proprietary and Confidential | 30 / 40

 Capgemini
INVESTING IN YOUR FUTURE

Array of Pointers:

As we have already seen, an array is an ordered collection of data items, each of the same type, and type of an array is the type of its data items.

When the data items are of pointer type, it is known as a pointer array or an array of pointers.

Since a pointer variable always contains an address, an array of pointers is collection of addresses. The addresses present in the array of pointers can be address of “isolated variables” or addresses of “array elements” or any other addresses.

For example: Consider the following declaration:

`char *day[7];`

It defines day to be an array consisting of seven character pointers.

Concept of Array of Pointers

- The elements of a pointer array, can be assigned values by following the array definition with a list of comma-separated initializers enclosed in braces.
 - For example: Consider the following declaration.

```
char *days[7] = {"Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"};
```

- Here, the necessary storage is allocated for the individual strings, and pointers to them are stored in array elements.

Array of Pointers - Example

- ar contains addresses of isolated integer variables m, n, o, and p.
- The for loop in the program picks up addresses present in ar, and prints the values present at these addresses.

```
/* Example :Array of Pointers */  
#include <stdio.h>  
void main(void)  
{  
    int m = 25, n = 50, o = 60, p = 74;  
    int i,*ar[4]={&m,&n,&o,&p}  
    for(i=0; i<4; i++)  
        printf("%d\t",*(ar+i));  
}
```

Array of Pointers - Example

- Memory representation is shown below:

| | | | | |
|-------------------------------|------|------|------|------|
| int variables | m | n | o | p |
| values stored in variables | 25 | 50 | 60 | 74 |
| addresses of variables | 4002 | 5013 | 3056 | 9860 |

| | | | | |
|-------------------------------------|-------|-------|-------|-------|
| array of pointers | ar[0] | ar[1] | ar[2] | ar[3] |
| Elements of an array of pointers | 4002 | 5013 | 3056 | 9860 |

- An array of pointers can contain addresses of other arrays.

6.6: Multidimensional Array using Pointers

Concept

- A multidimensional array can be expressed in terms of an array of pointers.
- A two-dimensional array can be defined as a one-dimensional array of pointers by the following expression:

```
data-type *array[expression 1];
```

January 18, 2016 | Proprietary and Confidential | 34 /

 Capgemini
INNOVATING. TRANSFORMING. REDEFINING.

Multidimensional Array using Pointers:

A multidimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

In general terms, a two-dimensional array can be defined as a one-dimensional array of pointers by writing the following:

```
data-type *array[expression 1];
```

This is instead of writing the conventional array definition as follows:

```
data-type array[expression 1][expression 2];
```

Notice that the array name and its preceding asterisk are not enclosed in parentheses in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named object as an array. The preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the last (the rightmost) expression is omitted when defining an array of pointers, whereas the first (the leftmost) expression is omitted when defining a pointer to a group of arrays.

Concept

- Note that the array name and its preceding asterisk are not enclosed in parentheses in this declaration.
- Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named data item as an array.
- The preceding asterisk then establishes that the array will contain pointers.

Concept

➤ Example:

- Suppose that x is a two-dimensional char array having 10 rows and 20 columns, then it can be defined as:

```
char *x[10];
```

- Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on.
- Note that the number of elements within each row is not explicitly specified.

Multidimensional Array using Pointers:

Suppose that x is a two-dimensional char array having 10 rows and 20 columns. Then we can define x as a one-dimensional array of pointers by writing the following:

```
char *x[10];
```

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on.

Note that the number of elements within each row is not explicitly specified.

Concept

- In individual array element, such as $x[2][5]$, can be accessed by writing the following:

```
*( $x[2]+5$ );
```

- In this expression, $x[2]$ is a pointer to the first element in row 3, so that $(x[2]+5)$ points to index 5 (actually, the sixth element) within row 3. The data item of this pointer, $*(x[2]+5)$, therefore refers to $x[2][5]$.

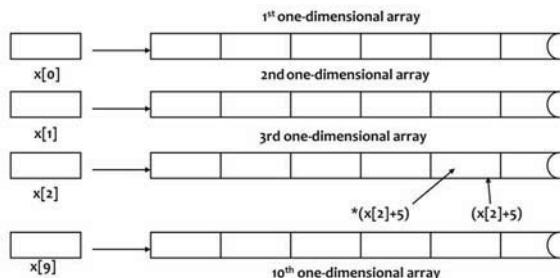
Multidimensional Array using Pointers:

In individual array element, such as $x[2][5]$, can be accessed by writing

```
*( $x[2]+5$ );
```

In this expression, $x[2]$ is a pointer to the first element in row 2, so that $(x[2]+5)$ points to element 5 (actually, the sixth element) within row 2. The object of this pointer, $*(x[2]+5)$, therefore refers to $x[2][5]$.

Concept



- Here `x` is an array of 10 pointers (`x[0]` to `x[9]`).
- Memory from the heap has to be allocated for each pointer so that valid data can be stored in it.

January 18, 2016 | Proprietary and Confidential | - 38 -



Multidimensional Array using Pointers:

Here `x` is an array of 10 pointers (`x[0]` to `x[9]`).

Memory from the heap has to be allocated for each pointer so that valid data can be stored in it. Now `x[0]` to `x[9]` can be treated as normal character pointers.

For example: Consider the following statement.

`puts (x[i]);`

It will print the string to which `x[i]` points to.

6.7: Pointer to Pointer

Concept of Pointer to Pointer

- The data item pointed to by a pointer can be an address of another data item.
- Thus a given pointer can be a pointer to a pointer to a data item.
- Accessing this data item from the given pointer, requires two levels of indirection:
 - The given pointer is dereferenced to get the pointer to the given data item.
 - The later pointer is dereferenced to get to the data item.

January 18, 2016 | Proprietary and Confidential | + 39 +

 Capgemini
INVESTING IN YOUR FUTURE

Pointer to Pointer:

A pointer provides the address of the data item pointed to by it. The data item pointed to by a pointer can be an address of another data item.
Thus, a given pointer can be a pointer to a pointer to an object. Accessing this object from the given pointer requires two levels of indirection.

First, the given pointer is dereferenced to get the pointer to the given object, and
Subsequently, this later pointer is dereferenced to get to the object.

Concept of Pointer to Pointer

- The general format for declaring a pointer to pointer is:
 - data-type **ptr_to_ptr;
- Here, the variable ptr_to_ptr is a pointer to a pointer pointing to a data item of the type data_type.
 - For example: Consider the following declarations:
 - * int i=1; int *p, **q;
 - * These declare i as an integer, and p as a pointer to an integer.
 - * The address can be assigned as follows:
 - p=&i; /* p points to i */
 - q=&p; /* q points to p */

January 18, 2016 | Proprietary and Confidential | 40 |



Pointer to Pointer:

The general format for declaring a pointer to pointer is:

data-type **ptr_to_ptr;

The declaration implies that the variable ptr_to_ptr is a pointer to a pointer pointing to a data object of the type data_type.

For example: Consider the following declarations:

```
int i=1;  
int *p;
```

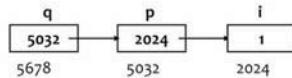
These declarations declare an integer i, and p as a pointer to an integer.

We can assign the address as follows

```
p=&i; /* p points to i */  
q=&p; /* q points to p */
```

Concept of Pointer to Pointer

- The relationship between i, p and q is pictorially depicted as shown:



- This implies that q is the pointer to p, which in turn points to the integer i.

- Consider the following expression:

- `**q;`
 - * It means "apply the dereferencing operator to q twice"
 - * The variable q is declared as:
 - `int **q;`

Concept of Pointer to Pointer

- To get the value of i, starting from q, we go through two levels of indirection. The value of *q is the content of p which is the address of i, and the value of **q is *(qi) which is 1.
- It can be written in different ways as follows:
 - $*(*q) = i = 1$
 - $*(p) = i = 1$
 - $*(*(\&p)) = i = 1$
 - $**(q) = i = 1$

January 18, 2016 | Proprietary and Confidential | + 41 +



Pointer to Pointer:

Considering the illustration in the above slide, each of the following expressions has the value 2.

```
i-1;  
*p+1;  
**q+1;
```

There is no limit on the number of levels of indirection, and a declaration such as given below is possible:

```
int ***p;
```

Concept of Pointer to Pointer

- Thus each of the expressions has the value 2:
 - `i+1;`
 - `*p+1;`
 - `**q+1;`
- There is no limit on the number of levels of indirection, and a declaration such as given below is possible:
 - `int ***p;`
- Thus, `***p` is an integer.
- `**p` is an pointer to an integer.

Demo on Pointer to Pointer



Sample Code for Pointer to Pointer

- Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, and so on.

```
/* Example :Pointers to Pointers */
#include <stdio.h>
void main(void)
{
    int data;
    int *iptr; /* pointer to an integer data */
    int **ptrptr; /* pointer to int pointer */
    iptr = &data; /* iptr points to data */
    ptrptr = &iptr; /* ptrptr points to iptr */
    *iptr = 100; /* same as data=100 */
```

Note:

Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, and so on.

Sample Code for Pointer to Pointer

```
printf("Variable data :%d \n", data);
**ptrptr = 200; /* same as data=200 */
printf("variable data :%d \n", data);
data = 300;
printf("ptrptr is pointing to :%d \n", **ptrptr);
}
```

6.8: Dynamic Memory Allocation

Concept of Dynamic Memory Allocation

- During Dynamic Memory Allocation, the memory is allocated and de-allocated at runtime.
- C provides a collection of dynamic memory management functions.
- Their prototypes are declared in alloc.h header file (under Borland C) and in malloc.h header file (under Unix and Windows).

January 18, 2016 | Proprietary and Confidential | + 47 +



Dynamic Memory Allocation:

In many programs, the number of objects to be processed by the program and their sizes are not known.

C provides a collection of dynamic memory management functions that enable storage to be allocated as needed and released when no longer required. Their prototypes are declared in alloc.h header file (under Borland C) and in malloc.h header file (under Unix).

The allocation of memory in this manner, as it is required, is known as “Dynamic Memory Allocation”.

Concept of Dynamic Memory Allocation

➤ Allocation Functions:

| Function Prototype | Description |
|--|---|
| <code>void* malloc(size);</code> | Allocates memory of given size in terms of bytes and returns the address of the first location |
| <code>void *calloc(nitems, size)</code> | Similar to malloc(), except it needs two arguments, that is, the no. of items to be allocated & the size of each item |
| <code>void *realloc(void *block, size);</code> | Resizes the block, which is already allocated according to the given size |

Concept of Dynamic Memory Allocation

➤ malloc():

- Suppose that x is to be defined as a one-dimensional, 10-element array of integers. Then it is possible to define x as a pointer variable rather than as an array. Thus we can write:

```
int *x;  
instead of  
int x[10];  
or instead of  
# define SIZE 10  
int x[SIZE];
```

- However, x is not automatically assigned a memory block when it is defined as a pointer variable. Whereas block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array

Dynamic Memory Allocation:

malloc() is used to obtain storage for an object. The allocation of storage by calling yields a pointer to the beginning of the storage allocated. It is suitably aligned, so that it may be assigned to a pointer to any type of object.

Suppose that x needs to be defined as a one-dimensional, 10-element array of integers. Then it is possible to define x as a pointer variable rather than as an array. Thus, we can write as follows:

```
int *x;  
instead of  
int x[10];  
or instead of  
# define SIZE 10  
int x[SIZE];
```

However, x is not automatically assigned a memory block when it is defined as a pointer variable. Whereas block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array.

Malloc-Example

```
#include<stdio.h>
#include<stdlib.h>
#define Null 0
main()
{
    char *buffer;
    /*allocating memory*/
    if((buffer = (char*)malloc(10))==NULL)
```

January 18, 2016 | Proprietary and Confidential | 30 / 4



Program to create a 1-D Array using malloc.

```
#include<malloc.h>
#include<stdio.h>
void main()
{
    int *a,n,i;
    printf("Enter array size:");
    scanf("%d",&n);
    a=(int *)malloc(n*sizeof(int));
    if(a==NULL)
    {
        printf("Insufficient Memory");
        return;
    }
    for(i=0;i<n;i++)
    {
        printf("Enter element[%d]:",i);
        scanf("%d",&a[i]);
    }
    for(i=0;i<n;i++)
        printf("%d\n",a[i]);
    free(a);
}
```

Malloc-Example

```
{  
    printf("malloc failed");  
    exit(1);  
}  
printf("buffer of size %d created \n", _msize(buffer));  
strcpy(buffer,"hyderabad");  
printf("\n buffer contains : %s \n",buffer);  
/*reallocation*/  
if((buffer = (char*)realloc(buffer,15))==NULL)
```

January 18, 2016 | Proprietary and Confidential | 31 /



```
#include<malloc.h>  
#include<stdio.h>  
void main()  
{  
    int **a,r,c,i,j;  
    printf("Enter row size:");  
    scanf("%d",&r);  
    printf("Enter col size:");  
    scanf("%d",&c);  
  
    a=(int **)malloc(r*sizeof(int *));  
    if(a==NULL)  
    {  
        printf("Insufficient Memory");  
        return;  
    }
```

Contd...

Malloc-Example

```
{  
    printf("realloc failed");  
    exit(1);  
}  
printf("buffer size modified \n");  
strcpy(buffer,"secunderbad");  
printf("\n buffer now contains : %s \n",buffer);  
/*freeing memory*/  
free(buffer);}
```

January 18, 2016 | Proprietary and Confidential | 53 |



```
for(i=0;i<r;i++){  
    a[i]=(int *)malloc(c*sizeof(int));  
    if(a[i]==NULL) {  
        printf("Insufficient Memory");  
        return;  
    }  
    for(j=0;j<c;j++) {  
        for(j=0;j<c;j++) {  
            printf("Enter element[%d][%d]:",i,j);  
            scanf("%d",&a[i][j]);  
        }  
    }  
    for(j=0;j<c;j++) {  
        for(j=0;j<c;j++) {  
            printf("%d\t",a[i][j]);  
        }  
        printf("\n");  
    }  
    free(a);  
}
```

Concept of Dynamic Memory Allocation

➤ **calloc():**

- The calloc() function works exactly similar to malloc(). The exception is that it needs two arguments as against the one argument required by malloc().
- General format for memory allocation using calloc() is

* where,

void *calloc(nitems, size);

- nitems: The number of items to allocate
- size: Size of each item

Concept of Dynamic Memory Allocation

➤ **calloc()** (contd.):

- For example:

```
ar=(int *)calloc(10,sizeof(int));
```

- * The above allocates the storage to hold an array of 10 integers and assigns the pointer to this storage to ar. While allocating memory using calloc(), the number of items which are allocated, are initialized.

Difference between malloc() and calloc()

- There is one major difference and one minor difference between the two functions.
 - The major difference is that malloc() does not initialize the allocated memory. The first time malloc() gives you a particular chunk of memory, the memory might be full of zeros.
 - calloc() fills the allocated memory with all zero bits. That means that anything that you are going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you are going to use as a pointer is set to all zero bits.
 - The minor difference between the two is that calloc() returns an array of objects; malloc() returns one object. Some people use calloc() to make clear that they want an array.

January 18, 2016 | Proprietary and Confidential | 35/4



Difference between malloc() and calloc():

Both the malloc() and the calloc() functions are used to allocate dynamic memory. Each operates slightly different from the other.

malloc() takes a size and returns a pointer to a chunk of memory at least that big:
`void *malloc(size_t size);`

calloc() takes a number of elements, and the size of each, and returns a pointer to a chunk of memory at least big enough to hold them all: `void *calloc(size_t numElements, size_t sizeOfElement);`

There is one major difference and one minor difference between the two functions.

The major difference is that malloc() does not initialize the allocated memory. The first time malloc() gives you a particular chunk of memory, the memory might be full of zeros.

If memory has been allocated, freed, and reallocated, it probably has whatever junk was left in it.

That means, unfortunately, that a program might run in simple cases (when memory is never reallocated) but break when used harder (and when memory is reused).

calloc() fills the allocated memory with all zero bits. That means that anything that you are going to use as a char or an int of any length, signed or unsigned, is guaranteed to be zero. Anything you are going to use as a pointer is set to all zero bits. That is usually a null pointer, but it is not guaranteed. Anything you are going to use as a float or double is set to all zero bits. That is a floating-point zero on some types of machines, but not on all.

Concept of Dynamic Memory Allocation

➤ realloc():

- General format for memory allocation using realloc() is as follows:

```
void *realloc(void *block, size);
```

- * where,

- Block: Points to a memory block previously obtained by calling malloc(), calloc(), or realloc()
- Size: New size for allocated block.

January 18, 2016 | Proprietary and Confidential | 56 |



Dynamic Memory Allocation:

General format for memory allocation using realloc() is as shown in the slide. realloc() returns a pointer to the new storage and NULL if it is not possible to resize the object, in which case the object (*block) remains unchanged. The new size may be larger or smaller than the original size.

If the new size is larger, then the original contents are preserved and the remaining space is uninitialized;

If the new size is smaller, then the contents are unchanged up to the new size.

The function realloc() works like malloc() for the specified size if block is a null pointer.

Concept of Dynamic Memory Allocation

➤ realloc() (contd.):

- For example:
 - * Suppose the following is true.

```
char *cp;
cp=(char *)malloc(sizeof("computer"));
strcpy(cp,"computer");
cp=(char *)realloc(cp,sizeof("compute"));
```

- * Then cp points to an array of 9 characters containing string "computer".

