Elementary Programming with C

Elementary Programming with C
Trainer's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2014

Aptech®
Unleash your potential

Aptech™
COMPUTER EDUCATION
Unleash your potential

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to fin the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profil Survey is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profil (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibilit is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of Web based learning –acquiring information, organizing information and acting on the basis of insufficien information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as define by the learning objectives.

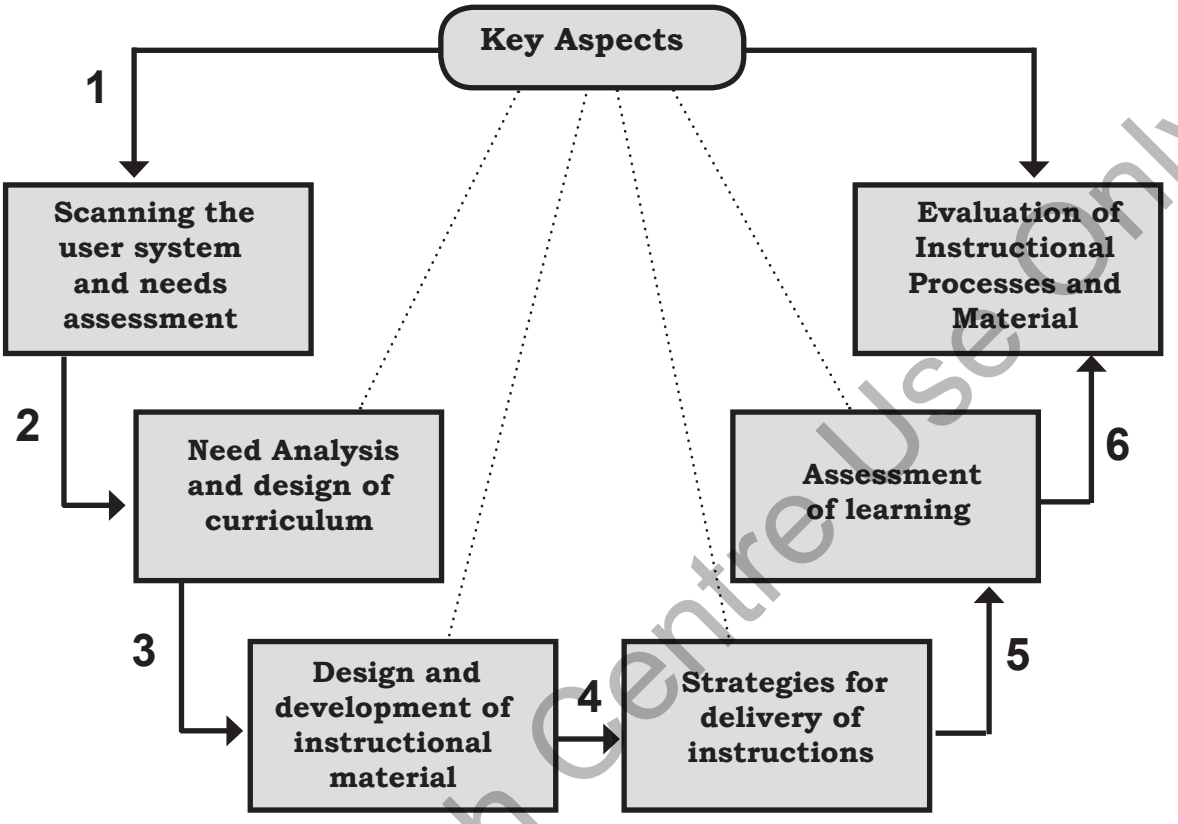
- Evaluation of instructional process and instructional materials

The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model




Any expansion is life
all contraction is death
For Aptech Centre Use Only

Preface

The Trainer's Guide for **Elementary Programming with C** begins with explaining the basic concept of a program, including variable, data types, and expressions. The Trainer's Guide then explains the basic flow control of a C program. It then explains arrays, structures, pointers, basic arithmetic and other basic data structures. Finally, the Trainer's Guide concludes with a description of file handling concepts.

The faculty/trainer should teach the concepts in the theory class using the slides. This Trainer's Guide will provide guidance on the flow of the session and also provide tips and additional examples wherever necessary. The trainer can ask questions to make the session interactive and also to test the understanding of the students.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team


Learning how to learn is
life's most important skill
For Aptech Centre Use Only

Table of Contents

Sessions

Knowing is not enough

we must apply;

Willing is not enough,

we must do

Session 1 - Basics of C

Note: This TG maps to Session 1 of the book.

1.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

1.1.1 Objectives

By the end of this session, the learners will be able to:

- Differentiate between Command, Program, and Software
- Explain the beginning of C
- Explain when and why is C used
- Discuss the C program structure
- Discuss algorithms
- Draw flowcharts
- List the symbols used in flowcharts

1.1.2 Teaching Skills

You should be well-versed in Programming concepts and logic building for teaching this session. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

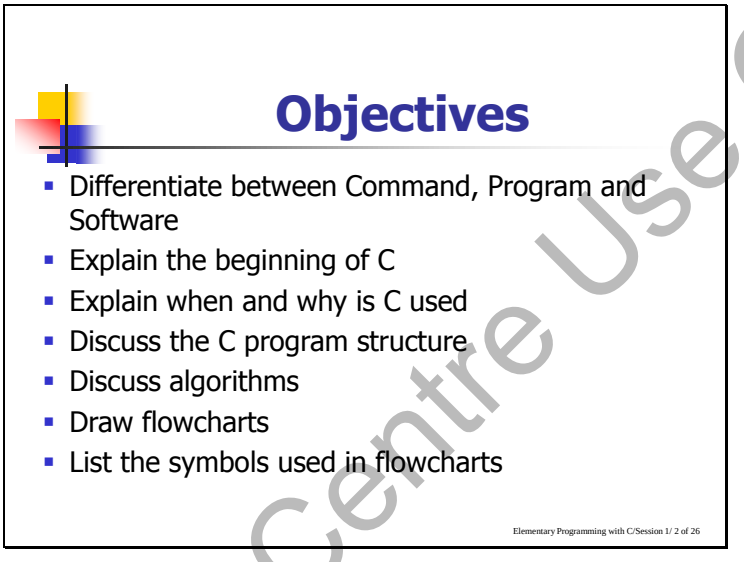
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to make the students understand the difference between software, program, and instruction. Emphasize that software is classified into two types: System and Application. Then explain that programmers use high-level languages to create software. The session then introduces the students to C programming language. It also discusses algorithms and flowcharts; using these, the students will be able to develop the logic underlying the program.

1.2 In-Class Explanations

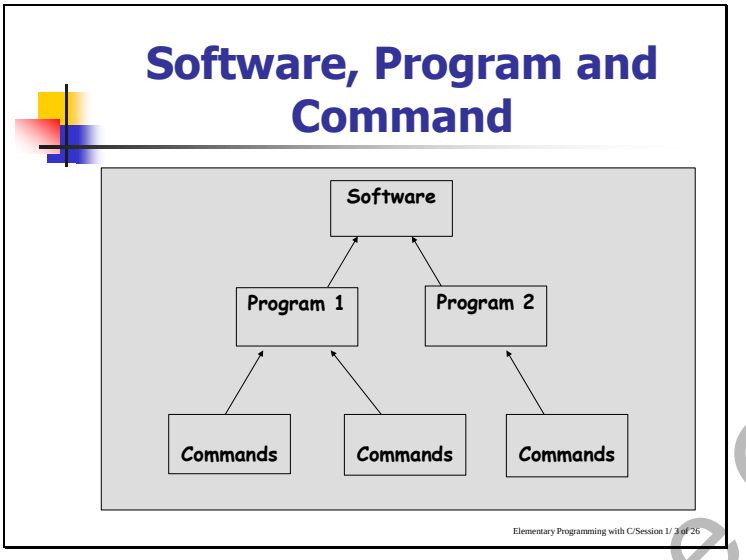
Slide-2



Max Time-2 Minutes

Subject: You should list out the objectives as given in the slide and inform the students what they would be learning in the session.

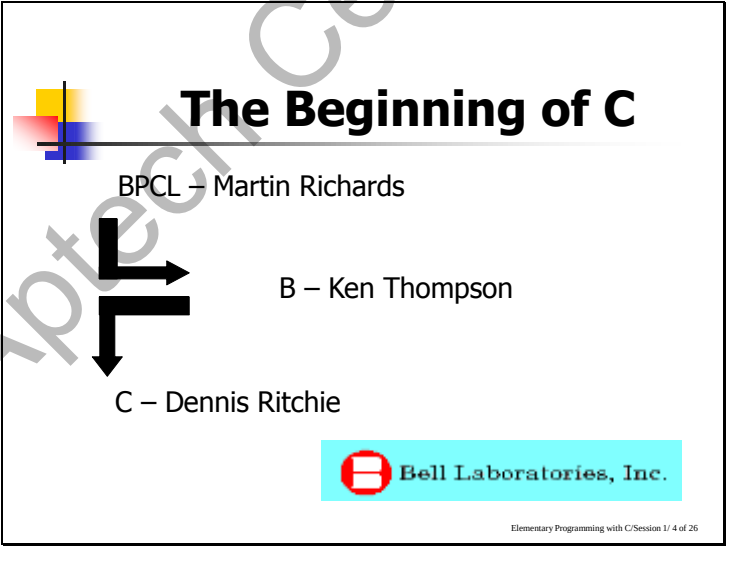
Slide-3



Max Time-5 Minutes

Subject: The focus of this section is to make the students understand the difference between software, program, and instruction. Emphasize that software is classified into two types: System and Application. Then explain that programmers use high-level languages to create software. End the explanation for this slide by highlighting that C is one such language, which is the most preferred high level programming language.

Slides-4, 5, 6, and 7



Application Areas Of C

- C was initially used for systems programming
- A system program forms a portion of the operating system of the computer or its support utilities
- Operating Systems, Interpreters, Editors, Assembly programs are usually called system programs
- The UNIX operating system was developed using C
- There are C compilers available for almost all types of PC's

Elementary Programming with C/Session 1/ 5 of 26

Middle Level Language

High Level Language

C

Assembly Language

Elementary Programming with C/Session 1/ 6 of 26

Structured Language

- C allows compartmentalization of code and data
- It refers to the ability to sectic off and hide all information a instructions, necessary to perform a specific task, from rest of the program

```
do
{
    i = i + 1;
    .
    .
    .
} while (i < 40);
```

- Code can be compartmentalized in C by using functions or code blocks.

Elementary Programming with C/Session 1/ 7 of 26

Max Time: 10 Minutes

Subject: Computer understands 0's and 1's only. However, it is difficult for a human to write programs in 0's and 1's. Hence the high level languages were invented which are very much English like. This makes it easy for the programmers to code their logic in these languages. C is one such language that has the power of both High level and Assembly language.

Additional Information:

In addition, you can use the additional material given here to explain the concept.

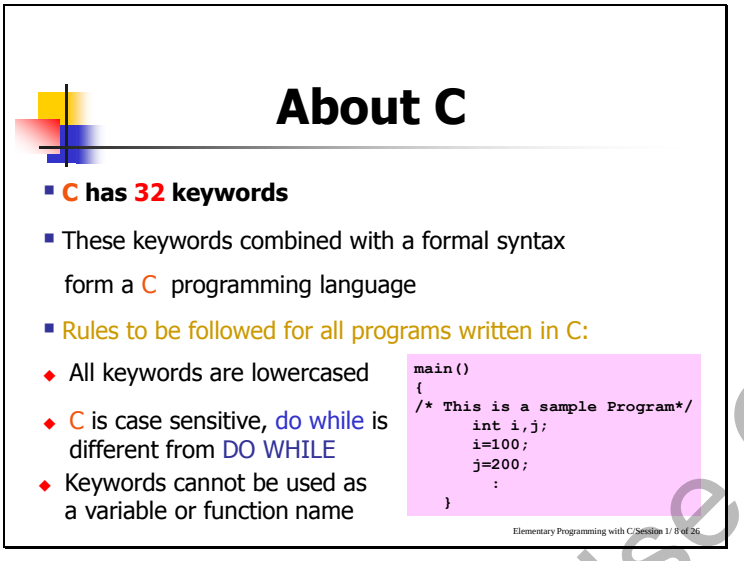
"C was originally developed by Dennis Ritchie of the Bell Laboratories in 1972. The language was named "C" because it was the successor to a language named "B"(no lasting fame, however). C was developed as a high level language that could be used to rewrite the UNIX operating system. Today, most of the UNIX operating system and most of the programs that run under UNIX are written in C."
-C Programming Language Earl L. Adams 1992

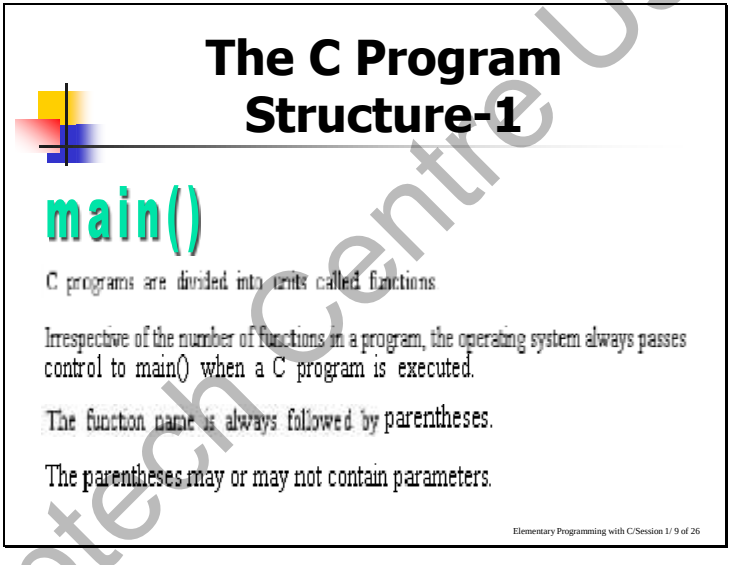
Benefits of C

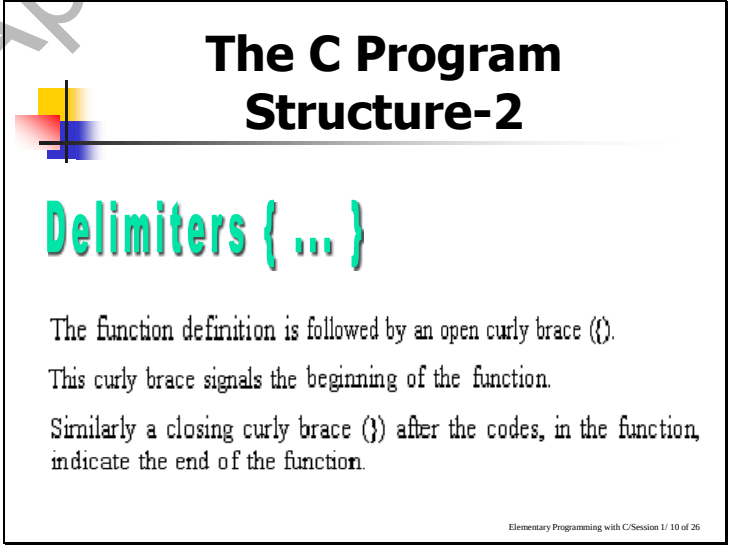
C has an extensive library for mathematical computations, character analysis, input and output functions, hardware structure, and graphics. While some functions are used more than others, they are all offered and can be used by the best programmers. Why write a code to find square roots or alter strings, when there are "header" files that can be used to call these special functions. C is also a relatively easy language to learn, so that you can also write some programs for your everyday life. Yes, there are more advance programs, but it is easy enough to learn to write simple programs.

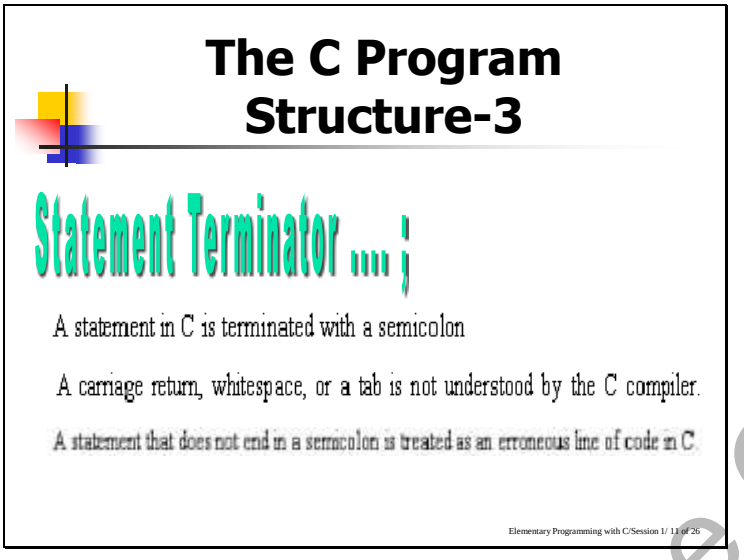
Did I mention that programmers were also starting at 40K a year, right after college; programmers can make a lot of money doing what they do best and what they enjoy. I would not advise anyone to go into programming if they did not really have a love for the science. At times it is very time demanding and it can really stress you out! There are many fields programmers can go into, from business applications, to video games, to Hollywood; programmers can be found providing the essential computer work for companies everywhere.

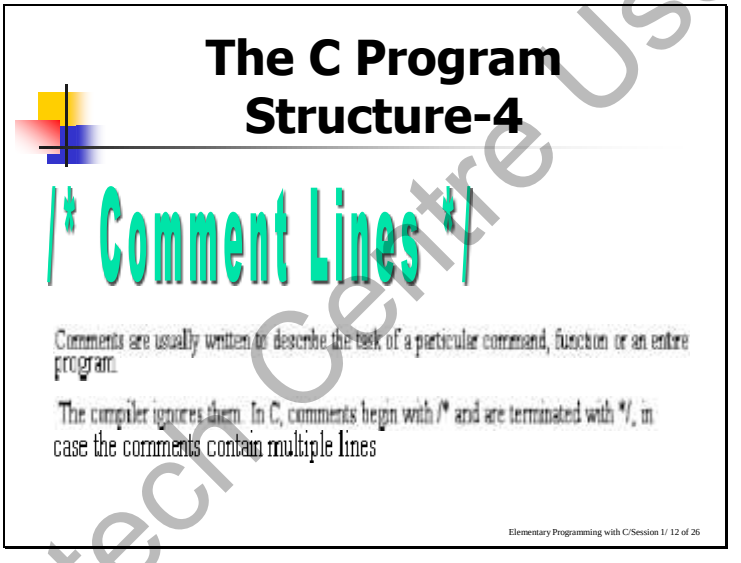
Slides-8, 9, 10, 11, and 12











Max Time: 10 Minutes

Subject: This section explains the structure of a C program in detail.

1. The main function has to be written as shown here:

```
main ()
{
…
…
}
```

2. Every block of code is enclosed between a set of curl parenthesis in C, {}.

3. Every statement in C has to end with a semicolon (;) as it is the statement terminator in C.

4. Comments in C can be of single line or multiple lines. // is used to indicate the single line of comment. The multiple lines of comment can be enclosed within /* and */

For example:

```
// This is a single line comment
/* multiple lines of comments
  are given here */
```

Additional Information:

You can use the following additional material:

Source:

http://www.cs.columbia.edu/~aya/W3101-01/lectures/lec1/tsld011.htm

Standard C Program Structure

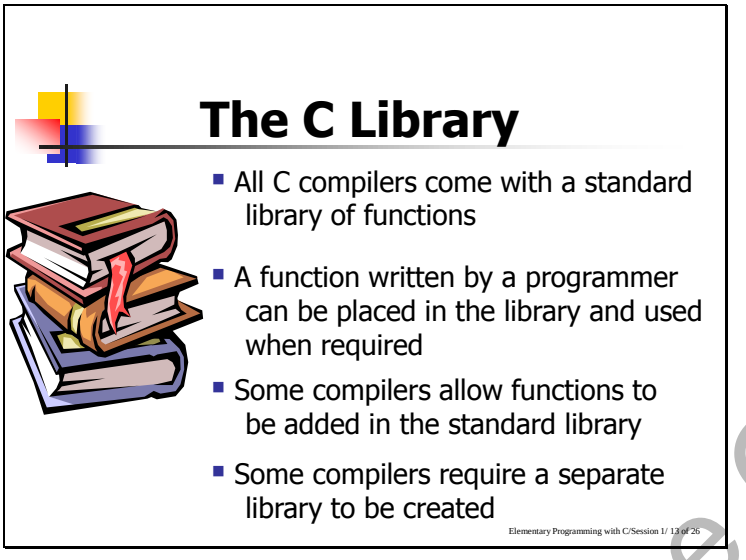
A simple C program consists of the function main()
The body of this function consists of declaration and execution statements enclosed within braces { and }

```
preprocessing directives
main() {
  declarations;
  execution statements;
}
```

C - Program Structure

- Every C program has a main() function which is an entry point: main() { printf("Hello everybody\n"); }
- Program is built out of blocks {…}
- Building blocks can also be functions, loops.
- Functions have: - arguments (optional) - return value (optional) - statements - variables (optional)
- Example: `main() { int j = 1; j = 8+1; }`

Slide-13



Max Time: 5 Minutes

Subject: C has a huge function library that can be included into the program using the include statement. For example, the standard input output functions are stored in the stdio.h header file. A header file in C is a file that contains certain functions, which are grouped, based on the usage. When included inside a C program, the C compiler will include the definition of the function used in the program into the program itself. Thus, the contents of the function are attached during compile time.

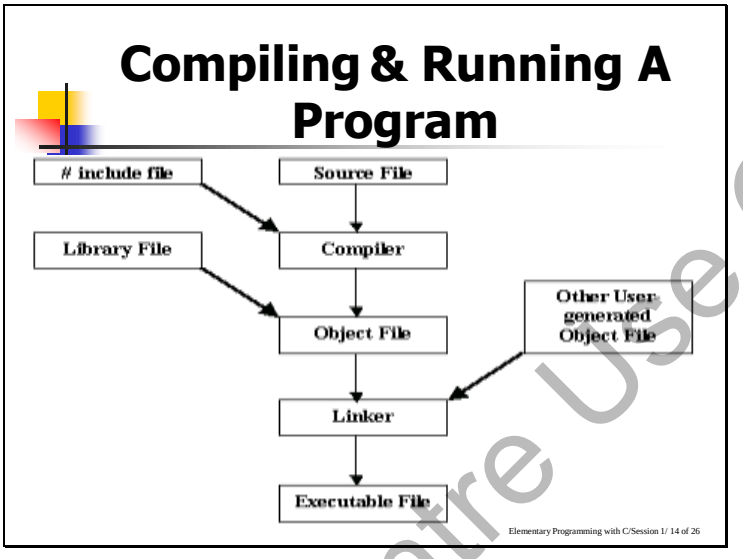
The functions in C are grouped in various groups. To name few, string.h contains the string functions; stdlib.h contains the standard library functions; math.h contains the mathematical functions and so on.

The format of the include statement is shown here:

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

Explain the concept using the given material.

Slide-14



Max Time: 10 Minutes

Subject: Programs written in high-level languages are either interpreted or compiled to output executable code. This executable code can then be executed to get the output.

Make use of the additional material given here to explain the same to the students.

Additional Information:

Source: http://www.cs.cf.ac.uk/Dave/C/node3.html#SECTION00320000000000000000

The C Compilation Model
We will briefly highlight key features of the C Compilation model (Figure1.1) here.

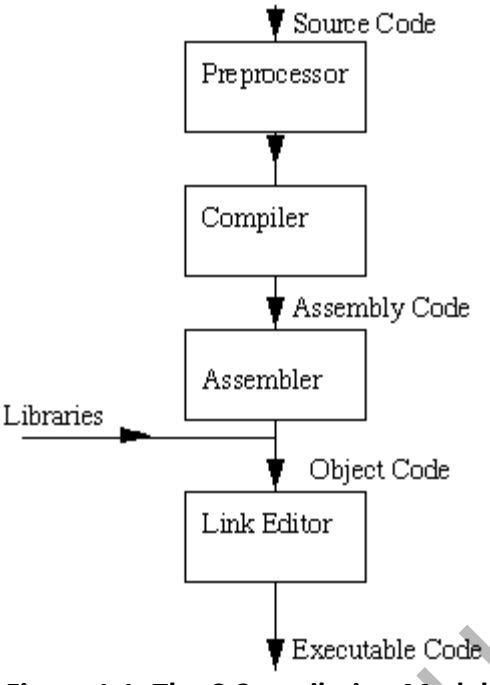


Figure 1.1: The C Compilation Model

The Preprocessor

We will study this part of the compilation process in detail later. However, we need some basic information for some C programs.

The Preprocessor accepts source code as input and is responsible for,

- Removing comments
- Interpreting special ***preprocessor directives*** denoted by `#`.
  For example
- `#include` -- includes contents of a named file. Files usually called ***header*** files. ***e.g***
  - `#include <math.h>` -- standard library maths file
  - `#include <stdio.h>` -- standard library I/O file
- `#define` -- defines a symbolic name or constant. Macro substitution
  - `#define MAX_ARRAY_SIZE 100`

C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler

The assembler creates object code. On a UNIX system you may see files with a `.o` suffix (`.OBJ` on MSDOS) to indicate object code files.

Link Editor

If a source file references library functions or functions defined in other source files the ***link editor*** combines these functions (with `main()`) to create an executable file. External Variable references resolved here also.

Some Useful Compiler Options

Now that we have a basic understanding of the compilation model we can introduce some useful and sometimes essential common compiler options.

`-c`

Suppress the linking process and produce a `.o` file for each source file listed. Several can be subsequently linked by the `cc` command, for example:

```
cc file1.o file2.o ..... -o executable
```

-llibrary

Link with object libraries. This option must follow the source file arguments. The object libraries are archived and can be standard, third party or user created libraries. Probably the most commonly used library is the math library (`math.h`). You must link in this library explicitly if you wish to use the maths functions (**note** do not forget to `#include <math.h>` header file), for example:

```
cc calc.c -o calc -lm
```

-Ldirectory

Add directory to the list of directories containing object-library routines. The linker always looks for standard and other system libraries in `/lib` and `/usr/lib`. If you want to link in libraries that you have created or installed yourself (unless you have certain privileges and get the libraries installed in `/usr/lib`) you **will** have to specify where your files are stored, for example:

```
cc prog.c -L/home/myname/mylibs mylib.a
```

-Ipathname

Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash /).

By default, the preprocessor first searches for #include files in the directory containing source file, then in directories named with -I options (if any), and finally, in /usr/include. So to include header files stored in `/home/myname/myheaders` you would do:

```
cc prog.c -I/home/myname/myheaders
```

Note: System library header files are stored in a special place (`/usr/include`) and are not affected by the `-I option`. System header files and user header files are included in a slightly different manner.

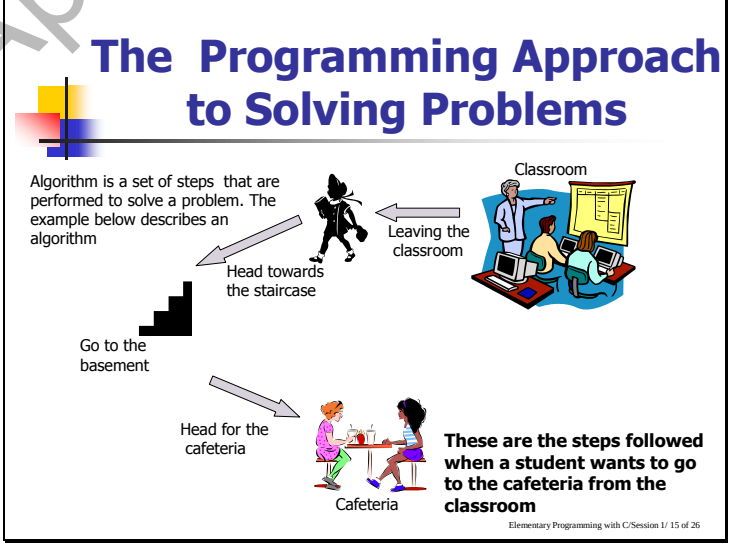
-g

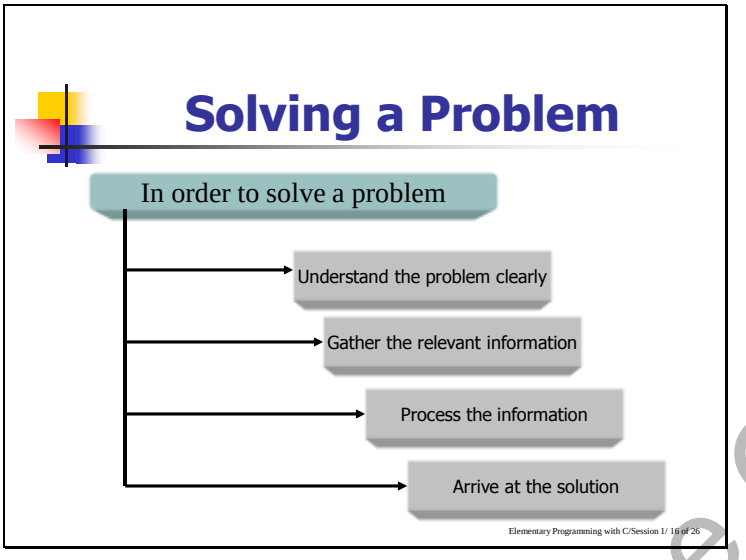
Invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

-D

Define symbols either as identifiers (`-D`***identifer***) or as values (`-D`***symbol=value***) in a similar fashion as the `#define` preprocessor command.

Slides-15 and 16





Max Time: 5 Minutes

Subject: The programming process is a set of activities listed here:

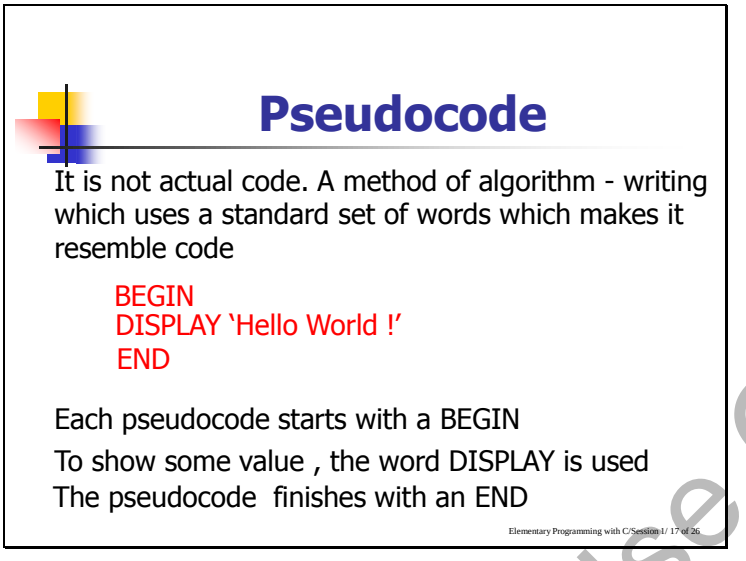
1. Understand Program Specification
2. Design a Program model
3. Determine correctness of the program
4. Code the program
5. Test and debug the program
6. Document the program

Once the problem is clearly defined, an algorithm can be developed. An algorithm can be defined as a definite sequence of steps to solve a problem. Developing an algorithm is the most creative part of programming.

To be useful as a basis for writing program, the algorithm must:

1. Arrive at a correct solution within a finite time
2. Be clear, precise, and unambiguous

Slide-17



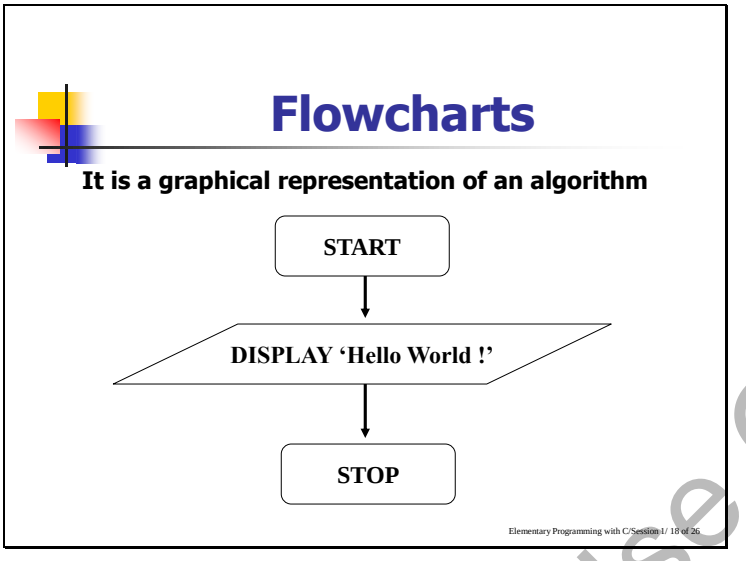
Max Time: 10 Minutes

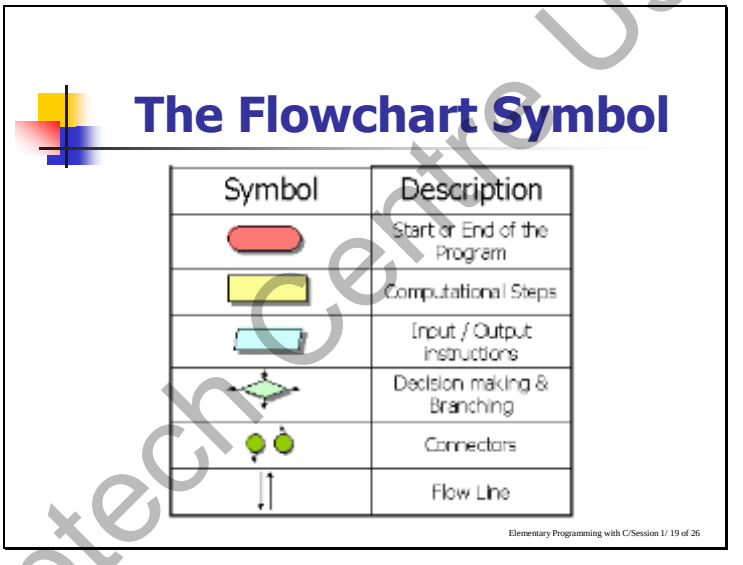
Subject: Pseudo codes mean writing the program logic in a simple English-like language. Pseudo codes use statements that are a bridge between actual programming and ordinary English. In pseudo code each step is written using a simple English phrase which is also called as a construct.