Concept of Dynamic Memory Allocation

➤ realloc() (contd.):

- For example (contd.):

- * The following function call, discards the trailing '\0' and makes cp point to an array of 8 characters containing the string "compute".

```
cp=(char *)realloc(cp,sizeof("compute"));
```

- * Whereas the following call, makes cp point to an array of 16 characters.

```
cp=(char *)realloc(cp,sizeof("computerization"));
```

Concept of Dynamic Memory Allocation

➤ De-allocation Function:

- The function prototype is as follows:

```
free(ptr);
```

- It de-allocates memory which was allocated by malloc(), calloc() and realloc(). ptr is the pointer returned by allocation function.

January 18, 2016 | Proprietary and Confidential | 10 / 10



Dynamic Memory Allocation:

Deallocation Function:

free():

The memory allocated by the malloc(), calloc, or realloc() function is not destroyed automatically. It has to be cleared by using the function free().

```
free(ptr);
```

where:

ptr is a pointer variable to which memory was allocated, free (ptr) clears the memory to which ptr points to.

6.9: Command Line Arguments

Concept of Command Line Arguments

- Parameters or Values can be passed to a program from the command line which are received and processed in the main function.
- Since the arguments are passed from the command line, they are called as "command line arguments".
- They are used frequently to create command files.
- All commands on the Unix Operating System use this concept.

Concept of Command Line Arguments

➤ For example:

```
C:\>CommLineTest.exe arg1 arg2 arg3...
```

- where,
 - * CommLineTest.exe: It is the executable file of the program.
 - * arg1, arg2, arg3... : They are the actual parameters for the program

Concept of Command Line Arguments

- Two built in formal parameters are used to accept parameters in main.

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

— where,

- * argc : contains number of command line arguments. (type-int)
- * argv : A pointer to an array of strings where each string represents a token of the arguments passed. It is a character array of pointers

Concept of Command Line Arguments

➤ For example:

```
C:\>Tokens.exe abc 10 xyz
```

- The value of argc will be 4.
- The contents of argv will be as follows:
 - * argv[0] : "Tokens.exe"
 - * argv[1] : "abc"
 - * argv[2] : "10"
 - * argv[3] : "xyz"

6.10: Common Best Practices

Pointers-functions

> Avoid returning the address of local variable from function.

- Variables that are local to a function go on “the stack”. When the function returns/ends, all the data on the stack is popped off (discarded). So if you return a pointer to it, you are returning a pointer to a variable that no longer exists (returning an address that is no longer valid) and may be overwritten / garbage.

Bad Practice

```
char *foo()
{
    static temp[80] = {NULL};
    //do something with temp
    return temp;
} //returning local variable
```

January 18, 2016 | Proprietary and Confidential | 8.4 +

 Capgemini
INVESTING IN YOUR FUTURE

Pointers-functions:

There are basically two types of memory in C. Variables that are local to a function go on “the stack”. When the function returns/ends, all the data on the stack is popped off (discarded). So, if you return a pointer to a bit of local memory (like your local char *), you are returning an address that is no longer valid and may be overwritten/garbage.

The other type of memory is “the heap”. Memory allocated on the heap remains allocated until it is explicitly freed. So, you can allocate some memory, put your value(s) in it, and return a completely valid pointer to that address and there's no problem.

To allocate a chunk of memory on the heap, you use the malloc() function from stdlib.h:

```
char *pStr;
pStr = malloc(sizeof(char) * 100);
```

The argument to malloc() is the size of the memory chunk you want to allocate. In this case, I am allocating enough memory for 100 chars [sizeof(char) returns the amount of memory used by one char].

However, once you have allocated a chunk of memory you are responsible for it. If I just reassign pStr to something else, I will not have any reference to that memory anymore. There is no such thing as a garbage collector in C, so that memory stays allocated, even though I cannot use it because I have no reference to it. This is called a memory leak. Do not do it. You have to make sure when you are done with a chunk of memory that you

```
free() it (also in stdlib.h):
free(pStr);
```

Pointers-functions (contd.):

That is basically it. Yeah, it is a lot more complicated than just declaring it static. However, THIS is what C programming is. Memory management is a huge part of it and if you do not know how to do it right, you can't code in C.

"static" local variable is essentially a global variable, except that you can only use its reference while inside the function that declares it. This is why the memory address remains valid when you use static. The reason it is generally the wrong way to do it, is that if the function is called again before you have used the value you want, the value could change. Usually you can probably get away with it but it can cause bad things to happen, and if you are scared of allocating your own memory correctly you should not be coding in C.

Pointers-functions

- This is not a good practice as this will give warning at compile time and error at run-time.
- If you want to return a variable from a function, you either need to make the variable static which means it doesn't get deleted once the function ends or you need to allocate memory for the variable (using malloc())

Using MALLOC

```
char*returnMessage(char*message)
{
    char *msg = (char*)
    malloc(sizeof(char)* 100);
    char msgTest[] = "Hello how are
    you";
    strcpy(msg, msgTest,
    sizeof(msgTest));
    return msg;
}
```

Using STATIC

```
char *foo()
{
    static temp[80] =
    {NULL};
    //do something
    //with temp
    return temp;
}
```

January 18, 2016 | Proprietary and Confidential | 88 / 4



Pointers-malloc

- **malloc (0) return may return a null pointer or a pointer to 0 byte.**
 - The behavior is implementation dependent.
 - Lesson to be learnt is that your code must take care not call malloc (0) or be prepared for the possibility of a null return;
- **Be careful while the assignment of malloc :**

```
char *p  
*p = malloc(10); //bad practice  
p = malloc(10); //good practice
```

Pointers-strings

- Prefer to use pointer strings instead array of character as string manipulation becomes easy.

```
char str1[]="hello";
char *s ="hello";
str1 = 'Bye';//not allowed
s = "bye"; //allowed
```

```
char str1[]="hello";
char str2[10];
char *s ="good morning";
char *p;
str2 = str1;//not allowed
p = s; //allowed
```

Summary

➤ In this lesson, you have learnt:

- The address of a data item is called a “**pointer**” to the data item and a variable that holds an address is called a “**pointer variable**”.
- The address of (&) operator, when used as a prefix to the variable name, gives the address of that variable.
- The relational comparisons <, <=, >, >= are permitted between pointers of the same type and the result depends on the relative location of the two data items pointed to.



Review Questions

- Question 1: The data item can be accessed if we know the ___ of the first memory cell.
- Question 2: Null Pointer is same as un initialized pointer.
 - True/False
- Question 3: In pnum = &num, pnum is called as: ___.



Review – Match the Following

1. int p(char *a[]);

2. int (*P)(char *a);

3. int *p(void);

4. int *p;

1. p is a function that returns pointer to integer, argument is void

2. p is pointer to integer.

3. P is a pointer to a function that returns integer, argument is char pointer

4. P is a function that returns integer, accepts array of pointers to character as argument.



Lab

➤ Lab 5



January 18, 2016 | Proprietary and Confidential | • 71 •

 Capgemini
INNOVATING THROUGHOUT THE WORLD

Add the notes here.

C Programming

Lesson 6: Structures

January 18, 2018 Proprietary and Confidential © 2018 Capgemini

 Capgemini
CONNECTING BUSINESS AND TECHNOLOGY

Table of Contents

- **Structures:**
 - Basics of Structures
 - Nested Structures
 - Structures and Arrays
 - Structures and Pointers
 - Structures and Functions
 - Unions
 - Difference between union and structure
 - Typedef statement
 - Enumerated datatype



January 18, 2016 Proprietary and Confidential 4 / 2

 Capgemini
CONNECT. INSPIRE. DELIVER.

Lesson Objectives

➤ In this lesson, you will learn:

- The concept of structure declaration and definition
- The method to process structure variables using pointers
- The method to access the structure members



January 18, 2016 Proprietary and Confidential 43



Lesson Objectives

➤ In this lesson, you will cover the following topics:

- Structures:
 - Basics of Structures, Declaration
 - Structure Variables, tag, and initialization
 - Accessing Structure Members
- Nested Structures
- Arrays of Structures
- Pointers to Structures, Accessing members with structure pointers
- Allocating memory for a pointer to a structure
- Structures as Function Arguments and Function Values



January 18, 2016 Proprietary and Confidential - 4 -



7.1: Arrays of Structures

Concept of Arrays of Structures

- Array contains individual structures as its elements
- They are commonly used when a large number of similar records are required to be processed together
- For example:
 - The data of motor containing 1000 parts can be organized in an array of structure as follows:

```
struct item motor[1000];
```

- It declares motor to be an array containing 1000 elements of the type struct item.

January 18, 2016 Proprietary and Confidential © Capgemini

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Add the notes here.

Arrays of Structures – Sample Code

Let us see a sample code on “arrays of structures”

```
/* Example- An array of structures */
#include<stdio.h>
void main(void)
{
    struct book
    {
        char name[15];
        int pages;
        float price;
    };
    struct book novel[10];
    int index;
    printf("\n Enter name, pages and price of the book\n");
    /* accessing elements of array of structures */
```

January 18, 2016 Proprietary and Confidential 48



Add the notes here.

Arrays of Structures – Sample Code (contd.)

```
for(index=0;index<9;index++)
{
    scanf("%s%d",novel[index].name,&novel[index].pages);
    scanf("%f",&novel[index].price);
    printf("\n");
}
printf("\n Name, Pages and Price of the book :\n");
for(index=0;index<9;index++)
{
    printf("%s %d",novel[index].name,novel[index].pages);
    printf("%f",novel[index].price);
}
```

January 18, 2016 Proprietary and Confidential - 9 -



7.2: Arrays within Structures

Concept of Arrays within Structures

➤ A structure may contain arrays as members

- They are used when a string needs to be included in a structure.
- For example: The structure date can be expanded to include the names of the day of the week and month as shown below:

```
struct date
{
    char weekday[10];
    int day;
    int month;
    char monthname[10];
    int year;
};
```

January 18, 2016 Proprietary and Confidential 48 /

 Capgemini
CONNECTING BUSINESS AND TECHNOLOGY

Concept of Arrays within Structures

- A structure variable today can be declared and initialized as shown below:

```
struct date today={"Sunday",1,10,"November",2011};
```

- An element of an array contained in a structure can be accessed using the dot and array subscript operators

```
printf("%c",today.monthname[2]);
```

7.3: Pointer to Structures

Concept of Pointer to Structures

- The beginning address of a structure can be accessed by the address of (&) operator.
- It is mainly used to create complex data structures such as Linked lists, trees, graphs, and so on
- Pointer variables holding address of structure are called Structure Pointers

➤ For example: Consider the following declaration:

```
struct date today,*ptrndate;
```

- It declares today to be a variable of type struct date, and ptrndate to be a pointer to a struct date variable.

January 18, 2016 Proprietary and Confidential • 10 / 10

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Accessing Pointer members in Structures

- Consider the following structure declaration:

```
struct account
{
    int acct_no;
    char acct_type;
    char name[20];
    float balance;
    struct date lastpayment;
};

struct account customer,*pc;
```

- In this example, customer is a structure variable of type account, and pc is a pointer variable whose object is a structure of type account.

January 18, 2016 Proprietary and Confidential 4 / 10



Add the notes here.

Accessing Pointer members in Structures

- The pointer variable pc can now be used to access the member variables of customer using the dot operator as:
 - (*pc).acct_no;
 - (*pc).acct_type;
 - (*pc).name;
- The parentheses are necessary because the dot operator (.) has higher precedence than that of the dereferencing operator(*) .

Accessing Pointer members in Structures

- The members can also be accessed by using a special operator called structure pointer or arrow operator (`->`).
- The general form for the use of the operator `->` is as shown below:
 - `pointer_name->member_name;`
- Thus,

```
if pc=&customer;  
pc->balance=(*pc).balance=customer.balance
```

— where, balance is member of structure customer

7.4: Structures and Functions

Structures as Function Arguments

- C provides three methods of passing structures to a function:
- Passing Structure Member to Function
- Passing Entire Structure to Function
- Passing Structure Pointers to Functions

January 18, 2016 Proprietary and Confidential • 144 •

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

#1: Passing Structure Member to Function

- This method involves supplying structure members as the arguments in a function call.
- These arguments are then treated as separate non-structure values, unless they themselves are structures.
- **Disadvantage:**
 - The relationship between the member variables encapsulated in a structure is lost in the called function.
 - It should only be used if a few structure members need to be passed to the called function.

Demo - Passing Structure Member to function

- Demo struct function.c program



 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

January 18, 2016 Proprietary and Confidential • 148 •

Sample Code

```
/* Example- structure member as function arguments */
#include <stdio.h>
#define CURRENT_YEAR 2011
typedef struct
{
    char name[20];
    struct date
    {
        int day,month,year;
    }birthday;
    float salary;
} emprec;
```

January 18, 2014 Proprietary and Confidential © 2012



Sample Code (Contd..)

```
/* give increments to employees if age is greater than 30*/
float increment(float sal,int year,int inc)
{
    if(CURRENT_YEAR - year > 30)
        sal+= inc;
    return(sal);
}
void main(void)
{
    int n=500;
    emprec per={"Rohit Tamhane",5,9,1979,4000.50};
    printf(" *** Employee Details ***\n");
    printf("Name :%s \n",per.name);
```

January 18, 2016Proprietary and Confidential

Sample Code (Contd..)

```
printf("Birthdate:%d:%d:%d\n",per.birthday.day,  
      per.birthday.month, per.birthday.year);  
printf("Salary :%6.2f \n\n",per.salary);  
per.salary=increment(per.salary,per.birthday.year,n);  
printf(" *** Employee Details *** \n");  
printf("Name :%s \n",per.name);  
printf("Birthdate: %d:%d:%d \n",per.birthday.day,  
      per.birthday.month, per.birthday.year);  
printf("Salary :%6.2f \n",per.salary);  
}
```

January 18, 2016 Proprietary and Confidential - 108 -



#2: Passing Entire Structure to Function

- This method involves passing the complete structure to a function by simply providing the name of the structure variable as the argument in the function call.
- The corresponding parameter in the called function must be of the same structure type.

January 18, 2016 Proprietary and Confidential - 2/31



Structures and Functions:

#2: Passing Entire Structure to Function:

The second method involves passing the complete structure to a function by simply providing the name of the structure variable as the argument in the function call. The corresponding parameter in the called function must be of the same structure type.

To illustrate this method, an example is given on the further slide.

Demo on passing Entire Structure to Function

Demo on Struct_variable_asparameter.c program



 Capgemini
CONNECT. INSPIRE. DELIVER.

January 18, 2016 Proprietary and Confidential - 2 of 1

Sample Code

```
/* Example- Entire structure as function arguments */
# include<stdio.h>
struct book
{
    char name[20];
    char author[10];
    int pages;
};
void main(void)
{
    void display(struct book);
    struct book tech_book={"Programming in C", "Stephen", 300};
    display(tech_book);
}
```

January 18, 2016 Proprietary and Confidential - 13 / 1

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Structures and Functions:

#2: Passing Entire Structure to Function (contd.):

Sample Code:

Structure book has made global by defining it outside main(), so all the functions can access the structure book.

When a structure variable b1 is passed directly as an argument to the function, it is passed by value like an ordinary variable.

Sample Code contd..

```
void display(struct book tech_book)
{
    printf("Name :%s\n Author :%s\nPages :%d\n", tech_book.name,
           tech_book.author,tech_book.pages);
}
```

Output :
Name: Programming in C
Author: Stephen
Pages: 300

January 18, 2016 Proprietary and Confidential - 13 -



#3: Passing Structure Pointers to Function

- This method involves passing pointers to the structure variables as the function arguments.
- In the situations where, more than one member variable is computed in the function, pointers to structures are used.
- If the pointer to a structure is passed as an argument to a function, then any changes that are made in the function are visible in the caller.

January 18, 2016 Proprietary and Confidential | 234 |



Structures and Functions:

#3: Passing Structure Pointers to Functions:

The third method involves passing pointers to the structure variables as the function arguments. In the situations where, more than one member variable is computed in the function, pointers to structures are used.

If the pointer to a structure is passed as an argument to a function, then any change that are made in the function are visible in the caller.

Demo - Passing Structure Pointer to Function



January 18, 2016 Proprietary and Confidential © 2016

Sample Code

```
# include <stdio.h>
# include "str2.h"
# define CURRENT_YEAR 2011
void increment(emprec *x)
{
    if(CURRENT_YEAR - x->birthday.year > 30)
        x->salary += 500;
}
void main(void)
{
    emprec per={"Khan",27,10,62,5500};
    printf(" *** Employee Details ***\n");
}
```

January 08, 2014 Proprietary and Confidential • 2 of 1



Sample Code

```
printf("Name :%s \n",per.name);
printf("Birthdate: %d:%d:%d \n", per.birthday.day,
       per.birthday.month,per.birthday.year);
printf("Salary :%6.2f\n",per.salary);
/* give increments to employees if age is greater than 30 */
increment(&per);
printf(" *** Employee Details ***\n");
printf("Name :%s \n",per.name);
printf("Birthdate: %d:%d:%d \n", per.birthday.day,
       per.birthday.month,per.birthday.year);
printf("Salary :%6.2f\n",per.salary);
}
```

January 18, 2016 Proprietary and Confidential - 10 / 7



Structures as Function Values

- Structures can be returned from functions just as variables of any other type.
- Instead of accepting a pointer to a structure, it can construct a structure by itself and return this structure variable.

Demo on Structure as Function Values

- Demo on struct_asReturn_type.c program



Sample Code

```
/* Example- structures and functions */
# include<stdio.h>
struct time
{
    int min,hr,sec;
};
void main(void)
{
    struct time time_udt(struct time);
    struct time or_time,nx_time;
    printf("Enter time(hh:mm:ss):");
    scanf("%d:%d:%d",&or_time.hr,&or_time.min,
        &or_time.sec);
```

January 18, 2016 Proprietary and Confidential - 2/81



Structures and Functions:

Structures as Function Values:

The address of the per structure variable is passed to increment().

This method becomes particularly attractive when large structures have to be passed as function arguments because it avoids copying overhead.

Sample Code

```
nx_time = time_udt(or_time);
printf("Updted time is %2d:%2d:%2d\n", nx_time.hr,
       nx_time.min,nx_time.sec);
}
struct time time_udt(struct time now)
{
    struct time new_time;
    new_time=now;
    ++new_time.sec;
    if(new_time.sec==60)
    {
        new_time.sec=0;
        ++new_time.min;
```

January 18, 2016 Proprietary and Confidential - 2 of 1

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Structures and Functions:

Structures as Function Values:

A structure or_time is passed as an argument to the function time_udt().

The function returns a value of type struct time.

The program prompts the user for the current time, updates the time by one second and prints it.

Sample Code

```
if(new_time.min==60)
{
    new_time.min=0;
    ++new_time.hr;
    if(new_time.hr==24)
        new_time.hr=0;
}
return(new_time);
```

January 18, 2016 Proprietary and Confidential © 2016



Structures and Functions:

Structures as Function Values:

A structure or _time is passed as an argument to the function time_udt().

The function returns a value of type struct time.

The program prompts the user for the current time, updates the time by one second and prints it.

Common Best Practices in C Programming



7.5: Common Best Practices
Structures

➤ The following program works correctly, but it dumps core after it finishes. So be careful while structure declaration.

```
struct list
{ char *item; struct list *next; } //missing semicolon

/* Here is the main program. *
main(argc, argv) ...
```

— A missing semicolon causes the compiler to believe that main returns a structure. Since struct-valued functions are usually implemented by adding a hidden return pointer, the generated code for main() tries to accept three arguments, although only two are passed.

January 18, 2016 Proprietary and Confidential - 234 -

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Structures

➤ Compiler will leave an unnamed, unused hole between members of structure for appropriate alignment.

- Use #pragma directive for removing the padding effect .
Use the following code:
`#pragma pack(push,1)`

| | |
|--|---|
| <pre>struct emp { char name[3]; char role[2]; double salary; }; Output: 16 byte</pre> | <pre>struct emp { int empid; char band; char gender; }; Output: 8 byte</pre> |
|--|---|

January 18, 2016 Proprietary and Confidential © Capgemini

 Capgemini
CONNECT. INSPIRE. DELIVER.

Structures:

Many machines access values in memory most efficiently when the values are appropriately aligned. (For example, on a byte-addressed machine, short ints of size 2 might best be placed at even addresses, and long ints of size 4 at addresses which are a multiple of 4.) Some machines cannot perform unaligned accesses at all, and require that all data be appropriately aligned.

Therefore, if you declare a structure like

```
struct { char c;
int i; }
```

the compiler will usually leave an unnamed, unused hole between the char and int fields, to ensure that the int field is properly aligned.

Suppose you have this structure:

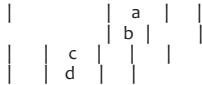
```
struct {
    char a[3];
    short int b;
    long int c;
    char d[3];
};
```

Now, you might think that it ought to be possible to pack this structure into memory as shown below:

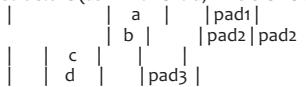
```
|   | a |   | b |
|   |   | b |   | c |   |
| c |   | d |   |
```

Structures (contd.):

However, it is much easier on the processor if the compiler arranges it as shown below:



In the “packed” version, notice how it is at least a little bit hard for you and me to see how the b and c fields wrap around? In a nutshell, it is hard for the processor, as well. Therefore, most compilers will “pad” the structure (as if with extra, invisible fields) as shown below:



If you are worried about wasted space, you can minimize the effects of padding by ordering the members of a structure based on their base types, from largest to smallest.

Structure Padding: Most processors require specific memory alignment on variables certain types. Normally, the minimum alignment is the size of the basic type in question, for instance a common char variables can be byte aligned and appear at any byte boundary.

short (2 byte) variables must be 2 byte aligned, they can appear at any even byte boundary. This means that 0x10004567 is not a valid location for a short variable but 0x10004566 is a valid location.

long (4 byte) variables must be 4 byte aligned, they can only appear at byte boundaries that are a multiple of 4 bytes. This means that 0x10004566 is not a valid location for a long variable but 0x10004568 is a valid location.

Structure padding occurs because the members of the structure must appear at the correct byte boundary. To achieve this, the compiler puts in padding bytes (or bits if bit fields are in use) so that the structure members appear in the correct location. Additionally, the size of the structure must be such that in an array of the structures all the structures are correctly aligned in memory. So there may be padding bytes at the end of the structure too.

```
struct example { char c1;
short s1;
char c2;
long l1;
char c3; }
```

In this structure:

c1 can appear at any byte boundary. However, s1 must appear at a 2 byte boundary so there is a padding byte between c1 and s1.
c2 can then appear in the available memory location. However, l1 must be at a 4 byte boundary so there are 3 padding bytes between c2 and l1.
c3 can then appear in the available memory location. However, the structure contains a long member. Hence the structure must be 4 byte aligned and must be a multiple of 4 bytes in size.

Structures (contd.):

Therefore there are 3 padding bytes at the end of the structure. It would appear in memory in the following order:

```
c1  
padding byte  
s1 byte 1  
s1 byte 2  
c2  
padding byte  
padding byte  
padding byte  
l1 byte 1  
l1 byte 2  
l1 byte 3  
l1 byte 4  
c3  
padding byte  
padding byte  
padding byte
```

The structure would be 16 bytes long.

re-written like this struct example, { long l1; short s1; char c1; char c2; char c3; }

Then l1 appears at the correct byte alignment, s1 will be correctly aligned so no need for padding between l1 and s1. c1, c2, c3 can appear at any location. The structure must be a multiple of 4 bytes in size since it contains a long, so 3 padding bytes appear after c3. It appears in memory in the following order:

```
l1 byte 1  
l1 byte 2  
l1 byte 3  
l1 byte 4  
s1 byte 1  
s1 byte 2  
c1  
c2  
c3  
padding byte  
padding byte  
padding byte
```

It is only 12 bytes long. I should point out that structure packing is platform and compiler (and in some cases compiler switch) dependent.

Structures

- Structures cannot be compared as byte-by-byte comparison can be invalidated by random bits present in unused “holes” in the structure
- If you need to compare two structures, you will have to write your own function to do so, field by field.
- Size of report a larger size than expect for a structure type due to padding effect.

January 18, 2016 Proprietary and Confidential - 2/8

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

Structures (contd.):

There is not a good way for a compiler to implement structure comparison (that is, to support the == operator for structures) which is consistent with C's low-level flavor. A simple byte-by-byte comparison can founder on random bits present in unused “holes” in the structure (such padding is used to keep the alignment of later fields correct). A field-by-field comparison might require unacceptable amounts of repetitive code for large structures. Any compiler-generated comparison cannot be expected to compare pointer fields appropriately in all cases: for example, it is often appropriate to compare char * fields with strcmp rather than == .

If you need to compare two structures, you will have to write your own function to do so, field by field.

Checking a string to see if it matches a particular value.

```
char *string; ... if(string == "value")  
{ /* string matches "value" */ ... }
```

This code is not working.

Strings in C are represented as arrays of characters, and C never manipulates (assigns, compares, etc.) arrays as a whole. The == operator in the code fragment above compares two pointers - the value of the pointer variable string and a pointer to the string literal “value” – to see if they are equal, that is, if they point to the same place. They probably do not, so the comparison never succeeds.

To compare two strings, you generally use the library function strcmp:

```
if(strcmp(string, "value") == 0)  
{ /* string matches "value" */ ... }
```

Unions

```
union record {  
    char *name;  
    int refcount : 4;  
    unsigned dirty : 1; //bitfield  
};
```

- Remember that the colon notation for specifying the size of a field in bits is valid only in structures (as in union); you cannot use this mechanism to specify the size arbitrary variables.

January 18, 2016 Proprietary and Confidential - 2/91



Unions:

```
struct record {  
    char *name;  
    int refcount : 4;  
    unsigned dirty : 1;  
};
```

This is bitfield; the number gives the exact size of the field in bits.
Bitfields are used primarily in structures having several binary flags or other small fields. It saves space. Remember that the colon notation for specifying the size of a field in bits is valid only in structures (as in union). You cannot use this mechanism to specify the size arbitrary variables.

Typedef

- The type defined with a **typedef** is exactly like its counterpart as far as its type declaring power is concerned. However, it cannot be modified like its counterpart
 - `typedef int MYINT`
- Now you can declare an int variable either with:
 - `int a;` or `MYINT a;`
- However, you cannot declare an unsigned int (using the **unsigned** modifier) with `unsigned MYINT a;`; although `unsigned int a;` would be perfectly acceptable

Typedef

- **typedefs can correctly encode pointer types. Whereas #DEFINES are just replacements done by the preprocessor**

- **For example,**

```
typedef char * String_t;  
#define String_d char *  
String_t name, content; String_d subject, grade;  
name, content, and subject are all declared as char *, but grade is declared as a char,  
which is probably not the intention.
```

Typedef

- It is extremely useful to make code more compact and easier to read
typedef also allows to declare arrays,

```
typedef char char_arr[];  
char_arr my_arr = "Hello World!\n";
```

- This is equal to

```
char my_arr[] = "Hello World!\n";
```

Summary

➤ In this lesson, you have learnt:

- Individual members cannot be initialized inside the structure declaration.
- Structures may be passed as function arguments and functions may return structures.
- Memory from the heap needs to be allocated for a pointer to a structure if you want to store some data. This is done by using malloc() function.
- Unions like structures, contain members whose individual data types may differ from one another.



January 18, 2016 Proprietary and Confidential - 4/41

 Capgemini
CONNECT. INSPIRE. DELIVER.

Add the notes here.

Review Question

- Question 1: Data items that make up a structure can be of different types.
 - True / False
- Question 2: The structure variables can be accessed using operator __
- Question 3: It is not possible to create an array of pointer to structures.
 - True / False



January 18, 2016 Proprietary and Confidential • 444 •

Add the notes here.

Review Question: Match the Following

1. `->`

2. `.`

3. `=`

1. Operator used to access structure element through structure variable

2. Operator used to assign elements of one structure variable to another structure variable.

3. Operator used to access structure element through pointer to structure



Lab Session

➤ Lab 6



January 18, 2016 Proprietary and Confidential - Page 1

 Capgemini
CONNECTING BUSINESS & TECHNOLOGY

C Programming

Lesson 7: File Handling

January 28, 2016 | Proprietary and Confidential | v 1.0



Table of Contents

➤ File Handling:

- Creation and Access
- Text files and Data Files
- Random access and Error Handling



Lesson Objectives

- In this lesson, you will learn the method to process files and using functions to do it.

Lesson Coverage

➤ In this lesson, you will cover:

- Linked List: Read from file, build, save on file
- Error Handling Functions: perror(), perror()
- File Positioning: fseek(), ftell(), rewind()
- File Handling Overview:
 - Unformatted high-level disk I/O functions: fopen(), fclose()
 - Character Input/Output in files: getc(), putc()
 - String (line) Input/Output in Files: fgets(), fputs()
 - Formatted high-level disk I/O functions: fscanf(), fprintf()
 - Direct Input/Output: fread(), fwrite(), feof()

Sample Code

```
/*This program takes the contents of a text file and copies into another text file,
character by character */
void main(void) {
    FILE *fileptr_read,*fileptr_write;
    char content;
    fileptr_read=fopen("pri.c","r"); /* open file in read mode */
    if(fileptr_read==NULL)
    {
        puts("Cannot open source file");
        exit(0);
    }
    fileptr_write=fopen("pr2.c","w"); /*open file in write mode*/
```

January 05, 2016 | Proprietary and Confidential | - 3 -



Note:

Now we have seen functions fopen(), fclose(),getc(),putc(), and so on.
As a practical use of the above functions we can copy the contents of one file
into another.

Sample Code

```
if(fileptr_write==NULL) {  
    puts("Cannot open target file");  
    fclose(fileptr_read);  
    exit(0);  
}  
while(1) {  
    content=getc(fileptr_read);  
    if(content==EOF)  
        break;  
    putc(content,fileptr_write);  
}  
fclose(fileptr_read);  
fclose(fileptr_write);  
}
```



8.1: String Input/Output in Files

Functions Used

- They are used to read and write a string of characters from / to a file.
- Functions used are as follows:
 - fgets():
 - It reads a line of text from a file.
 - The general format is as shown below:

```
char *fgets( char *s, int n, FILE *fp);
```

- It reads character from the stream fp into the character array 's' until a newline character is read, end-of-file is reached, or until n-1 characters are read.

 Capgemini
TEKsystems TEKsystems TEKsystems

January 06, 2016 | Proprietary and Confidential | - 2 -

String Input/Output in Files:

We have seen putc() and getc() functions as character I/O in files. However, reading or writing strings of characters from and to files is as easy as reading and writing individual characters.

The functions fgets() and fputs() can be used for string I/O.

The routine fgets() is used to read a line of text from a file.

The general format of fgets() is as shown below:

```
char *fgets( char *s, int n, FILE *fp);
```

The function fgets() reads character from the stream fp into the character array 's' until a newline character is read, end-of-file is reached, or n-1 characters have been read. It then appends the terminating null character after the last character read and returns 's'. If end-of-file occurs before reading any character or an error occurs during input, then fgets() returns NULL.

The routine fputs() is used to write a line of text from a file.

The general format of fputs() is as shown below:

```
int fputs(const char *s, FILE *fp);
```

The function fputs() writes to the stream fp except the terminating null character of string 's'. It returns EOF, in case an error occurs during output otherwise it returns a nonnegative value.

Functions Used

➤ Functions used are as follows (contd.):

- fputs():
 - It writes a line of text from a file.
 - The general format is as shown below:

```
int fputs(const char *s, FILE *fp);
```

- It writes to the stream fp except the terminating null character of string 's'.
- It returns the non-negative value, on success. It returns EOF, in case of an error.



Add the notes here.

Demo on String I/O in file

- Demo fputs_fgets.c



Add the notes here.

Sample Code

➤ Let us see a program using String Input/Output functions:

```
/* Receives strings from user and writes them to file. */

void main(void)
{
    FILE *fp;
    char data[80];
    fp=fopen("test.txt","w");
    if(fp==NULL)
    {
        puts("Cannot open file");
        exit(0);
    }
}
```

January 05, 2016 | Proprietary and Confidential | + 10 s



Note:

In the program on the slide, we have set up a character array to receive the string. The fputs() function then writes the contents of the array to the disk. Since the fputs() function does not automatically add a newline character, we have done this explicitly.

Sample Code

```
printf("Enter few lines of text \n");
while(strlen(gets(data))>0)
{
    fputs(data,fp);
    fputs("\n",fp);
}
fclose(fp);
```

January 05, 2016 | Proprietary and Confidential | 11 of 11



Add the notes here.

8.2: Direct Input/Output Functions

Functions Used

- They provide facilities to read and write a certain number of data items of specified size.
- Functions used are as follows:
 - int fread(ptr,size,nitems,fp) :
 - It reads into buffer ptr nitems of data items of size size from the stream fp.
 - It returns the number of items read, on success.
 - It returns EOF, if any error occurs.
 - int fwrite(ptr,size,nitems,fp) :
 - It appends at the most nitems item of data of size size in the file, from the array which ptr points to.
 - It returns the number of items written, on success.
 - It returns EOF if an error is encountered.

January 05, 2016 | Proprietary and Confidential | - 12 -

 Capgemini
CONSULTING | IT | MANAGEMENT | BUSINESS SERVICES

Direct Input/Output Functions:

Direct input/output functions provide facilities to read and write a certain number of data items of specified size. The functions are fread() and fwrite(). The general format of fread() is as shown below:

```
int fread(ptr,size,nitems,fp)
char *ptr;
int size,nitems;
FILE *fp;
```

The function fread() reads into array ptr upto nitems data items of size size from the stream fp and returns the number of items read.

If an error is encountered fread() returns EOF otherwise returns the number of items read.

The file position indicator is advanced by the number of characters successfully read.

For example: Assuming 4-byte integers, consider the following statement:

```
rchar=fread(buf,sizeof(int),20,input);
```

It reads 80 characters from input into the array buf and assigns 80 to rchar, unless an error or end-of-file occurs.

8.3: Error Handling Functions

Usage of Error Handling Functions

Let us see some of the Error Handling Functions:

- `int feof(FILE *fp);`
 - It returns true (non-zero) if the end of the file pointed to by fp has been reached; otherwise it returns zero.
- `int ferror(FILE *fp);`
 - It returns a non-zero value if the error indicator is set for the stream fp; otherwise it returns zero.
- `void perror(const char *s);`
 - It writes the string 's' followed by a colon and a space to the standard error output stderr, and then an implementation-defined error message corresponding to the integer in errno, terminated by a newline character.