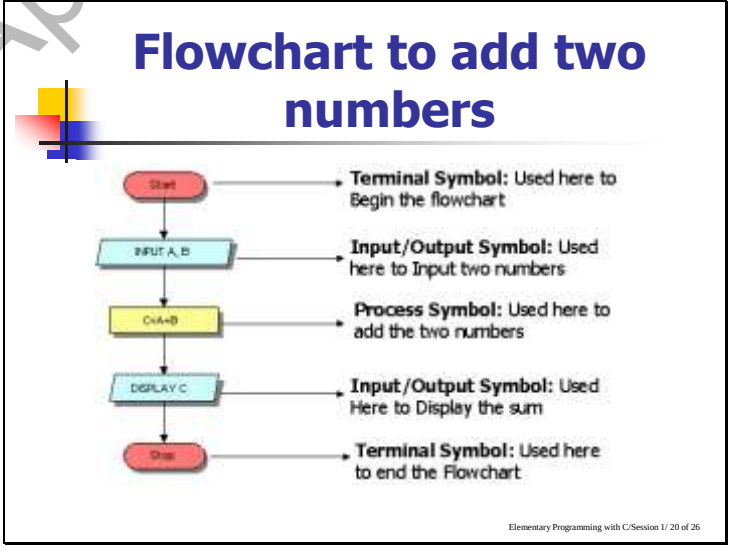
Some of the conventions, which are used while writing pseudo codes, are as follows:

1. All statements in a loop should be indented.
2. All alphanumeric values should be enclosed in single or double quotes.
3. The beginning and end of a pseudo code is marked with keywords like 'start' / 'begin', and 'end' respectively.
4. All statements must include certain key words, which denote an operation.

Slides-18, 19, and 20







Max Time: 15 Minutes

Subject: A flowchart is a pictorial representation of the program logic. You can make use of the additional material given here.

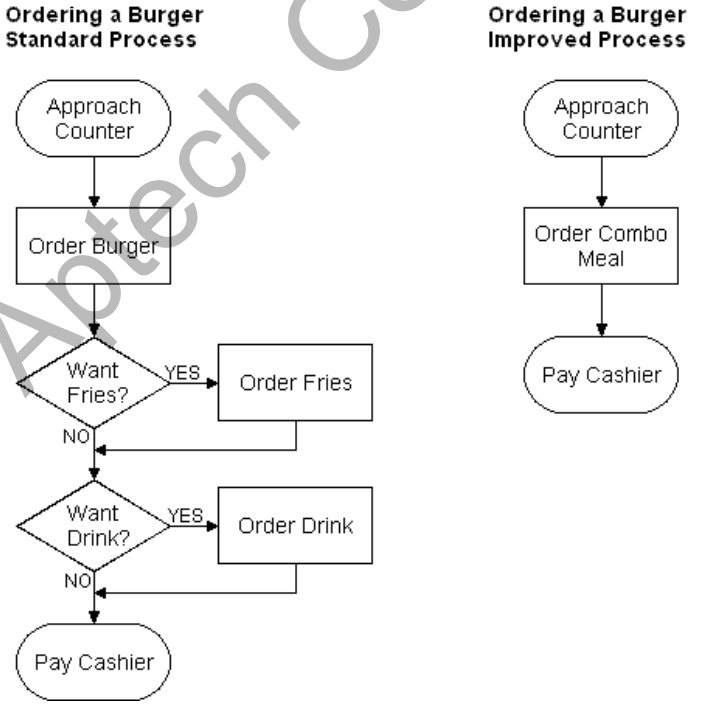
Additional Information:

Source:
http://www.smartdraw.com/articles/flowchart/what-is-flowchart.htm
http://www.smartdraw.com/software/flowchart-types.htm

A flowchart illustrates the steps in a process. By visualizing the process, a flowchart can quickly help identify bottlenecks or inefficiencies where the process can be streamlined or improved.

Example: Two Flowcharts for a Common Process

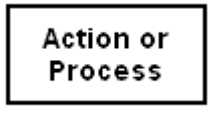
Suppose your research revealed that you always want fries and a drink with your burger. You decide to streamline your process by ordering the combo meal, which automatically includes fries and a drink. The two flowcharts show at a glance that you omit two decisions and two order steps by using the streamlined order process.



Basic Flowcharting Shapes

Flowcharts use special shapes to represent different types of actions or steps in a process. Lines and arrows show the sequence of the steps and the relationships among them.

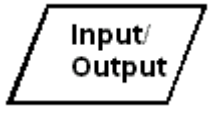
The terminator symbol marks the starting or ending point of the system. It usually contains the word 'Start' or 'End'.



A box can represent a single step ('add two cups of flour') and entire sub-process ('make bread') within a larger process.

A printed document or report.

A decision or branching point. Lines representing different decisions emerge from different points of the diamond.

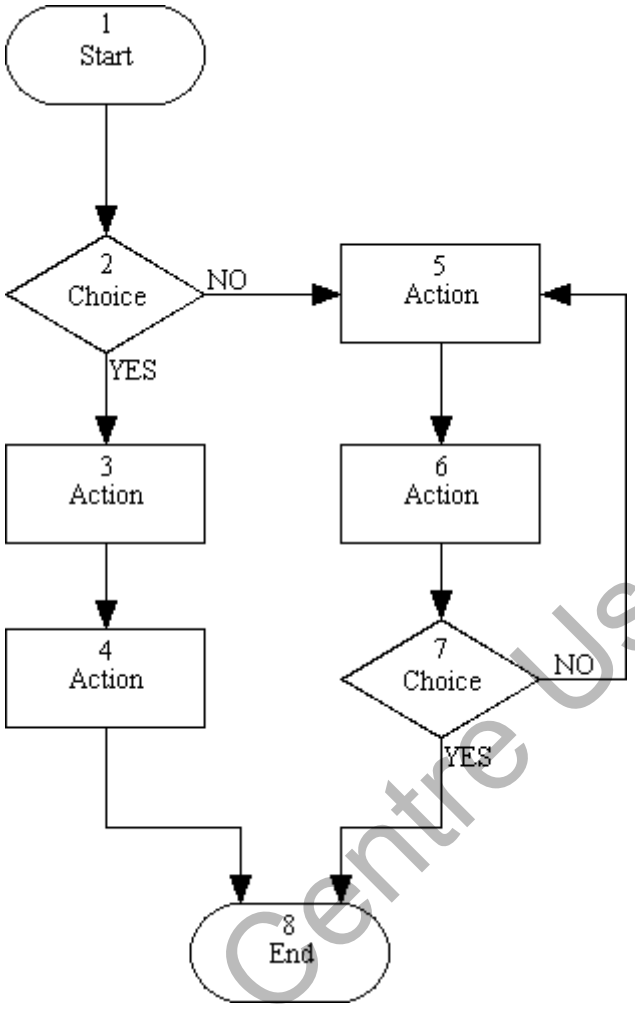


Represents material or information entering or leaving the system, such as customer order (input) or a product (output).

Indicates that the flow continues on another page, where a matching symbol (containing the same letter) has been placed.

Lines indicate the sequence of steps and the direction of flow.

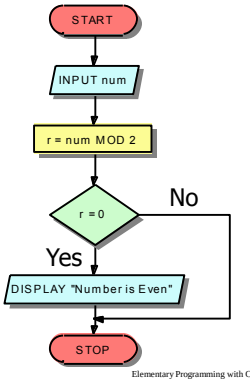
A basic flowchart identifies the starting and ending points of a process, the sequence of actions in the process, and the decision or branching points along the way.



Slides-21, 22, 23, 24, and 25

The IF Construct

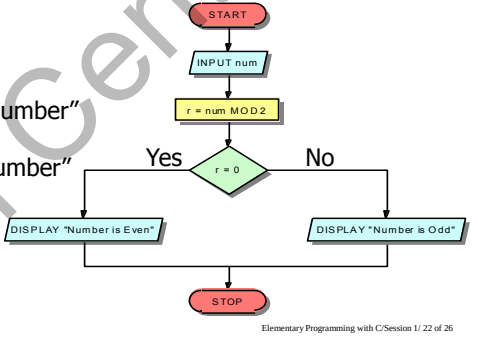
```
BEGIN
INPUT num
r = num MOD 2
IF r=0
Display "Number is even"
END IF
END
```



Elementary Programming with C/Session 1/ 21 of 26

The IF-ELSE Construct

```
BEGIN
INPUT num
r=num MOD 2
IF r=0
  DISPLAY "Even Number"
ELSE
  DISPLAY "Odd Number"
END IF
END
```



Elementary Programming with C/Session 1/ 22 of 26

Multiple criteria using AND/OR

```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10 AND bizDone >=5000000
        DISPLAY "Classified as an MVS"
ELSE
        DISPLAY "A little more effort required!"
END IF
END
```

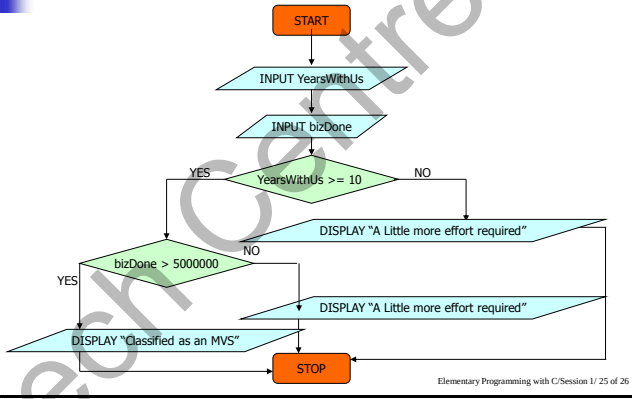
Elementary Programming with C/Session 1/ 23 of 26

Nested IFs-1

```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10
IF bizDone >=5000000
        DISPLAY "Classified as an MVS"
          ELSE
                DISPLAY "A little more effort required!"
END IF
ELSE
        DISPLAY "A little more effort required!"
END IF
END
```

Elementary Programming with C/Session 1/ 24 of 26

Nested IFs-2



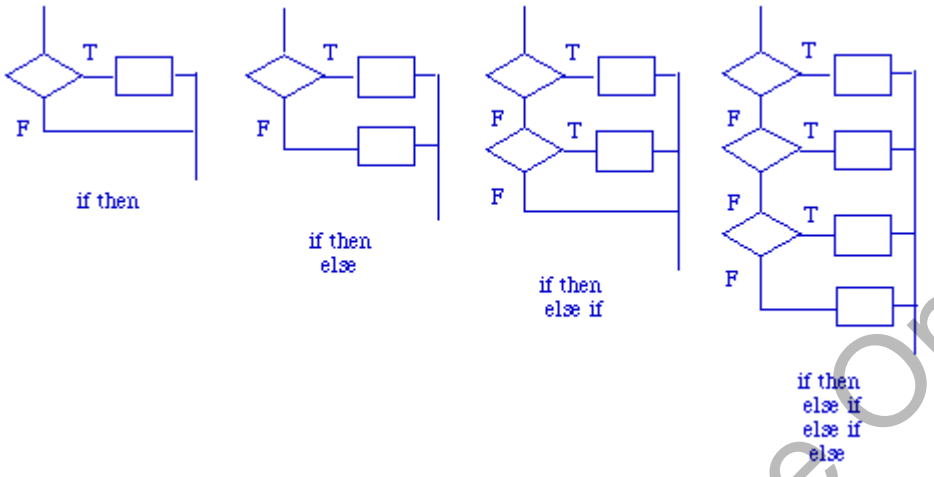
Elementary Programming with C/Session 1/ 25 of 26

Max Time: 20 Minutes

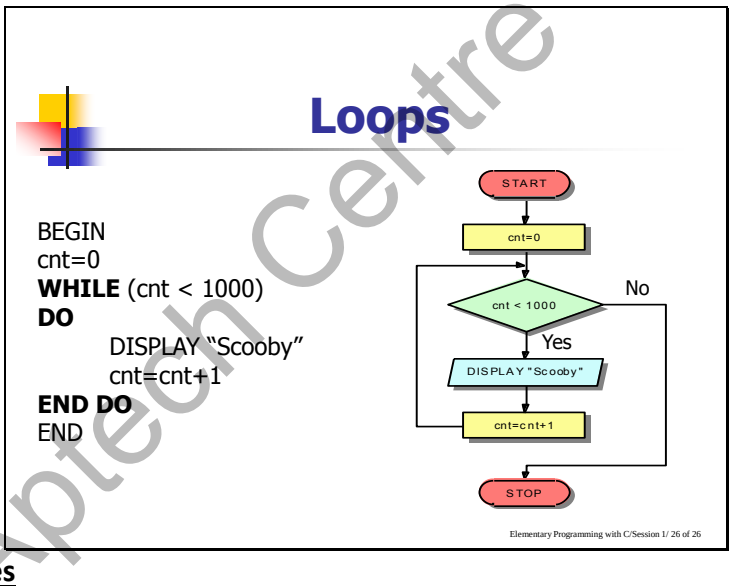
Subject: In flowcharts the decision making symbol is used to depict the diversion of paths. In pseudo codes the selection construct is used for the same purpose.

The `IF` construct may be used to select for execution at most one true block from one or more in the construct. Selection of a path is based on the value of one or more conditions. It permits several paths of control flow.

Additional Information:



Slide-26



Max Time: 10 Minutes

Subject: The constructs we saw until now will perform a set of steps only once. However, in real life we may have to repeat certain steps for a particular number of times. For example, if we have to calculate the average mark of 100 students, the sum of marks of 100 students needs to be calculated. We need to accept marks of the 100 students and add them. That is the same process of accepting a student's mark and adding it to a grand sum needs to be repeated 100 times. This is called iteration/looping.

There are various iteration constructs as listed here. The difference between each of them needs to be explained to the students.

Additional Information:
Source:
http://www.strath.ac.uk/IT/Docs/Ccourse/subsection3_8_3.html#SECTION00083000000000000000

C gives you a choice of three types of loop, while, do while, and for.

- The '**while**' loop keeps repeating an action until an associated test returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The '**do while**' loops are similar, but the test occurs after the loop body is executed. This ensures that the loop body is run at least once.
- The '**for**' loop is frequently used, usually where the loop will be traversed a fixed number of times. It is very flexible and novice programmers should take care not to abuse the power it offers.

Solutions to Check Your Progress

1. compartmentalization
2. flowchart
3. True
4. False
5. Looping or iterative construct

Solutions to Do It Yourself

Solution 1:

```
BEGIN
INPUT C
F=1.8*C + 32
DISPLAY F
END
```

Solution 2:

```
BEGIN
INPUT PHYSICS_MARKS
INPUT CHEMISTRY_MARKS
INPUT PHYSICS_MARKS
INPUT BIOLOGY_MARKS
TOTAL_MARKS= CHEMISTRY_MARKS + PHYSICAL_MARKS+BIOLOGY_MARKS
DISPLAY TOTAL_MARKS
END
```

1.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 2-Variables and Data Types

Note: This TG maps to Session 2 of the book.

2.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. The session introduces the students to the different data types and arithmetic operators available in C. It also deals with how to create and use the variables in a C program.

2.1.1 Objectives

By the end of this session, the learners will be able to:

- Discuss variables
- Differentiate between variables and constants
- List the different data types and make use of them in C programs
- Discuss arithmetic operators

Difficulties

The students may find difficulty in understanding the following topics:

- What is a variable
- Usage of data type with variable
- Difference between variable and a constant

Hence, the mentioned topic should be handled carefully.

2.1.2 Teaching Skills

You should be well-versed in C Programming basics including variables, data types, and operators for teaching this session. The session is to be taught in a T/L Format. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

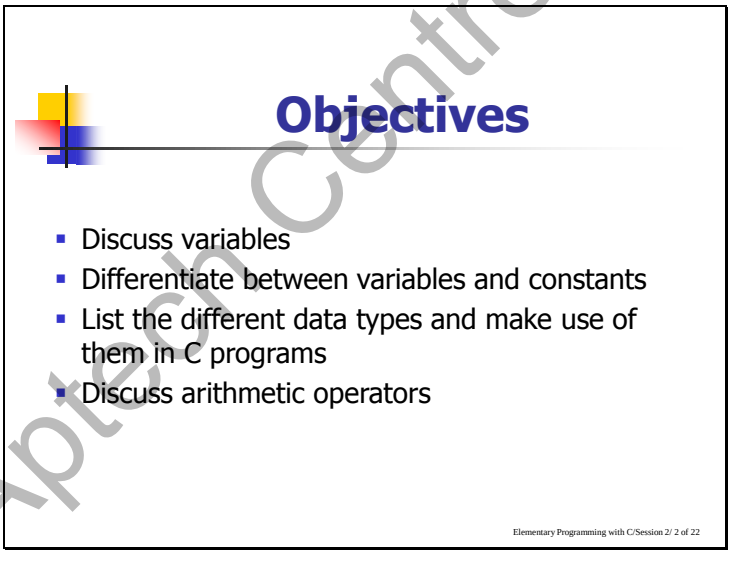
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to make the students understand the concept of variables, different data types, and arithmetic operators available in C. The session also demonstrates with ample examples how to create and use the variables in a C program.

2.2 In-Class Explanations

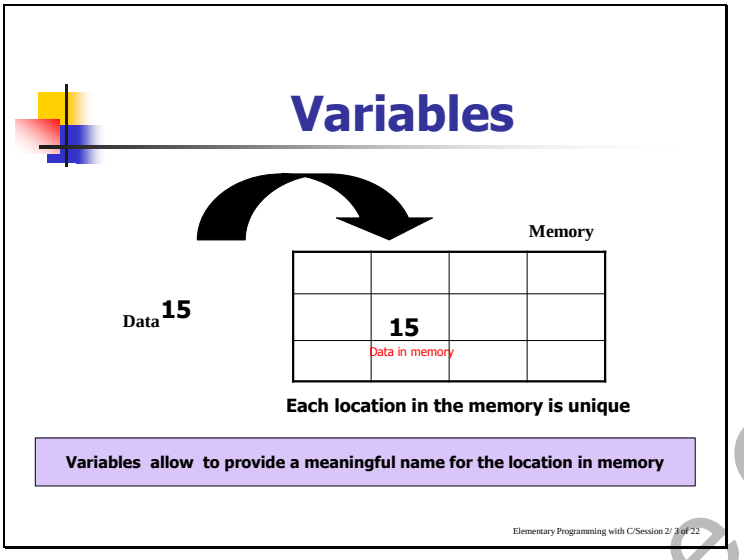
Slide 2

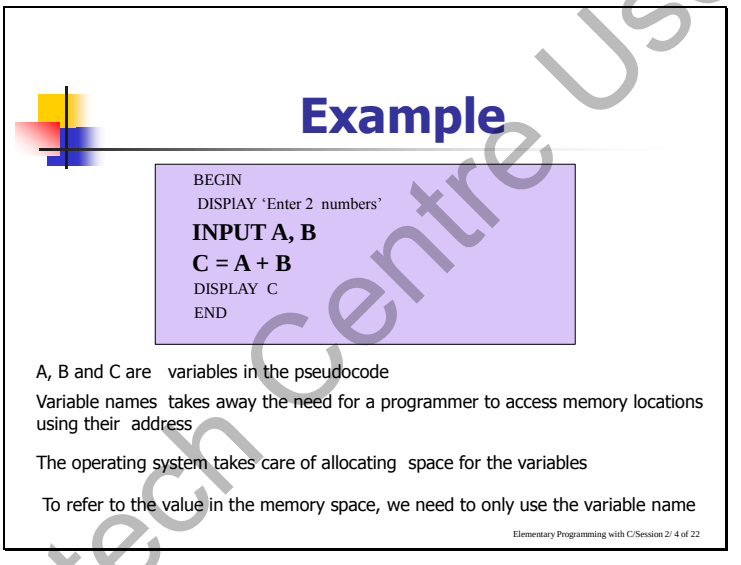


Max Time: 4 Minutes

Subject: List out the objectives as given in the slide and inform the students what they would be learning in the session.

Slides-3 and 4

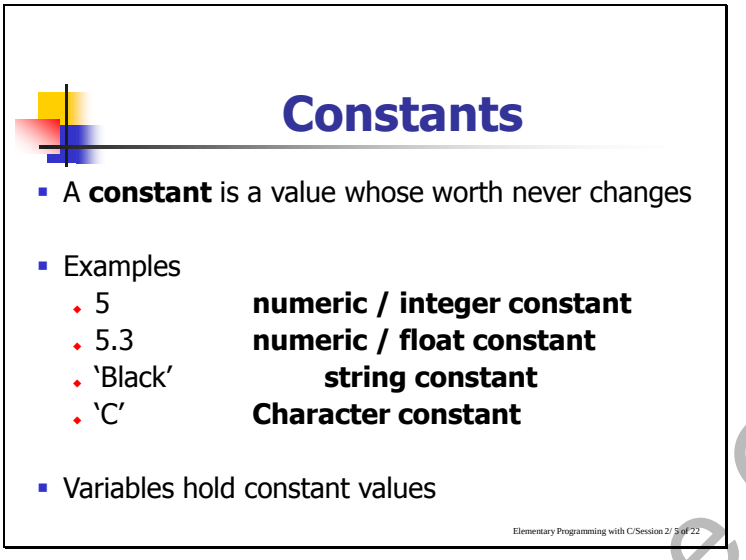




Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students for variables. Define variable and explain its importance. You should clarify the term variable with the example given in the slide. This section should be covered in 4-5 minutes.

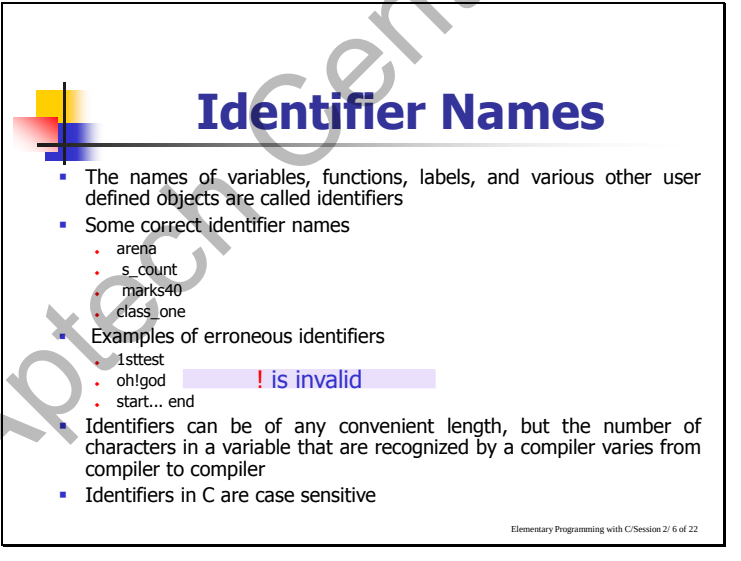
Slide-5

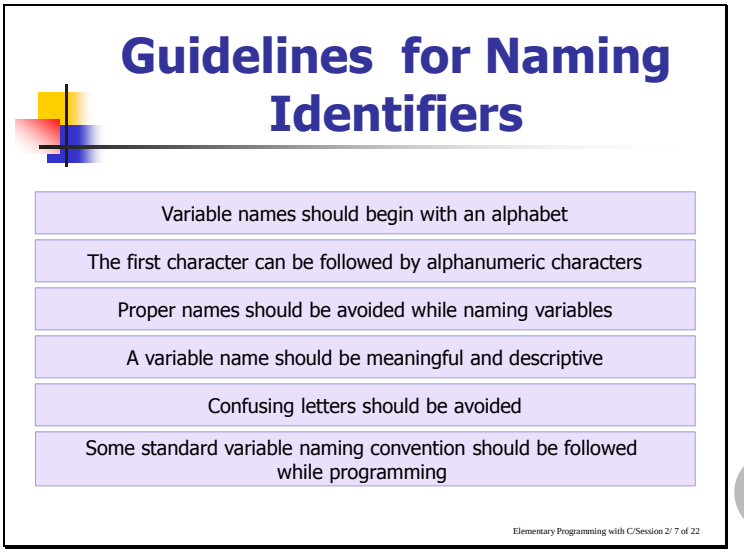


Max Time: 6 Minutes

Subject: The teaching of the session should start with the answers given by the students for constants. You should define constant and explain its importance. Clarify the term constant with the example given in the slide.

Slides-6 and 7

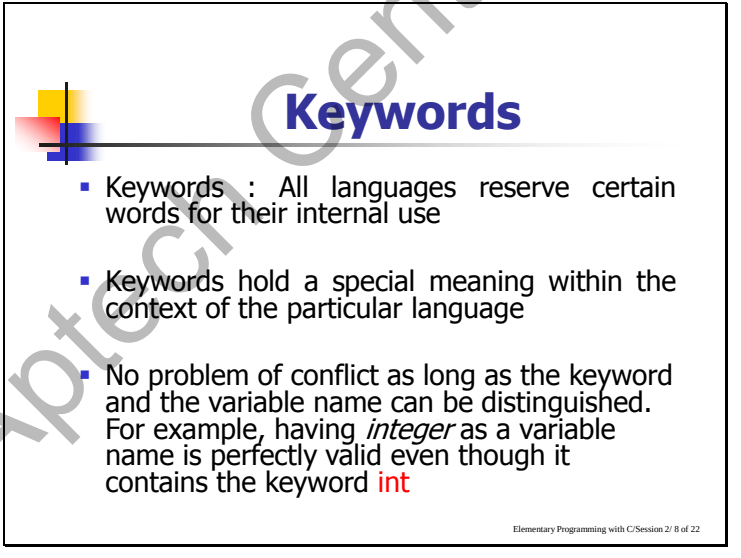




Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students for identifiers. Define identifier and explain its importance. You should clarify the term variable with the example given in the slide. Also explain the naming guidelines defined for identifiers. This section should be covered in 3-4 minutes.

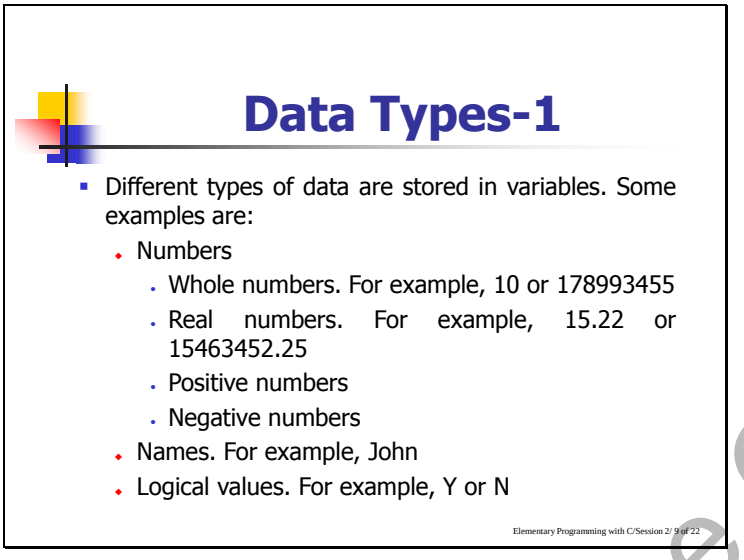
Slide-8

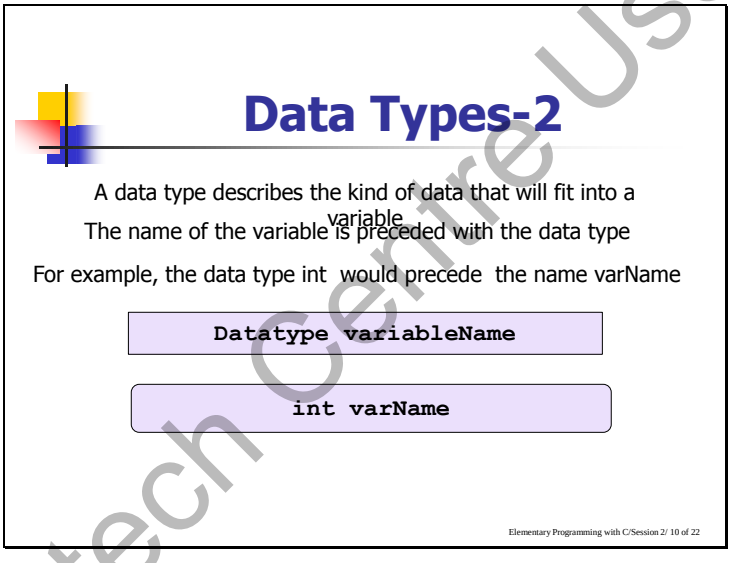


Max Time: 6 Minutes

Subject: The teaching of the session should start with the answers given by the students for keyword. You should define keyword and its usage.

Slides-9 and 10





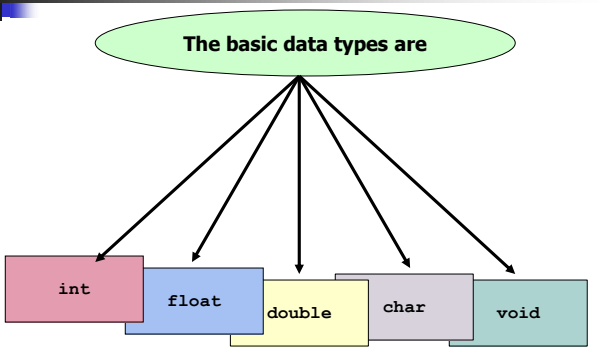
Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students for data types. You should define data type and explain its importance in the life of a variable. You should clarify how to define variable of a particular data type with the example given in the slide. This section should be covered in 3-4 minutes.

Slides-11, 12, 13, 14, 15, and 16

Basic Data Types

The basic data types are



Elementary Programming with C/Session 2/ 11 of 22

Type int

- Stores numeric data

```
int num;
```

- Cannot then store any other type of data like "Alan" or "abc"
- 16 bits (2 bytes)
- Integers in the range -32768 to 32767
- Examples: 12322, 0, -232

Elementary Programming with C/Session 2/ 12 of 22

Type float

- Stores values containing decimal places

```
float num;
```

- Precision of upto 6 digits
- 32 bits (4 bytes) of memory
- Examples: 23.05, 56.5, 32

Elementary Programming with C/Session 2/ 13 of 22

Type double

- Stores values containing decimal places

```
double num;
```

- Precision of upto 10 digits
- 64 bits (8 bytes) of memory
- Examples: 'a', 'm', '$' '%' , '1', '5'

Elementary Programming with C/Session 2/ 14 of 22

Type char

- Stores a single character of information

```
char gender;
gender='M';
```

- 8 bits (1 byte) of memory
- Examples: 'a', 'm', '$' '%' , '1', '5'

Elementary Programming with C/Session 2/ 15 of 22

Type void

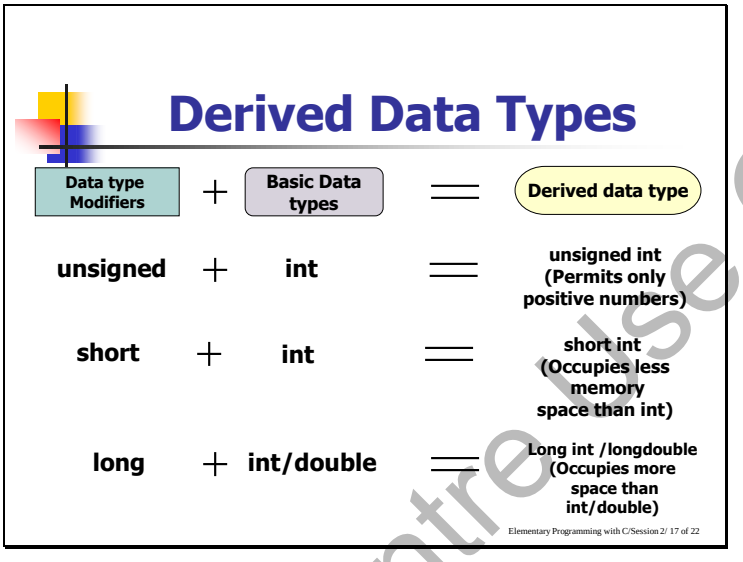
- Stores nothing
- Indicates the compiler that there is nothing to expect

Elementary Programming with C/Session 2/ 16 of 22

Max Time: 30 Minutes

Subject: The teaching of the session should start with the answers given by the students for different data types available in C. Define each basic data type and explain its limitation. Clarify when to use which kind of data type. This section should be covered in 7-8 minutes.

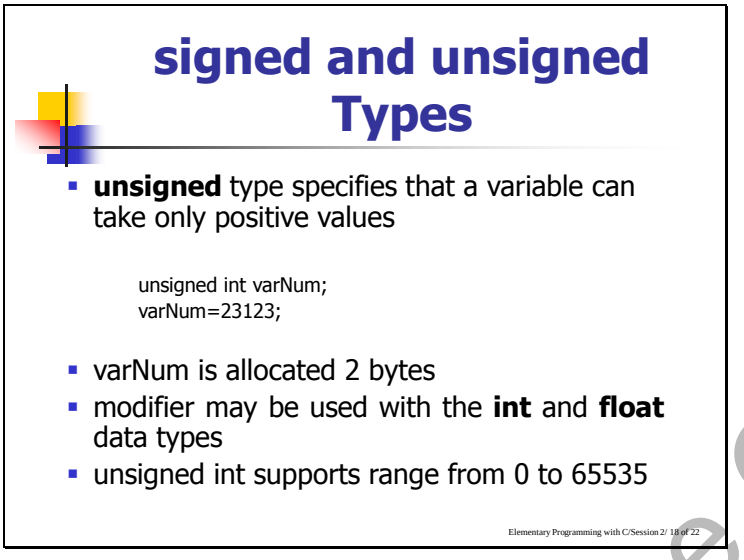
Slide-17

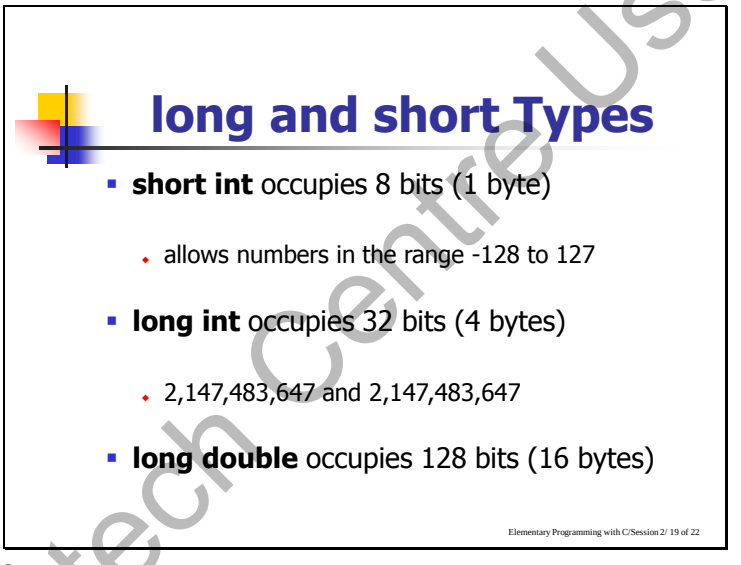


Max Time: 6 Minutes

Subject: The teaching of the session should start with the answers given by the students for their understanding of derived data types. Define derived data type and explain its importance. You should clarify how to use derived data type.