January 05, 2016 | Proprietary and Confidential | - 13 -

 Capgemini
CONSULTING. IT. INNOVATION. INSPIRATION.

Error Handling Functions:

The error handling functions provide facilities to test whether EOF returned by a function indicates an end-of-file or an error.

The function feof():

The buffered file system is designed to handle both text and binary files. Hence it is necessary that there should be some way other than the return value ofgetc() to determine that the end-of-file mark is also a valid integer value that can occur in a binary file.

The general format of feof() is as shown below:

```
int feof(FILE *fp);
```

where fp is a valid file pointer

The function feof() returns true (non-zero) if the end of the file pointed to by fp is reached, otherwise it returns zero.

The function ferror():

The general format of ferror() is as shown below:

```
int ferror(FILE *fp);
```

The function ferror() returns a non-zero value if the error indicator is set for the stream fp, and otherwise it returns zero.

Sample Code

```
FILE *fp,*fpr; char another='Y';
struct emp
{
    char name[40];
    int age;
    float bs;
};
struct emp e; fp=fopen("emp.dat","w");
if(fp==NULL)
{ puts("Cannot open file");
exit(0);
}
while(another=='Y')
{
    printf("\n enter name , age basic salary\n");
    scanf("%s%d%f",&e.name,&e.age,&e.bs);
    fwrite(&e,sizeof(e),1,fp);
```

January 05, 2016 | Proprietary and Confidential | + 14 +



Note:

The program in the slide receives records from keyboard, writes them to a file and also display them on the screen.

Sample Code

```
printf("Add another record (Y/N)");
fflush(stdin);another=getchar();
}
fclose(fp);
fpr=fopen("emp.dat","r");
if(fpr==NULL)
{
    puts("Cannot open file");
    exit(0);
}
while(fread(&e,sizeof(e),1,fpr)==1)
    printf("%s %d %f\n",e.name,e.age,e.bs);
fclose(fpr);
```



Add the notes here.

Functions Used

- `long ftell(FILE *fp);`
 - * It returns the current value of the file position indicator associated with fp.
- `void rewind(FILE *fp);`
 - * It resets the current value of the file position indicator associated with fp to the beginning of the file.
 - * It allows a program to read through a file more than once without having to close and open the file again.

January 05, 2016 | Proprietary and Confidential | + 10 +



Random Access Functions:

The Function `ftell()`:

The general format of `ftell()` is as follows:

`long ftell(FILE *fp);`

The function `ftell()` returns the current value of the file position indicator associated with fp.

The function `rewind()`:

The general format of `rewind()` is as follows:

`void rewind(FILE *fp);`

The function `rewind()` resets the current value of the file position indicator associated with fp to the beginning of the file.

The call `rewind(fp);` has the same effect as `void fseek(fp,0,SEEK_SET);`

The use of `rewind()` allows a program to read through a file more than once without having to close and open the file again.

Summary

➤ **In this lesson, you have learnt:**

- The getc() and putc() functions can be used for character I/O.
- The main() function takes two arguments called argv and argc.
- fread() and fwrite() functions provide facilities to read and write a certain number of data items of specified size.



Review Question

- Question 1: stderr, stdin and stdout are FILE pointers.
 - True/False
- Question 2: The structure of FILE is defined in ____ header file.
- Question 3: A File written in text mode can be read back in the binary mode.
 - True/False



Add the notes here.

Review Question: Match the Following

| | |
|--------------------------|---|
| 1. fseek(fp,n,SEEK_CUR) | 1. sets cursor back from current position by n bytes |
| 2. fseek(fp,-n,SEEK_CUR) | 2. sets cursor ahead from current position by n bytes |
| 3. fseek(fp,0,SEEK_END) | 3. sets cursor to the beginning of the file |
| 4. fseek(fp,0,SEEK_SET) | 4. sets cursor to the end of the file |



Add the notes here.

Lab Session

➤ Lab 7



January 05, 2016 | Proprietary and Confidential | + 30 s

 Capgemini
TENAGI LINGKARAN KERAS DAN KONSEP KONSEP

Add the notes here.

C Programming

Lesson 8: Preprocessor

January 18, 2018 Proprietary and Confidential 11



Lesson Objectives

- In this lesson, you will learn Preprocessor topics related to C Programming.



Lesson Coverage

➤ In this lesson, you will cover:

- Introduction to Preprocessor
- Macro substitution

9.1: Preprocessor

Introduction to Preprocessor

- Preprocessor is a program that processes the source text of a C program before it is passed to the compiler
- It can be an independent program or its functionality may be embedded in the compiler
- It offers a collection of special statements, called “preprocessor directives”

January 18, 2018 | Proprietary and Confidential | 44

 Capgemini
Digitizing Business Experience

Pre-processor:

The C pre-processor is exactly that its name implies. It is a collection of special statements, called directives. It can be an independent program or its functionality may be embedded in the compiler.

It is a program that processes the source text of a C program before the program is passed to the compiler.

It has four major functions:

- Macro replacement
- Conditional compilation
- File inclusion
- Error generation

Introduction to Preprocessor

➤ Preprocessor has four major functions:

- Macro replacement
- Conditional compilation
- File inclusion
- Error generation

9.2: Preprocessor Directives

Preprocessor Directives

- Preprocessor Directives begin with a # symbol
- Some of the preprocessor directives are as follows:
 - #define directive
 - #include directive
 - #undef directive
 - #error directive

Conditional compilation directives

- #if
- #ifdef
- #ifndef
- #else
- #endif

January 18, 2018 | Proprietary and Confidential | - 8 -

 Capgemini
ENTERPRISE LEARNING

Pre-processor:

Pre-processor Directives:

The C pre-processor offers several features called pre-processor directives. Each of these pre-processor directives begin with a # symbol.

We will learn the following pre-processor directives in this lesson:

```
#define directive  
#include directive  
#undef directive  
#error directive  
Conditional compilation directives
```

9.3: Macro Substitution

Concept of Macro Substitution

- During macro substitution, the preprocessor replaces every occurrence of a simple macro in the program text by a copy of the body of the macro
- The body of the macro may itself contain other macros
- It is achieved using the #define directive
- The general syntax is as shown below:

```
#define macro-name sequence-of-tokens
```

Note:

Macro Substitution is a very useful feature.

Concept of Macro Substitution

- The macro-name is associated with the sequence-of-tokens that appear from the first blank after the macro-name to the end of the file

— For example:

```
/* Associates macro name MIN with value -5 */  
#define MIN -5
```

Sample Code

➤ The program given below shows #define directive used to define operators

```
#include <stdio.h>
#define AND &&
#define OR ||
void main(void)
{long int salary=;
char gender='F';
if((salary>20000) AND (gender=='M' OR gender=='F'))
    printf("You have to pay Income Tax.....");
else
    printf("No Tax.....");}
```

January 18, 2018 | Proprietary and Confidential | - 9 -



Note:

The program given in the above slide shows #define directive used to define operators.

9.4: Parameterized Macros

Concept of Parameterized Macros

Parameterized Macros are macros with arguments

- For example:

```
#include <stdio.h>
#define AREA( r ) (3.14*r*r)
void main(void)
{
    float radius;
    printf("Enter the radius \t");
    scanf("%f",&radius);
    printf("\nArea of the circle is %f", AREA(radius));
}
```

January 16, 2018 | Proprietary and Confidential | 10 / 10

 Capgemini

Note:

The macros that we have used so far are called simple macros.
However, macros can have arguments, as well. They are also called as parameterized macros.

9.5: Nested Macros

Concept of Nested Macros

➤ In Nested Macros, one macro can be used in the definition of another macro, as shown below:

```
/* This program shows use of nesting of macros */
#include <stdio.h>
#define SQUARE(num) (num*num)
#define CUBE(num)      (SQUARE(num) * num)
void main(void)
{
    int no;
    printf("Enter the number ");
    scanf("%d",&no);
    printf("\nSquare of a number is %d",SQUARE(no));
    printf("\nCube of a number is %d",CUBE(no));
}
```

January 16, 2018 | Proprietary and Confidential | - 11 -

 Capgemini

Note:

We can also use one macro in the definition of another macro. That is to say that macro definitions may be nested.

For instance, consider the macro definition in the above slide.

9.6: Un-defining a Macro

Concept of Un-defining a Macro

➤ Un-defining a Macro is useful to restrict the definition only to a particular part of the program

- Syntax:

#undef identifier

- For example:

#undef SQUARE

January 18, 2018 Proprietary and Confidential - 12 -

 Capgemini

Un-defining a Macro:

A defined macro can be undefined, using the following statement:

#undef identifier

This is useful when we want to restrict the definition only to a particular part of the program.

In the program given below, macro-name SQUARE and CUBE can be undefined using the undef statement.

#undef SQUARE
#undef CUBE

9.7: File Inclusion

Concept of File Inclusion

- This preprocessor directive causes one file to be included in another
- It is useful to include the library file that contains common library functions or necessary macros in another file

January 10, 2018 | Proprietary and Confidential | - 12 -

 Capgemini
ENTERPRISE LEARNING

File Inclusion:

This pre-processor directive causes one file to be included in another. This feature is used in two cases.

If we have a very large program, it is good programming practice to keep different sections in separate files. These files are included at the beginning of main program file.

Many a times we need some functions or some macro definitions almost in all programs that we write. In such a case, commonly needed functions and macro definitions can be stored in a file and that file can be included wherever necessary.

Concept of File Inclusion

➤ Syntax:

```
#include <filename>
```

- It searches the specified file in default directory and includes, if it exists.
or

```
#include "filename"
```

- It searches the specified file in default directory and also in current directory and includes, if it exists

File Inclusion (contd.):

Two ways exist for writing #include statements. These are as shown below:

```
#include <filename>
```

```
#include "filename"
```

File Inclusion - Example

➤ **For example:**

- Suppose we have the following three files:

| | |
|--------------|----------------------------------|
| • function.c | contains some functions |
| • proto.h | contains prototypes of functions |
| • test.c | contains test functions |

- Then we can make use of a definition or function contained in any of these files by including them in the program as shown on the next slide

File Inclusion - Example

➤ For example (contd.):

```
#include <stdio.h>
#include "function.c"
#include "proto.h"
#include "test.c"
#define MAX 50
void main(void)
{
    /* Here the code in the above three files */
    /* is added to the main code */
    /* and the file is compiled */
}
```

9.8: Conditional Compilation

Concept of Conditional Compilation

- Conditional Compilation allows selective inclusion of lines of source text on the basis of a computed condition
- Conditional Compilation is performed using the preprocessor directives shown below:
 - #ifdef
 - #ifndef
 - #elif
 - #else
 - #endif

January 18, 2018 Proprietary and Confidential - 12 -

 Capgemini

Conditional Compilation:

Conditional compilation is performed using the pre-processor directives shown in the above slide.

We can have the compiler skip over, part of a source code by inserting the pre-processing commands #ifdef and #endif.

The general form is as shown below:

```
#ifdef macroname  
    statement 1;  
    statement 2 ;  
#else  
    statement 3 ;  
#endif
```

9.8: Conditional Compilation

Concept of Conditional Compilation

- Logical directive tests whether an identifier exists as a result of having been created in a previous #define directive.
- C Conditional Compilation takes the following form:

```
#if defined identifier
...
#endif
```
- If identifier is defined, statements between #if and #endif are included in the program code. If the identifier isn't defined, the statements between the #if and the #endif will be skipped in the absence of an identifier.
- with the help of #elif and #else these can be handled well. In fact, this tends to be used more frequently than the form you've just seen.
- #ifndef is used to check whether a particular symbol is defined.

January 18, 2018 | Proprietary and Confidential | 18 / 20

 Capgemini
ENTERPRISE LEARNING

Example : 1

```
#define USA 1
#define EUP 1
#include <stdio.h>
#if (defined (USA))
    #define currency_rate 46
#elif (defined (EUP))
    #define currency_rate 100
#else
    #define currency_rate 1
#endif
int main()
{
    int rs;
    rs = 10 * currency_rate;
    printf ("%d\n", rs);
    return 0;
}
```

Example : 2

```
#define USA 1
//#define EUP 1
#include <stdio.h>
#ifndef USA
    #define currency_rate 100
#endif
#endif EUP
    #define currency_rate 46
#endif
int main()
{
    int rs;
    rs = 10 * currency_rate;
    printf ("%d\n", rs);
    return 0;
}
```

9.9: Error Generation

Concept of Error Generation

- Error Generation is useful to display the user-defined error message on occurrence of an error
- The format of the directive is:
 - #error token sequence
- It causes the implementation to produce a diagnostic message containing the token sequence

Concept of Error Generation

➤ For example:

```
#ifndef PI  
    #error "PI NOT DEFINED"  
#endif
```

➤ If PI is not defined, pre-processor will print the error message “PI NOT DEFINED” and the compilation will stop

Library

- A library is a package of code that is meant to be reused by many programs
- There are two types of libraries: static libraries and dynamic libraries
 - There are two types of libraries: static libraries and dynamic libraries

January 18, 2018 | Proprietary and Confidential | - 21 -

 Capgemini
ENTERPRISE LEARNING

Library:

A library is a package of code that is meant to be reused by many programs. Some libraries may be split into multiple files and/or have multiple header files. Libraries are precompiled for several reasons.

First, since libraries rarely change, they do not need to be recompiled often. It would be a waste of time to recompile the library every time you wrote a program that used them.

Second, precompiled objects are in machine language. Hence it prevents people from accessing or changing the source code, which is important to businesses or people who do not want to make their source code available for intellectual property reasons.

There are two types of library:

- Static
- Dynamic

Static Library

- A static library (also known as an archive) consists of routines that are compiled and linked directly into your program
- When you compile a program that uses a static library, all the functionality of the static library becomes part of your executable
- On Windows, static libraries typically have a .lib extension. On Linux, static libraries typically have an .a (archive) extension

Static Library:

A static library (also known as an archive) consists of routines that are compiled and linked directly into your program. When you compile a program that uses a static library, all the functionality of the static library becomes part of your executable. On Windows, static libraries typically have a .lib extension, whereas on linux, static libraries typically have an .a (archive) extension.

One advantage of static libraries is that you only have to distribute the executable in order for users to run your program. Because the library becomes part of your program, this ensures that the right version of the library is always used with your program. Also, because static libraries become part of your program, you can use them just like functionality you've written for your own program.

On the downside, because a copy of the library becomes part of every executable that uses it, this can cause a lot of wasted space. Static libraries also can not be upgraded easy — to update the library, the entire executable needs to be replaced.

Dynamic Library

- A dynamic library (also called a shared library) consists of routines that are loaded into your application at run time
- When you compile a program that uses a dynamic library, the library does not become part of your executable — it remains as a separate unit
- On Windows, dynamic libraries typically have a .dll (dynamic link library) extension, whereas on Linux, dynamic libraries typically have a .so (shared object) extension

January 18, 2018 Proprietary and Confidential 133



Dynamic Library:

A dynamic library (also called a shared library) consists of routines that are loaded into your application at run time.

When you compile a program that uses a dynamic library, the library does not become part of your executable — it remains as a separate unit.

On Windows, dynamic libraries typically have a .dll (dynamic link library) extension, whereas on Linux, dynamic libraries typically have a .so (shared object) extension. One advantage of dynamic libraries is that many programs can share one copy, which saves space. Perhaps a bigger advantage is that the dynamic library can be upgraded to a newer version without replacing all of the executables that use it. One advantage of dynamic libraries is that many programs can share one copy, which saves space. Perhaps a bigger advantage is that the dynamic library can be upgraded to a newer version without replacing all of the executables that use it. Because dynamic libraries are not linked into your program, programs using dynamic libraries must explicitly load and interface with the dynamic library. This mechanism can be confusing, and makes interfacing with a dynamic library awkward. To make dynamic libraries easier to use, an import library can be used.

Lab

➤ Lab 8



Hands On

January 18, 2018 | Proprietary and Confidential | 134 |



Summary

- The preprocessor directive causes one file to be included in another.
- C provides a facility called type definition, which allows users to define new data types. They are equivalent to existing data types.



Review Questions

➤ **True or False?**

1. A macro must always be defined in capital letters.
2. Macro calls and function calls work exactly similar.
3. Every C program will contain AT LEAST ONE preprocessor directive.



Review Question: Match the Following

- | | |
|----------------------------|-------------|
| 1. Macro Replacement | 1. #include |
| 2. Conditional Compilation | 2. #error |
| 3. File Inclusion | 3. #undef |
| 4. Error Generation | 4. #define |



C Programming

Lesson 9: Algorithms

Table Of Contents

➤ Algorithms

- Algorithm analysis
- Comparisons of Searching and Sorting Algorithms
- Time vs Space Complexity
- Example C program to demonstrate time Complexity



Lesson Objectives

In this lesson, you will learn about:

- This course introduces the analysis and design of computer algorithms
- Various notations best, worst, average cases. Big Oh, small oh and theta notations
- Comparing all sorting and searching algorithms



9. Algorithms

Concept of Algorithms

- An Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An Algorithm is thus a sequence of computational steps that transform the input into the output. It is a tool for solving a well - specified computational problem.
- The analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them.
- Time Complexity vs Space Complexity

January 26, 2016 | Proprietary and Confidential | + 4 +



Most algorithms are designed to work with inputs of arbitrary length.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Concept of Algorithm analysis Contd...