Slides-18 and 19





Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students about their understanding of derived data types. You should explain different derived data types and explain limitations. This section should be covered in 3-4 minutes.

Slides-20, 21, and 22

Data Types and their range-1

Type	Approximate Size in Bits	Minimal Range
char	8	-128 to 127
unsigned	8	0 to 255
signed char	8	-128 to 127
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
signed int	16	Same as int
short int	16	Same as int
unsigned short int	8	0 to 65, 535

Elementary Programming with C/Session 2/ 20 of 22

Data Types and their range-2

Type	Approximate Size in Bits	Minimal Range
signed short int	8	Same as short int
signed short int	8	Same as short int
long int	32	-2,147,483,647 to 2,147,483,647
signed long int	32	0 to 4,294,967,295
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	128	Ten digits of precision

Elementary Programming with C/Session 2/ 21 of 22

Sample Declaration

```
main ()
{
   char abc;       /*abc of type character */
   int xyz;   /*xyz of type integer */
   float length;  /*length of type float */
   double area;   /*area of type double */
   long liteyrs;  /*liteyrs of type long int */
   short arm;      /*arm of type short integer*/
}
```

Elementary Programming with C/Session 2/ 22 of 22

Max Time: 10 Minutes

Subject: You should tell about the range of different data types used in C and explain its usage. You should clarify how to define variable of a particular data type with the example given in the slide.

Solutions to Check Your Progress

1 True
2 Whole number
3 False
4 Float
5 void
6 Unary and Binary
7 ++ and --

Solutions to Try it yourself

1.

Column A	Column B
8 23	Integer Constants
10.34	Floating Point Numbers
ABC abc _A1	Valid Identifier Names
$abc	Invalid Identifier names
12112134.8686 8686886	Double
'A'	Character Constants

2.

a. a=256

b. a=23

c. a=11
   b=20

d. a=-5
   b=5

2.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the OnlineVarsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

You can also put a few questions to students to search additional information.

Session 3-Operators and Expressions

Note: This TG maps to Session 4 of the book.

3.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

3.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain Assignment Operators
- Understand Arithmetic Expressions
- Explain Relational and Logical Operators
- Understand Bitwise Logical Operators and Expressions
- Explain Casts
- Understand the Precedence of Operators

Difficulties

The students may find difficulty in understanding the following topics:

- Bitwise Logical operators
- Mixed Mode Expressions and Type Conversions
- Casts

Hence, the mentioned topics should be handled carefully.

3.1.2 Teaching Skills

You should be well-versed in C Programming basics such as operators and expressions for teaching this session. The session is to be taught in a T/L Format. The session is to be taught in a T/L Format. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

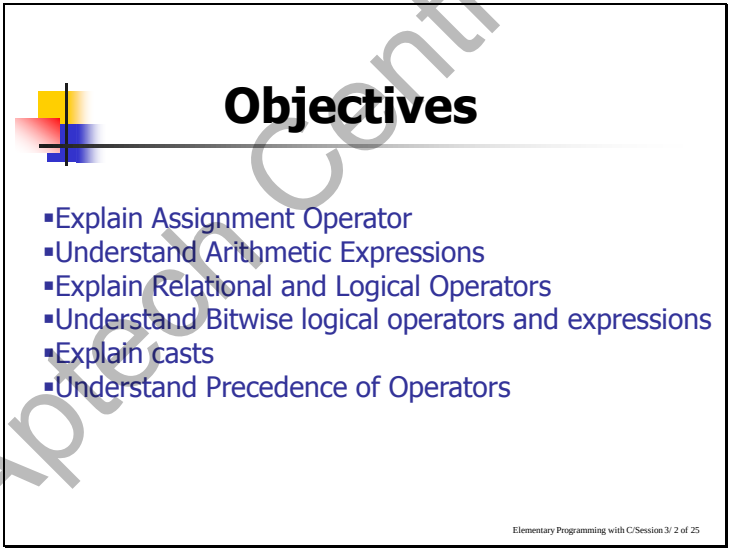
Follow the order as given here for the In-Class activities.

Overview of the Session:

The session introduces the students to the concepts of different types of operators and expressions in detail. The focus of this session is to make the students understand the concept of operators, their types and usage, expressions and type conversions.

3.2 In-Class Explanations

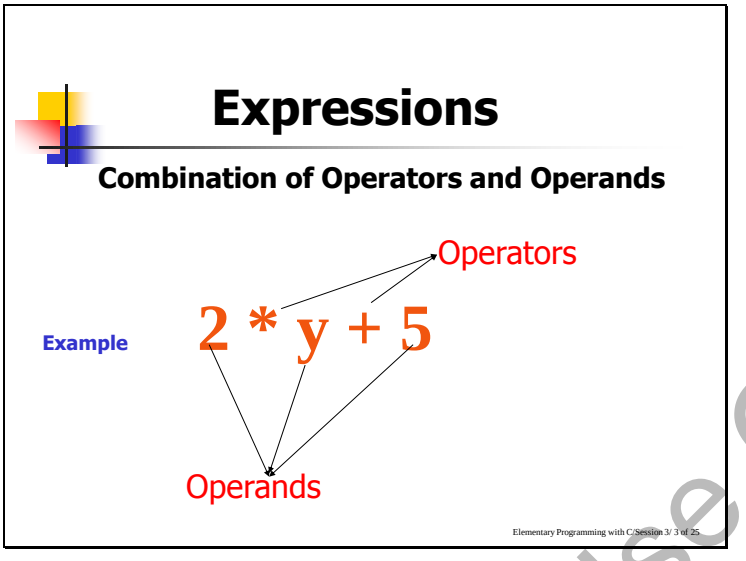
Slide 2



Max Time: 4 Minutes

Subject: You should list out the objectives as given in the slide and inform the students what they would be learning in the session.

Slide-3

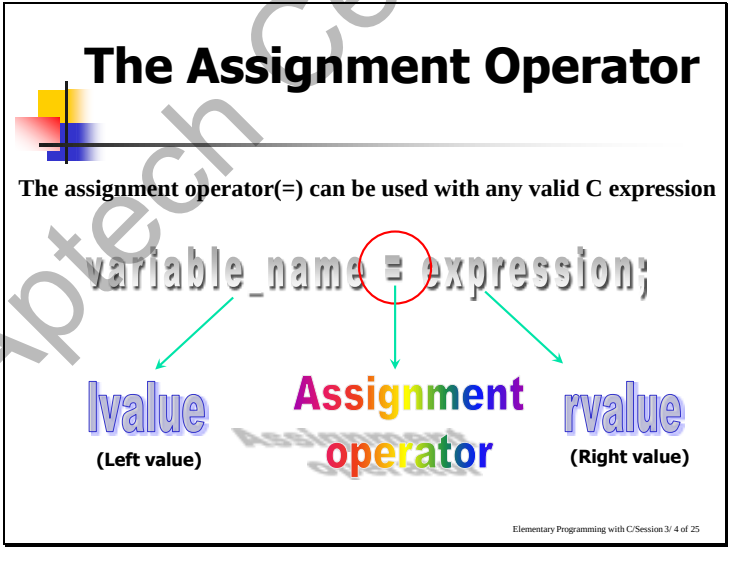


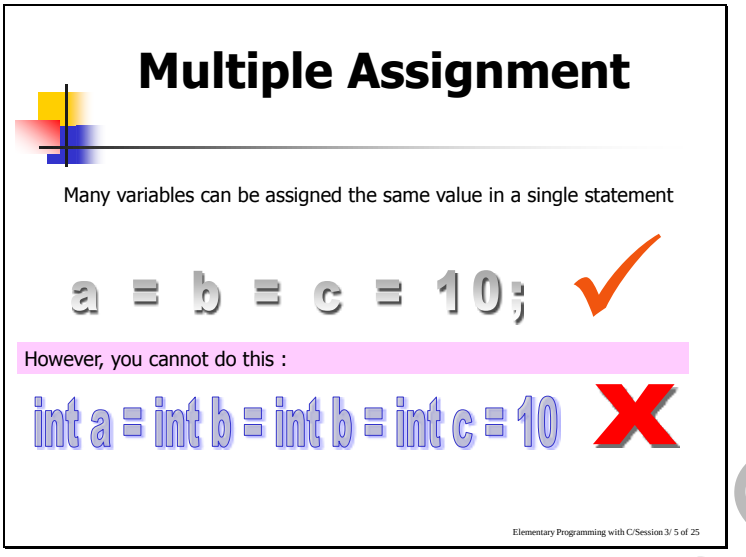
Max Time: 4 Minutes

Subject: The session should start with the answers given by the students for the expressions, operators, and operands. You should explain the concept of expressions clearly.

The following example can be used to explain an expression.
2 + 6 * (4 - 2), here the operators used are +, *, -, and operands used are 2, 6, 4, 2.

Slides-4 and 5





Max Time: 8 Minutes

Subject: You should explain the use of multiple assignment statement. Give some examples to make the concept clear. Also, explain the error message displayed when the data type is given more than once in a C statement.

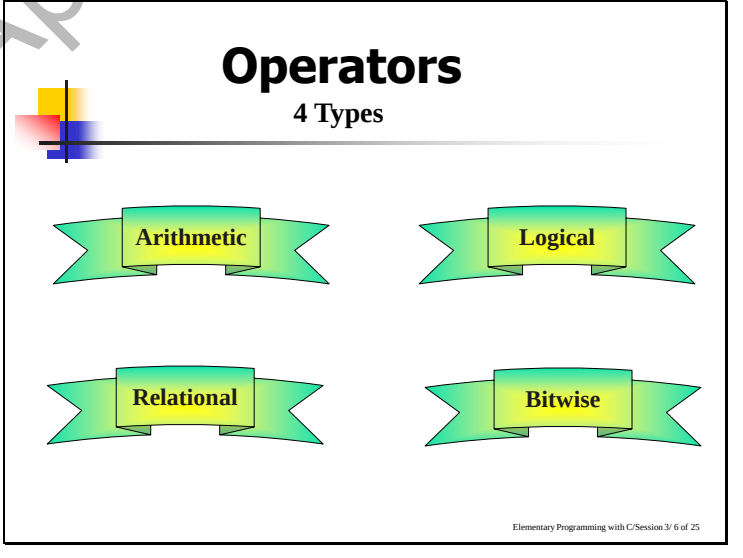
Additional Information:

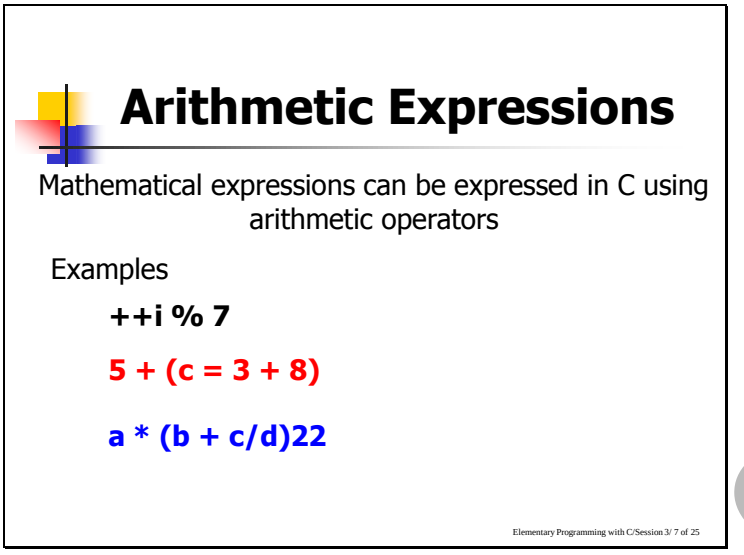
Reference Examples:

1) A = 10 , here A is the variable
2) C= a*b
3) A = b = c =10
4) Int a = int b = int c= 0;

Statement 4 gives an error because the data type (int) is defined more than once.

Slides-6 and 7





Max Time: 5 Minutes

Subject: Explain the difference between post and pre increment/decrement with examples. The program mentioned in the session should be explained to students clearly and output expected should be justified. Some reference program can be written and students should be asked to identify the output. Later, ask the students to write a simple program of their own. Precedence of operators can be discussed later.

Example: **10.0/100.0 * amount + 2.0 * sales**, where 'amount' and 'sales' are variables of the type ***float***

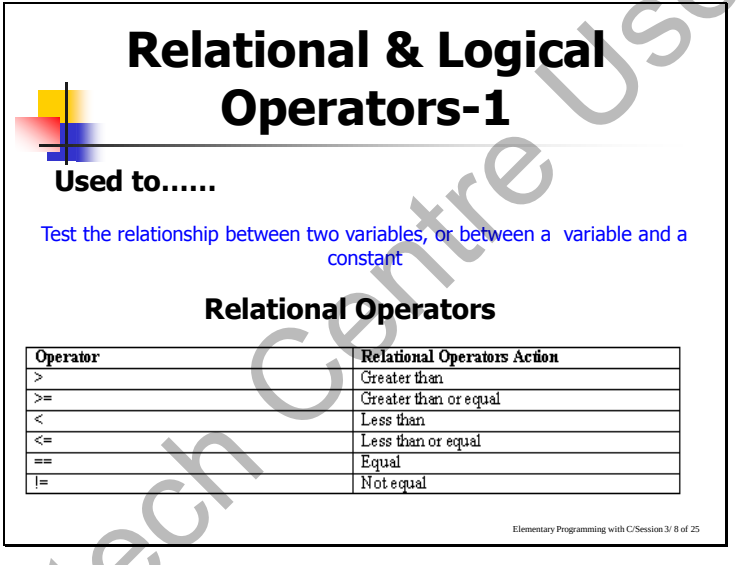
Additional Information:

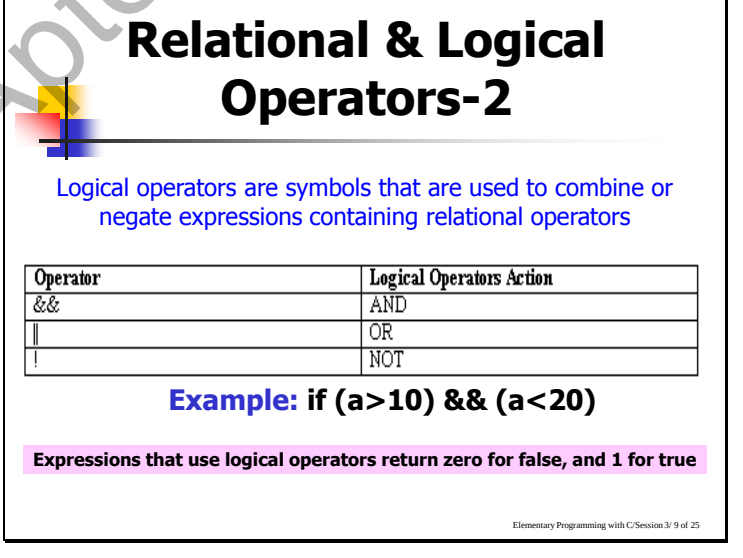
Reference Example:

```
#include <stdio.h>
    void main ()
    {
        int a=10,b=2;
        int i,j;
        printf ( "The numbers are            : %d & % d\n",a,b);
        printf ( "The addition gives         : %d\n",a+b);
        printf ( "The subtraction gives      : %d\n",a-b);
        printf ( "The multiplication gives   : %d\n",a*b);
        printf ( "The division gives         : %d\n",a/b);
        printf ( "The modulus gives          : %d\n",a%b);
        printf ( " After in                  : %d\n",a/b);
        printf ( "The modulus gives          : %d\n",a%b);

        i= 5;
        j=i++;
        printf("\n PostFix: j= %d,\t i=%d\n",j,i);
        i=5;
```

```
        j = ++i;
        printf("\n PreFix: j= %d,\t i=%d\n",j,i);
        i=5;
        printf("\nPostfix Operation\nBefore : i=%d\n",i);
        printf("During: i= %d\n",i++);
        printf ("After :i = %d\n",i);
        i=5;

        printf("\nPrefix Operation\nBefore i=%d\n",i);
        printf("During : i=%d\n",++i);
        printf("After : i=%d\n",i);
        printf("\nPostfixOperation:\nBefore:i=%d,\tAfter:i=%d\n",i,i++);

printf("\nPrefixOperation:\nBefore:i=%d,\t After:i=%d\n", i,++i);

    }
```

Slides-8 and 9





Max Time: 10 Minutes

Subject: You should clearly explain the use of relational and logical operators and how they are used in expressions. Explain clearly the use of logical operators, AND (&&), OR (||), and NOT (!). Explain the C statement given here:

```
printf ("%d", 5> 4 + 3);
```

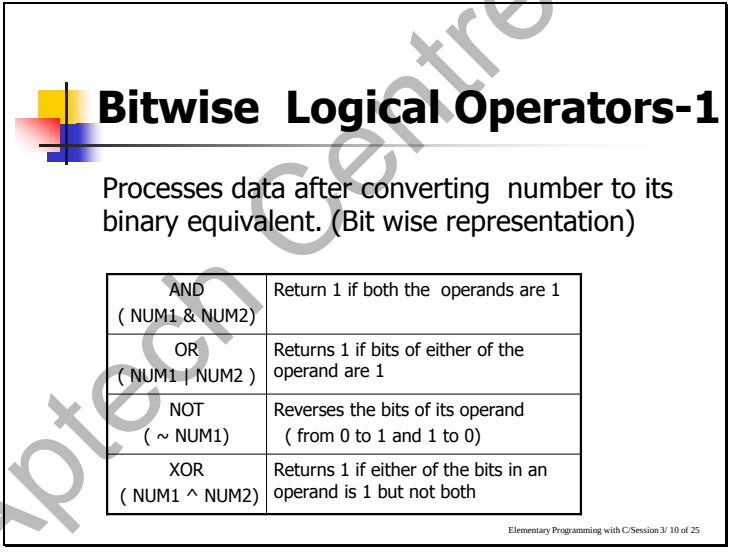
Will give

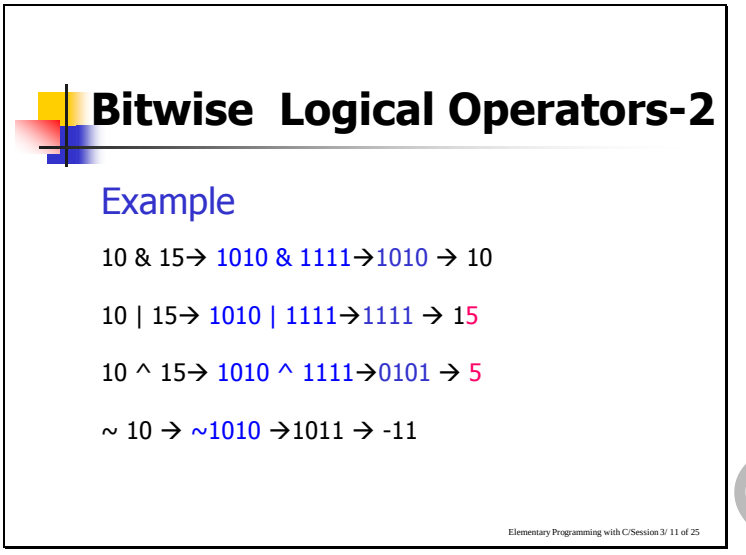
0

as 5 is less than 7 (4 + 3).

Both relational and logical operators are lower in precedence than the arithmetic operators are. For example, **5 > 4 + 3** is evaluated as if it is written as **5 > (4 + 3).** So 4+3 will be evaluated first and then the relational operation will be carried out. The result for this will be false, that is, zero will be returned.

Slides-10 and 11

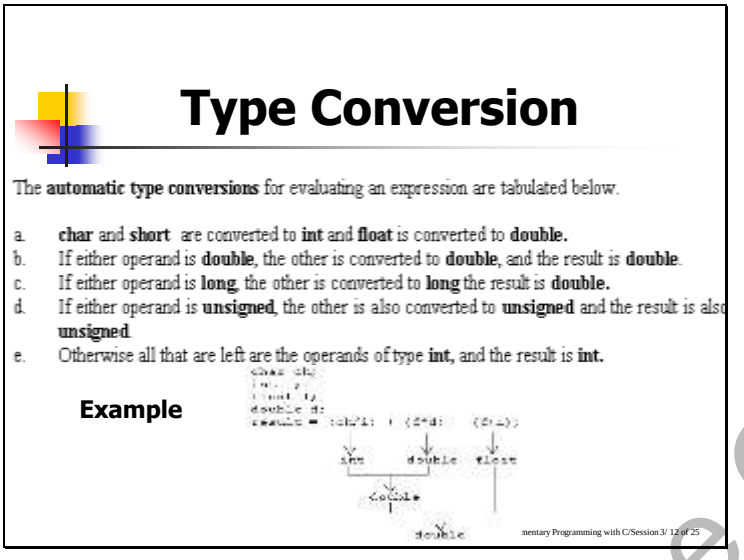


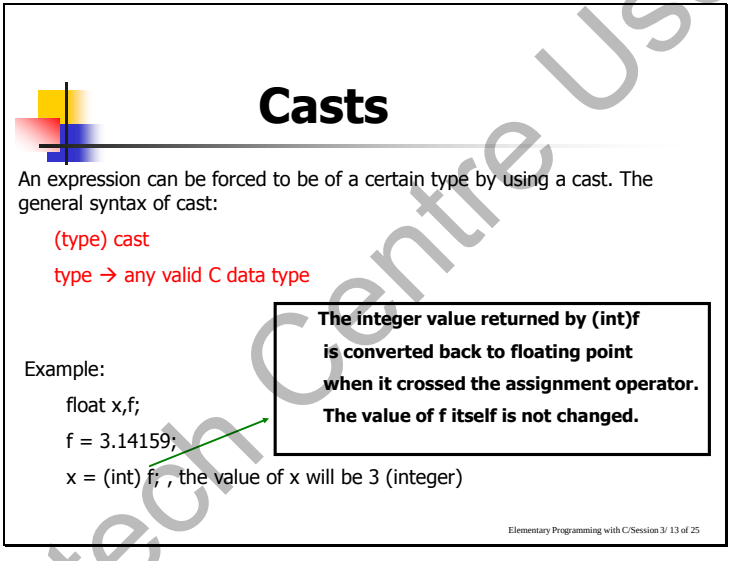


Max Time: 15 Minutes

Subject: You should explain the various Bitwise Logical Operators with examples. The students should be asked to solve some problems.

Slides-12 and 13





Max Time: 14 Minutes

Subject: You should explain the automatic type conversion and explicit conversion. Explain thoroughly the various points given in the slide 12. Reference example is given here to make the concepts very clear.

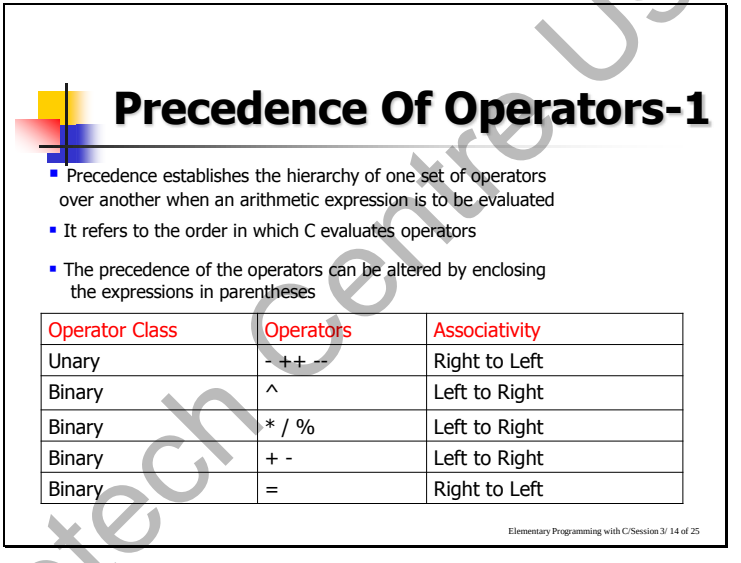
Additional Information:

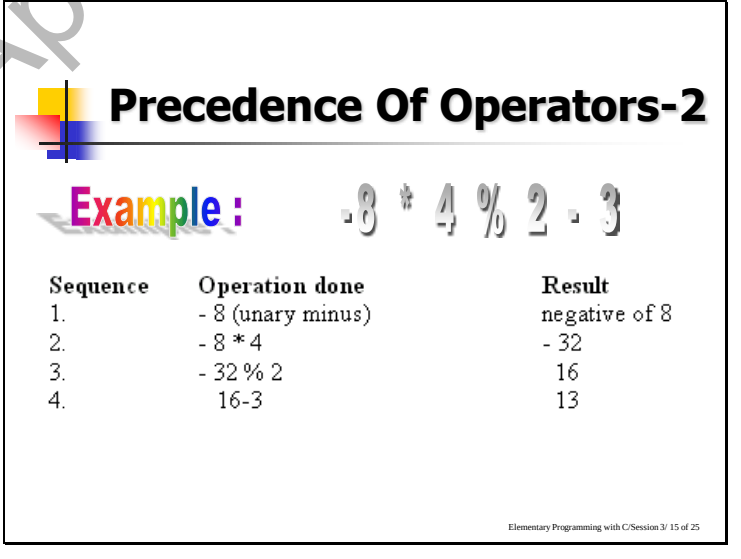
Reference example:

```
#include <stdio.h>

void main()
{
    char c1,c2;
    int i1,i2,i3;
    float f1,f2,f3,f4;
    c1= 'a';   /* no conversion */
```

```
    c2=1128; // int demoted to char
    printf("\nConversion to character :\n");
    printf("'a' = %c,1128 = %c \n",c1,c2);
    i1 = 'a';              // char promoted to int
    i2= 1128;          // no conversion
    i3=6.73223;          // float demoted to int
    printf("\nConversion to integer: \n");
    printf("'a' = %d, 1128 = %d , 6.73223 = %d\n",i1,i2,i3);
    f1 = 'a';              // char promoted to float
    f2 = 1128;          // int promoted to float
    f3 = 6.73223;     // no conversion
    f4 = 6.02e23;   // Approximate storage of float
    printf("\nConversion to float: \n");
    printf("  'a'  =  %f,  1128  =  %f  ,  6.73223  =  %f  \n  6.02e23  =  %f
    \n",f1,f2,f3,f4);
}
```

Slides-14, 15, 16, 17, and 18





Precedence between comparison Operators

Always evaluated from left to right

Elementary Programming with C/Session 3/ 16 of 25

Precedence for Logical Operators-1

Precedence	Operator
1	NOT
2	AND
3	OR

When multiple instances of a logical operator are used in a condition, they are evaluated from right to left

Elementary Programming with C/Session 3/ 17 of 25

Precedence for Logical Operators-2

Consider the following expression
False OR True AND NOT False AND True

This condition gets evaluated as shown below:
False OR True AND [NOT False] AND True

NOT has the highest precedence.
False OR True AND [True AND True]

AND is the operator of the highest precedence and operators of the same precedence are evaluated from right to left
False OR [True AND True]
[False OR True]
True

Elementary Programming with C/Session 3/ 18 of 25

Max Time: 20 Minutes

Subject: You should clearly explain the precedence of various operators with examples.

Additional Information:

Reference Example

```
#include <stdio.h>
void main()
{
    int a,b,c,d,e,f;

    b=15;
    c=8;
    d=3;
    e=32;
    f=5;
    a=b*c+e/d+f;

    printf("The value of 'a' = %d",a);
}
```

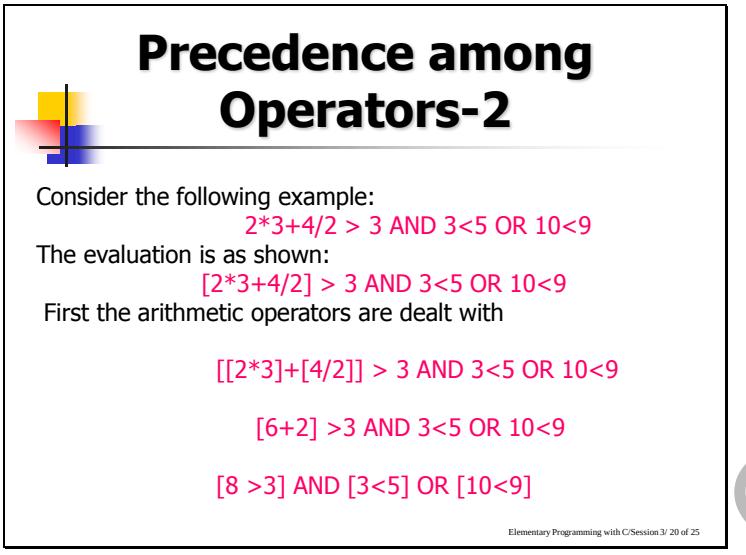
Slides-19, 20, and 21

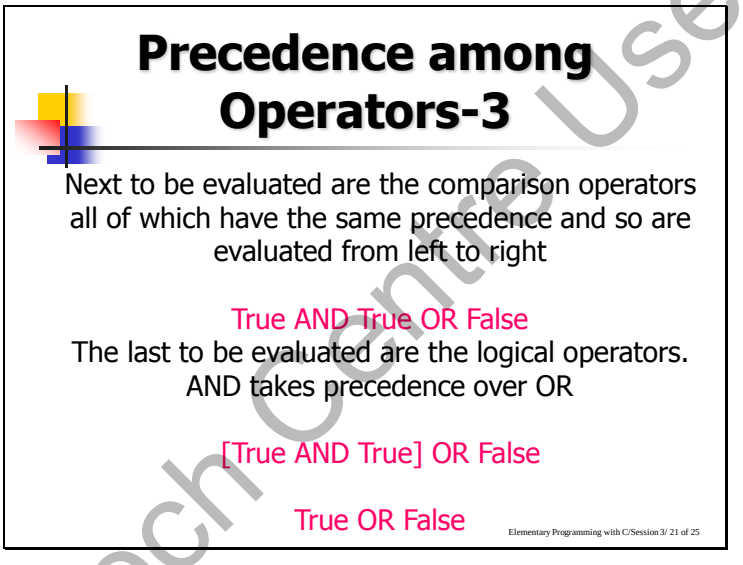
Precedence among Operators-1

When an equation uses more than one type of operator then the order of precedence has to be established with the different types of operators

Precedence	Type of Operator
1	Arithmetic
2	Comparison
3	Logical

Elementary Programming with C/Session 3/ 19 of 25



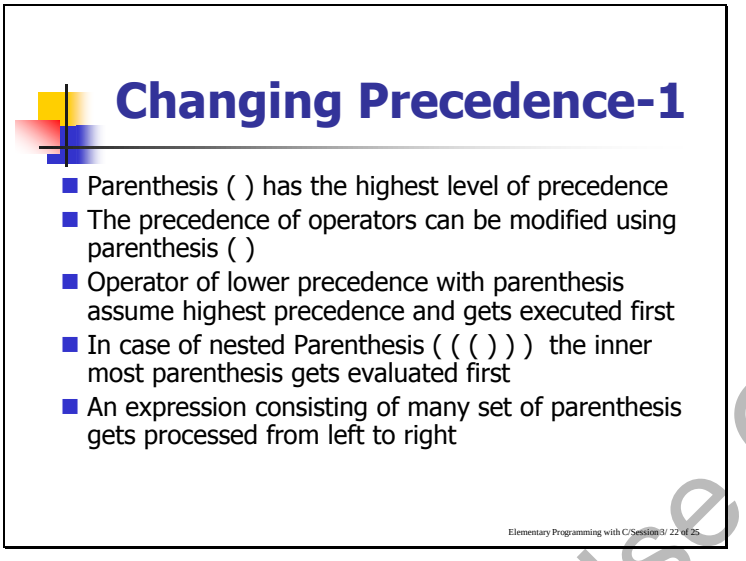


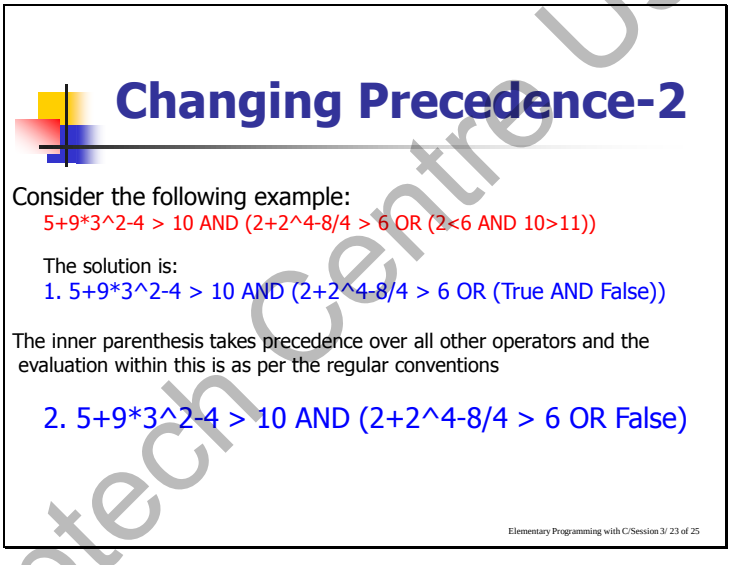
Max Time: 15 Minutes

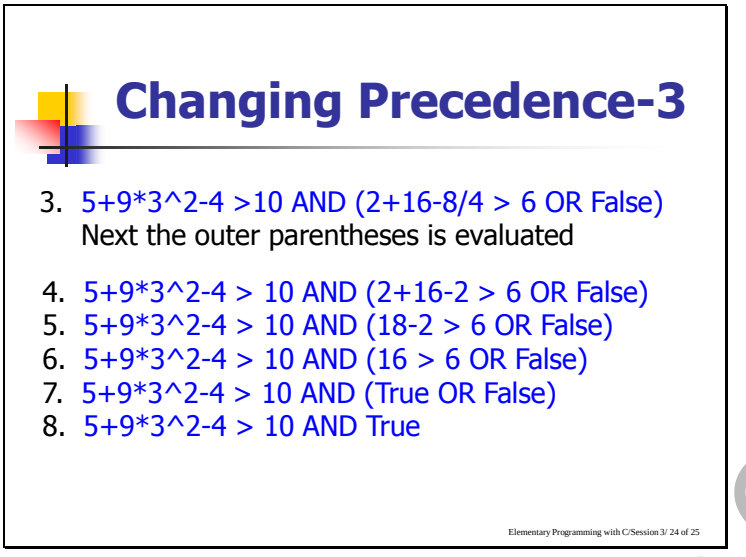
Subject: You should explain the precedence of various operators when given in expressions. You should also explain the following rules.

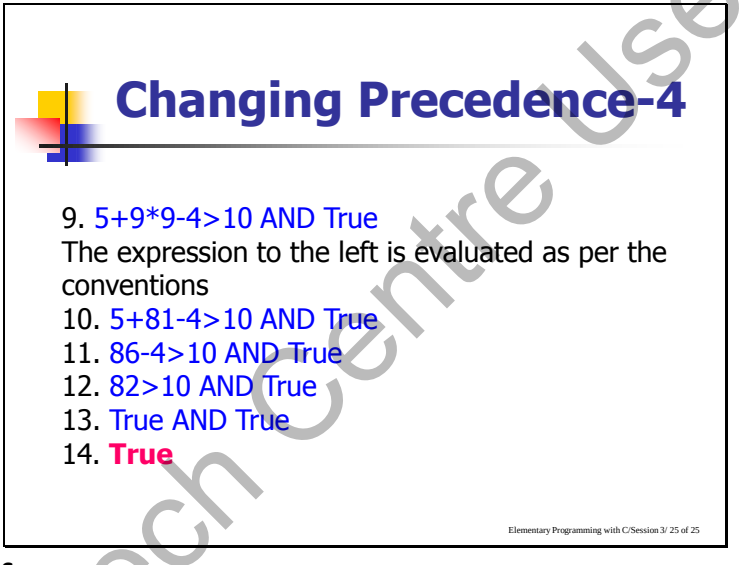
In an equation involving all three types of operators, the arithmetic operators are evaluated first followed by comparison and then logical.

Slides-22, 23, 24, and 25









Max Time: 25 Minutes

Subject: You should explain the effect of parentheses on expressions. The students should be clear about the example given in slides 24 and 25.

Solutions to Check Your Progress

1. Operators
2. Expression
3. Precedence
4. Mixed Mode expression
5. Cast
6. Logical operators.
7. &, | , ~ ,^.
8. 10. Parentheses.

Solutions to Do It Yourself

1.

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

void main()
{
   clrscr();
   int a,b,c,d;
   printf("Enter the First Number ");
   scanf("%d",&a);
   printf("\nEnter the Second Number ");
   scanf("%d",&b);
   printf("\nEnter the Third Number ");
   scanf("%d",&c);
   d = a + b + c;
   printf("\nSum of 3 numbers = %d",d);
   getch();
}
```

2.

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

void main()
{
   int a= 10, b = 7 , d= 4 ,e= 2;
   float  c= 15.75, f= 5.6, z;
         z = a*b+(c/d)-e*f;

   printf("Value of z= %f",z);
}
```

3.

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

void main()
{
  int breadth, height; // of rectangle
  int perimeter, area; // of rectangle
    // input breadth and height
  printf("Enter breadth and height: ");
  scanf("%d %d",&breadth,&height);
    // calculate perimeter and area
  perimeter = 2*(breadth+height);
  area = breadth*height;

printf("\n Perimeter of a rectangle = %d",perimeter);
printf("\n Area of a rectangle = %d",area);
}
```

4.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   clrscr();
   float radius, height, volume;
   /* declare variables as floating point values */

   printf("This program computes the volume of a cylinder      given
   the radius and height.\n");
   printf("Enter the radius and height:");
   scanf("%f %f",&radius,&height);
   volume=3.1416*radius*radius*height;
   printf("\nThe volume is %f\n",volume);
}
```

5.

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

void main()
{
   clrscr();
   int basic_sal = 12000, hra = 150, ta = 120 ,  others=450;
   float da ,pf,it , net_sal;
   da= basic_sal*12/100;
   pf = basic_sal * 14/100;
   it = basic_sal * 15/100;
   net_sal= (basic_sal + da + hra + ta +  others -(pf + it) );
   printf("\n Net salary = %f ",net_sal);
   getch();
}
```

3.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 4 - Input and Output in 'C'

Note: This TG maps to Session 6 of the book.

4.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. The session introduces the students to the concepts of different types of operators and expressions in detail.

4.1.1 Objectives

By the end of this session, the learners will be able:

- To understand formatted I/O functions scanf() and printf()
- To use character I/O functions getchar() and putchar()

Difficulties

The students may find difficulty in understanding the following topics:

- Modifiers for Format commands

Hence, the mentioned topic should be handled carefully.

4.1.2 Teaching Skills

You should be well-versed in the input/output functions and their usage for teaching this session. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

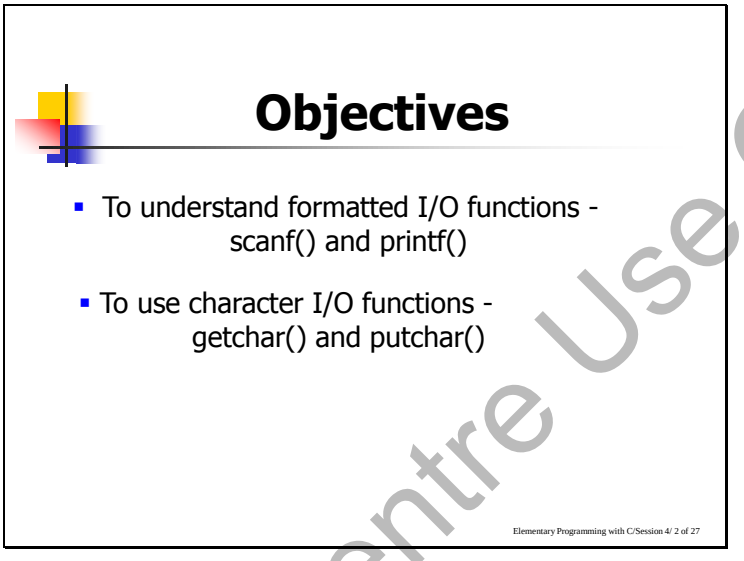
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the concept of input and output functions.

4.2 In-Class Explanations

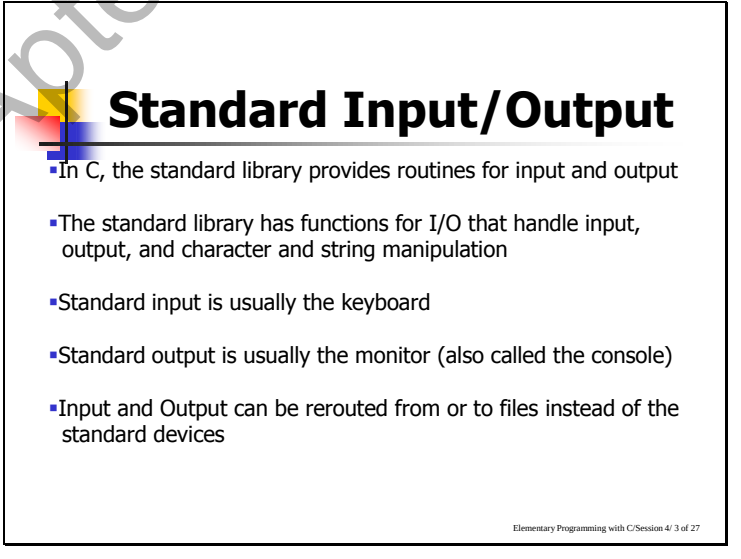
Slide 2



Max Time: 1 Minute

Subject: List out the objectives given on Slide 2. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

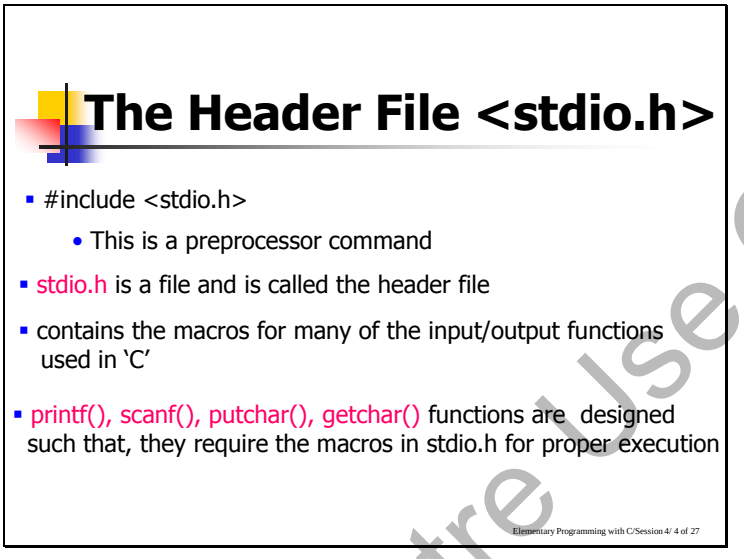
Slide-3



Max Time: 4 Minutes

Subject: The teaching of the section should start with the various input and output devices. Also explain that the input and output can be rerouted from or to files instead of the standard devices

Slide-4

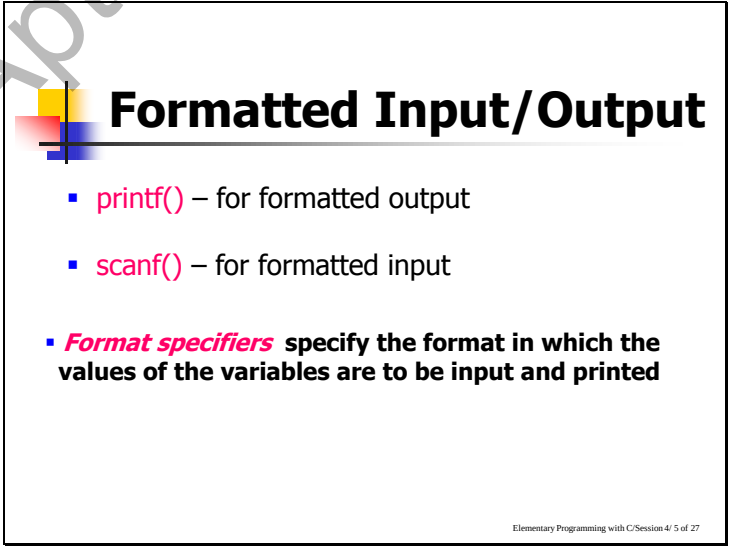


Max Time: 3 Minutes

Subject: You should explain the use of header files in 'C'. The students should be clear about using the *<stdio.h>* header file.

You should explain the concept of compiling and linking. The students should be clear with the concept of the linking process.

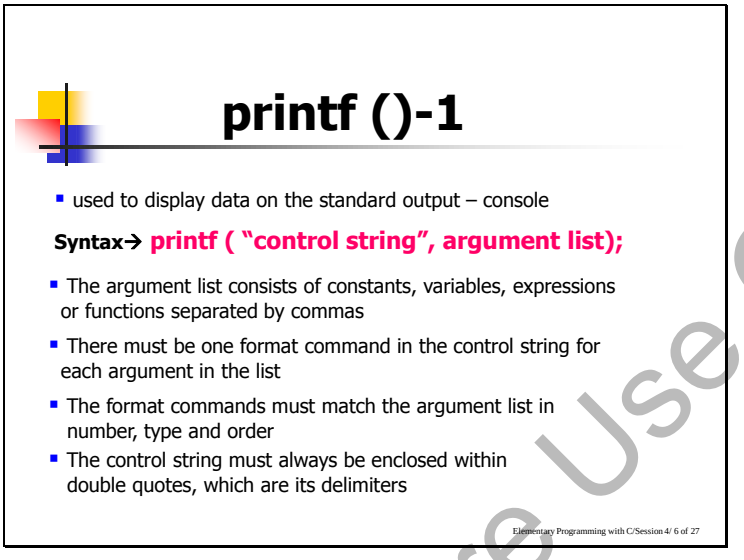
Slide-5

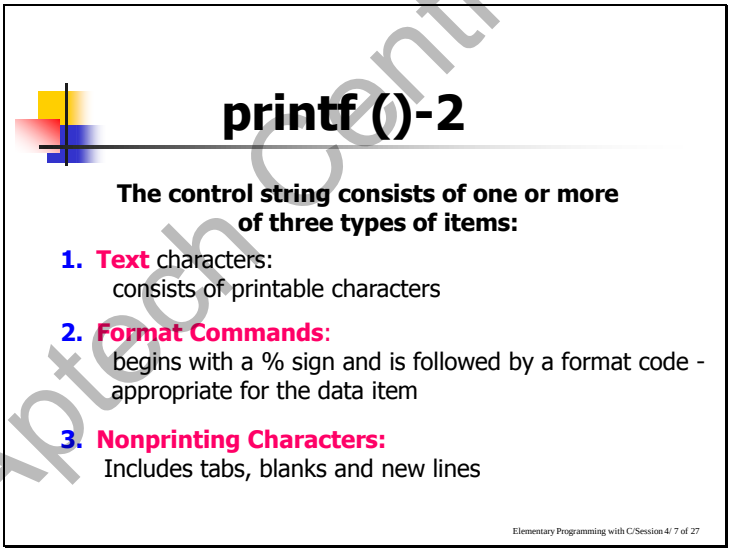


Max Time: 3 Minutes

Subject: You should explain the concept of formatted functions. Also explain the usage of *printf( )* and *scanf( )*.

Slides-6 and 7





Max Time: 11 min

Subject: You should clearly explain the syntax of *printf( )*, items that can be included in the control string and what is the argument list. Also explain the various rules for the control list and the argument list.

Slides-8, 9, and 10

Format codes-1

Format	printf()	scanf()
Single Character	%c	%c
String	%s	%s
Signed decimal integer	%d	%d
Floating point (decimal notation)	%f	%f or %e
Floating point (decimal notation)	%lf	%lf
Floating point (exponential notation)	%e	%f or %e
Floating point (%f or %e , whichever is shorter)	%g	
Unsigned decimal integer	%u	%u
Unsigned hexadecimal integer (uses "ABCDEF")	%x	%x
Unsigned octal integer	%o	%o

In the above table c, d, f, lf, e, g, u, s, o and x are the type specifiers

Elementary Programming with C/Session 4/ 8 of 27

Format codes-2

Format Code	Printing Conventions
%d	The number of digits in the integer
%f	The integer part of the number will be printed as such. The decimal part will consist of 6 digits. If the decimal part of the number is smaller than 6, it will be padded with trailing zeroes at the right, else it will be rounded at the right.
%e	One digit to the left of the decimal point and 6 places to the right , as in %f above

Elementary Programming with C/Session 4/ 9 of 27

Control String Special Characters

\\	to print \ character
\ "	to print " character
%%	to print % character

Elementary Programming with C/Session 4/ 10 of 27

Max Time: 23 Minutes

Subject: The various format codes used in *printf( )* can be explained. You should clearly explain which format codes should be used for numbers, characters, strings etc. Slide 10 should be introduced to explain how to print some special characters like \, “, %.

Slides-11 and 12

control strings & format codes

No	Statements	Control String	What the control string contains	Argument List	Explanation of the argument list	Screen Display
1.	printf("%d",300);	%d	Consists of format command only	300	Constant	300
2.	printf("%d",10+5);	%d	Consists of format command only	10 + 5	Expression	15
3.	printf("Good Morning Mr. Lee.");	Good Morning Mr. Lee.	Consists of text characters only	Nil	Nil	Good Morning Mr. Lee.
4.	int count = 100; printf("%d",count);	%d	Consists of format command only	count	variable	100
5.	printf("\nhello");	\nhello	Consists of nonprinting character & text characters	Nil	Nil	hello on a new line
6.	#define str "Good Apple " ..... printf("%s",str);	%s	Consists of format command only	Str	Symbolic constant	Good Apple

7.	.....
	int count,stud_num;
	count=0;
	stud_nim=100;
	printf("%d %d\n",count, stud_num);
		%d %d	Consists of format command and escape sequence	count, stud_num	two variables	0 , 100

Example for printf()

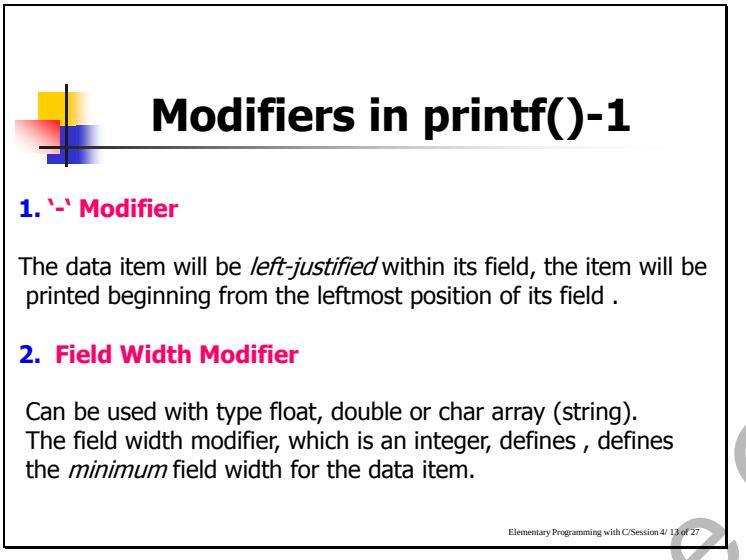
Program to display integer, float , character and string

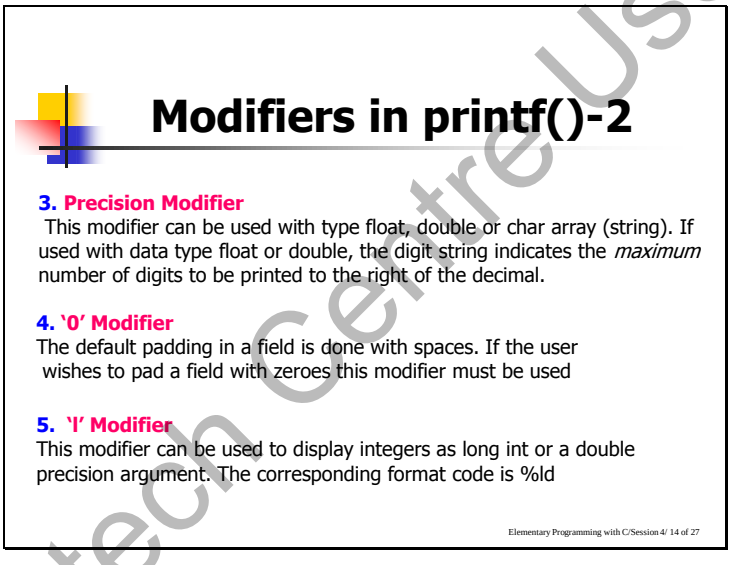
```
#include <stdio.h>
   void main()
   {
                int a = 10;
                float b = 24.67892345;
                char ch = 'A';

                printf("Integer data = %d", a);
                printf("Float Data = %f",b);
                printf("Character = %c",ch);
                printf("This prints the string");
                printf("%s","This also prints a string");
    }
```

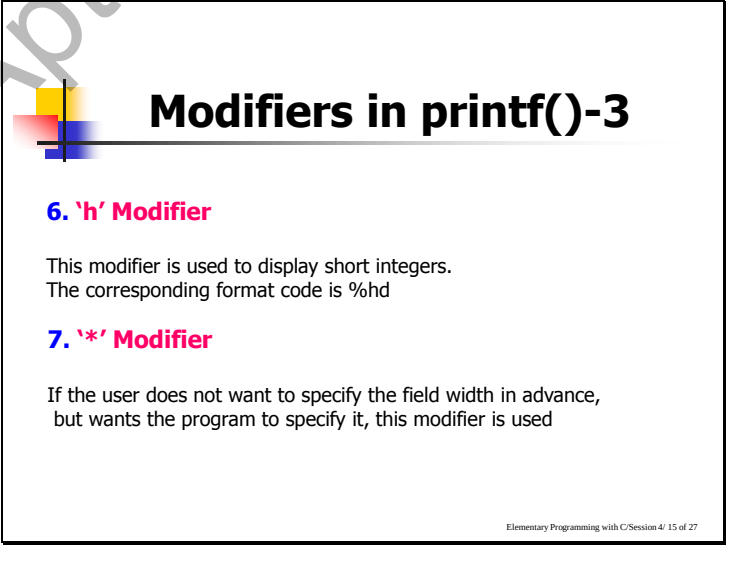

Max Time: 18 Minutes

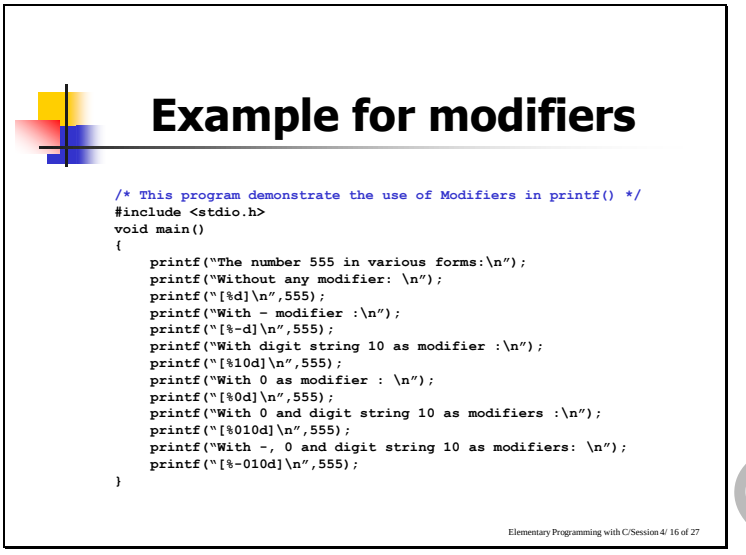
Subject: The example given in slide 11 can be used to explain the control string parameters, the argument list, the output printed by using the various format commands. Slide 12 explains how to print an integer, float, character, and string using *printf( )*. You should clarify the difference between integers and float values.

Slides-13, 14, 15, and 16





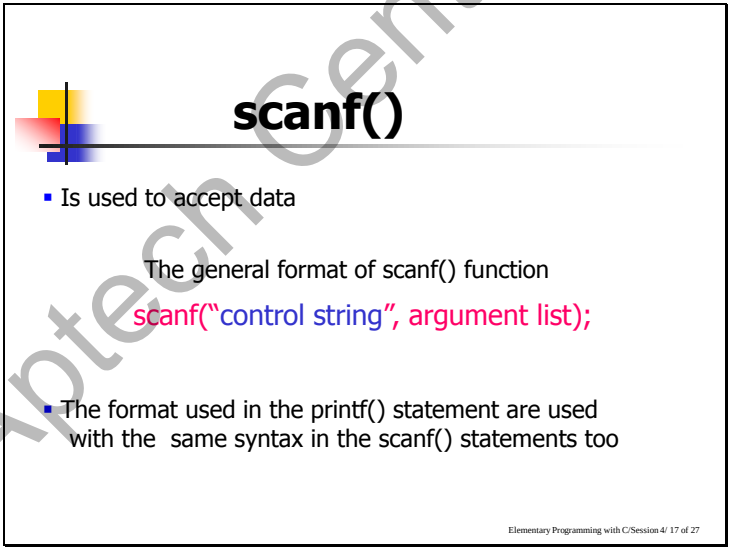




Max Time: 16 Minutes

Subject: The modifiers for format commands in *printf( )* should be explained here. The students should clearly understand the usage of modifiers like '-', Field width, precision, 0, h, *. Slide 16 gives an example how to support the concepts taught in the slides.

Slides-17, 18, 19, and 20



Differences in argument list of between printf() and scanf()

- printf() uses variable names, constants , symbolic constants and expressions
- scanf() uses pointers to variables

When using scanf() follow these rules, for the argument list:

- If you wish to read in the value of a variable of basic data type, precede the variable name with a **&** symbol
- When reading in the value of a variable of derived data type, do not use a **&** before the variable name

Elementary Programming with C/Session 4/ 18 of 27

Differences in the format commands of the printf() and scanf()

- **There is no %g option**
- **The %f and %e format codes are in effect the same**

Elementary Programming with C/Session 4/ 19 of 27

Example for scanf()

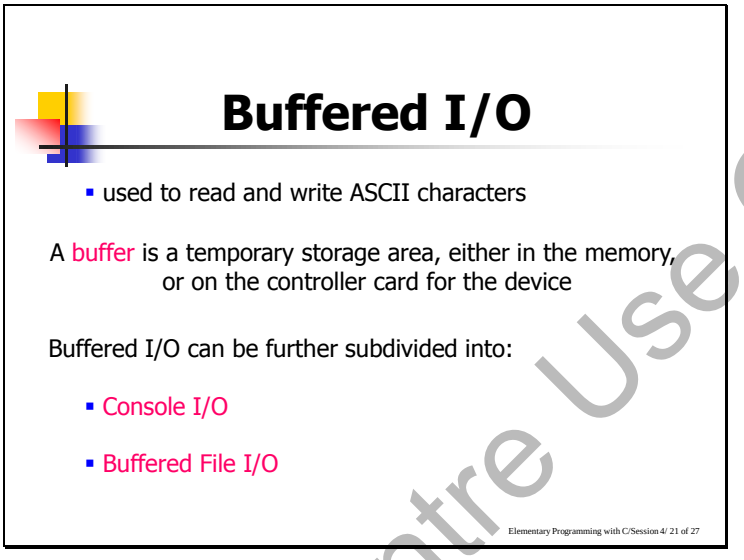
```
#include <stdio.h>
  void main()
  {
      int a;
      float d;
      char ch, name[40];

   printf("Please enter the data\n");
   scanf("%d %f %c %s", &a, &d, &ch, name);

   printf("\n The values accepted are :
      %d, %f, %c, %s", a, d, ch, name);
  }
```

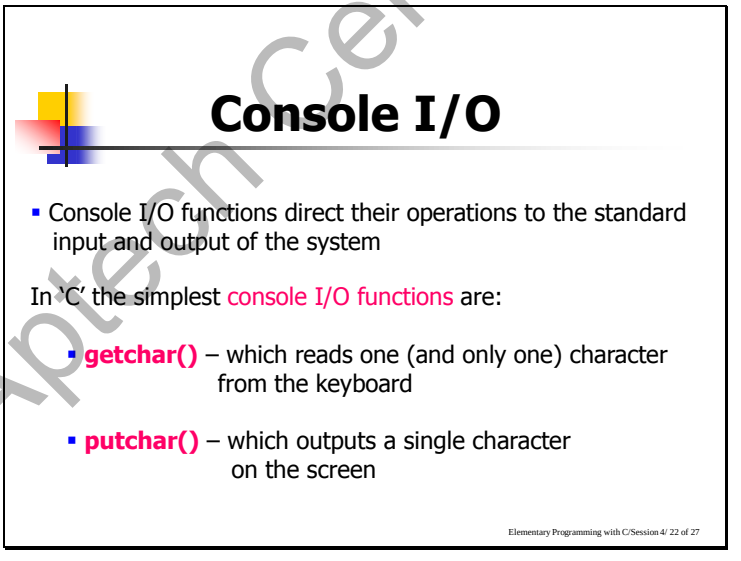
Elementary Programming with C/Session 4/ 20 of 27

Max Time: 14 Minutes

Subject: You should explain the use of *scanf( )* and its syntax. Students should be clear about the difference of the argument list and the format commands used with *printf( )* and *scanf( )*. Explain the example in detail how we can accept different values from the keyboard using *scanf( )*.

Slides-21 and 22

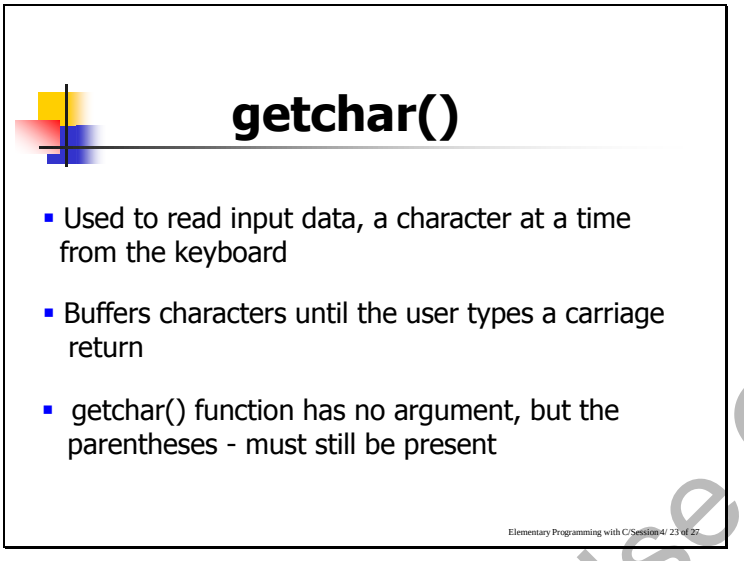


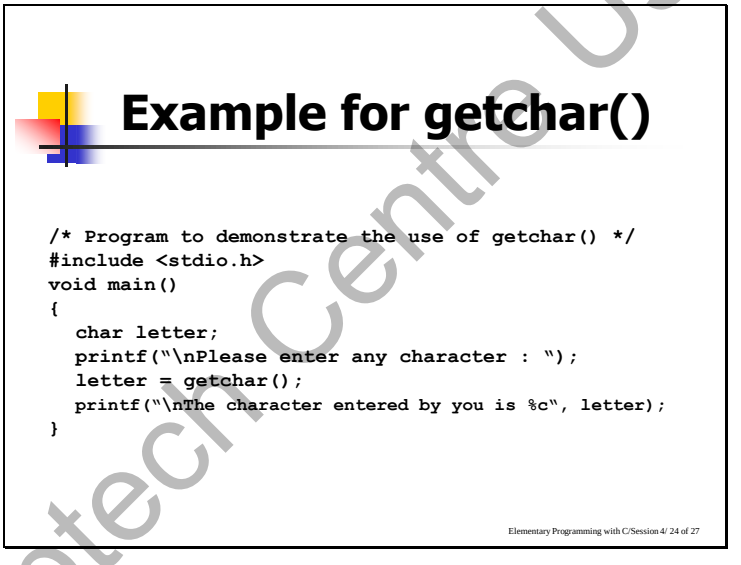


Max Time: 6 Minutes

Subject: You should explain what is buffer and Buffered I/O. Then he/she can explain the console I/O function *getchar( )* and *putchar( )* using Slide 22.

Slides-23 and 24





Max Time: 6 Minutes

Subject: You should explain how to use *getchar( )* and what is the exact use of getchar( ). The given example should be explained in detail to understand the use of *getchar( )*.

Slides-25, 26, and 27

putchar()

- Character output function in 'C'
- requires an argument

Argument of a putchar() function can be :

- A single character constant
- An escape sequence
- A character variable

Elementary Programming with C/Session 4/ 25 of 27

putchar() options & effects

Argument	Function	Effect
character variable	putchar(c)	Displays the contents of character variable c
character constant	putchar('A')	Displays the letter A
numeric constant	putchar('5')	Displays the digit 5
escape sequence	putchar('\t')	Inserts a tab space character at the cursor position
escape sequence	putchar('\n')	Inserts a carriage return at the cursor position

Elementary Programming with C/Session 4/ 26 of 27

putchar()

```
/* This program demonstrates the use of constants and escape
sequences in putchar() */

#include <stdio.h>
void main()
{
   putchar('H'); putchar('\n');
   putchar('\t');
   putchar('E'); putchar('\n');
   putchar('\t'); putchar('\t');
   putchar('L'); putchar('\n');
   putchar('\t'); putchar('\t'); putchar('\t');
   putchar('L'); putchar('\n');
   putchar('\t'); putchar('\t'); putchar('\t');
   putchar('\t');
   putchar('O');
}
```

Example

Elementary Programming with C/Session 4/ 27 of 27

Max Time: 15 Minutes

Subject: You should explain the use of *putchar( )* function and how the arguments can be used in *putchar( )*. Example given in the slide show the various arguments to explain the use of *putchar( )*.

Solutions to Check Your Progress

1. printf() and scanf()
2. pointers
3. Format specifier
4. %
5. T
6. buffer
7. F

Solutions to Do It Yourself

1. A.

```
   a) printf("%d", sum);
   b) printf("Welcome\n");
   c) printf("%c", letter);
   d) printf("%f",discount);
   e) printf("%.2f",dump);
```

B.

```
   a) scanf("%d",&sum);
   b) scanf("%f",&discount_rate);
```

2.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   clrscr();

      printf("%d",'A');
      printf("%d",'b');
}
```

3.

```
#include <stdio.h>
void main()
{
   int breadth;
   float length, height;
   scanf("%d%f%f",&breadth,&length,&height);
   printf("%d %f  %f",breadth, length, height);
}
```

4.

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

void main()
{
   char name[20];
   int basic,daper;

   float bonper,loandet,salary;

   clrscr();


   printf("\nEnter Employees Name : ");
   scanf("%s",name);

   printf("\nEnter Basic Salary : ");
   scanf("%d",&basic);

   printf("\nEnter percentage of DA : ");
   scanf("%d",&daper);

   printf("\nEnter percentage for bonus : ");
   scanf("%f",&bonper);

   printf("\nEnter Loan amount : ");
   scanf("%f",&loandet);

   salary = basic + basic * daper /100 + bonper * basic/10 -
              loandet;

      printf("\nName\t\t\tBasic\t\t\tSalary\n");

      printf("%s\t\t\t%d\t\t\t%0.0f",name,basic,salary);
}
```

5.

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

void main()
{
        char fname[20], lname[20];

        clrscr();

        printf("\n Enter First Name : ");
        scanf("%s",fname);

        printf("\n Enter Last Name : ");
        scanf("%s",lname);
```

```
            printf("\n%s %s",lname,fname);
}
```

4.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 5-Condition

Note: This TG maps to Session 7 of the book.

5.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. The session introduces the students to the concepts of different types of operators and expressions in detail.

5.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain the Selection Construct
  - *If statement*
  - *If – else statement*
  - *Multi if statement*
  - *Nested if statement*
  - *Switch statement*

Difficulties

The students may find difficulty in understanding the following topics:

- Nested if's
- Switch statements

Hence, the mentioned topic should be handled carefully.