- These estimates provide an insight into reasonable directions of search for efficient algorithms.
- In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input.
- Big O notation, Big-omega notation and Big-theta notation are used to the theoretical analysis .
- For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in $O(\log(n))$, colloquially "in logarithmic time".

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem.

Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

9.1 Cost Model

Cost model

- Time efficiency estimates depend on what one define to be a step.
- For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant.
- One must be careful here; for instance, some analyses count an addition of two numbers as one step.
- For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.
- Two cost models are
 - The uniform cost model
 - The logarithmic cost model

January 26, 2006 Proprietary and Confidential - 6 -

 Capgemini
CONSULTING TECHNOLOGY SERVICES

The uniform cost model, also called uniform-cost measurement (and similar variations), assigns a constant cost to every machine operation, regardless of the size of the numbers involved

The logarithmic cost model, also called logarithmic-cost measurement (and variations thereof), assigns a cost to every machine operation proportional to the number of bits involved

9.2 Best, worst and average case

Best, worst and average case

- Best, worst and average cases of a given algorithm express what the resource usage is at least, at most and on average, respectively.
- Usually the resource being considered is running time, i.e. time complexity, but it could also be memory or other resources.
- Average performance and worst-case performance are the most used in algorithm analysis. Less widely found is best-case performance, but it does have uses.
- For example, where the best cases of individual tasks are known, they can be used to improve the accuracy of an overall worst-case analysis.

January 18, 2016 | Proprietary and Confidential | +7+

 Capgemini
CONSULTING TECHNOLOGY SOLUTIONS

In real-time computing, the worst-case execution time is often of particular concern since it is important to know how much time might be needed in the worst case to guarantee that the algorithm will always finish on time.

The terms are used in other contexts; for example the worst- and best-case outcome of a planned-for epidemic, worst-case temperature to which an electronic circuit element is exposed, etc. Where components of specified tolerance are used, devices must be designed to work properly with the worst-case combination of tolerances and external conditions.

The term best-case performance is used in computer science to describe an algorithm's behavior under optimal conditions. For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

Development and choice of algorithms is rarely based on best-case performance: most academic and commercial enterprises are more interested in improving Average-case complexity and worst-case performance. Algorithms may also be trivially modified to have good best-case running time by hard-coding solutions to a finite set of inputs, making the measure almost meaningless.

worst-case: The situation that would require the most amount of processing.
best-case: The situation that would require the least amount of processing.
average-case: The average amount of required processing. This option is the most difficult to compute because you must consider the probabilities of all possible cases and the amount of processing required for each case.

Computer scientists use probabilistic analysis techniques, especially expected value, to determine expected running times.

9.2 Best, worst and average case

Best, worst and average case (cont..)

Worst-Case, Best-Case, and Average-Case

```
algorithm SequentialSearch(A[0..n - 1], K)
    // Searches for a value in an array
    // Input: An array A and a search key K
    // Output: The index where K is found or -1

    for i ← 0 to n - 1 do
        if A[i] = K then return i
    return -1
```

- Basic Operation: The comparison in the loop
- Worst-Case: n comparisons
- Best-Case: 1 comparison
- Average-Case: $(n+1)/2$ comparisons assuming each element equally likely to be searched.

January 16, 2016 | Proprietary and Confidential | 8 | Capgemini CONSULTING TECHNOLOGY SERVICES

Examples: Sequential and Binary Searches of an Array of n Items

Sequential Search - Assume that the data in the array is in no particular order.

worst-case: Not found or searching for the item that is in the last position. $f(n) = n$

best-case: Found at the first position. $f(n) = 1$

average-case: Assuming the probability that the item is found at any position is equally likely, $f(n) = 1/n + 2/n + 3/n + \dots + n/n = (1 + 2 + 3 + \dots + n)/n = (n+1)/2$

Ordered Sequential Search - Requires the precondition that the data in the array is ordered by the key field that will be used for the search. This additional overhead will affect the efficiency of the search, but for now will not be considered

worst-case: Searching for the item that is greater than or equal to the item in the last position. $f(n) = n$

best-case: Searching for the item that is less than or equal to the item in the first position. $f(n) = 1$

average-case: --- cannot, in general, be determined ---

Binary Search - Requires the precondition that the data in the array is ordered by the key field that will be used for the search. This additional overhead will affect the efficiency of the search, but for now will not be considered

worst-case: Not found or searching for the item that would be the last one checked before determining it is not in the array. $f(n) = \log_2 n$

best-case: Found at position $n/2$ (the first value checked). $f(n) = 1$

average-case: --- in general, can't be determined ---

| Comparison of all sorting Algorithms | | | | | |
|--------------------------------------|----------------|----------------------|-------------------------|-----------------------|------------------------|
| Algorithm | Data Structure | Time Complexity:Best | Time Complexity:Average | Time Complexity:Worst | Space Complexity:Worst |
| Quick Sort | Array | $O(n \log(n))$ | $O(n \log(n))$ | $O(n^2)$ | $O(\log(n))$ |
| Merge sort | Array | $O(n \log(n))$ | $O(n \log(n))$ | $O(n \log(n))$ | $O(n)$ |
| Heap sort | Array | $O(n \log(n))$ | $O(n \log(n))$ | $O(n \log(n))$ | $O(1)$ |
| Bubble sort | Array | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ |
| Insertion sort | Array | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ |
| Selection sort | Array | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ |

Capgemini
CONSULTING TECHNOLOGY SERVICES

Insertion sort applied to a list of n elements, assumed to be all different and initially in random order. On average, half the elements in a list $A_1 \dots A_j$ are less than element A_{j+1} , and half are greater.

Therefore the algorithm compares the $j+1$ -st element to be inserted on the average with half the already sorted sub-list, so $t_j = j/2$. Working out the resulting average-case running time yields a quadratic function of the input size, just like the worst-case running time.

Quicksort applied to a list of n elements, again assumed to be all different and initially in random order. This popular sorting algorithm has an average-case performance of $O(n \log n)$, which contributes to making it a very fast algorithm in practice.

But given a worst-case input, its performance degrades to $O(n^2)$. Also, when not implemented with the "shortest first" policy, the worst-case space complexity degrades to $O(n)$.

| Data structure | Time Complex: Avg: Search | Time Complex: Avg: Insertion $O(n^2)$ | Time Complex: Avg: Deletion | Time Complex: Worst: Search | Time Complex: Worst: Insertion | Time Complex: Worst: Deletion | Space Complex: Worst |
|--------------------|---------------------------|---------------------------------------|-----------------------------|-----------------------------|--------------------------------|-------------------------------|----------------------|
| Basic Array | $O(n)$ | - | - | $O(n)$ | - | - | $O(n)$ |
| Dynamic array | $O(n)$ | $O(n)$ | - | $O(n)$ | $O(n)$ | - | $O(n)$ |
| Singly linked list | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Doubly linked list | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Binary Search Tree | $O((\log n))$ | $O((\log n))$ | $O((\log n))$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |

January 26, 2016 | Proprietary and Confidential | +10+

 Capgemini
CONSULTING TECHNOLOGY SERVICES

Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list. However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only $n/2$ elements.

9.3 Big o, Omega and Theta

Big O, Omega and Theta

- Big O notation is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in input size
- The difference between Big O notation and Big Omega notation is that Big O is used to describe the worst case running time for an algorithm.
- Big Omega is used to represent the lower bound, which is also the “best case” for that algorithm
- It is also possible to consider the "greater than or equal to" relation and "equal to" relation in a similar way. Big-Omega is for the former and big-theta is for the latter.

January 26, 2016

Proprietary and Confidential

» 11 «



But, Big Omega notation, on the other hand, is used to describe the best case running time for a given algorithm. Big O is used to represent the upper bound running time for an algorithm, which is also the “worst case”. Big-oh concerns with the “less than or equal to” relation between functions for large values of the variable.

9.3 Big o, Omega and Theta

Big O, Omega and Theta With an example C Code

- For Example in an array of n elements, If we wanted to access the first element of the array this would be O(1) since it doesn't matter how big the array is, it always takes the same constant time to get the first item.

```
int array[n];
x = array[0];
for(int i = 0; i < n; i++)
{
    if(array[i] == numToFind)
        { return i; }
} // to find a number in the list:
```

- This would be O(n) since at most we would have to look through the entire list to find our number.
- The Big-O is still O(n) even though we might find our number the first try and run through the loop once because Big-O describes the upper bound for an algorithm (omega is for lower bound and theta is for tight bound).



Big O gives the upper bound for time complexity of an algorithm. It is usually used in conjunction with processing data sets (lists) but can be used elsewhere.

When we get to nested loops:

```
for(int i = 0; i < n; i++){
    for(int j = i; j < n; j++){
        array[j] += 2;
    }
}
```

This is O(n^2) since for each pass of the outer loop (O(n)) we have to go through the entire list again so the n's multiply leaving us with n squared.

9.4 Time Complexity vs Space Complexity

Time vs Space Complexity

- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input.
- The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms.
- Space Complexity, It represents the total amount of memory space that a "normal" physical computer would need to solve a given computational problem with a given algorithm.
- Space Complexity corresponds to the amount of physical computer memory needed to run a given program.

January 26, 2016 | Proprietary and Confidential | +13+



Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

The time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$ for any n (bigger than some n_0), the asymptotic time complexity is $O(n^3)$.

Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as $T(n)$, which is defined as the maximum amount of time taken on any input of size n . Less common, and usually specified explicitly, is the measure of average-case complexity. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(Mn)$ and $M > 1$ is said to be an exponential time algorithm.

9.5 Example C Program to calculate Time Complexity

Example C Program to calculate Time Complexity

```
#define LISTSIZE 100000 //Number of integers to be generated
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <conio.h>

int main()
{
    clock_t start;
    double d;
    int primesFound;
    long int list[LISTSIZE],i,j;
    int listMax = (int)sqrt(LISTSIZE), primeEstimate = (int)(LISTSIZE/log(LISTSIZE));

    for(i=0;i < LISTSIZE; i++)
        list[i] = i+2;
    start=clock();
```

9.5 Example C Program to calculate Time Complexity

Example C Program (Contd...)

```
for(i=0; i < listMax; i++)
{
    //If the entry has been set to 0 ('removed'), skip it
    if(list[i] > 0)
    {
        //Remove all multiples of this prime
        //Starting from the next entry in the list
        //And going up in steps of size i
        for(j = i+1; j < LISTSIZE; j++)
        {
            if((list[j] % list[i]) == 0)
                list[j] = 0;
        }
    }
}
```

9.5 Example C Program to calculate Time Complexity Example C Program (Contd...)

```
d=(clock()-start)/(double)CLOCKS_PER_SEC;

//Output the primes
primesFound = 0;
for(i=0; i < LISTSIZE; i++)
{
    if(list[i] > 0)
    {
        primesFound++;
        printf("%d\n", list[i]);
    }
}
printf("\n%f", d);
getch();
return 0;
}
```



Summary

➤ In this lesson, you have learnt:

- The Definition of Algorithm Analysis
- The various cost analysis models
- Best, worst and average case
- Comparison of all sorting Algorithms
- Comparison of all Data Structures
- The definitions of Big O, Omega and Theta
- Example for Big O, Omega and Theta understandings
- Time VS Space Complexity



Review Questions

1. Which of the following case does not exist in complexity theory
 - a) Best case
 - b) Worst case
 - c) Average case
 - d) Null case
2. The Worst case occur in linear search algorithm when
 - a) item is somewhere in the middle of the array
 - b) Item is not in the array at all
 - c) Item is the last element in the array
 - d) Item is the last element in the array or is not there at all
3. Two main measures for the efficiency of an algorithm area.
 - a) Processor and memory
 - b) Complexity and capacity
 - c) Time and space
 - d) Data and space





C and Data Structure

Lab Book

Copyright © 2011 IGATE Corporation (a part of Capgemini Group). All rights reserved.

No part of this publication shall be reproduced in any way, including but not limited to photocopy, photographic, magnetic, or other record, without the prior written permission of IGATE Corporation (a part of Capgemini Group).

IGATE Corporation (a part of Capgemini Group) considers information included in this document to be confidential and proprietary.

Document Revision History

| Date | Revision No. | Author | Summary of Changes |
|--------------|--------------|-------------------------|---|
| 05-Apr-2009 | 0.1D | Kishori Khadilkar | Content Creation |
| 01-Jul-2009 | | CLS Team | Review |
| 25-May-2011 | 0.2D | Rathnajothi Perumalsamy | Revamp/Refinement |
| 28-Sept-2012 | | Vaishali Kunchur | Revamped for additional assignments |
| 17-Mar-015 | 0.3D | Vinod P Manoharan | Revamped for additional assignments and fixing existing errors. |

Table of Contents

| | |
|--|----|
| Document Revision History | 2 |
| Table of Contents | 3 |
| Getting Started..... | 5 |
| Overview..... | 5 |
| Setup Checklist for Data structure..... | 5 |
| Instructions | 5 |
| Learning More (Bibliography if applicable) | 5 |
| Lab 1. Introduction to C operators | 6 |
| Lab 2. Type Casting and Loop Constructs | 7 |
| | 8 |
| Lab 3. Functions | 9 |
| Lab 4. Arrays..... | 10 |
| Lab 5. Pointers..... | 12 |
| Lab 6. Structures..... | 13 |
| Lab 7. File handling..... | 14 |
| Lab 8. Preprocessor | 15 |
| Lab 9. Sorting and Searching..... | 16 |
| 1.1: Bubble Sort | 16 |
| 1.2: Insertion Sort | 17 |
| 1.3: Quick Sort | 18 |
| 1.4: Extra Assignments | 19 |
| 1.5: Sequential Search | 19 |
| 1.6: Binary Search..... | 20 |
| Lab 10. Linked List | 22 |
| 1.1: Employee Linked List | 22 |
| 1.2: List of characters from string | 26 |
| Lab 11. Stacks and Queues | 27 |
| 1.1 Stacks using LinkedList..... | 27 |
| 1.2 Queue using LinkedList | 29 |
| Assignment 1: Job Queue <>To Do> | 31 |
| 1.3: Stretched Assignments | 31 |
| Lab 12. Trees | 32 |

| | |
|---|----|
| 1.1 Binary Tree Traversal..... | 32 |
| Assignment 1: Job Queue <> To Do> | 35 |
| Appendix I - Debugging Tips..... | 36 |
| Appendix II - Coding Standards | 41 |
| Appendix III - Table of Figures..... | 51 |

Getting Started

Overview

This lab book is a guided tour for learning C Programming. It comprises ‘To Do’ assignments. Work out the ‘To Do’ assignments that are given.

Setup Checklist for Data structure

Here is what is expected on your machine for the lab to work.

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 32MB of RAM (64MB or more recommended)

Please ensure that the following is done:

- Visual studio 2005 is installed on your machine.

Instructions

- Create a directory by your name in drive <drive>. In this directory, create a subdirectory C_assgn. For each lab exercise create a directory as lab <lab number>.
- If under the Windows OS, .prj is not created please create all the functions and the main function under a single .c file.

Learning More (Bibliography if applicable)

- “Pointers in C” by Kanetkar Yashawant
- “Test Your C” by Kanetkar Yashawant
- “C: The Complete Reference” by Schildt Herbert
- “Let Us C” by Kanetkar Yashawant
- “Programming in ANSI C” by Balagurusamy

Lab 1. Introduction to C operators

| | |
|-------|--|
| Goals | Write and execute simple C programs using various operators. |
| Time | 120 Minutes |

1. Write a function that inverts the bits of an unsigned char x and stores answer in y.

Your answer should print out the result in binary form although input can be in decimal form.

Your output should be like this:

x = 10101010 (binary)

x inverted = 01010101 (binary)

2. Write a program to perform the following task:

- a. Input two numbers and work out their sum, average and sum of the squares of the numbers.

3. Write a program that works out the largest and smallest values from a set of 10 inputted numbers.

4. Write a program to read a "float" representing a number of degrees Celsius, and print as a "float" the equivalent temperature in degrees Fahrenheit. Print your results in a form such as
 - b. 100.0 degrees Celsius converts to 212.0 degrees Fahrenheit.

5. Write a program to read a positive integer at least equal to 3, and print out all possible permutations of three positive integers less or equal to than this value.

6. Write a program to read a number of units of length (a float) and print out the area of a circle of that radius. Assume that the value of pi is 3.14159
 - a. Your output should take the form: The area of a circle of radius ... units is.... units.
 - b. It should print an error message "Error: Negative values not permitted" if the input value is negative.

7. Given as input a floating (real) number of centimeters, print out the equivalent number of feet (integer) and inches (floating, 1 decimal), with the inches given to an accuracy of one decimal place.
 - a. Assume 2.54 centimeters per inch, and 12 inches per foot.
 - b. If the input value is 333.3, the output format should be:
 - c. 333.3 centimeters is 10 feet 11.2 inches.

8. Given as input an integer number of seconds, print as output the equivalent time in hours, minutes and seconds. Recommended output format is something like 7322 seconds is equivalent to 2 hours 2 minutes 2 seconds.