5.1.2 Teaching Skills

You should be well-versed in programming constructs such as conditional statements for teaching this session. You must know the various types of conditional statements and how to write them in C. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

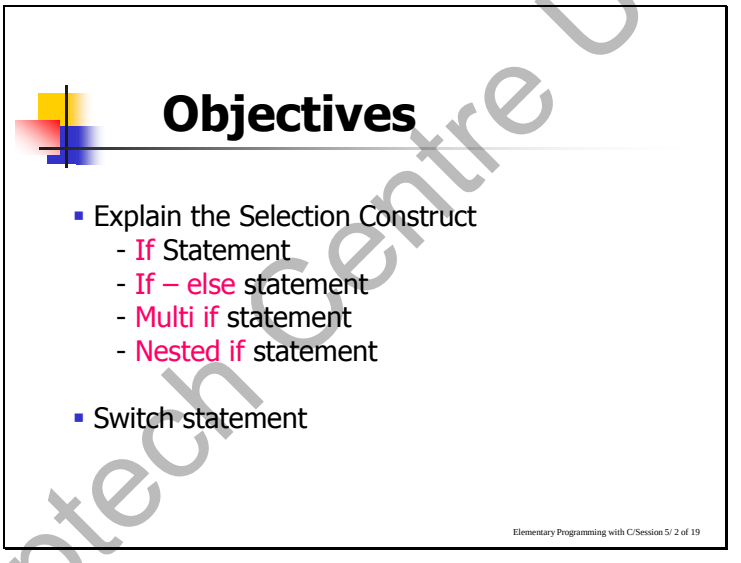
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the concept of selection constructs.

5.2 In-Class Explanations

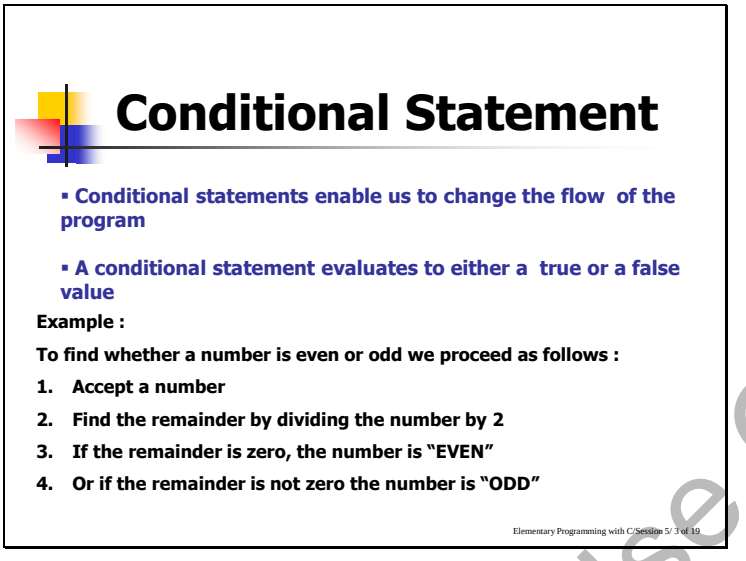
Slide 2



Max Time: 4 Minutes

Subject: List out the objectives using Slide 2 informing students what they are going to learn in the next 2 hours. By the end of slide 2, the students should have an idea of the topics they will be learning in the session.

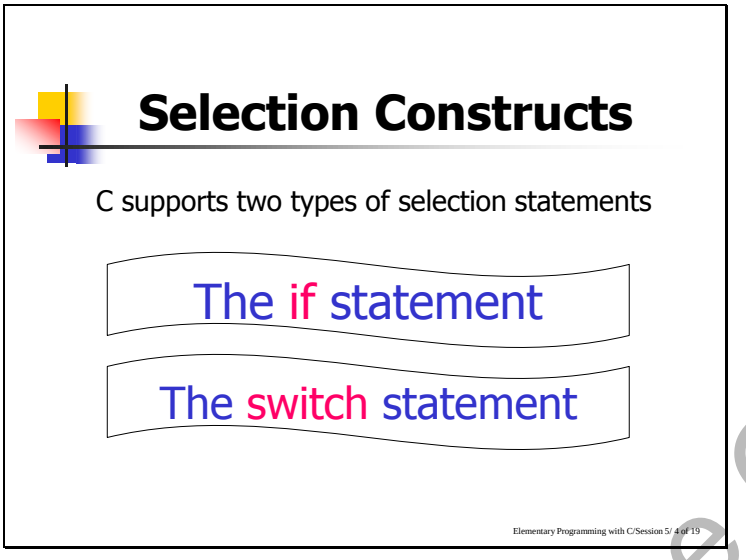
Slide-3



Max Time: 5 Minutes

Subject: The teaching of the session should start with the answers given by the students for the various selection statements. You can explain the conditional statement with an example.

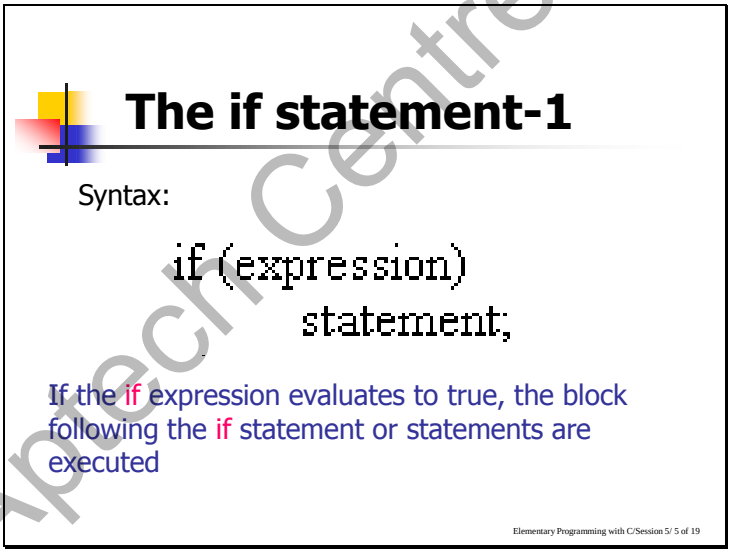
Slide-4

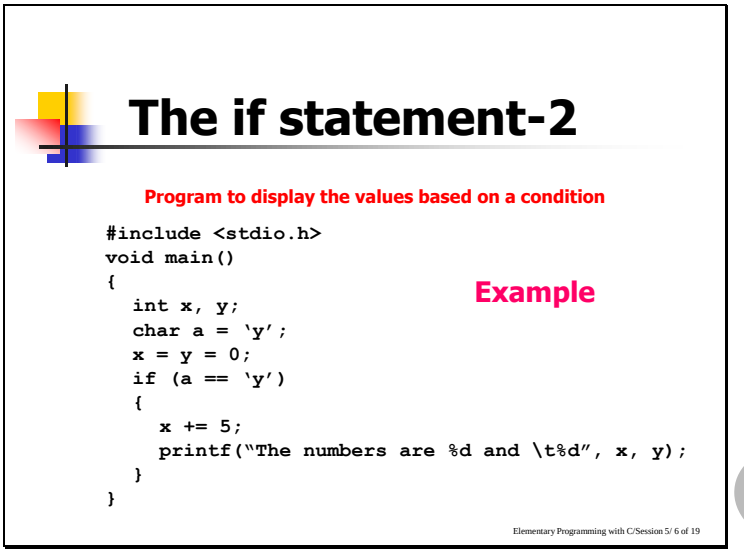


Max Time: 5 Minutes

Subject: You should explain the different selection constructs. Name the selection constructs.

Slides-5 and 6





Max Time: 15 Minutes

Subject: You have to explain the use of the *if* statement and mention clearly the syntax of the *if* statement. The example has to be handled carefully so that the students get an idea of the *if* statements with and without curly brackets. You can give a simple program and ask them to write using the *if* statement.

Additional Information:

Reference Example:

Accept a number and display the message "The number is greater than 10", if the number is greater than 10.

```
#include <stdio.h>

void main()
{
      int num;

      printf("Enter a Number :");
      scanf("%d",&num);

      if(num > 10)
            printf("The number is greater than 10");
}
```

Slides-7, 8, and 9

The if – else statement-1

Syntax:

```
if (expression)
        statement;
else
        statement;
```


The if – else statement-2

- If the if expression evaluates to true, the block following the if statement or statements are executed
- If the if expression does not evaluate to true then the statements following the else expression take over control
- The else statement is optional. It is used only if a statement or a sequence of statements are to be executed in case the if expression evaluates to false

The if – else statement -3

Program to display whether a number is Even or Odd

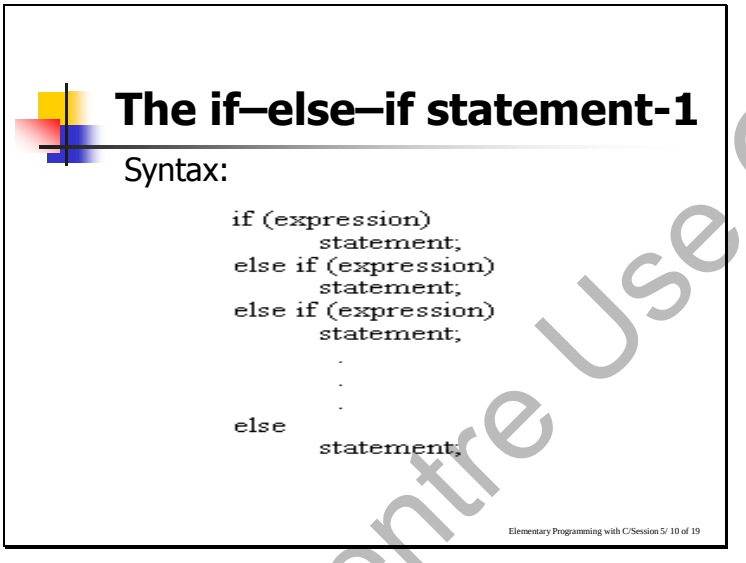
```
#include <stdio.h>
void main()
{
   int num , res ;

   printf("Enter a number :");
   scanf("%d",&num);
   res = num % 2;
   if (res == 0)
        printf("Then number is Even");
   else
        printf("The number is Odd");
}
```

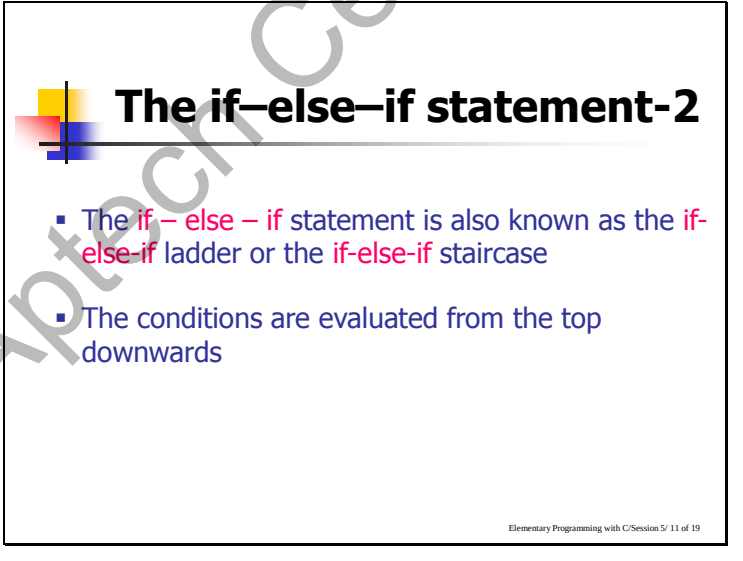
Example

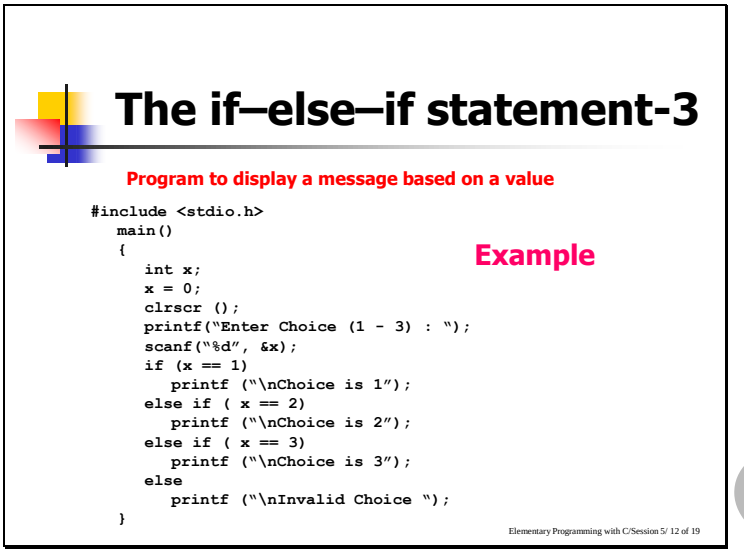
Max Time: 18 Minutes

Subject: You have to explain the use of the *if - else* statement and mention clearly the syntax of the *if -else* statement. The example has to be handled carefully so that the students get an idea and all students are clear about it. You can give a simple program and ask them to write using the *if -else* statement.

Slides-10, 11, and 12







Max Time: 20 Minutes

Subject: You have to explain the use of the *if – else –if* statement and mention clearly the syntax of the *if –else-if* statement. The example has to be handled carefully so that the students get an idea and all students are clear about it. You can give a simple program and ask them to write using the *if –else-if* statement.

Additional Information:

Reference Example:

Accept user input in the range 1 to 2. If the user enters 1, then display the message, **"PASCAL is easy ".** If the user enters 2, then display the message **"C is fun".** If the user enters 3, then display the message **"Invalid entry. Valid 1 and 2 ".**

```
#include <stdio.h>
void main()
{
     int num;
     printf("Enter a number");
     scanf("%d",&num);

     if(num==1)
          printf("PASCAL is easy");
     else if (num == 2)
          printf("C is fun");
     else if(num == 3)
          printf("Invalid entry. Valid 1 and 2");
     else
          printf("Type a number 1 or 2);
}
```

Slides-13, 14, and 15

Nested if-1

- The nested **if** is an **if** statement, which is placed within another **if** or **else**
- In C, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** statement and is not already associated with an **if**

Nested if-2

Syntax:

```
if (exp1)
{
    if (exp2) statement1;
    if (exp3) statement2;
    else statement3;          /*with if (exp3) */
}
else statement4;              /* with if (exp1) */
```

- Note that the inner else is associated with **if(exp3)**
- According to ANSI standards, a compiler should support at least 15 levels of nesting

Nested if-3

Example

```
#include <stdio.h>
void main ()
 {
   int x, y;
   x = y = 0;
   clrscr ();
   printf ("Enter Choice (1 - 3) : ");
   scanf ("%d", &x);
   if (x == 1)
   {
      printf("\nEnter value for y (1 - 5) : ");
      scanf ("%d", &y);
      if (y <= 5)
            printf("\nThe value for y is : %d", y);
      else
            printf("\nThe value of y exceeds 5 ");
   }
   else
      printf ("\nChoice entered was not 1");
}
```

Max Time: 28 Minutes

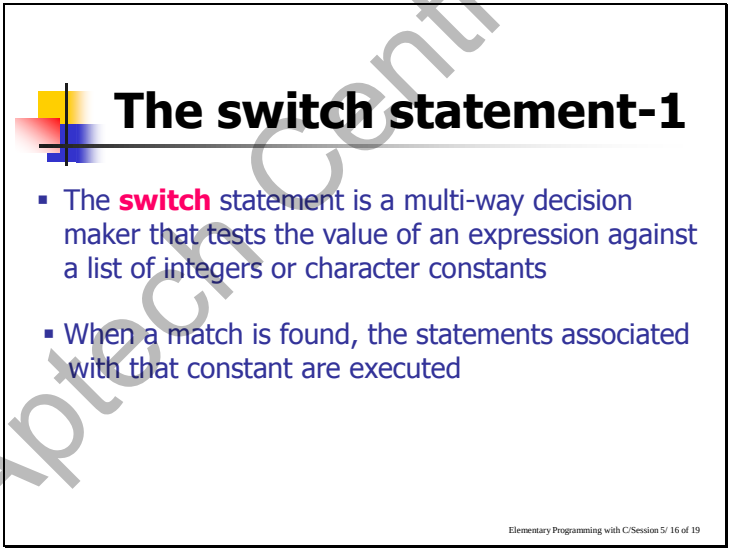
Subject: You have to explain the use of *nested if* statement and mention clearly the syntax of the *nested if* statement. The example has to be handled carefully so that the students get an idea and all students are clear about it. Later, students should be asked to write a simple program of their own.

Additional Information:
Reference Example:
This program calculates grades for a student in Mr. Smith's mathematics class.

```
#include <stdio.h>

void  main()
{
     //local declarations..
     int classification;          //classification of a student
     int tst1, tst2, tst3;        //three test scores
     float average;

     // input student's information
   printf( "Enter the student's classification 1 = freshman , 2 =
       sophomore, 3= junior , 4 = senior , 5 = graduate");

    scanf("%d",&classification);
     //check for a valid class and proceed accordingly
     if (classification >= 1 && classification <= 5)
     {
       //display classification
       printf( "Your classification is " );
        if (classification==1)
                    printf( "freshman");
        else if(classification==2)
                    printf ("sophomore");
        else if(classification==3)
                    printf("junior");
        else if(classification==4)
                    printf("senior");
        else if(classification==5)
                    printf("graduate");
        else
                    printf("Not a valid Number  ");
              printf("\n");

       // get the three test scores
       printf("Enter three test scores: ");
       scanf("%d%d%d",&tst1,&tst2,&tst3);

       //find average and display pass/fail status

       //calculate the average score
       average = (tst1 + tst2 + tst3)/3.0;

        //print the student's letter grade
```

```
        if (average >= 90)
            printf("Your grade is A \n");
       else   if (average >= 80)
            printf("Your grade is B\n");
        else if (average >= 70)
            printf("Your grade is C\n" );
        else if (average >= 60)
            printf("Your grade is D\n");
        else
            printf("Your grade is F\n");

       //determine pass/fail status
        if (average >= 70)
           printf("You passed! ");
        else
           printf("You failed! ");

     }
     else
       printf( "You entered an invalid class " );
}
```

Slides-16, 17, 18, and 19



The switch statement-2

Syntax:

```
switch (expression)
{
        case constant1:
                statement sequence
                break;
        case constant2:
                statement sequence
                break;
        case constant3:
                statement sequence
                break;
        .
        .
        default:
                statement sequence
}
```

Elementary Programming with C/Session 5/ 17 of 19

The switch statement-3

Program to check whether the entered lowercase character is vowel or 'z' or a consonant

Example

```
#include <stdio.h>
   main ()
   {
      char ch;
      clrscr ();

      printf ("\nEnter a lower cased alphabet (a - z) : ");
      scanf("%c", &ch);
```

contd.....

Elementary Programming with C/Session 5/ 18 of 19

The switch statement-4

Example

```
if (ch < 'a' || ch > 'z')
        printf("\nCharacter not a lower cased alphabet");
else
        switch (ch)
        {
                case 'a' :
                case 'e' :
                case 'i' :
                case 'o' :
                case 'u' :
                        printf("\nCharacter is a vowel");
                        break;
                case 'z' :
                        printf ("\nLast Alphabet (z) was entered");
                        break;
                default :
                        printf("\nCharacter is a consonant");
                        break;
                        }
        }
```

Elementary Programming with C/Session 5/ 19 of 19

Max Time: 25 Minutes

Subject: You need to explain the use of switch statement and mention clearly about the syntax of the *switch* statement. The example has to be handled carefully so that the students get an idea and that all students are clear about it. Later students should be asked to write a simple program of their own.

Solutions to Check your Progress

1. Conditional
2. T
3. Nested if
4. switch
5. statement 2

Solutions to Do It Yourself

1.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a,b ;
   clrscr();

   printf("Enter a number :");
   scanf("%d",&a);

   printf("Enter another number :");
   scanf("%d",&b);

   if( (a % b) ==0)
        printf("a is divisible by b");
   else
        printf("a is not divisible by b");
}
```

2.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int num1 , num2 ,prod;
   clrscr();

   printf("Enter a number :");
   scanf("%d",&num1);
```

```
   printf("Enter another number :");
   scanf("%d",&num2);

     prod = num1 * num2;

   if( prod >= 1000)
        printf("Product is greater than or equal to 1000");
   else
        printf("Product is less than 1000");
}
```

3.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int num1 , num2 ,diff;
   clrscr();

   printf("Enter a number :");
   scanf("%d",&num1);

   printf("Enter another number :");
   scanf("%d",&num2);

     diff = num1 - num2;

     if(diff == num1)
        printf("Difference is equal to value %d",num1);
     else if (diff == num2)
        printf("Difference is equal to value %d",num2);
   else
        printf("Difference is not equal to any of the values entered");

}
```

4.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int allow;
   float salary;
   char grade;
   clrscr();

   printf("Enter the grade :");
   scanf("%c",&grade);

   printf("\nEnter  the salary");
```

```
    scanf("%f",&salary);


      if(grade == 'A')
            allow = 300;
      else if (grade == 'B')
          allow = 250;
    else
          allow = 100;

    salary = salary + allow;

      printf("Salary = %f", salary);
}
```

5.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int marks;
   char grade;
   clrscr();

   printf("Enter the marks :");
   scanf("%d",&marks);


     if(marks > 75)
        printf(" Grade = A");
     else if (marks >= 60)
               printf(" Grade = B");
     else if (marks > =45)
               printf(" Grade = C");
     else if (marks >= 35)
              printf(" Grade = "D");
   else if(marks<35)
        printf(Grade = "E");
}
```

5.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 6 - Loop

Note: This TG maps to Session 9 of the book.

6.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

6.1.1 Objectives

By the end of this session, the learners will be able to:

- Understand 'for' loop in C
- Work with 'comma' operator
- Understand nested loops
- Understand the 'while' loop and the 'do-while' loop
- Work with break and continue statements
- Understand the exit( ) function

Difficulties

The students may find difficulty in understanding the following topic:

- Nested loops

Hence, the mentioned topic should be handled carefully.

6.1.2 Teaching Skills

You should be well-versed in iteration statements such as loops for teaching this session. You should be familiar with all types of loops in C and their usage and differences. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

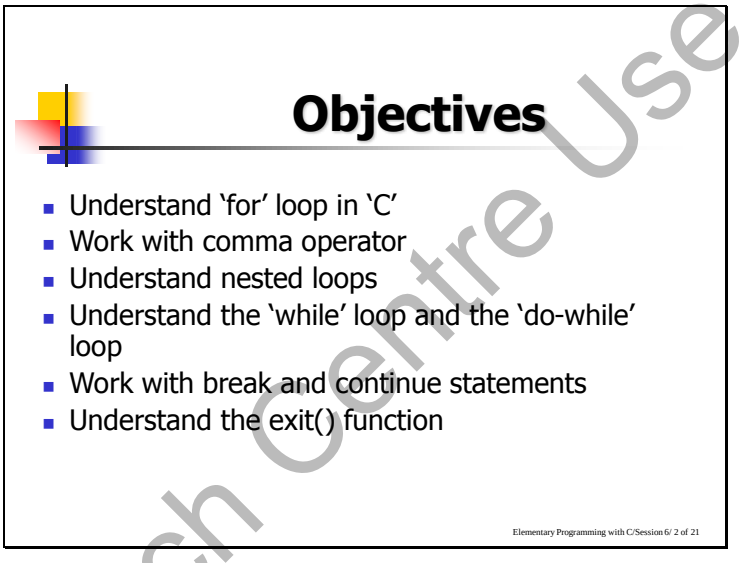
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to make the students understand the various types of loops and their usage in C and also the other statements that are used with loops such as break and exit.

6.2 In-Class Explanations

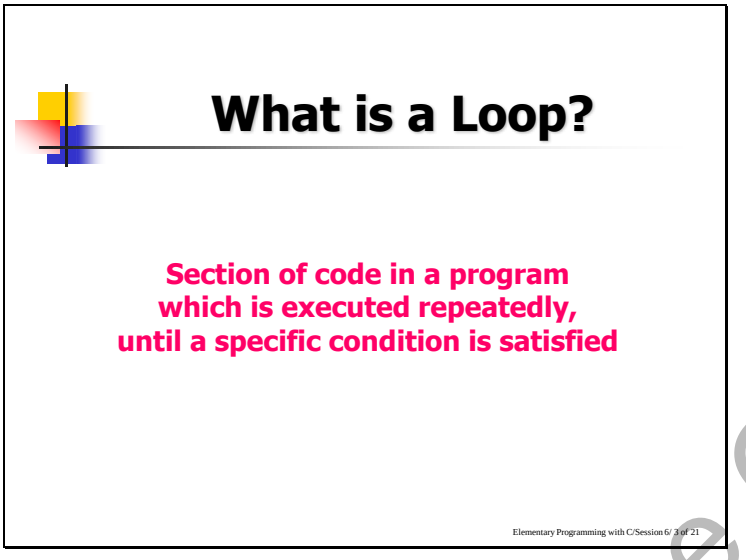
Slide 2

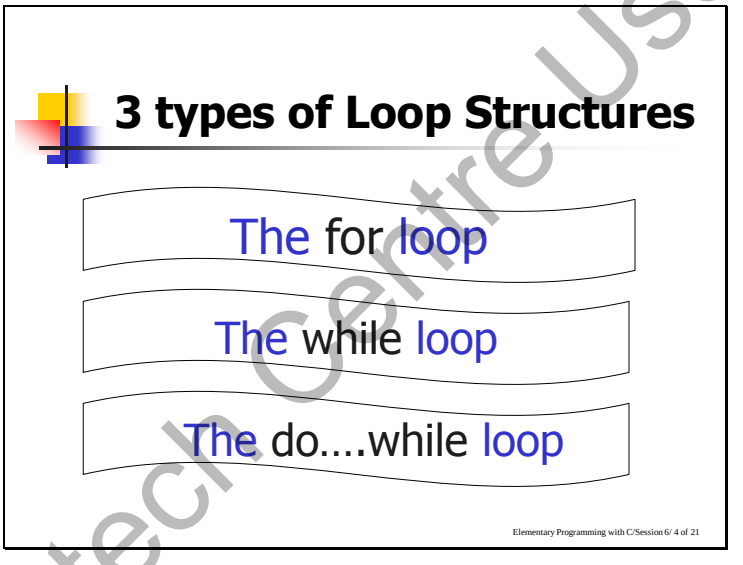


Max Time: 4 Minutes

Subject: List the objectives given on Slide 2. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slides-3 and 4

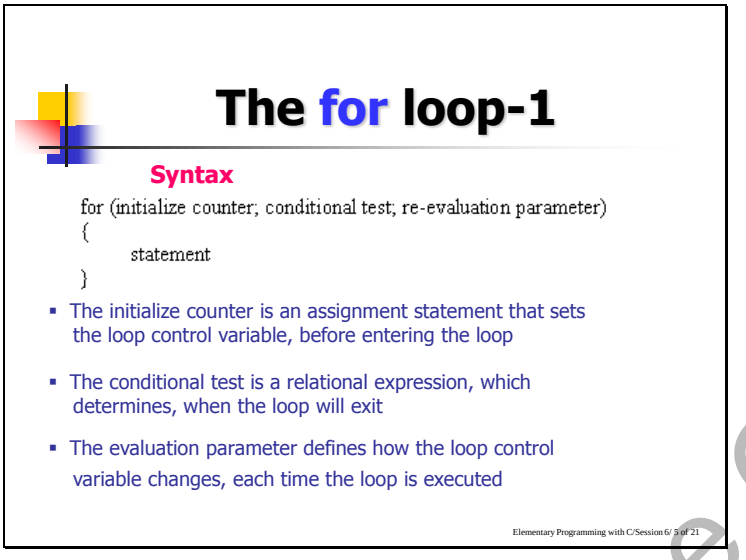


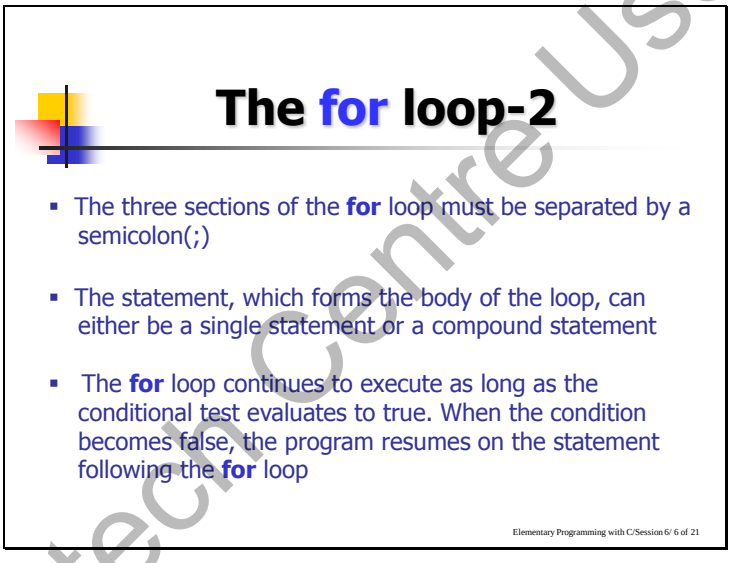


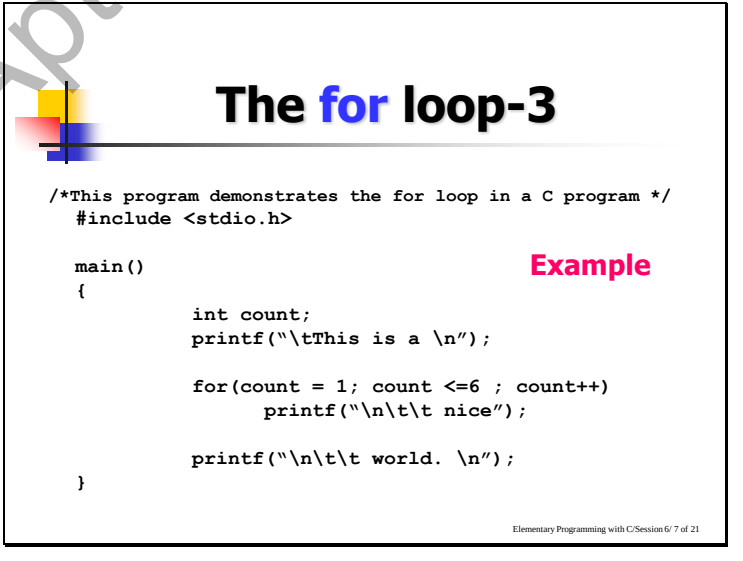
Max Time: 7 Minutes

Subject: The teaching of the session should start with the answers given by the students for the conditional statements. You should explain loops and explain the various loops in C.

Slides-5, 6, and 7







Max Time: 25 Minutes

Subject: The syntax for the **for** loop should be explained to students very clearly. An example is given on slide 7 to demonstrate the **for** loop. Also explain to the students how to increment /decrement the value of variable other than 1.

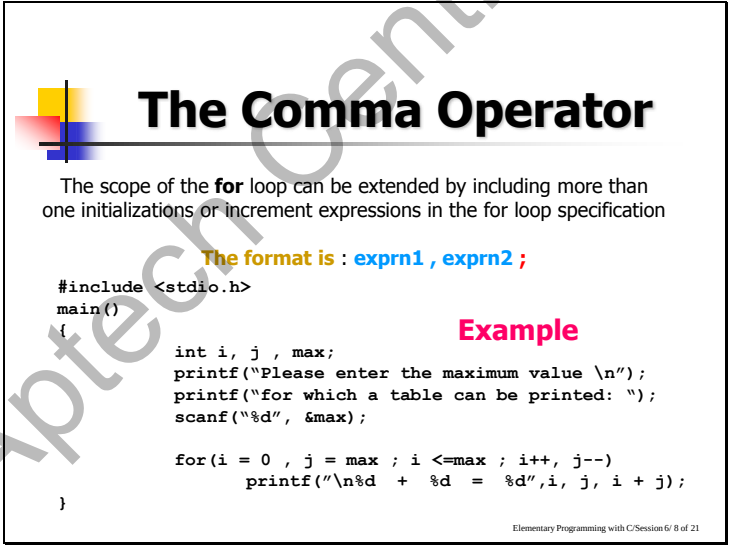
Additional Information:

In the **for** loop, we know that the value of the variable is incremented by 1 every time the loop is executed. This is automatically done and is called the 'default increment value'. If you want to increase this increment value, it can be done by using the corresponding arithmetic operator and the = operator.

Suppose we want to increment the variable each time by 2, then the statement can be written as:

```
for ( i=1; i<=10; i+=2 )
{
        ……….
}
```

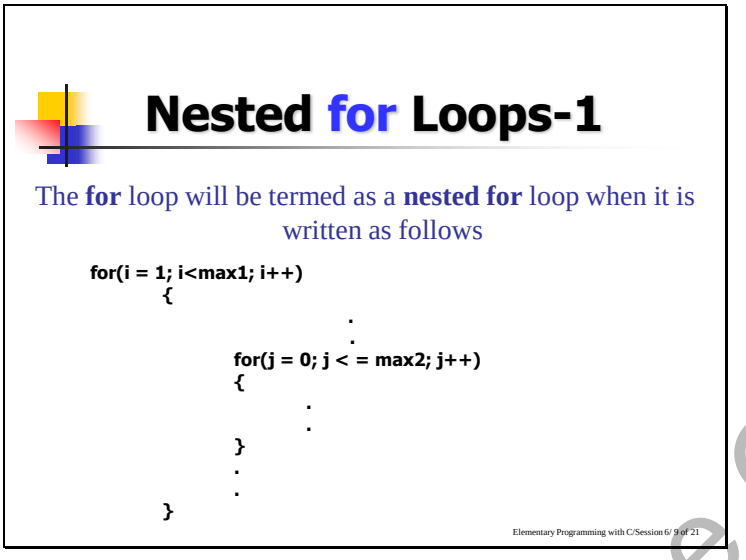
Slide-8

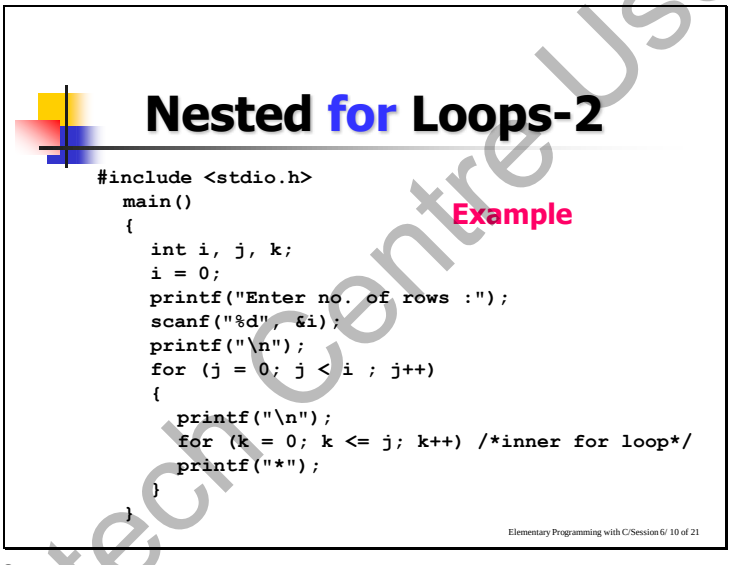


Max Time: 7 Minutes

Subject: Explain to the students the use of the **comma** operator with the example given on slide 8.

Slides-9 and 10

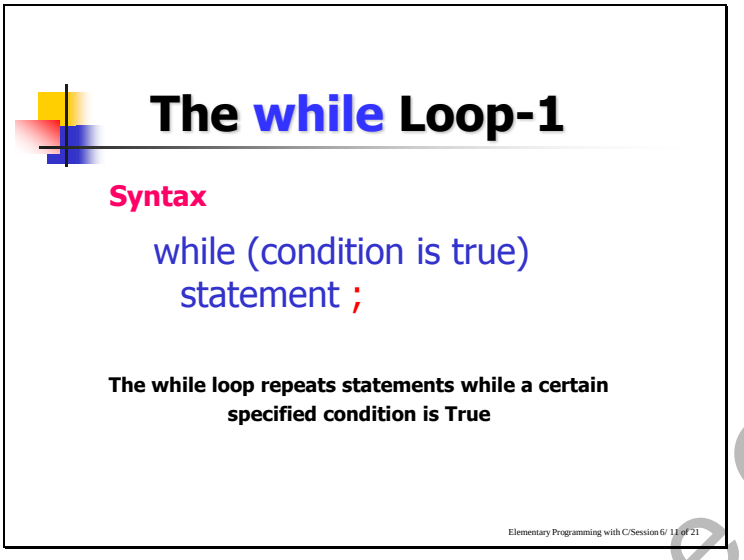


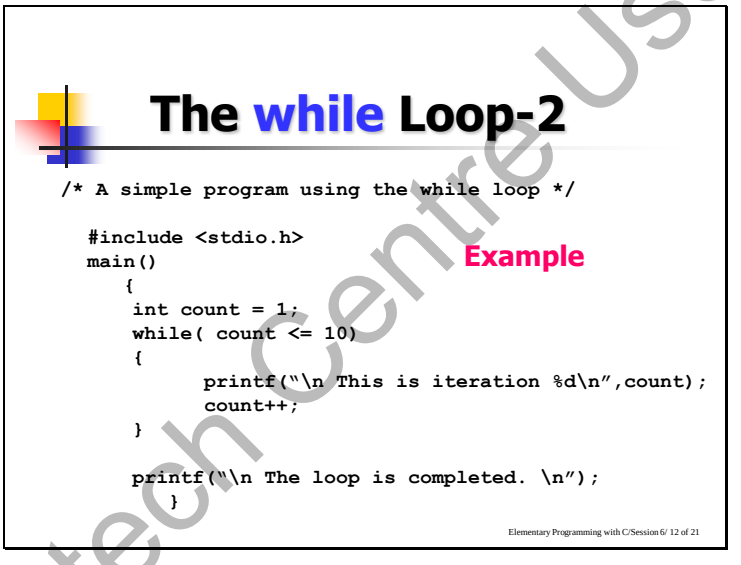


Max Time: 15 Minutes

Subject: Explain to the students the use of **nested for** loops and their working. This section can be better explained with the use of an example.

Slides-11 and 12





Max Time: 15 Minutes

Subject: Explain to the students the use of **while** loop and explain clearly the difference between the **for** loop and the **while** loop. Slide 12 can be used to explain the working of the **while** loop.

Additional Information:

After teaching them about the **While** Loop, you can ask what would happen in case we don't increment the variable. Discuss the necessity of the incrementing the variable.

Again, you can ask the students what is likely to happen if we don't initialize count to '1'.

We would have a compile time error, because in the condition, we would be checking if count<=10 when we have not initialized the variable at all.

We shall now write a program using the **while** loop to add 2, one thousand times.

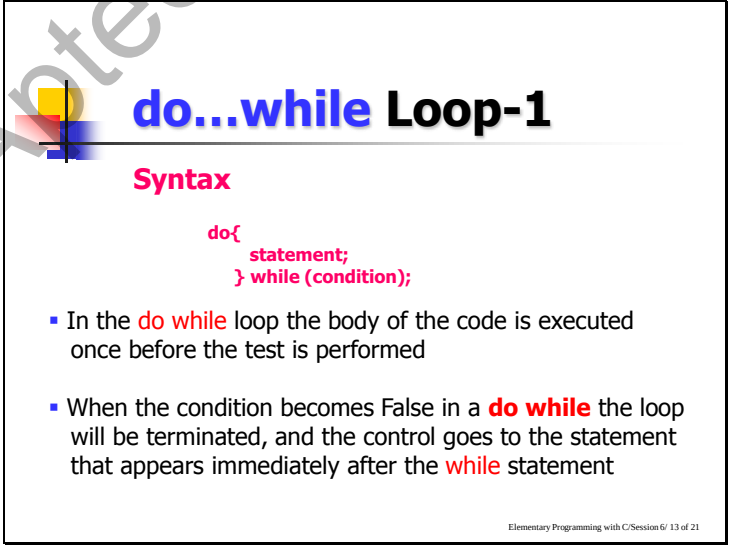
```
#include <stdio.h>

main()
{
      int sum = 0,times = 1;
      while( times<=1000)
      {
            sum = sum +2;
            times = times+1;
      }
      printf("Value of Sum = %d",sum);
}
```

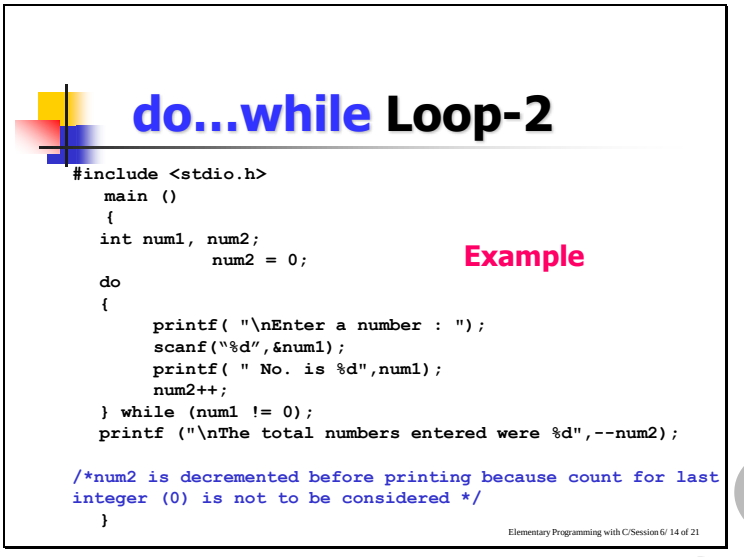
Here ***times*** is the variable which is commonly known as a counter because it performs the function of counting the number of times that the value 2 is added to *sum*.

In the given example, when the keyword **while** is met, the condition given with it is checked i.e., whether ***times*** is less than or equal to 1000. Depending on the result of this condition the loop may or may not be executed. In the beginning, ***times*** = 1 so the check proves to be **true**, then the two statements within the **while** loop construct are executed. The iteration or loop will continue 1000 times. The 1000th time, when 1 is added to ***times***, its value becomes 1001. The control is then transferred back to the **while** statement and when the check is performed, it turns out to **false** as 1001 is neither less than or nor equal to 1000.

When solving this problem, you can explain the concept of an accumulator, where the variable keeps accumulating the values, that it is similar to a counter variable. Instead of incrementing the value of the variable by some standard amount, the variable is incremented by whatever value is entered.

Slides-13 and 14





Max Time: 15 Minutes

Subject: First explain the working of the **do…while** loop and demonstrate as an example. Then You can explain the difference between the **while** loop and the **do..while** loop.

Additional Information:

We shall now write a program using the **do…while** loop to add 2, one thousand times.

```
#include <stdio.h>

main()
{
      int sum = 0,times = 1;
      do
      {
            sum = sum +2;
            times = times+1;
      } while( times<=1000);

      printf("Value of Sum = %d",sum);
}
```

With **while,** a condition is written which tells us that the statements within the **do… while** loop shall execute until that stated condition remains **true**. The statements are enclosed between the **DO – WHILE** keywords. The statements within the scope of a loop are called the **body** of the loop. This is a construct that executes a set of statements as long as the specified condition persists.

This is very similar to the **while** loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data and loops back to read again if it was unacceptable.

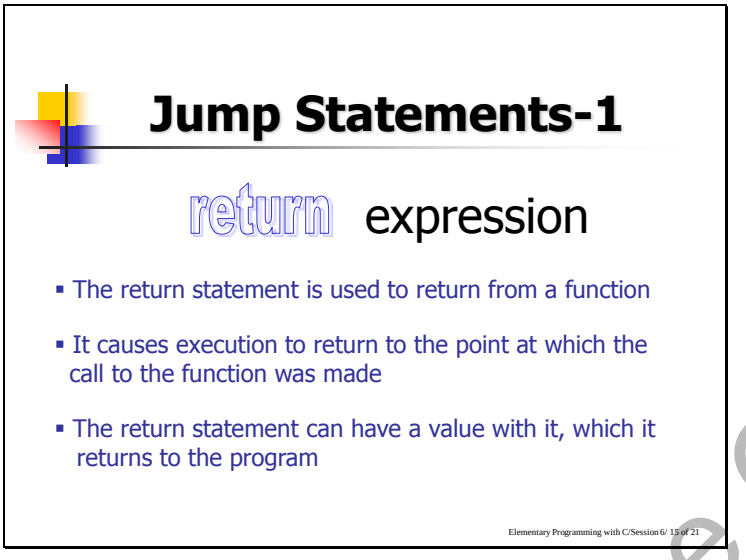
We shall now write a program using the **while** loop to accumulate 7, five thousand times.

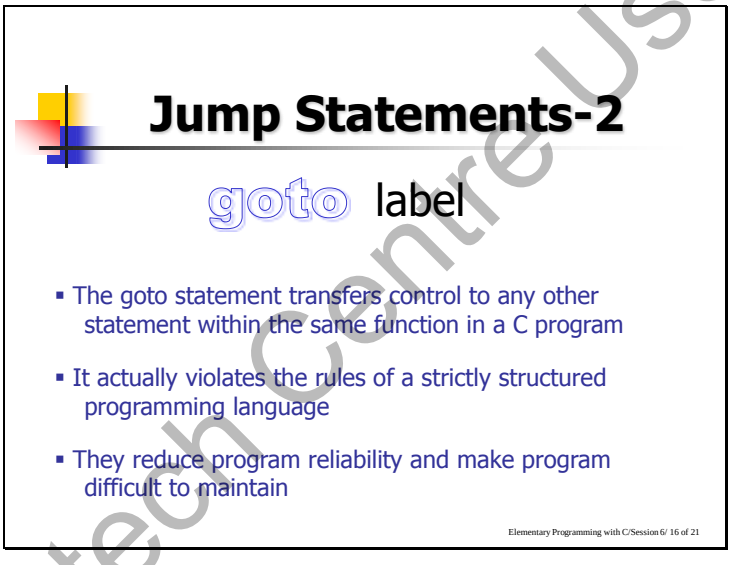
```
#include <stdio.h>

main()
{
      int sum = 0, times = 1;
      do
      {
            sum = sum +7;
            times = times+1;

      }while(times<=5000);
      printf("Sum = %d",sum);
}
```

Here ***times*** is a variable commonly known as a 'counter' because it performs the function of counting the number of times that 7 is added to *sum*.

In the given example, when the keyword **do** is encountered**,** the statement within the body is executed. After the statement is executed, the control reaches the **while** keyword. When **while** keyword is met, the condition given with it is checked i.e., whether ***times*** is less than or equal to 5000. Depending on the result of this condition the loop may or may not be executed further. The two statements within the ***body*** of the construct are executed. Then the **while** condition is checked and if it is **true,** then the control of the program is transferred back to the **do** statement and the statements are executed again. After that the **while** statement is checked once again. This iteration or loop will continue 5000 times. The 5000^{th} time, when 1 is added to ***times*** its value becomes 5001. The control is then transferred back to the **while** statement and when the check is performed, it turns out to be **false** as 5001 is neither less than nor equal to 5000.

Slides-15 and 16

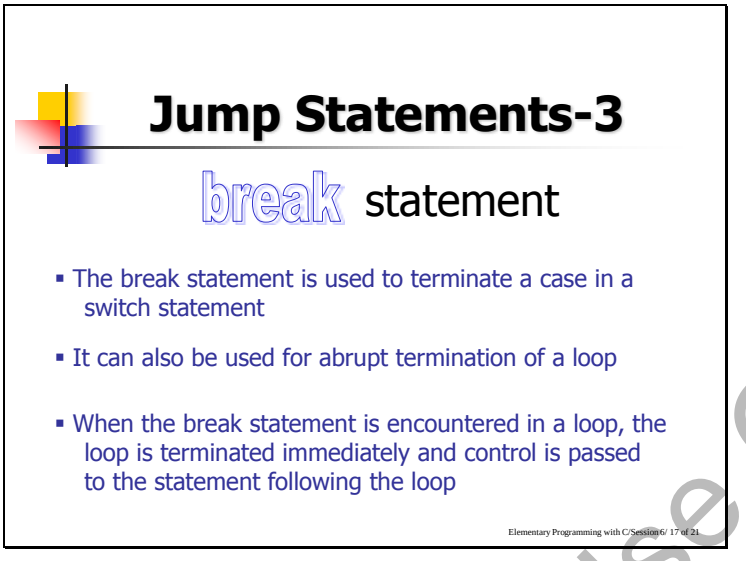


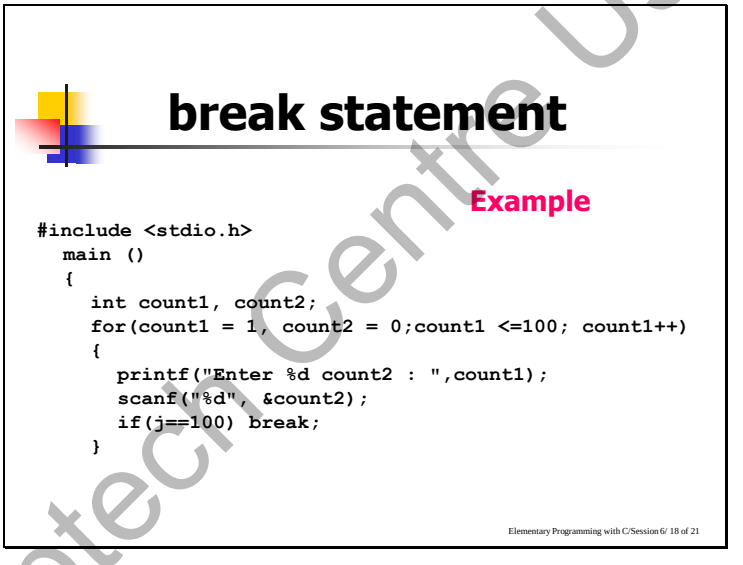


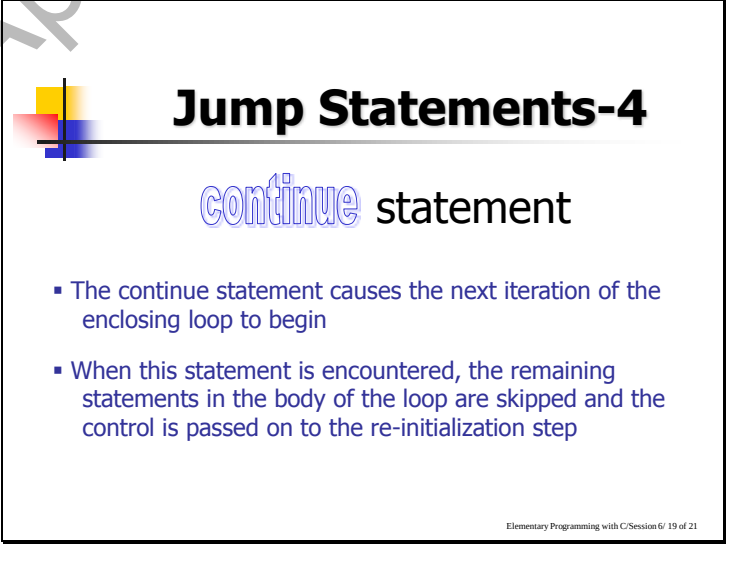
Max Time: 7 Minutes

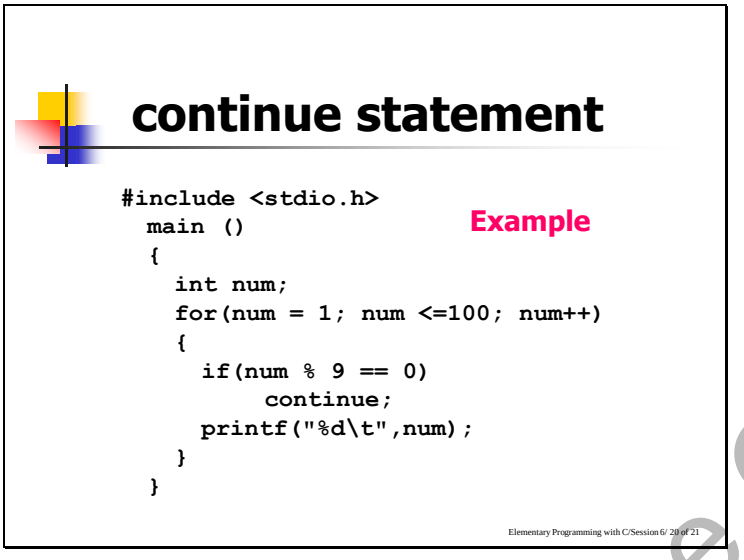
Subject: You can explain the use of the **return** and the **goto** statements in C programs. Along with this, you can also discuss the disadvantages of the use of the **goto** statements. This can be discussed with reference to its use in programs having large number of lines of code.

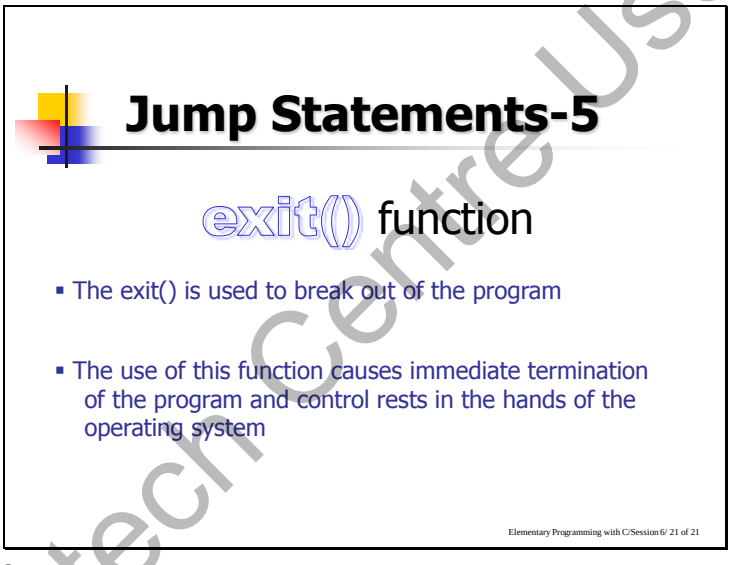
Slides-17, 18, 19, 20, and 21











Max Time: 27 Minutes

Subject: Slides 17 and 18 explain the use of the **break** statement with an example. Slides 19 and 20 explains the use of the **continue** statements with an example. Slide 21 can be used to show the use of the *exit( )* function.

Additional Information:

It is often not necessary to process the complete loop. For example, if we wish to find some particular data within a loop, we may break out of the loop as soon as we get what we need. It's not necessary to proceed with the loop till the end even after getting the required data; this can be achieved by the keyword **break** in C.

Additional Example of the 'break' statement

This loop searches a list of 1000 numbers to see if there is a 599 in it.

```
#include <stdio.h>
main()
{
      int i ;
      for(i=1;i<=1000;i++)
      {
                if( i == 599)
            break;
      }
}
```

In the given example, *i* is a variable with 1 as its start value and 1000 as the end value. When the loop is being executed, if it comes across the number (599) the loop will stop executing, because once it finds 599 it meets the **break** statement and exits from the loop.

You can discuss with the students the advantages of using the **break** statement.

Additional Example of the 'continue' statement

```
      #include <stdio.h>
      main()
      {
            for(i=1;i<=30;i++)
            {
                        if(i % 5 == 0 )
                              continue;
            printf("%d",i);
            }
      }
```

Here, our requirement is to print those numbers, which are not divisible by 5. To achieve this, we make use of the **continue** statement such that whenever we encounter a number that is a multiple of 5 we skip the rest of the loop. Only if we get a number that is not divisible by 5, we print it.

Solutions to Check Your Progress

1. Loop
2. while loop
3. semicolon
4. do…while loop
5. return
6. goto
7. exit

Solutions to Do It Yourself

1.

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

```
#include <conio.h>

void main()
{
   int num = 100;

   while(num>=5)
   {
         printf("%d\t",num);
         num = num - 5;
   }
}
```

2.

```
#include <stdio.h>
void main()
{
   int count,num1,num2;
   int sum;
   sum = 0;
   printf("\n Enter value for num1 : ");
   scanf("%d",&num1);
   printf("\n Enter value for num2 : ");
   scanf("%d",&num2);

   for(count=num1; count<=num2; count++)
   {
        if( count%2 != 0)
        {
         sum = sum + count;
          }
   }
   printf("The sum of all the odd numbers are :%d", sum);
}
```

3.

```
#include <stdio.h>
void main()
{
   int a,b,c,cnt;
   a=0;
   b=1;
   c=0;
   cnt=0;
   do
   {
       c= a + b;
       printf("%d\t",c);
       a=b;
       b=c;
       cnt++;
   } while(cnt <=10);
         }
```

4 .

a)

```
#include <stdio.h>
void main()
{
      int num,j;
      for(num=1; num<=5; num++)
      {
        for( j=1; j<=num; j++)
        {
           printf ("%d", j);
        }
         printf ("\n");
      }
}
```

b)

```
#include <stdio.h>
void main()
{
      int num, j;
      for(num= 5; num>=1; num--)
      {
            for( j=1; j<=num; j++)
            {
               printf ("%d", j);
            }
            printf("\n");
      }
 }
```

5.

```
#include <stdio.h>
void main()
{
    int num,j;
    for(num=7; num>=1; num--)
    {
        for( j=1; j<=num; j++)
        {
          printf ("*");
        }
        printf ("\n");
    }
}
```

6.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 7 - Arrays

Note: This TG maps to Session 11 of the book.

7.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

7.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain array elements and indices
- Define an array
- Explain array handling in C
- Explain how an array is initialized
- Explain string / character arrays
- Explain two dimensional arrays
- Explain initialization of multidimensional arrays

Difficulties

The students may find difficulty in understanding the following topics:

- The storage mechanism of arrays
- Using arrays in loops
- Example of Two-dimensional arrays

Hence, these topics should be handled carefully.

7.1.2 Teaching Skills

You should be well-versed in arrays and their related concepts for teaching this session.
Initializing of arrays, string and character arrays, one-dimensional, two-dimensional and multi-dimensional arrays are some of the concepts you should know well.

You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

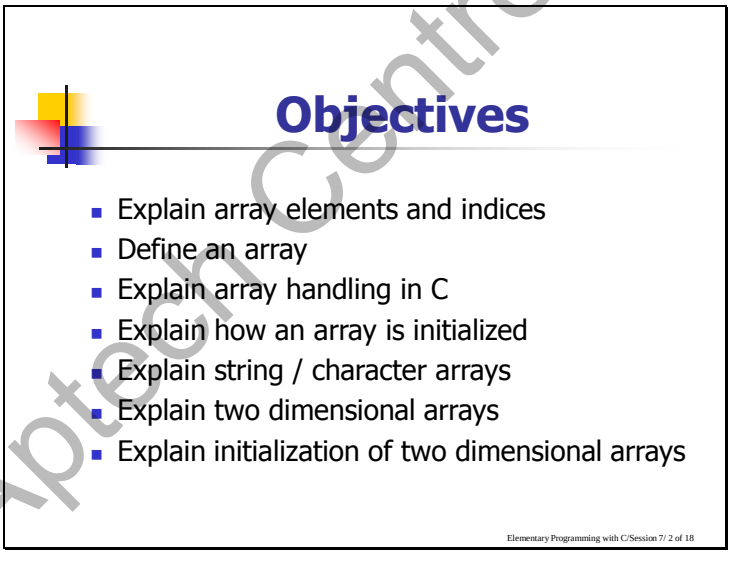
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the concept of arrays in C and teach them to write programs using arrays. It also familiarizes students with two-dimensional and Multi-dimensional arrays and their usage.

7.2 In-Class Explanations

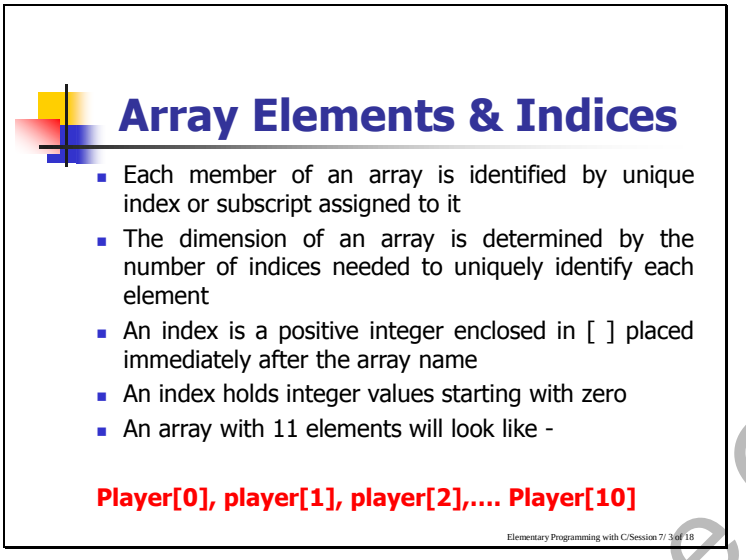
Slide 2



Max Time: 1 Minute

Subject: List the objectives given on Slide 2 and inform the students what they are going to learn in next 2 hours. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slide-3



Max Time: 4 Minutes

Subject: The session should start with the answers given by the students for the variables. You should define array and explain how it is different from variable. You should clarify the terms like, members, elements, dimension, and index of an array. Emphasis should be given on the fact that the index in arrays starts with 0 and not 1 and hence the last index will be N-1. The player example should be explained assigning a player name against each array element.

The following example can be used to differentiate between a normal variable and array:

Additional Information:

An array lets you declare and work with a collection of values of the same data-type. For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

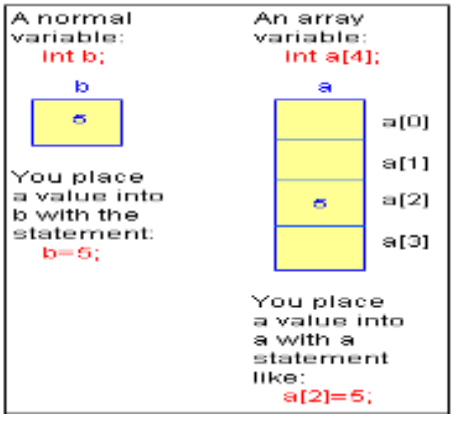
```
int a, b, c, d, e;
```

This is okay, but what if you needed a thousand integers? An easier way is to declare an array of five integers:

```
int a[5];
```

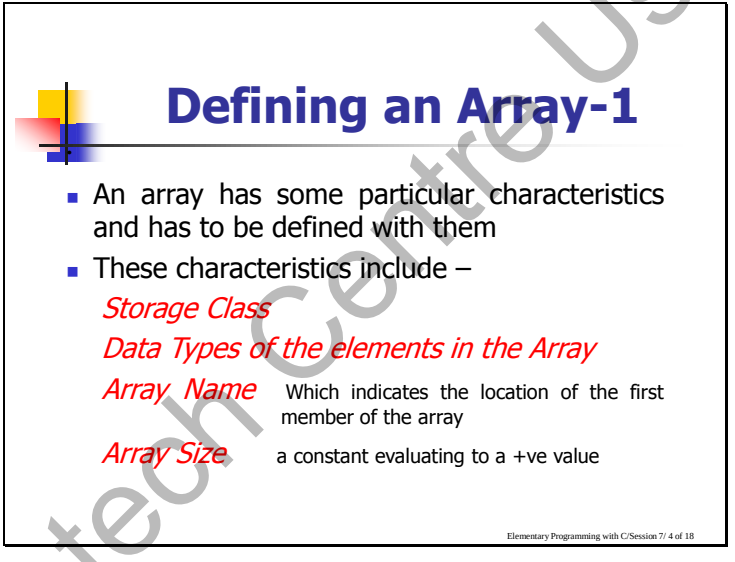
The five separate integers inside this array are accessed by an **index**. All arrays start at index zero and go to n-1 in C. Thus, **int a[5];** contains five elements. For example:

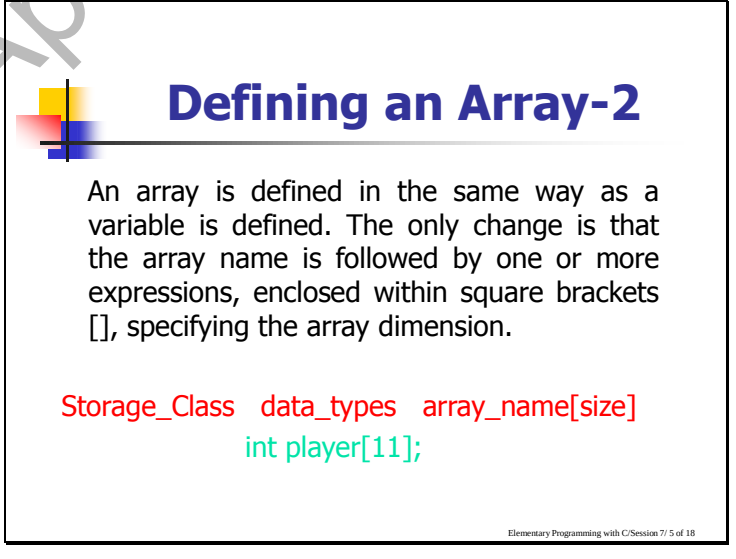
```
int a[5];

a[0] = 12;
a[1] = 9;
a[2] = 14;
a[3] = 5;
a[4] = 1;
```

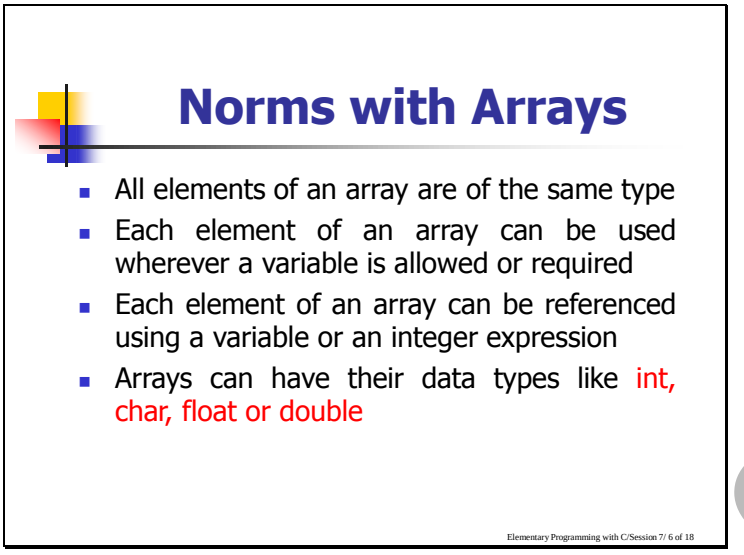


Note: In the given example Array declaration is used, but while teaching, that can be avoided as students are not taught defining an array.

Slides-4, 5, and 6







Max Time: 15 Minutes

Subject: The array declaration characteristics like Storage class, Data types, Array names, and Array size should be explained to students, their differences, and their uses. Slide 5 provides syntax for defining arrays and how it differs with variable declaration. It should be clarified that the array name and variable name have to be different in one program.

Additional Information:

Storage class specifiers tell the compiler how to store the subsequent variable. One of these specifiers (auto) is virtually never used since all non-global variables are, by default, assumed to be auto. The remaining specifiers are as follows:

extern

Because C allows separately compiled modules and because a global variable must be declared in only one of those files, there must be some way of telling all the files about the existence of that global variable. The extern specifier tells the compiler that the variables that follow have been declared in another module. When the linker links the two modules, all references to the external variables are resolved.

static

They differ from normal variables as they maintain their value between calls. The effect of static variable depends on whether it is applied to a local variable or a global variable:

√ **static Local Variables:** When the static specifier is applied to a local variable, the compiler creates a permanent storage for it, as it does for global variables. The difference is that a static local variable is still 'known' only to the code block in which it is declared.

√ static local variables are very important for the creation of stand-alone functions where the retention of values is required, given that they prevent the necessity to use global variables with the attendant risk of undesirable side effects.

√ **static Global Variables:** static applied to a global variable creates a global variable. Generally, a static global variable is used where a local static cannot do the job.

The students should be provided with only a brief definition of Storage class, it should not be dealt in depth. It should be clarified that storage class are optional while rest all characteristics are essential. The following point can be referred:

➢ **Arrays** are a way of implementing **fixed** length **vectors** of a given type

- We use square brackets, [ ], for arrays
- Syntax for definition:
  <type name> <variable name> [<length>]
  <length> must be a **constant integer** value i.e. one cannot use another variable here
- Examples:

```
int nums[5];
float x[128];
```

  defines an array of 5 integers called nums and an array of 128 floats called x
- Referring to arrays is easy - just write the variable name followed by the element number in square brackets

Convention - array elements in C starts at 0

You should explain the different data types used in arrays like int, char, float, or double. A student should be then asked to provide examples of each data types. The difference between **player[3]=player[2]+5; player[0]+=2;** and **player[i/2+1]** mentioned in the session should be clarified to the students.

The following points can be referred:

- Each item is stored sequentially in the array
- Each element can be accessed individually
  - give the array name and the index or position within the array
- Individual elements behave as if they were normal variables of the same type
- Arrays must be declared just like variables
  type arrayname[size];
  size must be a constant expression