Lab 2. Type Casting and Loop Constructs

| | |
|-------|--|
| Goals | Write and execute simple C programs to understand type conversion and type casting |
| Time | 30 Minutes |

1. Write a C program to calculate simple interest. Accept the principal amount, rate of interest in percentage and number of years from the user. Assume these 3 quantities to be integers. The interest calculated will be a floating-point number.
2. Write a program to accept conversion choice from the user:
 - a) centimeter to inch
 - b) inch to centimeterAccept the number of centimeters (a real number), and to print out the equivalent number of feet (integer) and inches (floating, 1 decimal), with the inches given to an accuracy of one decimal place. Similarly work out for b) option.

Assume 2.54 centimeters per inch, and 12 inches per foot.
If the input value is 333.3, the output format should be:
333.3 centimeters is 10 feet 11.2 inches.

| | |
|-------|--|
| Goals | Write and execute C programs to understand iterative constructs through while and do... while loops. |
| Time | 60 Minutes |

1. Modify the previous program so that it executes repetitively until user chooses to stop i.e. Accept the option "Y" or "N" from the user to indicate whether he wants to continue or not. Make use of do... while loop to achieve the same.
2. Generate all permutation as

Example : (n = 3)

```
1   2   3
1   3   2
2   1   3
2   3   1
3   1   2
3   2   1
```

Try to generate for any n.

- 3 Print a palindrome as shown below:

```
MALAYALAM
ALAYALA
LAYAL
AYA
Y
```

| | |
|-------|--|
| Goals | Write and execute C programs to understand iterative constructs through for loops. |
| Time | 90 Minutes |

- 1 Write a program to generate the following output

- 2 Write a program to print prime numbers from 1 to 1000
- 3 Read a positive integer value, and compute the following sequence: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value. Finally print out how many of these operations you performed.

Typical output might be:

Initial value is 9
Next value is 28
Next value is 14
Next value is 7
Next value is 22
Next value is 11
Next value is 34
Next value is 17
Next value is 52
Next value is 26
Next value is 13
Next value is 40
Next value is 20
Next value is 10
Next value is 5
Next value is 16
Next value is 8
Next value is 4
Next value is 2

Final value 1, number of steps 19

- If the input value is less than 1, print a message containing the word ‘Error’ and exit the execution.
- 4 Check whether a number is an Armstrong number or not. (An Armstrong number is the equal to sum of cubes of digits E.g. $153 = (1)^3 + (5)^3 + (3)^3$.
 - 5 Write a program to find hcf and lcm: The code below finds highest common factor and least common multiple of two integers. HCF is also known as greatest common divisor(GCD) or greatest common factor(gcf).

Lab 3. Functions

| | |
|--------------|---|
| Goals | Write and execute C programs to perform string operations using standard library functions. |
| Time | 120 min |

1. Write an interactive C program that will encode or decode a line of text. To encode a line of text, proceed as follows:
 - a. Convert each character including blank spaces to its ASCII equivalent.
 - b. Generate a positive random integer. Add this integer to the ASCII equivalent of each character. The same random integer will be used for the entire line of text.
 - c. Suppose the permissible values for ASCII code fall in the range of N1 and N2 (e.g. 0 to 255). If the number obtained in step ii above exceeds N2, then subtract the largest possible multiple of N2 from this number and then add the remainder to N1. Hence, the encoded number will always fall between N1 and N2, and will always represent some ASCII character.
 - d. Print the characters that correspond to encoded ASCII values.
 - e. Reverse the entire procedure to decode the line of text and print the original line of text again.
2. Write a program in C which performs the following operations on a string obtained. The following functions can be made for strings:
 - a. Reverse string
 - b. Check whether a string is palindrome or not
 - c. Convert to lower case
 - d. Convert to upper case
 - e. Convert to title case
 - f. Convert to sentence case
 - g. Convert to toggle case
 - h. Count vowels
 - i. Count consonants
 - j. Count digits
 - k. Count words
 - l. Extract n characters from the left
 - m. Extract n characters from the right
 - n. Extract n characters from m position
 - o. Delete all spaces from the string
 - p. Count the occurrence of a particular character
3. write a program to accept variable length of arguments , and display the sum of it all.
4. write a program to accept the list of names from the key board and pass it to a function with the help of variable length of arguments and display it all.

Lab 4. Arrays

| | |
|-------|--|
| Goals | Write and execute C programs to work with single dimensional arrays. |
| Time | 90 Minutes |

1. Write a program to store 11, 22, 33, 44, 55, 66 in an array.
 - a. Find the sum of all the integers
 - b. Insert a number at a specified position.
 - c. Delete a number from a specified position. Write a program to display the minimum and maximum number from an array of 10 integers along with their position in the array.
2. Write a program to evaluate the expression $z=x^2+y^2$, where x and y are 2 are 2 arrays. Each array holds 10 user entered elements. Store the result in another array and display it.
3. Write a program to store characters in an array and search the array for a given character. Also count the number of occurrences of the character in the array.
4. Accept a 2D Matrix and store only its nonzero value in the manner describe below :-

original Matrix Converted matrix

| | | |
|---------|-------|-------------------------|
| 1 0 0 4 | 3 4 4 | ---> 1st row (see note) |
| 0 0 0 2 | => | 0 0 1 |
| 0 6 0 0 | | 0 3 4 |
| | 1 3 2 | |
| | 2 1 6 | |

Note :-

1st row: It contains dimension of first matrix (m, n) and No. of nonzero elements.

Subsequent rows contain (row, col) of nonzero element & its value.

5. Write a program to merge two arrays into third array: Arrays are assumed to be sorted in ascending order. You enter two short sorted arrays and combine them to get a large array.

| | |
|-------|---|
| Goals | Write and execute C programs to understand character arrays and strings |
| Time | 30 Minutes |

1. Make a file which contains the functions to perform the following tasks and test it by including the file in another file which contains main()
 - a. Accept an array
 - b. Display an array
 - c. Sort array
 - d. Reverse array
 - e. Count odd
 - f. Count even
 - g. Count prime

- h. Maximum value in the array
- i. Minimum value in the array
- j. Search a value in the array

Lab 5. Pointers

| | |
|-------|---|
| Goals | Write and execute C programs to work with Pointers. |
| Time | 150 Minutes |

1. Write a program to accept list of values [Numeric] from user and store in it, the size of the array will be defined by the user.
2. Write a program to accept list of values [Alphabet] from user and store it in a dynamic array.
3. Write a program to accept list of strings / names from the user via keyboard and store it in a dynamic multi-dimensional array.
4. Create a n X m matrix numerical array with the help of “n” and “m” given by the user via keyboard. And make sure both “n” and “m” are not equal. Accept values from the user and displays it.
5. Create a n X m matrix string array with the help of “n” and “m” given by the user via keyboard. And make sure both “n” and “m” are not equal. Accept values from the user and display it.

Example : The Array should looks like for an input of 3 rows and 5 columns

```
Rahul rabino vinod rraaju reena  
Chandhu kiran kraunaa kareena meeru  
Mittu mibin madhu chandrasekaran ameerislahi
```

- Each strings / name should be a variable length array.
6. Create a function which accepts the array of values via parameter, and display it.
 7. Create a function which returns an array of values to the calling function.
 8. Create a global user defined dynamic array, via one function defines it's size and create the allocation. Via another function accepts the input from the user and stores in it. Via the third function displays the values of the array with the help of return by reference method. Main() function should simply call all these functions to execute the memory allocation, accepting values from the user, storing in the memory, and displaying it in the screen. Make sure all the list of following should be part of the program.
 - a. Static variable
 - b. Global variable
 - c. Call by value
 - d. Call by address / pass by references
 - e. Return by value
 - f. Return by reference / address
 - g. Dynamic array creation

Lab 6. Structures

| | |
|-------|---|
| Goals | Write and execute C programs to work with structures. |
| Time | 60 Minutes |

1. Define a structure to represent an employee record containing information like employee id, name and salary for an employee. Use this structure to store and print the values for the same. Implement name as a character pointer (`char*`) and allocate exact amount of memory required for the name entered by a user. Implement a function to raise the salary for an employee.
2. Write a program that defines a structure containing 2 members that describe an entry in a telephone book: one is the name of a person, another is his/her telephone number. Implement name as a `char *` and allocate memory dynamically to hold the name entered by the user. Use the same for storing and displaying information for 2 persons.
3. Modify the program you wrote in exercise 3. First it will ask how many entries will be needed. Then it will allocate an array of struct's and allow you to enter the data. Finally it will print and deallocate the array.

Note: Do not forget to release the memory allocated for the names before deallocating the entire structure array.

Note: The language definition states that for each pointer type, there is a special value—the “null pointer”—which is distinguishable from all other pointer values and which is “guaranteed to compare unequal to a pointer to any object or function”. That is, the address-of operator & will never yield a null pointer, nor will a successful call to malloc. (malloc does return a null pointer when it fails, and this is a typical use of null pointers: as a “special” pointer value with some other meaning, usually “not allocated” or “not pointing anywhere yet.”). A null pointer is conceptually different from an uninitialized pointer. A null pointer is known not to point to any object or function; an uninitialized pointer might point anywhere.

| | |
|-------|---|
| Goals | Write and execute C programs to work with structures. |
| Time | 30 Minutes |

1. Write a program that read a number from command line input and generates a random floating point number in the range 0 - the input number.
2. Write a program that reads a number that says how many integer numbers are to be stored in an array, creates an array to fit the exact size of the data and then reads in that many numbers into the array.
3. Write a program to dynamically allocate an array and find the sum of all the elements in the array,

Lab 7. File handling

| | |
|--------------|--|
| Goals | Write and execute C programs to do File handling |
| Time | 1.30 hrs |

1. Write a program to read a file which consists of only numbers. If the read number is an even number it has to write in a separate file. If the number read is an odd number then it has to write in a different file.
2. Create an utility which takes a file name as input from the user at command line and counts the no. of characters, words and lines and displays them.
3. Write a program which takes two file names from the command line and compares them character by character and displays if the contents are same. If no then the position of the mismatched character and the mismatched character from both the files should be displayed. If one file reaches end of file appropriate message should be displayed while the mismatched character should be displayed from the other file.
4. Enhance the Employee program and add code for the following tasks:
 - a. Search Employee
 - b. Modify Employee
 - c. Delete Employee

Lab 8. Preprocessor

| | |
|--------------|---|
| Goals | Write and execute simple C programs using Macros. |
| Time | 90 Minutes |

5. How would you arrange that the identifier MAXLEN is replaced by the value 100 throughout a program?
6. Define a preprocessor macro swap(t, x, y) that will swap two arguments x and y of a given type t.
7. write a program to illustrate the difference between functions with Macros.
8. write a program with the help of conditional macro to control the execution of the code block
9. write a program with the help of #ifdef and #undefine macros.

Lab 9. Sorting and Searching

| | |
|-------|---|
| Goals | Understand different sorting and searching algorithm. |
| Time | 180 minutes |

1.1: Bubble Sort

Step 1: Create a new C file named BubbleSort.c, and write the following code in it.

```
//include the required additional header files
#include <stdio.h>
//bubble sort function
void bubbleSort(int numbers[], int array_size)
{
    int i,j;
    int temp;
    //define bubble sort algorithm in this function
    for(=0;i<array_size-1;i++)
        for(j=i+1;j<array_size;j++)
            if(numbers[i]>numbers[j]){
                temp=numbers[i];
                numbers[i]=numbers[j];
                numbers[j]=temp;}
    //first parameter is the array to be sorted
    //second parameter is the size of the input array
}
```

Example 1: Bubble Sort

Step 2: Create header file bubblesort.h to include prototype of function.

```
#define NUM_ITEMS 5
void bubbleSort(int numbers[], int array_size);
```

Example 2: Bubble sort

Step 3: Create a new C source file named testBubbleSort.c, and write the following code.

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

```
int main()
{
    int numbers[NUM_ITEMS];
    //seed random number generator
    srand(getpid());
    for (i = 0; i < NUM_ITEMS; i++)
    {
        //generate elements of array randomly
        numbers[i] = rand();
        printf("%d\n", numbers[i]);
    }
    //perform bubble sort on array
    bubbleSort(numbers, NUM_ITEMS);
    printf("\nDone with sort.\n");
    for (i = 0; i < NUM_ITEMS; i++)
        printf("%i\n", numbers[i]);
    return 0;
}
```

Example 3: Bubble Sort

1.2: Insertion Sort

Step 1: Create a new C file named InsertionSort.c, and write the following code in it.

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

void insertionSort(int numbers[], int array_size)
{
    //first parameter is the array to be sorted
    //second parameter is the size of the input array
    <><to do>>
}
```

Example 4: Insertion sort

Step 2: Create Insertionsort.h file to add prototype of the following function:

```
#define NUM_ITEMS 5
void insertionSort(int numbers[], int array_size)
```

Example 5: Insertion sort

Step 3: Create a new C source file named testInsertionSort.c, and write the following code in it:

```
#include "InsertionSort.h"

int main()
{
    int numbers[NUM_ITEMS];
    int i;
    srand(getpid());
    /*srand seeds the random number generation
    function rand so it does not produce the same sequence of numbers
    The getpid() function returns the process ID of the calling process. */
    for (i = 0; i < NUM_ITEMS; i++)
    {
        //generate elements of array randomly
        numbers[i] = rand();
        printf("%d\n", numbers[i]);
    }
    insertionSort(numbers, NUM_ITEMS);
    printf("\nDone with sort.\n");

    for (i = 0; i < NUM_ITEMS; i++)
        printf("%i\n", numbers[i]);
    return 0;
}
```

Example 6: Insertion sort

1.3: Quick Sort

Step 1: Create a new C file named QuickSort.c, and write the following code in it. Also create QuickSort.h to add prototype of functions.

```
//include the required additional header files
#include <stdio.h>
//helper function quickSort
void quickSort(int numbers[], int array_size)
{
    q_sort(numbers, 0, array_size - 1);
}

//quick sort function
void q_sort(int numbers[], int left, int right)
{
    //define variables to hold left, right and pivot values
    int pivot, l_hold, r_hold;
    //define quick sort algorithm in this function
    //first parameter is the array to be sorted
    //second parameter is the index of the element on the left side
    //third parameter is the index of the element on the right side
}
```

Example 7: Quick sort

Step 2: Create a new C source file named testQuickSort.c, and write the following code:

```
// include the required header files
//define array size and array
#define NUM_ITEMS 100
int main()
{
    int i;
    int numbers[NUM_ITEMS];
    //seed random number generator
    srand(getpid());
    //fill array with random integers
    <<to do>>
        //perform quick sort on array
        quickSort(numbers, NUM_ITEMS);
        //print the sorted array
    <<to do>>
    return 0;
}
```

Example 8: Quick sort

1.4: Extra Assignments

Step 1: Write a C program that sorts the elements of a two dimensional array in the following manner:

- row wise
- column wise

1.5: Sequential Search

Step 1: Create a new C header file named SequenceSearch.c, and write the following code in it. Also create header file by name SequenceSearch.h to add prototype of function.

```
//include the required additional header files
#include <stdio.h>
//Search function
int Search(int numbers[], int array_size,int element)
{
    //define sequential search algorithm in this function
    //first parameter is the array to be sorted
    //second parameter is the size of the input array
    // Third parameter is element to be searched
    <<to do>>
    //return position of the number if found -1 otherwise
}
```

Example 9: Sequential Search

Step 2: Create a new C source file named testSequenceSearch.c, and write the following code:

```
#include<stdio.h>
#include "SequenceSearch.h"
//include required header files

#define NUM_ITEMS 5
int main()
{
    int pos,num;
    //seed random number generator
    srand(getpid());
    int numbers[NUM_ITEMS];
    //fill numbers array with random integers
    <<to do>>
    //Accept element to be searched
    //perform Sequential search on array
    pos=search(numbers, NUM_ITEMS,num);
    // Display appropriate message
    return 0;
}
```

Example 10: Sequential search

1.6: Binary Search

Step 1: Create a new C file named BinarySearch.c, and write the following code in it. Also create BinarySearch.h file to add prototypes of function.