```
char string[20];
int number[50];
```

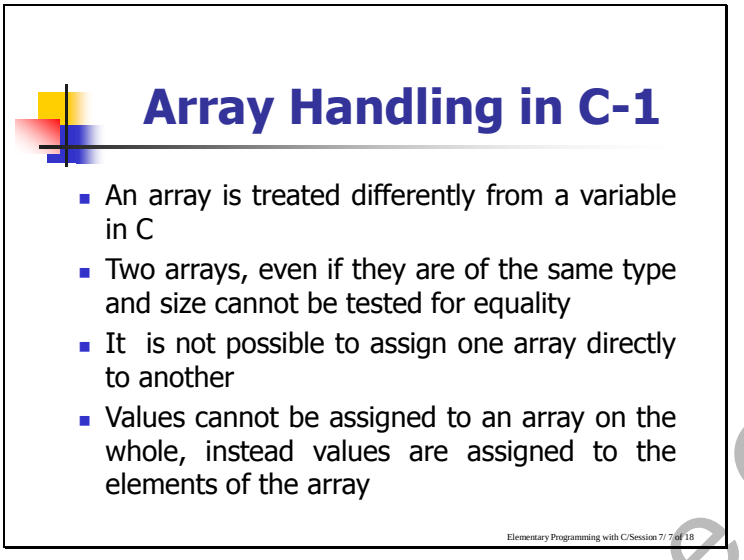
- Array elements in C are numbered from zero to the number the stated size

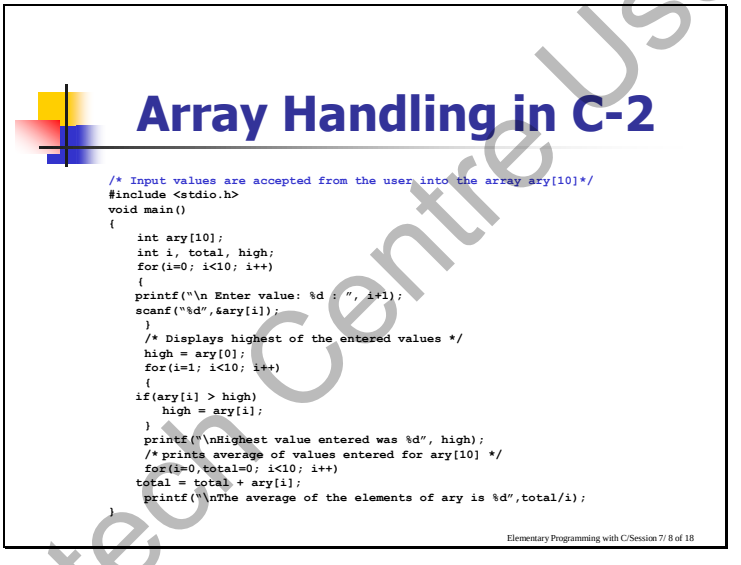
```
int num[10];
```

  -- declares an array with ten elements
  -- num[0] to num[9]
- These elements can be used in place of variables

```
num[0]=23;
num[4]=(num[2]+27)-num[3];
total=num[0]+num[1]+num[2]+num[3]+num[4]+num[5]+num[6]+num[7]+num[8]
+num[9];
```

Slides-7 and 8





Max Time: 15 Minutes

Subject: The program mentioned in the session should be explained to students clearly and output expected should be explained. This program should be handled carefully and should it be seen that all students are clear with it. Some reference program can be written and students should be asked to identify the output. Later students should be asked to write a simple program of their own.

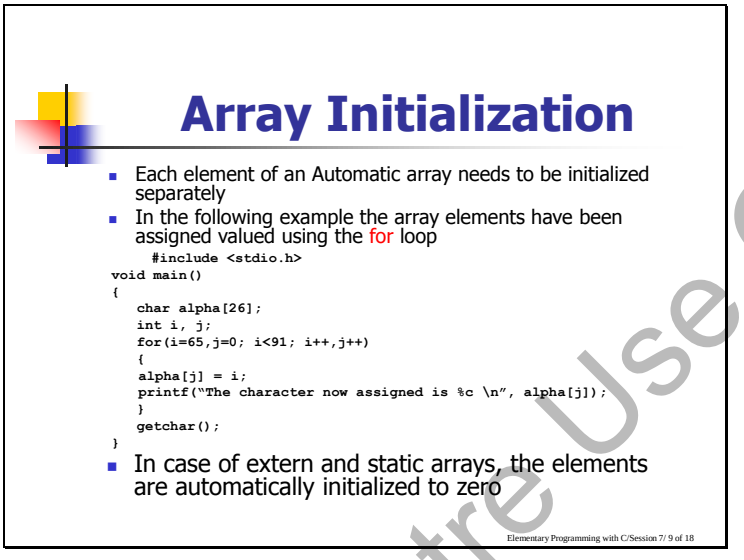
Additional Information:

The following code initializes the values in the array sequentially and then prints them out:

```
#include <stdio.h>

int main()
{
    int a[5];
    int i;
```

```
    for(i=0; i<5; i++)
        a[i] = i;
    for(i=0; i<5; i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

Slide-9



Max Time: 15 Minutes

Subject: You should clearly explain how to initialize an array, followed by explaining the program mentioned in the session. The following points can be referred for teaching the concept:

Additional Information:

Arrays may be initialized at the time of declaration. The following example initializes a ten-element integer array:

```
int i[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

Character arrays which hold strings allow a shorthand initialization, e.g.:

```
char str[9] = "I like C";
```

which is the same as:

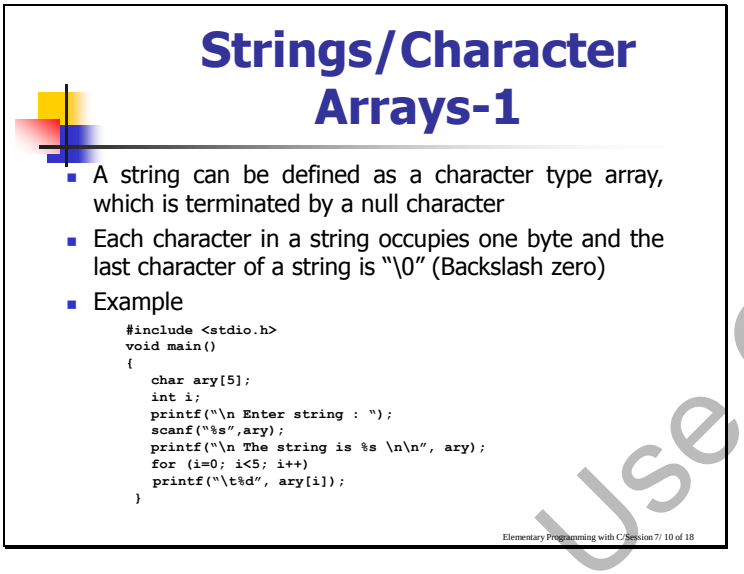
```
char str[9]={ 'I',' ','l','i','k','e',' ','C','\0' };
```

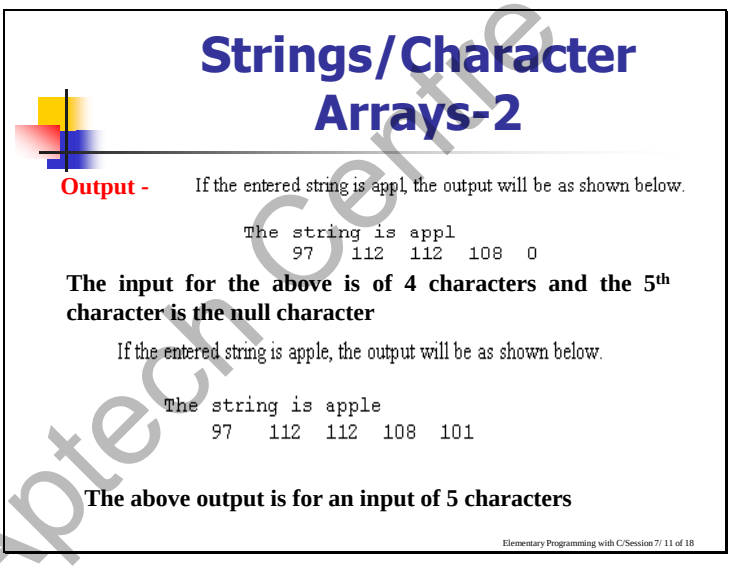
When the string constant method is used, the compiler automatically supplies the null terminator. Multi-dimensional arrays are initialized in the same way as single-dimension arrays, e.g.:

```
int sgrs[6][2] =
          {    1,1,
               2,4,
               3,9,
```

```
        4,16,
        5,25,
        6,36
};
```

Slides-10 and 11





Max Time: 15 Minutes

Subject: You should first explain the concept involved in string array and the switch over to explain the example. The "\0" concept should be made clear to the students. The difference between the values of **appl** and **apple** mentioned in the example should be explained.

Additional Information:

A string in C is simply an array of characters. The following line declares an array that can hold a string of up to 99 characters.

```
char str[100];
```

It holds characters as you would expect: **str[0]** is the first character of the string, **str[1]** is the second character, and so on. But why is a 100-element array unable to hold up to 100 characters?

Because, C uses *null-terminated strings*, which means that, the end of any string is marked by the ASCII value 0 (the null character), which is also represented in C as **'\ 0'**.

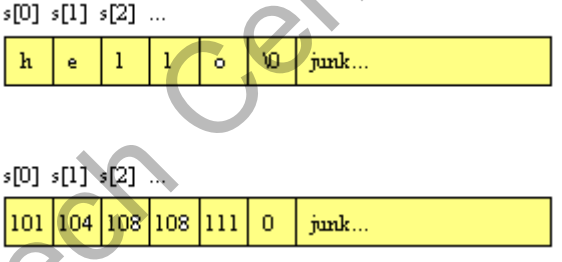
Null termination is very different from the way many other languages handle strings. For example, in Pascal, each string consists of array of characters, with a length byte that keeps count of the number of characters stored in the array. This structure gives Pascal a definite advantage when you ask for the length of a string. Pascal can simply return the length byte, whereas C has to count the characters until it finds **'\0'**. This fact makes C much slower than Pascal in certain cases, but in others it makes it faster, as we will see in the examples here.

As C provides no explicit support for strings in the language itself, all of the string-handling functions are implemented in libraries. The string I/0 operations (gets, puts, and so on) are implemented in <stdio.h>, and a set of fairly simple string manipulation functions are implemented in <string.h> (on some systems, <strings.h>).

The fact that strings are not native to C forces you to create some fairly roundabout code. For example, if you want to assign one string to another string; that is, you want to copy the contents of one string to another. You know that you cannot simply assign one array to another. You have to copy it element by element. The string library (<string.h> or <strings.h>) contains a function called **strcpy** for this task. Here is an extremely common piece of code to find in a normal C program:

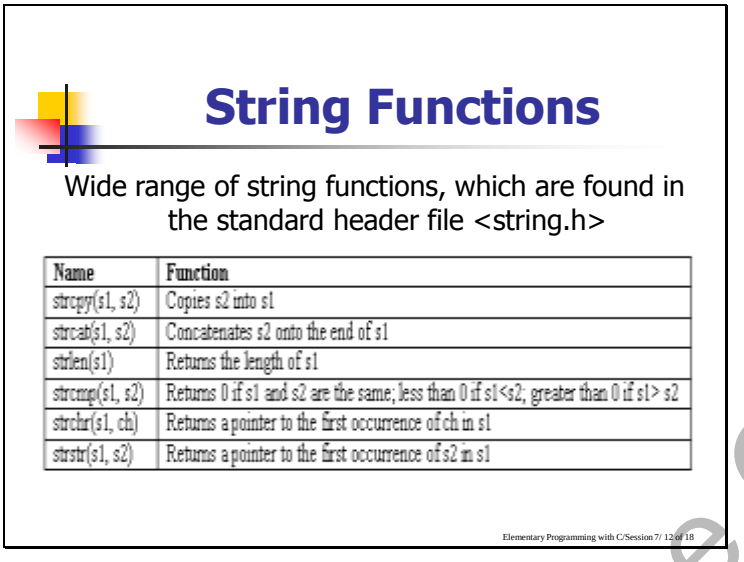
```
char s[100];
strcpy(s, "hello");
```

After these two lines execute, the following diagram shows the contents of s:



The top diagram shows the array with its characters. The bottom diagram shows the equivalent ASCII code values for the characters, and is how C actually thinks about the string (as an array of bytes containing integer values).

Slide-12



Max Time: 20 Minutes

Subject: You should explain all the string functions mentioned in the session and explain the difference between them. Sample examples to should be showed using the string functions. For few functions students should be asked to write program.

Additional Information:

String Copy & Concatenate

```
Strcpy(target_string,source_string);
strcpy(string, "coffee");

strcat(first_string, second_string);
```

Puts first_string followed by second_string into first_string, properly NULL terminated

```
strcpy(string1,"cof");
strcpy(string2,"fee");
strcat(string1, string2);
```

Results in string1 containing "coffee"
— (NULL terminated)

String Comparison

- Compare character by character
- Result depends on first character position that is different in two strings
- ASCII code order
- Shorter sting comes first
- dog before doghouse
- cat before dog

- result = strcmp(string1,string2);
- 0 if equal
- -1 if string1 comes before string2
- +1 if string1 comes after string2

String Length

- Number of characters in string
- Counts characters up to NULL terminator
- Does not count NULL terminator

```
char string[50];
strcpy(string,"Hello");
length = srtlen(string);
```

- length contains 5

Examples:

The following code shows how to use **strcpy** in C:

```
#include <string.h>
int main()
{
    char s1[100],s2[100];
    strcpy(s1,"hello"); /* copy "hello" into s1 */
    strcpy(s2,s1);      /* copy s1 into s2 */
    return 0;
}
```

strcpy is used whenever a string is initialized in C.

You use the **strcmp** function in the string library to compare two strings. It returns an integer that indicates the result of the comparison. Zero means the two strings are equal, a negative value means that **s1** is less than **s2**, and a positive value means **s1** is greater than **s2**.

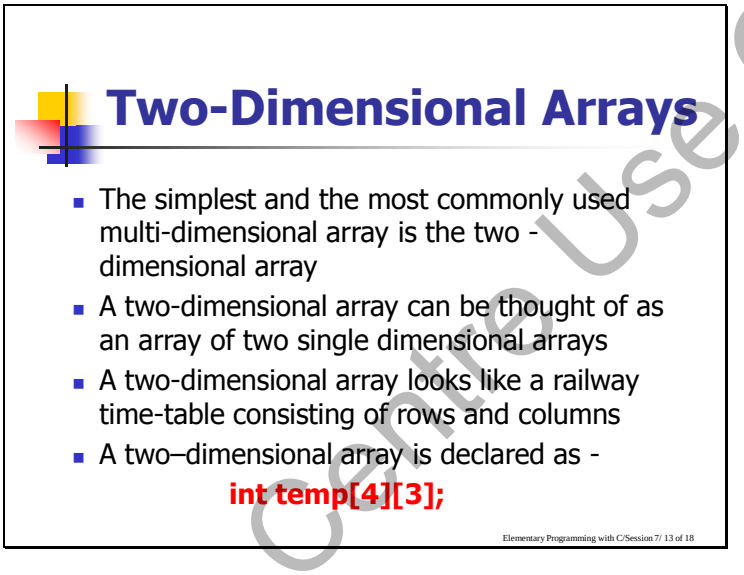
```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[100],s2[100];
     gets(s1);
    gets(s2);
    if (strcmp(s1,s2)==0)
        printf("equal\n");
    else if (strcmp(s1,s2)<0)
        printf("s1 less than s2\n");
    else
        printf("s1 greater than s2\n");
    return 0;
}
```

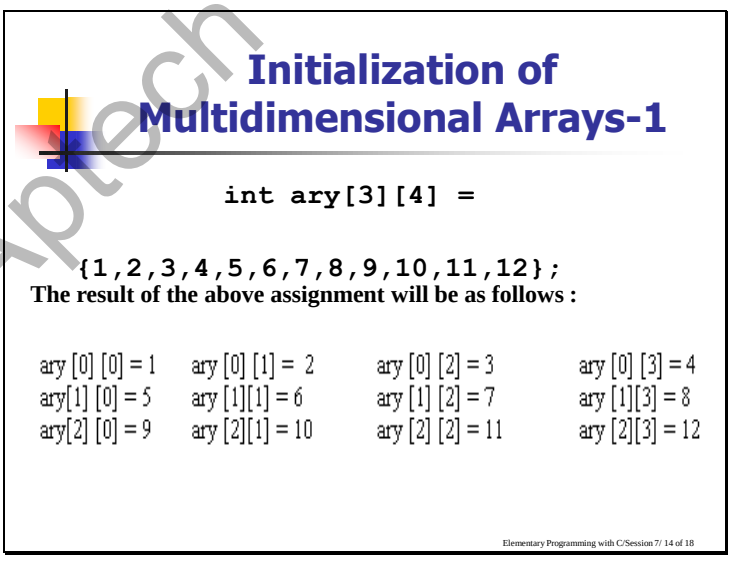
Other common functions in the string library include **strlen**, which returns the length of a string, and **strcat** which concatenates two strings. The string library contains a number of other functions, which you can peruse by reading the main page.

To get you started building string functions and to help you understand other programmer's code (everyone seems to have his or her own set of string functions for special purposes in a program), we will look at two examples, **strlen** and **strcpy**. Following is a strictly Pascal-like version of **strlen**:

```
int strlen(char s[])
{
    int x;
    x=0;
    while (s[x] != '\0')
         x=x+1;
    return(x); }
```

Slides-13 and 14





Max Time: 15 Minutes

Subject: You should differentiate between a Single-dimensional array and two-dimensional array. It should be explained that two-dimensional array is the most commonly used multi-dimensional array. The concept behind **int temp[4][3]** should be explained clearly, explaining how values will be

assigned in matrix format. The students should be clear that the values are assigned row wise in the example mentioned in slide 15. A sample array can be given to students and ask them to assign values.

The following example can be referred:

Additional Information:

Multi-dimensional Arrays:

- Number of indices an array has is called the dimension of the array
- We have looked at one dimensional arrays
- A two-dimensional array has rows and columns
  - Like a chess board or noughts & crosses
- A three-dimensional array has rows, columns, and planes
  - Three-dimensional chess board
  - Three-dimensional noughts & crosses

To **define** multi-dimensional arrays just add extra brackets: e.g.

```
float x[5][6];
```

defines a array of floats called 'x'

Using multi-dimensional arrays is just as easy: e.g.

```
x[5][3] = x[0][0];
```

Two-Dimensional Arrays:

In C, a **two-dimensional array** is declared as shown in the following example:

```
int d[10][20];
```

Two-dimensional arrays are stored in a row-column matrix. The first index indicates the row.

Eg:

```
8  6  9  52
2  14 24  87
13 76  5   3
```

- Declaration

```
int number[3][4];
```

- Accessing an element

```
answer = number[1][3];
```

- Result in answer is 87
  - Remember all array numbering starts at zero
  - Count row 0, row 1
  - Count column 0, column1, column2, column3

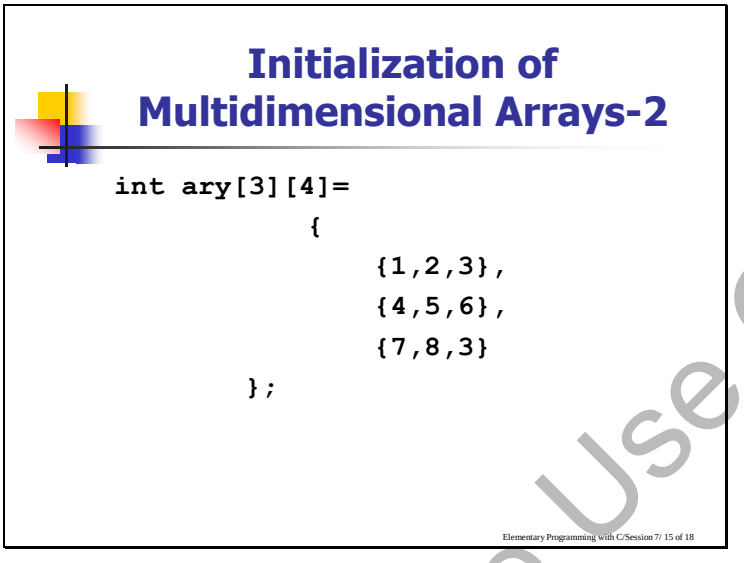
Single Dimensional Array:

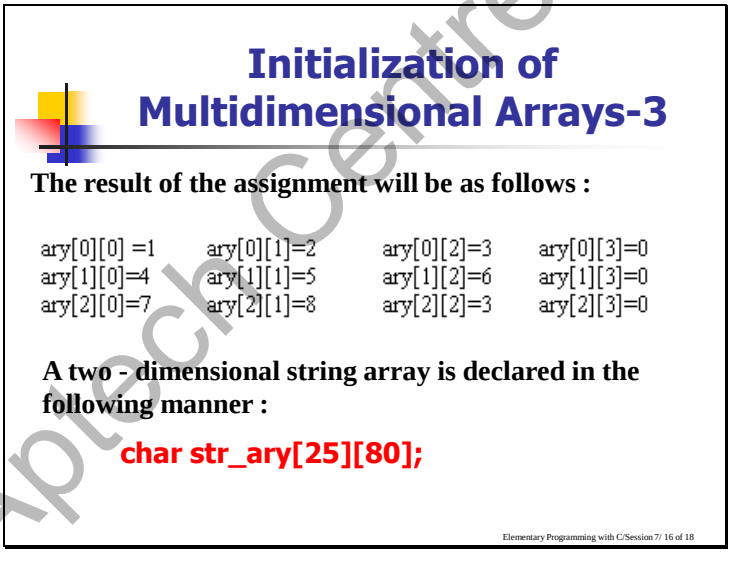
The general form for declaring a **single-dimension array** is:

type var_name[size];

In C, all arrays have zero as the index of their first element. Therefore, a declaration of char p[10]; declares a ten-element array (p[0] through p[9]).

Slides-15, 16, 17, and 18





Two-Dimensional Array-1

Example

```
#include <stdio.h>
#include <string.h>
void main ()
{
   int i, n = 0;
   int item;
   char x[10][12];
   char temp[12];

   clrscr();
   printf("Enter each string on a separate line\n\n");
   printf("Type 'END' when over \n\n");

   /* read in the list of strings */
   do
   {
      printf("String %d : ", n+1);
      scanf("%s", x[n]);
   } while (strcmp(x[n++], "END"));
/*reorder the list of strings */
```

contd….

Elementary Programming with C/Session 7/ 17 of 18

Two-Dimensional Array-2

Example

```
n = n - 1;
   for(item=0; item<n-1; ++item)
   {
      /* find lowest of remaining strings */
         for(i=item+1; i<n; ++i)
         {
                  if(strcmp (x[item], x[i]) > 0)
                  {
                           /*interchange two stings */
                           strcpy (temp, x[item]);
                           strcpy (x[item], x[i]);
                           strcpy (x[i], temp);
                  }
          }
     }
 /* Display the arranged list of strings */
 printf("Recorded list of strings : \n");
      for(i = 0; i < n ; ++i)
      {
         printf("\nString %d is %s", i+1, x[i]);
      }
 }
```

Elementary Programming with C/Session 7/ 18 of 18

Max Time: 20 Minutes

Subject: You should explain how to initialize a two-dimensional string array. Care should be taken while explaining the program to students and the output expected should be explained. At the end, the students should be asked to review the entire session.

Solutions to Check Your Progress

1. Array
2. index, subscript
3. False
4. False
5. Equality
6. NULL
7. True
8. strcmp, strcpy

Solutions to Do It Yourself

1. Program to arrange the following names in alphabetical order.

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

main()
{
   char name[8][20] =
   {"George","Albert","Tina","Abdul","Xavier","Roger",
         "Tim","Williams"};
   char temp[20];
   int i,j;

   clrscr();

   for(i=0;i<=7;i++)
   {
         for(j=i+1;j<8;j++)
         {
            if (strcmp(name[j],name[i])<1)
            {
              strcpy(temp, name[i]);
              strcpy(name[i],name[j]);
              strcpy(name[j],temp);

            }
         }
   }
   for(i=0;i<=7;i++)
         printf("%s\n",name[i]);

}
```

2. Program to count the number of vowels in a line of text.

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

```
main()
{
   char stat[20];
   int len,i,cnt=0;
   printf("Enter a word : ");
   scanf("%s",stat);

   for(i=0;i<strlen(stat);i++)
         if(stat[i]=='a'||stat[i]=='e'||stat[i]=='i'||stat[i]==
         'o'||stat[i]=='u')
         {
         printf("%c\n",stat[i]);
         cnt++;
         }
printf("\n\nThe number of vowels are : %d", cnt);
}
```

3. Reversing the array

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

main()
{
   int num[5]={34,45,56,67,89};
   int temp;
   int i,j,len;

   printf("\n\nOriginal Array :\n");
   for(i=0;i<=4;i++)
         printf("%d\n",num[i]);

   printf("\n\nArray in Reverse order\n");
   for(i=0,j=4;i<=2;i++,j--)
   {
      temp=num[i];
      num[i]=num[j];
      num[j]=temp;
   }

   for(i=0;i<=4;i++)
         printf("%d\n",num[i]);

}
```

7.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 8 - Pointers

Note: This TG maps to Session 13 of the book.

8.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

8.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain what a pointer is and where it is used
- Explain how to use pointer variables and pointer operators
- Assign values to pointers
- Explain pointer arithmetic
- Explain pointer comparisons
- Explain pointers and single dimensional arrays
- Explain Pointer and multidimensional arrays
- Explain how allocation of memory takes place

Difficulties

This chapter has few topics which the students may find difficult to grasp. They are listed here:

- Passing pointers as arguments to function
- Pointers to arrays
- Pointers to strings
- Allocation of memory using pointers

Hence, these topics should be handled carefully.

8.1.2 Teaching Skills

You should be well-versed in pointers and their related concepts for teaching this session. You should be familiar with creation and use of pointers, pointers as arguments, pointers to arrays, and other such concepts. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

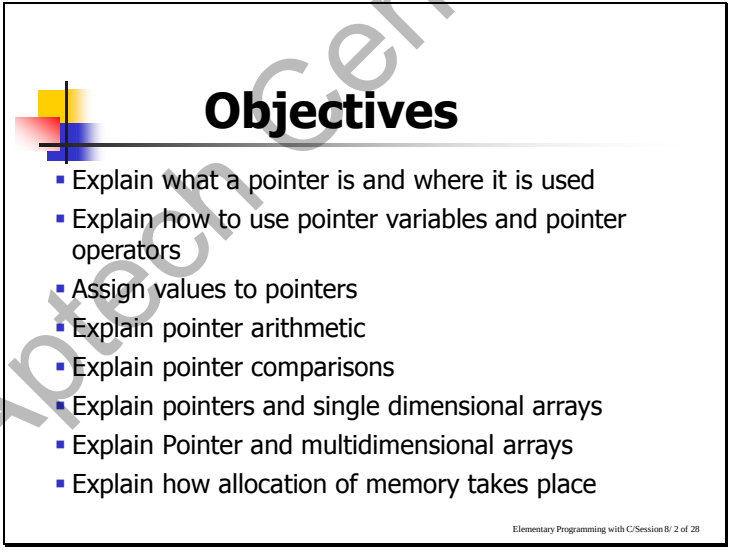
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the concept of pointers in C. The later part of this chapter deals with more advanced concepts such as how pointers are passed as arguments to functions and how pointers are used in relation to arrays. The students are also introduced to the concept of memory allocation and how pointers strive to make optimum usage of memory through dynamic allocation of memory.

8.2 In-Class Explanations

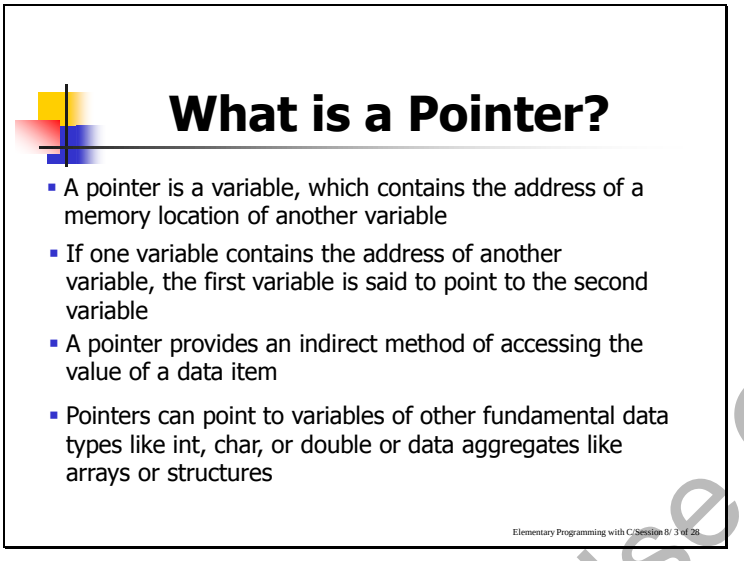
Slide 2



Max Time: 2 Minutes

Subject: List out the objectives as given on slide 2. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slide-3

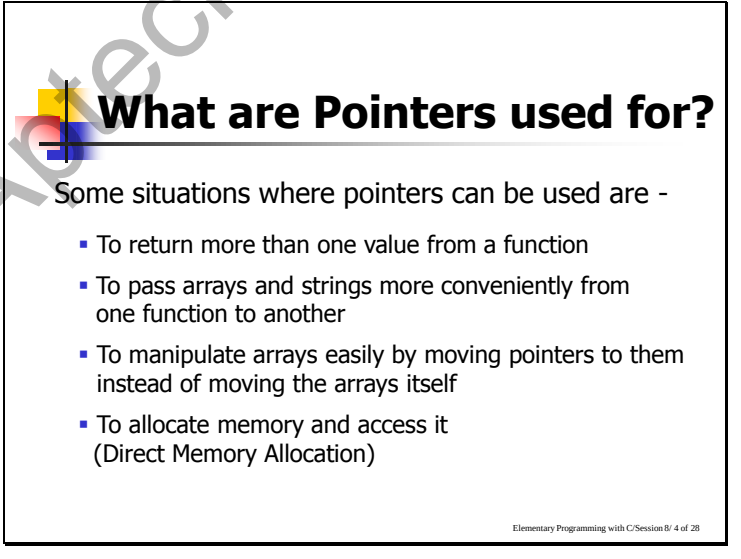


Max Time: 5 Minutes

Subject: Pointers are nothing but variables that contain addresses of other variables. Explain the second point on the slide with the help of a diagram. While reading out the third point, tell the students they will come to know the exact mechanism of accessing the variable indirectly as they proceed with the chapter.

Pointers are special variables therefore just like other variables they also should be of some fundamental data types. These pointers are used for pointing to fundamental data like int, char, float etc as well as composite data types like arrays and structures.

Slide-4



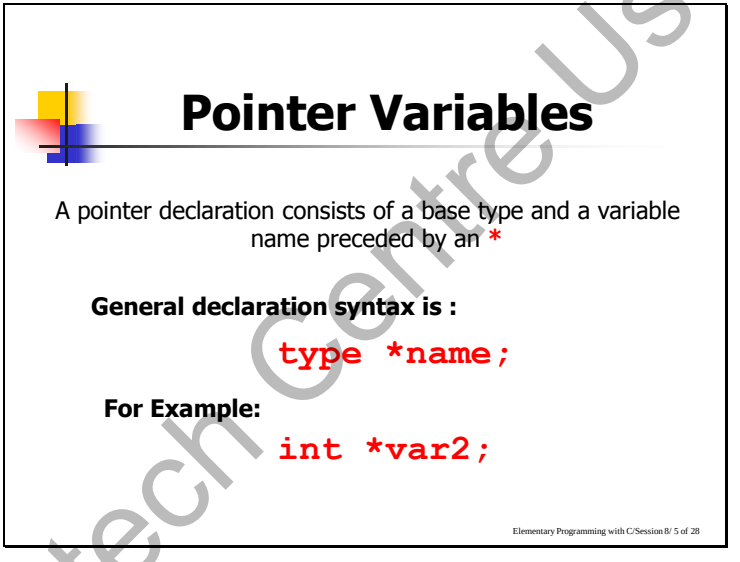
Max Time: 5 Minutes

Subject: Pointers can be used to pass aggregate data back to the calling function. Since aggregate data like structures and arrays are a collection of data, it means that more than one data can be sent back by the called function.

When arrays and strings are to be passed to functions normally each and every element of the data has to be passed individually. This means that the number of arguments passed to the function will depend on the number of elements in the aggregate data. However, when pointers are used the entire group of data can be passed to the function at a single point of time. Therefore, it is convenient to use pointers when passing arrays and structures to functions.

Since pointers contain addresses of other variables memory can be accessed directly with the use of pointers.

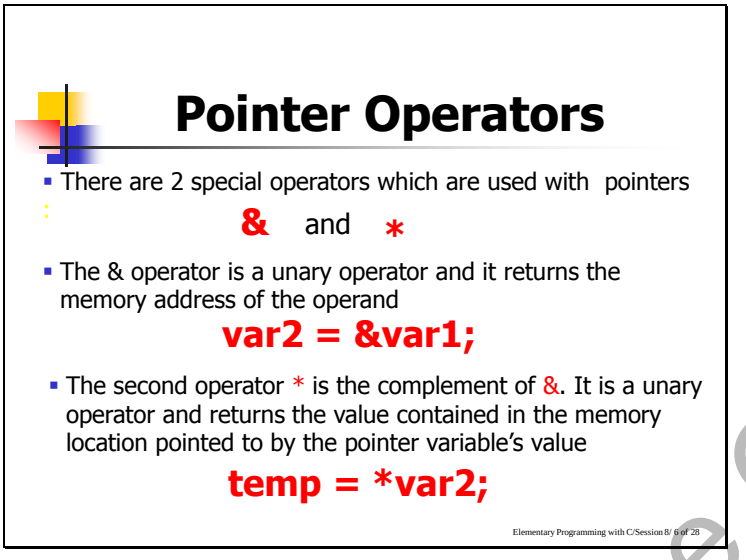
Slide-5



Max Time: 2 Minutes

Subject: Since at the beginning we defined pointers to be variables, they are to be declared. The syntax of declaration of variables states that a variable declaration should have a type. This has to be followed by the variable name preceded by * sign. The * sign differentiates the pointer from other variables.

Slide-6



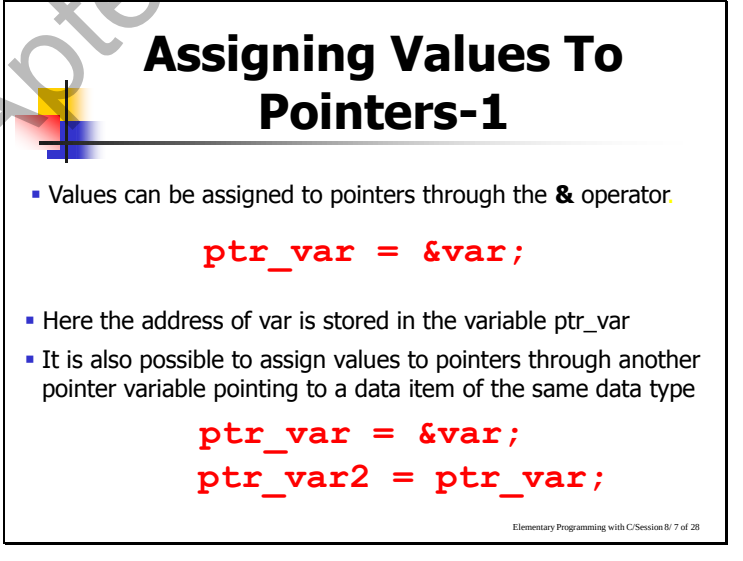
Max Time: 10 Minutes

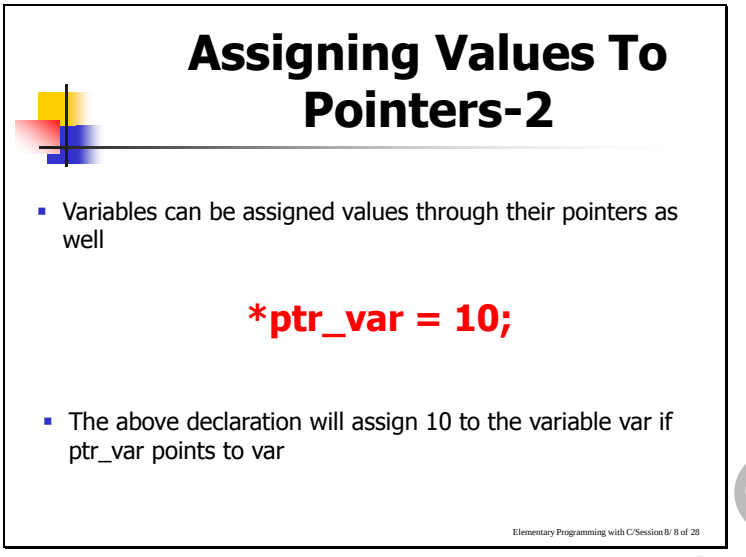
Subject: There are two pointer operators available to us. They are ***(star) and **&** (ampersand). Both are unary operators. At this point you can ask the student as to what is meant by unary operator. The probable answer is that the unary operator operates on a single operand.

The * operator is used for retrieving the value to which the pointer points. The '&' operator is used for returning the address of the variable. You can then point to the examples in the slide and ask the students what the variables on the left of the = sign will hold.

After this take up the program that is given in the student's guide and explain how the output is generated. Here, make sure that the students understand the number of bytes that are allocated for the different types of data.

Slides-7 and 8



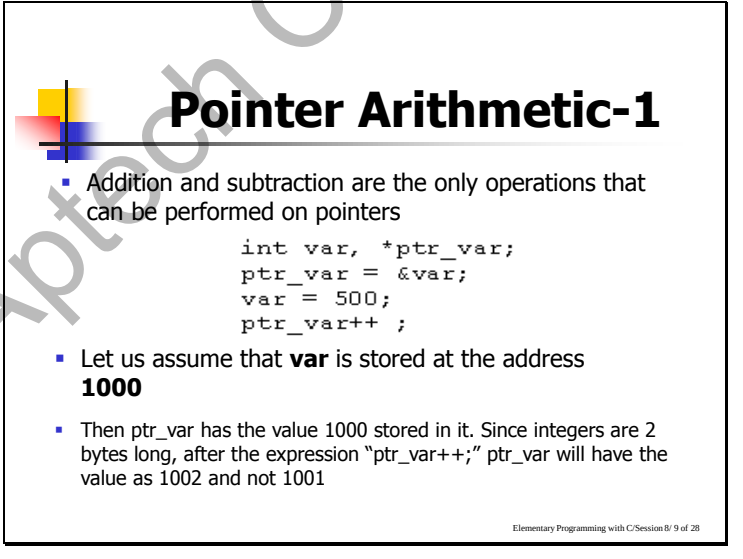


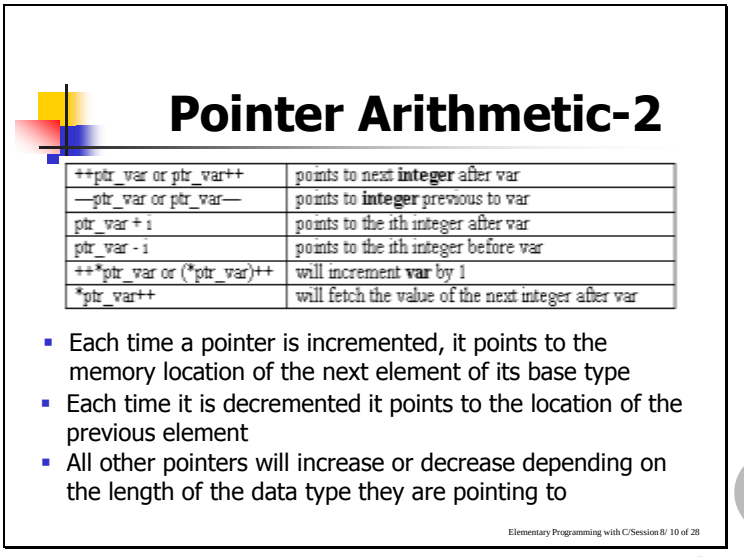
Max Time: 3 Minutes

Subject: The pointers can be initialized by using the & operator to the right hand side of the assignment operator. It is also possible to assign values to pointers through other pointer. At this point, take the example that is given at the end of Slide 8 and show the students how a pointer variable can be initialized.

There is yet another way to assign a value to a pointer. This can be done by assigning a value directly to the pointer.

Slides-9 and 10





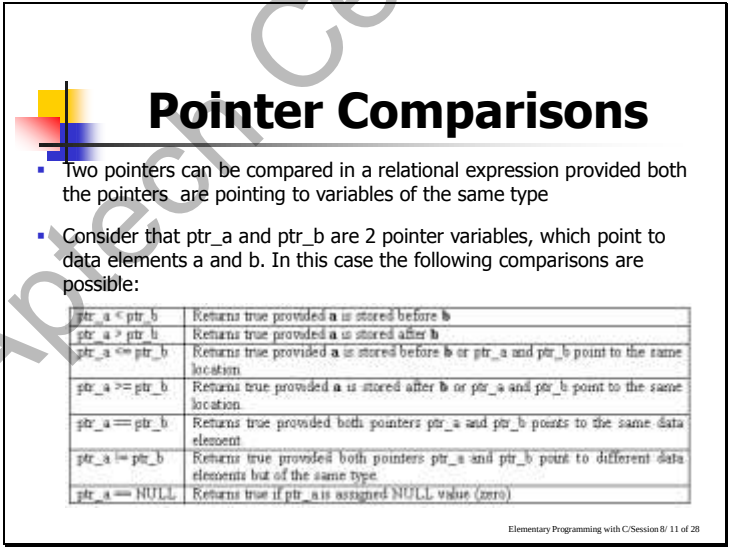
Max Time: 10 Minutes

Subject: Addition and subtraction are the only operations that can be performed on pointers.

Take the example of slide number 9 and explain it.

Take the table on slide 10 and explain each operation in detail. Highlight the fact that pointers will increase or decrease depending on the length of the data types they are pointing to.

Slide-11



Max Time: 5 Minutes

Subject: Pointers can be compared using relational operators. Pointers can be compared when they are pointing to variables of the same type. As a recap as the students to list all the relational operators they have learnt. Take the table in Slide 11 and then explain what each comparison returns.

Slides-12, 13, and 14

Pointers and Single Dimensional Arrays-1

- The address of an array element can be expressed in two ways :
  - By writing the actual array element preceded by the ampersand sign (&)
  - By writing an expression in which the subscript is added to the array name

Pointers and Single Dimensional Arrays-2

Example