```
//include the required additional header files
#include <stdio.h>
//Search function
int Search(int numbers[], int array_size, int element)
{
    //define binary search algorithm in this function
    //first parameter is the array to be sorted
    //second parameter is the size of the input array
    // Third parameter is element to be searched
    <<to do>>
    //return position of the number if found, -1 otherwise
}
```

Example 11: Binary Search

Step 2: Create a new C source file named testBinarySearch.c, and write the following code:

```
#define NUM_ITEMS 5
#include<stdio.h>
//include required header files
int main()
```

```
{ int pos,num;  
int numbers[NUM_ITEMS];  
//Accept array elements in the sorted order  
<<to do>>  
    //Accept element to be searched  
    //perform Binary search on array  
    pos=search(numbers, NUM_ITEMS,num);  
    // Display appropriate message  
    return o;  
}
```

Example 12: Binary Search

Lab 10. Linked List

| | |
|-------|---|
| Goals | Understand linked list and different operations performed on the list |
| Time | 120 minutes |

1.1: Employee Linked List

Step 1: The employee information (empid, name, sal) is stored in a text file EMPS.TXT. Write a program to read all the records from this file and store them in a linked list. Do the necessary update in the linked list such as adding new records, deleting and modifying existing records, etc. Once done, save all the records back to the EMPS.TXT file.

Step 2: Create a new file named employee.c, and type in the following code:

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

FILE * FP; /* global pointer to file */

struct emp
{
    int empid;
    char name[20];
    float sal;
    struct emp *next;
};

struct emp *list; /* global pointer to beginning of list */

struct emp * getnode() /*creates a node and accepts data from user*/
{
    struct emp *temp;
    temp=(struct emp *)malloc(sizeof(struct emp));
    printf("Enter the employee id:");
    scanf("%d",&temp->empid);
    fflush(stdin);
    printf("Enter the name:");
    scanf("%s",temp->name);
    fflush(stdin);
    printf("Enter salary:");
    scanf("%f",&temp->sal);
    fflush(stdin);
    temp->next=NULL;
```

```
return temp;}
```

Example 13: Linked List - getnode

```
struct emp * getrecord() /*creates a node and reads a record from file */
{
    struct emp *temp;
    temp=(struct emp *)malloc(sizeof(struct emp));

    fscanf(FP,"%d %s %f\n", &temp->empid, temp->name, &temp->sal);

    temp->next=NULL;
    return temp;
}

/* Search returns address of previous node; current node is */
struct emp * search(int cd,int *flag)
{
    struct emp *prev,*cur;
    *flag=0;
    if (list==NULL) /* list empty */
        return NULL;
    for(prev=NULL,cur=list;(cur) && ((cur->empid) < cd);
        prev=cur,cur=cur->next);
        if((cur) && (cur->empid==cd)) /* Node with given empid exists */
            *flag=1;
        else
            *flag=0;
    return prev;
}
int insert(struct emp *new)
{
    struct emp *prev;
    int flag;
    if (list==NULL) /* list empty */
    {
        list=new;
        return 0;
    }
    prev = search(new->empid,&flag);
    if(flag == 1) /* duplicate empid */
        return -1;

    if(prev==NULL) /*insert at beginning */
    {
        new->next=list;
        list=new;
    }
    else /* insert at middle or end */

```

```
{  
    new->next=prev->next;  
    prev->next=new;  
}  
return o;  
}  
  
void displayall()  
{  
    struct emp *cur;  
    if(list==NULL)  
    {  
        printf("The list is empty\n");  
        return;  
    }  
    printf("\n\nEmpid Name           Salary\n");  
    for(cur=list;cur;cur=cur->next)  
        printf("%4d %22s %8.2f\n",cur->empid,cur->name,cur->sal);  
}  
  
int delete(int cd)  
{  
    struct emp *prev,*temp;  
    int flag;  
    if (list==NULL)          /* list empty */  
        return -1;  
  
    prev=search(cd,&flag);  
    if(flag==0)              /* empid not found */  
        return -1;  
  
    if(prev==NULL) /* node to delete is first node (as flag is 1) */  
    {  
        temp=list;  
        list=list->next;  
        free(temp);  
    }  
    else  
    {  
        temp = prev->next;  
        prev->next = temp->next;  
        free(temp);  
    }  
    return o;  
}  
  
void BackToFile() /*save entire list to the file*/  
{  
    struct emp * cur =list;
```



```
switch(choice)
{
    case 1:
        new=getnode();
        if(insert(new)==-1)
            printf("error:cannot insert\n");
        else
            printf("node inserted\n");
        break;
    case 2:
        printf("Enter the employee id of the record to delete:");
        scanf("%d",&cd);
        fflush(stdin);
        if(delete(cd)==-1)
            printf("deletion failed\n");
        else
            printf("node deleted\n");
        break;
    case 3:
        displayall();
        break;
    case 0:
        BackToFile(); /* save the list back to the file*/
        exit(0);
}
}while(choice !=0);
}
```

Example 14: Linked List

Step 3: Create a text file named EMPS.TXT in the folder where SOL9_2_1.exe is stored. Add a few entries in the file, e.g., a snapshot of EMPS.TXT before executing the program may be as follows:



Figure 1: Notepad

1.2: List of characters from string

Step 1: Write a C program to get a String from the user, and convert it to a linked list of characters and have options to convert the list back to String.

Lab 11. Stacks and Queues

| | |
|-------|--------------------------------------|
| Goals | Understand use of stacks and queues. |
| Time | 90 minutes |

1.1 Stacks using LinkedList

Step 1: Create a new C header file named Stacks.h, and write the following code in it.

```
struct stack
{
    int data;
    struct stack *next;
}*top;

void InitStack();

void Push();

int Pop();

int IsEmpty();
```

Example 15: Stack structure

Step 2: Create a new C source file named Stacks.c, and write the following code:

```
#include <stdio.h>
#include <stdlib.h> /* for dynamic allocation */

typedef struct stack * stack_ptr;
#define NODEALLOC (struct stack *) malloc (sizeof(struct stack))

//Initializing stack
void InitStack()
{ top = NULL; }

//IsEmpty Function
int IsEmpty()
{ return (top == NULL);}

//Push function
```

```
void Push (int num)
{
    stack_ptr newnode;
    newnode=NODEALLOC;
    newnode->next=NULL;
    newnode->data=num;
    if(top==NULL)
        top=newnode;
    else
    {
        newnode->next=top;
        top=newnode;
    }
}

//pop function
int Pop()
{
    int num
    stack_ptr temp=top;
    num= top->data;
    top=top->next;
    free(temp);
    return(num);
}
```

Example 16: Stack operations

Step 3: Create a new C source file named Stacktest.c, and write the following code:

```
#include <stdio.h>
#include <string.h> /* for strlen() */
#include "stacks.h"

main()
{
    int n,choice;
    initstack() //initializing stack top
    do
    {
        printf("\n1. Push \n 2. Pop \n 3. Exit\n");
        printf("\nEnter your choice");
        scanf("%d",&choice);
        switch(choice)
```

```
{  
    case 1: //Push  
        printf("Enter the element to be pushed : ");  
        scanf("%d",&n);  
        Push(n);  
        break;  
    case 2: //pop  
        if(IsEmpty())  
            printf("\nThe stack is empty \n");  
        else  
            printf("the poped element : %d", Pop());  
        break;  
    }  
}while(choice!=3)  
getch();  
}//end of main
```

Example 17: Stack operations

1.2 Queue using LinkedList

Step 1: Write a C program queue.c to implement Queue.

```
#include<stdio.h>  
  
struct queue  
{  
    int data;  
    struct queue *next;  
}*front,*rear;  
  
Typedef struct queue QUEUE;  
//To initialize queue  
void initqueue()  
{  
    front=rear=NULL;  
}  
  
//to check for empty queue  
int emptyqueue()  
{  
    return(front==NULL);  
}  
  
//To remove element from queue  
int remove()
```

```
{  
    int num;  
    QUEUE *temp=front;  
    num=front->data;  
    front=front->next ;  
    free(temp);  
    if(front==NULL)  
        rear=NULL;  
    return(num);  
}  
  
//to add element in queue  
  
void insert(int num)  
{  
    QUEUE *temp;  
    temp=(QUEUE *) malloc(sizeof(QUEUE));  
    temp->data=num;  
    temp->next=NULL;  
    if(front==NULL)  
        rear=front=temp;  
    else  
    {  
        rear->next=temp;  
        rear=temp;  
    }  
}  
  
main()  
{  
    int choice,num,n;  
    initqueue();  
    do  
    {  
        printf(" \n\n 1: Add  \n 2: Delete \n 3: Exit\n");  
        printf("Enter your choice : ");  
        scanf("%d",&choice);  
        switch(choice)  
        {  
            case 1: //Add number in the queue  
                printf("Enter Element to be added in the queue");  
                scanf("%d",&num);  
                insert(num);  
                break;  
            case 2: //Remove data from the queue  
                if(emptyqueue())  
                    printf("\n\n Queue is empty");  
                else  
                    printf("\n\n The deleted element is %d ", dletequeue());  
        }  
    } while(choice!=3);  
}
```

```
        break;  
    case 3:  
        exit();  
    } // end of switch  
} while(choice!= 3)  
}//end of main
```

Example 18: Queue

Assignment 1: Job Queue <>To Do>>

Step 1: Suppose there are several jobs to be performed with each job having priority value of 1, 2, 3, 4, etc. Write a program that receives the job descriptions and the priorities. Create as many queues as the number of priorities, and queue up the jobs into appropriate queues.

For example:

- Suppose the priorities are 1, 2, 3 and 4, and the data to be entered is as follows:
 - ABC,2,XYZ,1,PQR,1,RTZ,3,CBZ,2,QQQ,3,XXX,4,RRR,1
- Then arrange these jobs as shown below:
 - Q1: XYZ,1,PQR,1,RRR,1
 - Q2: ABC,2,CBZ,2
 - Q3: RTZ,3,QQQ,3
 - Q4: XXX,4
- The order of processing should be: Q1, Q2, Q3, Q4
- Write a program to simulate the above problem.

1.3: Stretched Assignments

Assignment 1: Write a program to make a list of people entering in the hall. The person who enters first in the hall will come out last. Maintain a list, and display the list in an order in which people should come out of the hall.

Lab 12. Trees

| | |
|-------|---|
| Goals | Understand tree data structure and operations on tree |
| Time | 180 minutes |

1.1 Binary Tree Traversal

Solution: Create a program to construct a Binary Tree and perform the Inorder, Postorder, and Preorder Traversal

Step 1: Create a C program called TreeTraversal.h, and include the following code into it:

```
#include <stdio.h>

struct btreeNode
{
    btreeNode *leftchild;
    int data;
    btreeNode *rightchild;
}*root;

void InitTree( struct btreeNode *root );
void buildtree (btreeNode **root,int num );
static void insert ( btreeNode **sr, int num );
void traverse();
static void inorder ( btreeNode *sr );
static void preorder ( btreeNode *sr );
static void postorder ( btreeNode *sr );
static void del ( btreeNode *sr );
```

Example 19: Binary Tree

Step 2: Create a C program called TreeTraversal.c, and include the following code into it:

```
///
```

```
// build tree by calling insert()
void buildtree (btreenode **root,int num )
{
//in buildtree we are changing the address of root this changed //address we want to get back in
main function. So we want to //point to root and not contents of root hence using btreenode
/**root
    insert ( root, num );

}

// inserts a new node in a binary search tree
void insert ( btreenode **sr, int num )
{
    if ( *sr == NULL )
    {
        *sr = new btreenode;

        ( *sr )->leftchild = NULL ;
        ( *sr )-> data = num ;
        ( *sr )->rightchild = NULL ;
        return;
    }
    else // search the node to which new node will be attached
    {
        // if new data is less, traverse to left
        if ( num < ( *sr )-> data )
            insert ( & ( ( *sr )->leftchild ), num );
        else
            // else traverse to right
            insert ( & ( ( *sr )->rightchild ), num );
    }
    return;
}

// traverses btree
void traverse( btreenode *root)
{
    printf("\nIn-order Traversal: ");
    inorder( root );
    printf("\nPre-order Traversal: ");
    preorder( root );

    printf("\nPost-order Traversal: ");
    postorder( root );
}

// traverse a binary search tree in a LDR (Left-Data-Right) fashion
void inorder ( btreenode *sr )
{
```

```
if ( sr != NULL )
{
    inorder ( sr -> leftchild );

    // print the data of the node whose
    // leftchild is NULL or the path
    // has already been traversed
    printf("\t %d", sr -> data );
    inorder ( sr -> rightchild );
}
else
    return;
}

// traverse a binary search tree in a DLR (Data-Left-right) fashion
void preorder ( btreenode *sr )
{
    if ( sr != NULL )
    {
        // print the data of a node
        Printf("\t %d", sr -> data );
        // traverse till leftchild is not NULL
        preorder ( sr -> leftchild );
        // traverse till rightchild is not NULL
        preorder ( sr -> rightchild );
    }
    else
        return;
}

// traverse a binary search tree in LRD (Left-Right-Data) fashion
void postorder ( btreenode *sr )
{
    if ( sr != NULL )
    {
        postorder ( sr -> leftchild );
        postorder ( sr -> rightchild );
        printf( "\t %d",sr -> data );
    }
    else
        return;
}

// deletes nodes of a binary tree
void del ( btreenode *sr )
{
    if ( sr != NULL )
    {
        del ( sr -> leftchild );
```

```
        del( sr->rightchild );
    }
    free(sr);
}

void main( )
{
    btreenode *root ;
    int req, i = 1, num ;

    printf("Specify the number of items to be inserted: ");
    scanf("%d",&req);

    while ( i++ <= req )
    {
        Printf( "Enter the data: " );
        Scanf("%d", num) ;
        buildtree (&root, num ) ;
    }

    traverse(root );
    printf("Do you want to delete the tree?(y/n)");
    scanf("\n%c",&ans);
    if(ans=='Y') || (ans=='y')
        del(root);
}
```

Example 20: Binary Tree Traversal

Assignment 1: Job Queue <<To Do>>

Step 1: Create a file word.txt. The file includes various words one word on each line. Read all words from file arrange it in binary tree format. Accept a new word from user and search if it is there in the file. Display appropriate message.

Appendix I - Debugging Tips

In the development stage, you need to debug your code a number of times. A step-by-step execution of the code helps you monitor the execution path that gets followed. Monitoring/watching the values of the variables helps you catch the bugs (logical errors) in the code.

1. To start executing the code in a step-by-step mode, choose Debug -> Step over menu or press F10.

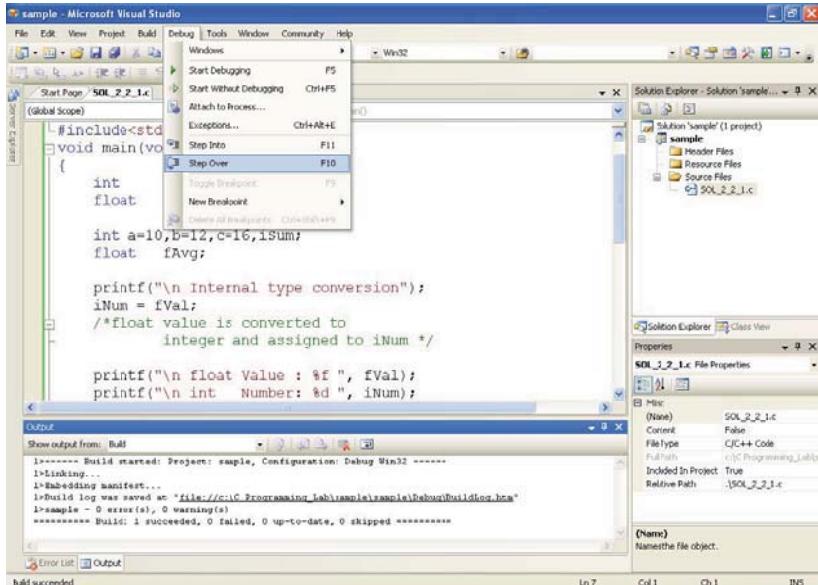


Figure 2: Step Over

2. The execution will start from main(). You can continue the execution by pressing the key, F11 every time. The line of code, which is going to get executed next, is highlighted.

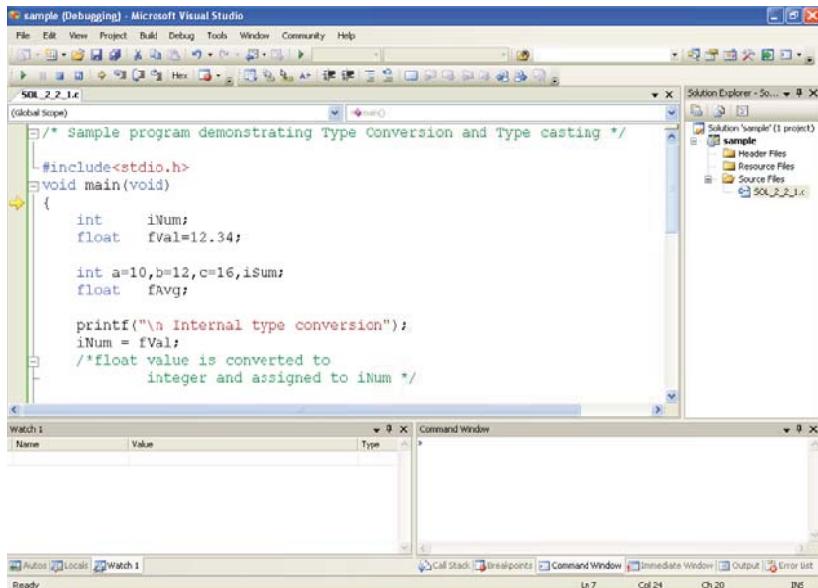


Figure 3: Start Debugging

- i) Whenever there is a function call, choose Debug -> StepIn or press F11 to step into the function code. To skip the step-by-step execution of a function you may continue with F5.
- ii) To monitor/display the current value of a variable throughout the program execution you can add a watch on that variable. To add a watch, right click on the editor in debug mode and select Add Watch from popup menu.

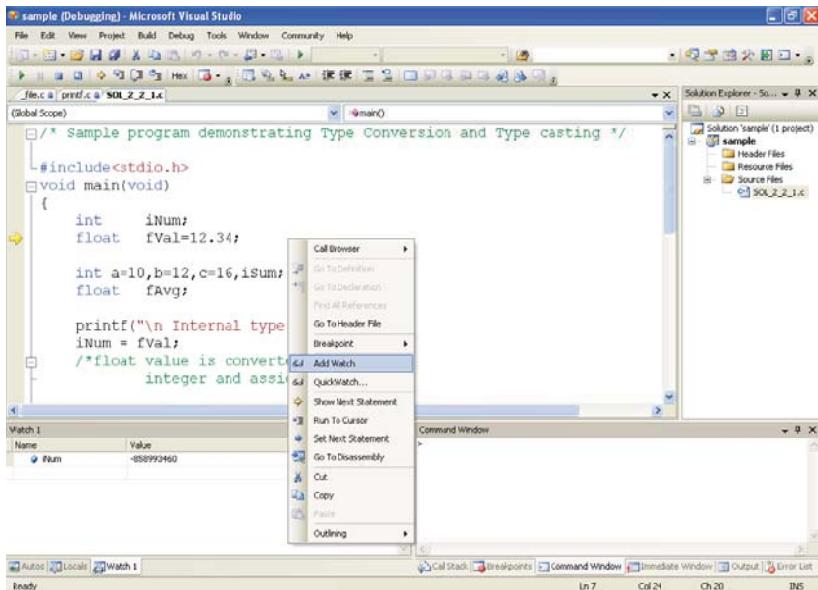


Figure 4: Add Watch Menu

- iii) The variable and its value are now displayed in the Watch window as shown below. You can add a watch on multiple variables in the same way.

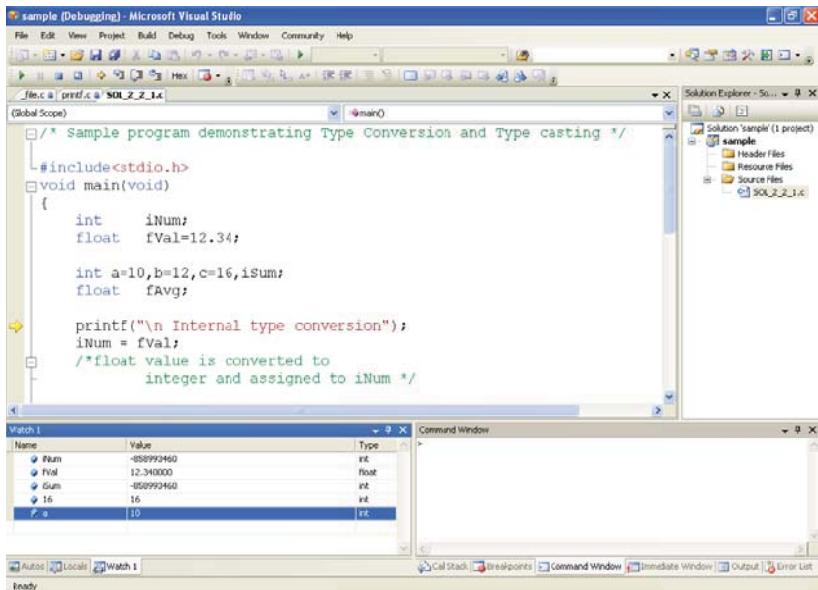


Figure 5: Watch Window

- v) At any point of time, if you want to skip the step-by-step execution, and continue with the normal execution, choose the Debug -> Start without Debugging or press Ctrl+F5.
- vi) You can insert a break-point at a particular statement in the code by selecting Debug -> Toggle Breakpoint menu option or by pressing F9 while your cursor is placed at that statement.

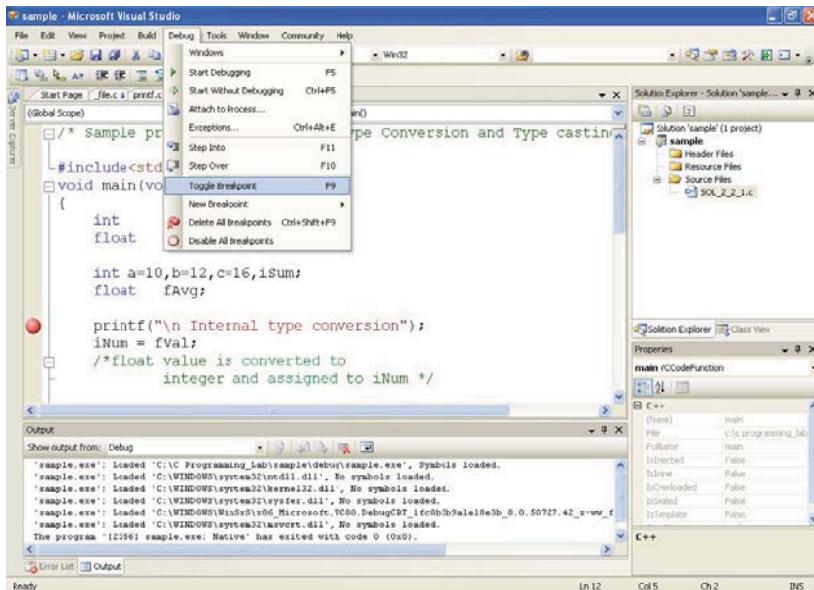


Figure 3: Breakpoints

- vii) When you run the program, the execution will proceed normally till the first breakpoint is reached. As soon as the break point is encountered, the execution halts at that point. You may continue with step-by-step mode of execution from that point onwards.
- viii) To remove the breakpoint, place the cursor at the statement having the break point and select Debug -> Toggle breakpoint menu or press F9.
- ix) To remove all the break points from the code, select Debug -> Delete all breakpoints menu option.
- x) At any point of time, if the program execution hangs you can return to Edit window from the User screen by selecting Debug -> Stop debugging or by pressing Shift+F5.

Appendix II - Coding Standards

The purpose of this document is to serve as a guiding standard for C programmers. This document will aid in achieving coding consistency and ANSI C conformance. It will also enhance the portability and maintainability of C source code. Any client specific standards are to override the standards given here in the event of contradiction between the two.

Naming conventions

The names of all variables, functions, etc., should be informative and meaningful. It is worth the time to devise and use informative names because good mnemonics lead to clear documentation, easy verification, maintenance and improved readability. The following rules should be followed to achieve this:

File Names:

1. File name must be of the format : <basename>.<ext>
2. All the characters forming the basename and the extension must be in lowercase.
3. Basename length must not be longer than 10 characters out of which first 8 characters should be unique and extension must not be longer than 3 characters.
4. Basename should always start with an alphabet and not with a digit or an underscore. It should not end with an underscore.
5. As far as possible digits should be avoided. If used they should be at the end of the basename.
6. Name of the file should always be related to the purpose of that file.
7. File name should never conflict with any system file names.

Source Code File Names :

1. Files belonging to a module should have a common prefix added to the basename, which would indicate the module to which the file belongs.
2. If a file contains only one function then the name of the file can be the same as the name of the function.

Identifier Names

General:

1. The names should be no longer than 31 characters.
2. Avoid similar looking names. e.g. systst and sysstst
3. Names for similar but distinct entities will be distinct.
 - a. e.g. 'goat' and 'tiger' instead of 'animal_1' and 'animal_2'
4. Consider limitations such as allowable length and special characters of compiler/linker on local machines.
5. Names will not have an underscore at their beginning or end.
6. Names should be related to the purpose of the identifiers.
7. Underscore '_' should be used to separate only significant words in names.

8. Avoid names that differ only in presence or absence of underscore '_'
9. Avoid names that differ only in case, eg. foo, Foo.
10. Avoid names conflicting with standard library names, variables and keywords.
 - ◆ Standardize use of abbreviations.
 - ◆ Avoid use of obscure abbreviations.
 - ◆ Do not abbreviate InputCharacter as inch.
 - ◆ Abbreviate common part of mnemonics
 - ◆ e.g. `next_char' is preferred over `n_character'

Local variables:

1. Local variables should be in lower case only.
2. Use of standard short names is allowed when the scope of the variable is limited to a few lines (about 22 lines e.g. following are a few frequently used short names
 - i. p ----- pointer
 - ii. c ----- char
 - iii. l, j -- index

Global Variables:

1. Relate names of variables to the name of the module where defined.
2. Append _g to the end of the name. Eg. int file_section_g;

Constants (#defines and enums):

1. Use of constants must be strictly avoided as they convey little meaning about their use. Instead use symbolic constants.
2. There can be exceptions to this usage where 0 and 1 may appear as themselves e.g.
3. for (i = 0; i < ARREND; i++)
4. All the symbolic constant names should be in capitals only. E.g.
5. #define ARREND 25 /* comment */
6. *The enumeration data type should be used to declare variables that take discrete set of values. e.g.
7. enum { BLACK = 0, WHITE = 1 } colors;

Tags:

1. This includes typedefs, structs, enums and unions.
2. Tags should be in lower case only with _t as a suffix. e.g.

```
typedef struct coord_t
{
    int x;
    int y;
}
```

Typedef names:

1. Typedef names should be in lower case only.

Macros:

1. Append `_M' to the macro names.

2. Macro name should be in all capitals. e.g.

```
#define MOD_M(a,b) <(b)>((b)-(a)):(((a)-(b))
```

Functions:

1. Relate the name of the function to the name of the module it belongs to. Convention for the abbreviation should be the same as that for the file names. e.g. function belonging to obj module will have Objxxxx.
2. Use capitalization to separate significant words. e.g. 'displayroutine' should be written as 'DispRoutine'

Pointers:

1. Append an '_p' to pointer names. e.g.

```
char *name_p;
```

Programming style:

General Rules:

1. Line length must be less than or equal to 80 characters.
2. File length should be less than or equal to 1000 lines.
3. Each function within a file should be limited to 200 lines.
4. Storage class, type and variables must be vertically aligned. Each item should start from the same column.
5. For union/enum/struct adopt following rules :

```
[ typedef ] enum/struct/union [ tag ]
{
...
...
}
```

6. Function prototypes and definitions should be as per the following style.

```
< return type >
< function name >(
    int count; /* the counter */
    ...
    ...
)
```

7. Indentation should be in multiples of 4 spaces. One shouldn't use tabs for indentation.
8. Any start and end brace should be alone on separate lines, and should be vertically aligned with the control keyword. e.g.

```
if (count != 0)
{
    .....
}
```

9. If a group of functions have alike parameters or local variables, then they should have same names. e.g.

Use

```
int
func1(
    int file_err)

int
func2(
    int file_err)
```

Instead of

```
int
func1(
    int file_problem)

int
func2(
    int file_invalid)
```

10. Avoid hiding identifiers across blocks. e.g.

```
/* Bad practice */

int a;

.....
.....
{

float a; /* This hides 'a' above */

.....
.....
}
```

11. For pointer definitions attach "*" to variable names and not to the types. e.g.

Use

char *char_p;

Instead of

char* char_p;

12. Put unrelated variables on the separate lines. e.g.

Use

```
int i, j, k;  
int count;  
int index;  
int flag;
```

Instead of

```
int i, j, k;  
int count, index, flag;
```

13. All the logical blocks should be separated by appropriate blank lines. There should be three blank lines between the definitions of any two functions and at least one blank line between other major blocks.
14. Static, automatic and register variable declarations should be demarcated by blank lines.
15. Variables declared with the register storage class must be declared in decreasing order of importance to ensure that the compiler allocates a register to the most important variables.
16. Avoid simple blocks i.e. without any control statements.
17. Do not depend upon any implicit types. E.g.

Use

static int a;

Instead of

static a;

18. Null body of the control statement must always be on a separate line and explicitly commented as /* Do nothing */.

Use

```
while (....)  
{  
    ;      /* Do nothing */  
}
```

instead of

while (....);

19. One line can have at the most one statement except when the multiple statements are closely related.
20. Return value of functions must be used.
21. An 'if' statement with 'else-if' clauses should be written with the else conditions left justified.

The format then looks like a generalized switch statement. e.g.

```
if (count == 0)
{
    ....
}
else if (count < 0)
{
    ....
}
else if (count > 0)
{
    ....
}
```

22. The body of every control statement must be a compound statement (enclosed in braces even if there is only one statement.)
23. Fall through in "case" statement must be commented.
24. "default" must be present at the end of "case" statement.
25. Condition testing must be done on logical expressions only. eg. With 'count' having the declaration:

```
int count;
Use
if (count == 0)
/* THIS REFLECTS THE NUMERIC (NOT BOOLEAN) NATURE OF THE TEST */
{
    ...
}
```

Instead of

```
if (!count)
{
    ...
}
```

26. "goto"s could be used only under error/exception conditions and that too only in forward direction. However, it is to be generally avoided.
27. Labels should be used only with goto. Labels should be placed at the first indentation.
28. If some piece of code is deleted or commented while modifying the file, the identifiers, which become unused must be deleted or commented respectively.
29. Check the side effects of ++ and --.
30. Use "," as a statement separator only when necessary.
31. Use ternary operator only in simple cases.
32. Repeat array size in the array declarations if the array was defined with size. E.g.

```
main()
{
    int arr[10];

    .....
    func(arr);
}

int
func(
    int passed_arr[10])
{
    .....
}
```

33. Initialize all the global variable definitions.
34. Do not assume the value of uninitialized variables.
35. Functions and variables used only in a particular file should be declared as static in that file.
36. Prototype a function before it is used.
37. Declare a Boolean type "bool" in a global include file. e.g.

```
typedef int bool;
#define FALSE 0
#define TRUE 1
```

Instead of checking equality with TRUE, check inequality with FALSE. This is because FALSE is always guaranteed to have a unique value, zero but this is not the case with TRUE. e.g.

Use

```
if (f() != FALSE)
```

instead of

```
if (f() == TRUE)
```

38. No tab characters should be saved in the file.

39. Data declarations should be written vertically aligned w.r.t. data types and identifier names both.
e.g.

```
int      count;
struct item_t item;
char    *buffer;
```

White Spaces:

- i) Add space before and after all the binary operators except "." and ">" for structure members.
- ii) Unary operators should not be separated from their single operand.
- iii) Horizontal and vertical spacing is equally important.
- iv) After each keyword one space is required.
- v) Long statements should be broken on different lines, splitting out logically separate parts and continue on the next line with an indentation of 4 spaces from the first line.

e.g.

Use

```
if (next_p == (node_p) NULL
&& count < now && now <= MAXALLOT)
{
    .....
}
```

instead of

```
if (next_p == (node_p) NULL && count < now
&& now <= MAXALLOT)
{
    .....
}
```

- vi) Use angle brackets to include a system header file
 - Angle brackets should be used to include system header files.
`#include <someFile.h>`
- vii) Use quotes to include a user defined header file
 - Quotes should be used to include user header files.
`#include "someFile.h"`
- viii) Every "if" statement written with the use of {} braces is the best practice. It will ensure the maintainability and readability of the code.

```
// Method 1
if (condition)
    function1(a, b);
for (i < 7 && j > 8 || k == 4)
    function2(i);
```

In Method1, naming conventions is not followed and {} braces is not used.

```
// Method 2
if (condition)
{
    get_details(emp_id, name);
}
for ((i < 7) && ((j < 8) || (k == 4)))
{
    display_info(emp_id);
}
```

In Method2, We have named functions using a verb (get, display), used variable names that describe the data they contain, and used brackets to help show what the for condition is. Not only that, but we have also included the braces for the control structures and used indentation to show which code appears under each structure.

Comments:

Comments are intended to help the reader understand the code. It is necessary to document source code. The following rules should be followed to achieve this:

- Keep code and comments visually separate.
- Use header comments for all files and functions.
- Use block comments regularly. Use trailing comments for special items.

File Header:

Each file shall begin with a File Header. The following file header template shall be used:

```
*****<File
Header>*****
/*!
@file <filename>
@brief Abstract: <Abstract>
[There must always be one line between the abstract & description]
>Description: <Description>
@note <Note>
@todo <Todo>
```

```
Author :<author>
Date: <date>
Change History : <change history>
*/
//*********************************************************************  
**/
```

Example:

```
//*********************************************************************<File
Header>*****  
/*!
@file SystemManager.c
@brief Abstract: This file contains the implementation of the SystemManager.

Description: The SystemManager provides a generalized mechanism to report
various system conditions.
@note Refer to System Manager Macros section in the FTS Software Coding Standards
for more information about when to use the System Manager.
@todo Unit test and review the code.
*/
//*********************************************************************  
**/
```

Function/Method Headers

Each function or method shall be preceded by a function header. The following function header template shall be used:

```
//*********************************************************************<Function
Header>*****  
/*!
@fn <FunctionName>
@brief Abstract: <Abstract>
[There must always be one line between the abstract & description]
Description: <Description>
@param <Parameter> <ParameterDescription>
@return <Return>
@remarks <Remark>
@warning <Warning>
@see <SeeAlso>
*/
//*********************************************************************  
**/
```

Example:

```
*****<Function
Header>*****
/*!
@fn factorial
@brief Abstract: This method accepts a limit number, and prints the factorial of a number
@param limit Limit of the factorial number.
@return This method returns the result of the factorial of a number
@warning **NOT_UNIT_TESTED**
*/
*****
```

Structure Header

- 1) Each structure and union shall have a header.
- 2) The following class header template shall be used:

```
*****<Structure
Header>*****
/*!
@struct <structurename>
@brief Abstract: <Abstract>
[There must always be one line between the abstract & description]
Description: <Description>
@remarks <Remarks>
@warning <Warning>
@see <SeeAlso>
*/
*****
```

You can substitute the keyword union instead of struct for union header

Appendix III - Table of Figures

| | |
|-----------------------------|----|
| Figure 2: Step Over | 35 |
| Figure 6: Breakpoints | 39 |