```
#include<stdio.h>
void main()
{
       static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
       int i;
       for (i = 0; i < 10; i ++)
       {
           printf("\ni=%d,ary[i]=%d,*(ary+i)=%d",i,
            ary[i],*(ary + i));
           printf("&ary[i]= %X,ary+i=%X",&ary[i],ary+i);
           /* %X gives unsigned hexadecimal  */
       }
}
```


Pointers and Single Dimensional Arrays-3

Output

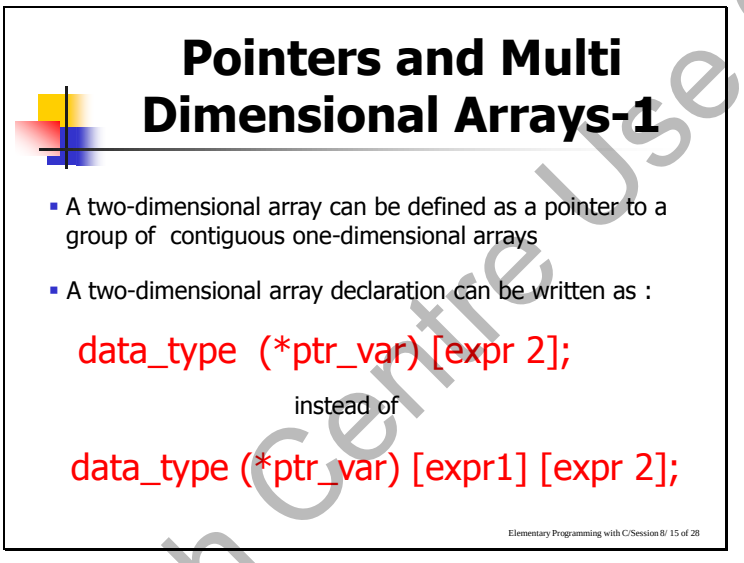
```
i=0   ary[i]=1   *(ary+i)=1   &ary[i]=194   ary+i = 194
i=1   ary[i]=2   *(ary+i)=2   &ary[i]=196   ary+i = 196
i=2   ary[i]=3   *(ary+i)=3   &ary[i]=198   ary+i = 198
i=3   ary[i]=4   *(ary+i)=4   &ary[i]=19A   ary+i = 19A
i=4   ary[i]=5   *(ary+i)=5   &ary[i]=19C   ary+i = 19C
i=5   ary[i]=6   *(ary+i)=6   &ary[i]=19E   ary+i = 19E
i=6   ary[i]=7   *(ary+i)=7   &ary[i]=1A0   ary+i = 1A0
i=7   ary[i]=8   *(ary+i)=8   &ary[i]=1A2   ary+i = 1A2
i=8   ary[i]=9   *(ary+i)=9   &ary[i]=1A4   ary+i = 1A4
i=9   ary[i]=10  *(ary+i)=10  &ary[i]=1A6   ary+i = 1A6
```

Max Time: 20 Minutes

Subject: When we are dealing with a single dimensional array, the array name is a pointer to the first element of the array. The address of the first element can either be the array can be referred as &<arrayname>[0] or simply as <arrayname> . Similarly, the second element can be referred as &<arrayname>[i] or simply as <arrayname>+1. Thus, each element of an array can be referred in two ways. At this point read out the subpoints given in slide 16.

After explaining the concept, take the example in slide 17 and explain it. Do a dry run and ask the students to compare the result that is shown in slide 18.

Slide-15



Max Time: 3 Minutes

Subject: Multi dimensional arrays are actually collections of contiguous single dimensional array. Similar to single dimensional array, multidimensional array can be represented in terms of a pointer and a subscript. Take an example of two dimensional array and draw the structure. Remember the rows in the 2-D array have to be drawn as continuous bytes. This will give a clear picture how a 2-D array is represented in the memory of the computer. Then show the students how and the array is declared in the program.

Slides-16 and 17

Pointers and Strings-1

Example

```
#include <stdio.h>
#include <string.h>
void main ()
{
        char a, str[81], *ptr;
        printf("\nEnter a sentence:");
        gets(str);
        printf("\nEnter character to search for:");
        a = getche();
        ptr = strchr(str,a);
        /* return pointer to char*/
        printf( "\nString starts at address: %u",str);
        printf("\nFirst occurrence of the character is at
        address: %u ",ptr);
        printf("\n Position of first occurrence(starting from
        0)is: % d", ptr_str);
}
```

Elementary Programming with C/Session 8/ 16 of 28

Pointers and Strings-2

Output

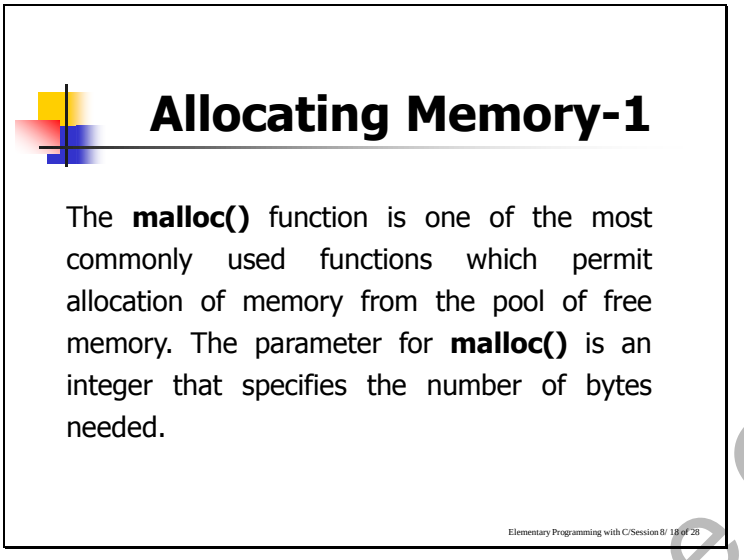
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17

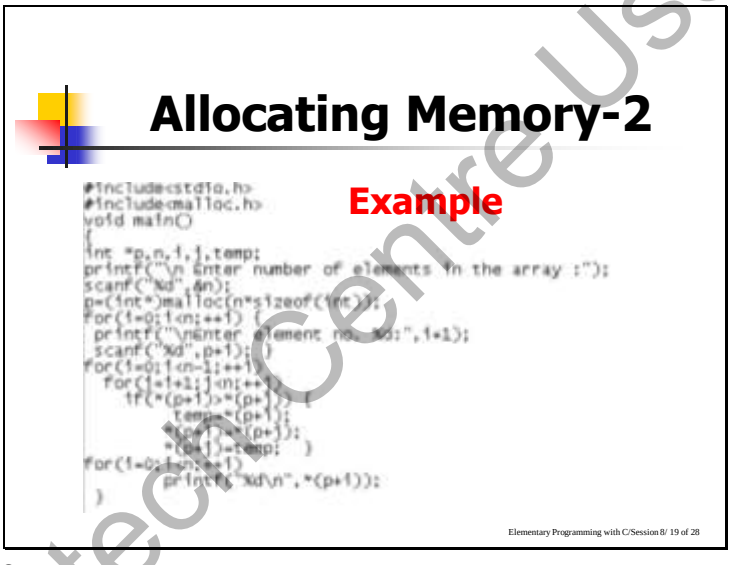
Elementary Programming with C/Session 8/ 17 of 28

Max Time: 10 Minutes

Subject: Take the example that is given in slide 20 and explain each step. Do a dry run and compare with the output that is given in slide 21.

Slides-18 and 19





Max Time: 10 Minutes

Subject: Whenever an array is declared where memory will be dynamic it has to use malloc() function. A pointer has to be declared and the pointer will be assigned the address of the array.

For example

```
int *ary;
```

This pointer ary will be assigned with the address of a dynamic array with the help of malloc() function.

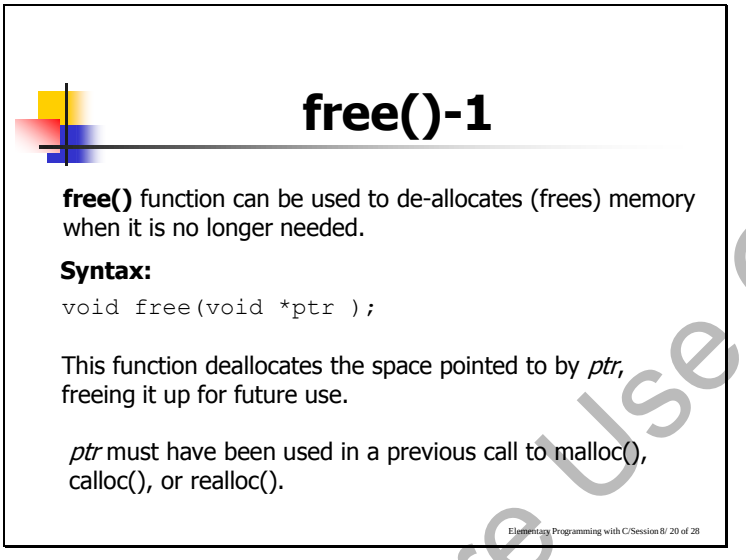
```
ary = (int *)malloc(20 * sizeof(int));
```

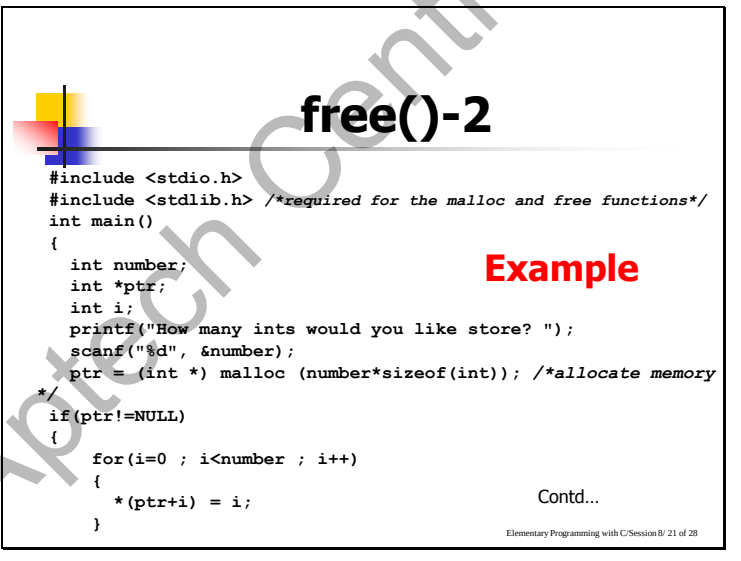
In the given example, the address of 20 contiguous integer bytes will be assigned to the pointer ary. The actual size of the array will depend upon the actual length of the data type. When memory has to be deallocated the free() function is used.

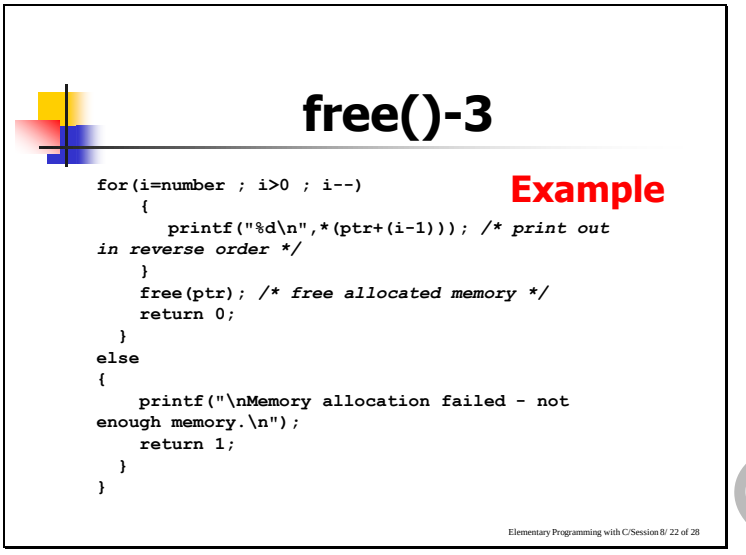
Both malloc() and free() are defined in the malloc.h header file.

After explaining the concept, take the example that is given in slide 23 and explain it.

Slides-20, 21, and 22



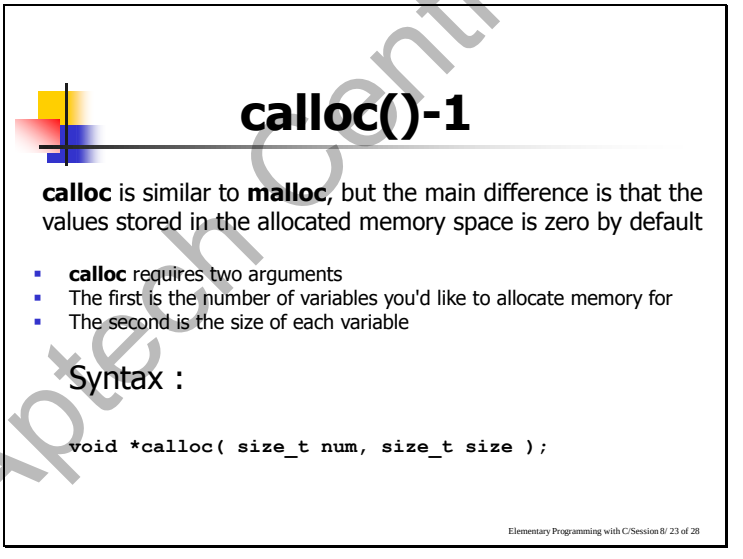




Max Time: 10 Minutes

Subject: You should explain that the free() function can be used to de-allocate (frees) memory when it is no longer needed. Explain the usage of this function with the given example.

Slides-23, 24, 25, 26, 27, and 28



calloc()-2

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  float *calloc1, *calloc2;
  int i;
  calloc1 = (float *) calloc(3, sizeof(float));
  calloc2 = (float *)calloc(3, sizeof(float));
  if(calloc1!=NULL && calloc2!=NULL)
  {
    for(i=0 ; i<3 ; i++)
    {
     printf("calloc1[%d] holds %05.5f ", i, calloc1[i]);
      printf("\ncalloc2[%d] holds %05.5f ", i,
           *(calloc2+i));
    }
```

Contd.....

Elementary Programming with C/Session 8/ 24 of 28

calloc()-3

Example

```
    free(calloc1);
    free(calloc2);
    return 0;
  }
  else
  {
      printf("Not enough memory\n");
      return 1;
    }
  }
```

Elementary Programming with C/Session 8/ 25 of 28

realloc()-1

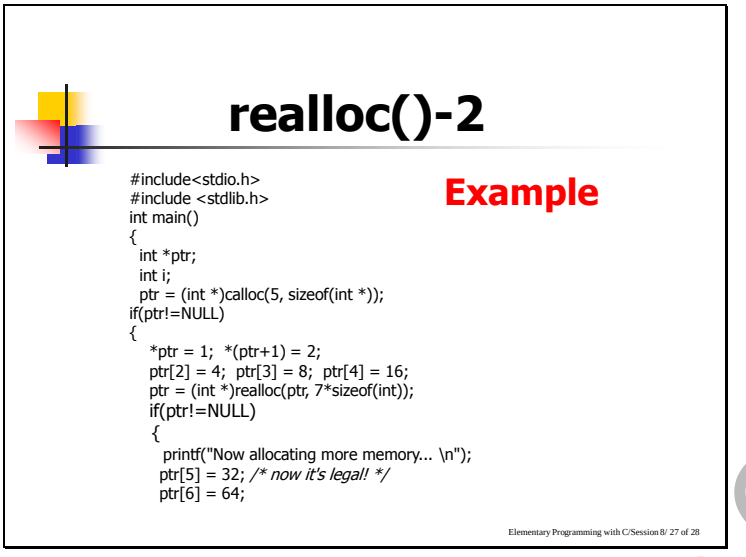
You've allocated a certain number of bytes for an array but later find that you want to add values to it.You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using **realloc**, without losing your data.

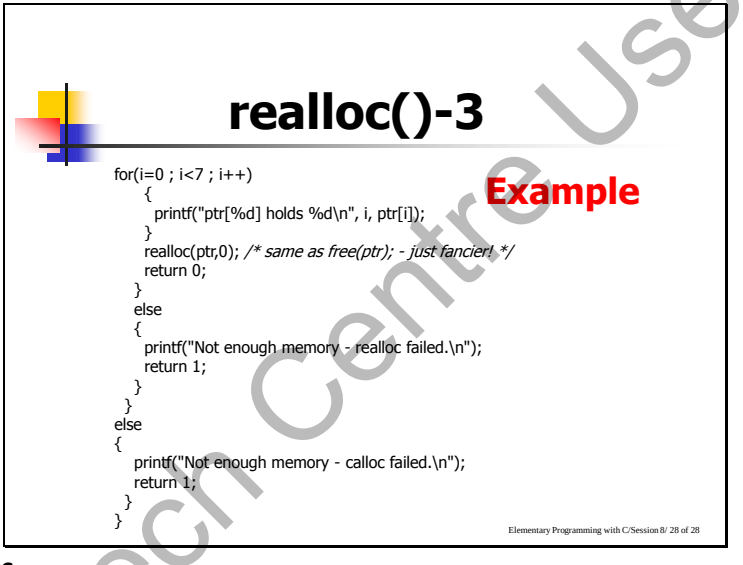
- **realloc** takes two arguments
- The first is the pointer referencing the memory
- The second is the total number of bytes you want to reallocate

Syntax:

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

Elementary Programming with C/Session 8/ 26 of 28





Max Time: 25 Minutes

Subject: You can start the session with the explanation of malloc function. Then explain that calloc and realloc can also be used to allocate memory dynamically. You have to tell the difference between malloc and calloc functions. The examples given on the slides should be handled carefully and should be noted that all students are clear with it.

Solutions to Check Your Progress

1. pointer
2. False
3. type
4. * and &
5. Addition , Subtraction
6. False
7. Dynamic Memory Allocation

Solutions to Do It Yourself

1.

```
#include<stdio.h>
#include<string.h>
main()
{
   char pal[30];
   char reverse[30];
   int i,j,len,k;
   char *palin;
   char a,b;
   char palind;

   printf("Enter a string : ");
   scanf("%s",pal);
   printf("\nThe string is : %s",pal);
   len=strlen(pal);
   palin=pal;
   printf("\nThe length string is : %d",len);
   for(i=0;i<len;i++,palin++);

   printf("\nThe reverse string is : ");
   for(j=len;j>=0;j--,palin--)
          printf("%c",*palin);

   printf("\n\n");

   for(i=0;i<len;i++,palin++);
   for(i=0;i<len;i++,palin--)
      if (pal[i]==*palin)
           palind='T';
      else
           palind='F';

   if(palind=='T')
          printf("\n\nThe string '%s' is a palindrome",pal);
   else
          printf("\n\nThe string '%s' is not a
        palindrome",pal);
}
```

2.

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

main()
{

   char animal[30],bird[30];
   char ani_plural[30],bird_plural[30];
   char *an, *bi;

   an=animal;
   bi=bird;
```

```
    printf("Enter the name of the animal : ");
    scanf("%s",an);
    printf("Enter the name of the bird : ");
    scanf("%s",bi);

    strcpy(ani_plural,strcat(an,"s"));
    printf("\nPlural of animal is %s",ani_plural);
    strcpy(bird_plural,strcat(bi,"s"));
    printf("\nPlural of bird is %s",bird_plural);
}
```

Session 9 -Functions

Note: This TG maps to Session 15 of the book.

9.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

9.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain the use of functions
- Explain the structure of a function
- Explain function declaration and function prototypes
- Explain the different types of variables
- Explain how to call functions
- Call by Value
- Call by Reference
- Explain the scope rules for a function
- Explain functions in multifile programs
- Explain Storage classes
- Explain function pointers

Difficulties

The students may find difficulty in understanding the following topics:

- Differentiate call by value and call by reference
- Function Pointers

Hence, the mentioned topic should be handled carefully.

9.1.2 Teaching Skills

You should be well-versed in programming concepts, especially modular programming such as functions, for teaching this chapter. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

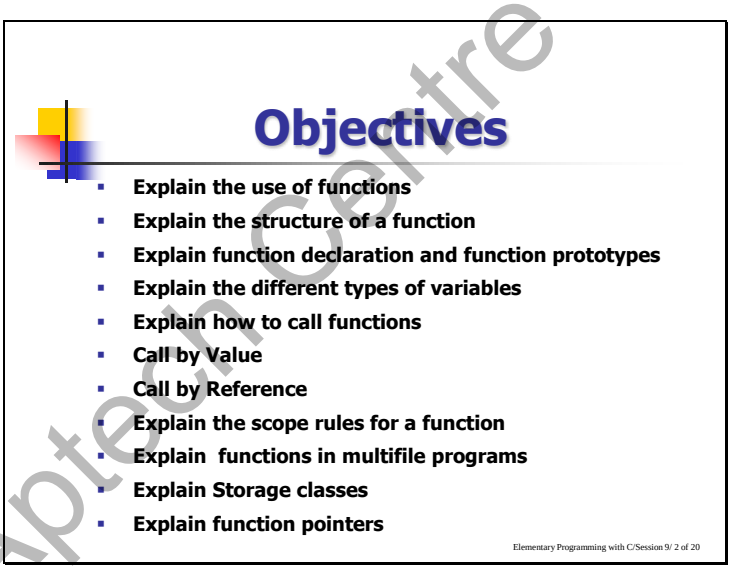
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the concept of functions in C. At the end of it, students are able to write modular programs using functions.

9.2 In-Class Explanations

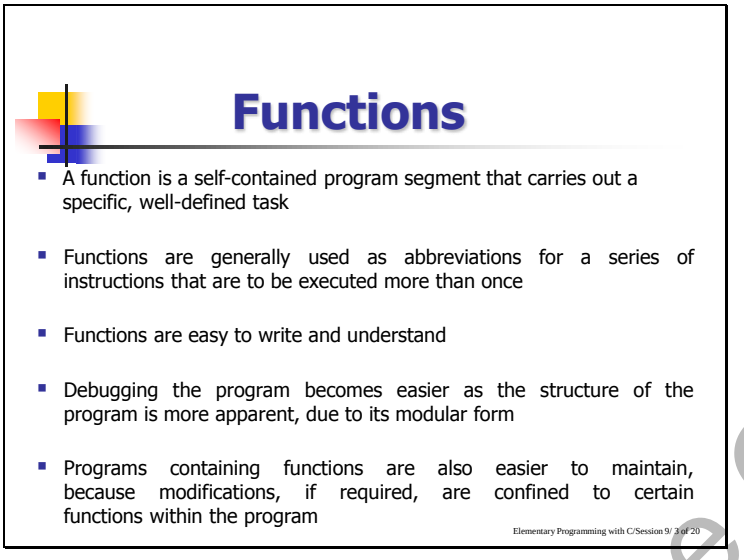
Slide 2



Max Time: 2 Minutes

Subject: List out the objectives as given on slide 2 and inform students what they are going to learn in next 2 hours. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slide-3



Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students for the execution of repeated steps. You should define a function and explain how it is different from loops. You should clarify the need for modular programming.

The following material could be used to explain what modular programming is.

Additional Information:

The role of modular programming is to allow the programmer to extend the built in capabilities of the language. It packages up bits of program into modules that we can 'plug in' to our programs. The first form of module was the subroutine which was a block of code that you could jump to (rather like the GOTO mentioned in the branching section) but when the block completed, it could jump back to wherever it was called from. That specific style of modularity is known as a procedure or function.

The following additional material could be used to teach the students.

Why Functions?

There are several questions that you should always ask whenever anyone gives you advice about writing programs:

1. Writing small programs is easy. Does the advice make writing large programs easier?
2. Writing a program by yourself is much easier than writing one with others. Does the advice make cooperative programming easier?
3. Writing a program is much easier than understanding or modifying a program written by others (or even by yourself). Does the advice make it easier to understand and modify existing programs?

Dividing programs into functions satisfies all of these questions. Let's consider each question in turn.

How does using functions make writing large programs easier?

Imagine a program with 1000 lines. (This may seem large to you, but it is actually quite small.) It is much easier to write a properly designed program consisting of 20 50-line functions than it is to write a program consisting of one 1000-line main program. Functions can be written and understood independently of one another. It is much easier to get your mind around a 50-line thing than a 1000-line thing.

Another benefit is that code can be reused between projects. If someone has previously written a function that you would like to use, it is a simple matter to incorporate it into your program.

A third benefit is that the use of functions can eliminate the duplication of code. If the duplicated code is long enough, the use of functions can actually shorten a program.

How does using functions make cooperative programming easier?

A multi-person programming project proceeds by first deciding what functions need to be implemented and then implementing the functions. Once the functions are designed, they can be divided up among the available programmers and implemented separately. Compare that with having multiple programmers work on a single main programmer.

How does using functions make modifying programs easier?

Take a look at the example program given here. In its original form, if we want to change the way that base two logarithms are computed, we have to search through the program and change it in three places. In the improved form, we need only change the LOG2 function. It is much easier figuring out what needs to change and it is much easier to actually make the change.

Program to calculate base 2 logarithm of numbers - Original

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


void main ()
{

  float x;

  printf("Please enter a float: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, log10(x)/log10(2));

  printf("Please enter a real number: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, log10(x)/log10(2));

  printf("Please enter a real number: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, log10(x)/log10(2));

}
```

Program to calculate base 2 logarithm of numbers – with Function

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


/* This returns the log base two of x. */

float LOG2 (float x)
{
  return(log10(x)/log10(2.0));
}

void main ()
{

  float x;

  printf("Please enter a float: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, LOG2(x));

  printf("Please enter a real number: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, LOG2(x));

  printf("Please enter a real number: ");
  scanf("%f", &x);
  printf("The base two logarithm of %f is %f\n", x, LOG2(x));

}
```

Slides-4, 5, 6, and 7

The Function Structure

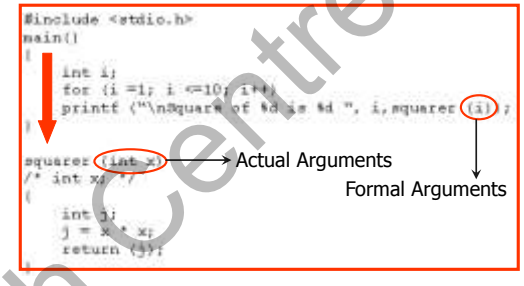
- The general syntax of a function in C is :

```
type_specifier function_name (arguments)
{
        body of the function
}
```

- The type_specifier specifies the data type of the value, which the function will return.
- A valid function name is to be assigned to identify the function
- Arguments appearing in parentheses are also termed as formal parameters.

Elementary Programming with C/Session 9/ 4 of 20

Arguments of a function



- The program calculates the square of numbers from 1 to 10
- The data is passed from the main() to the squarer() function
- The function works on data using arguments

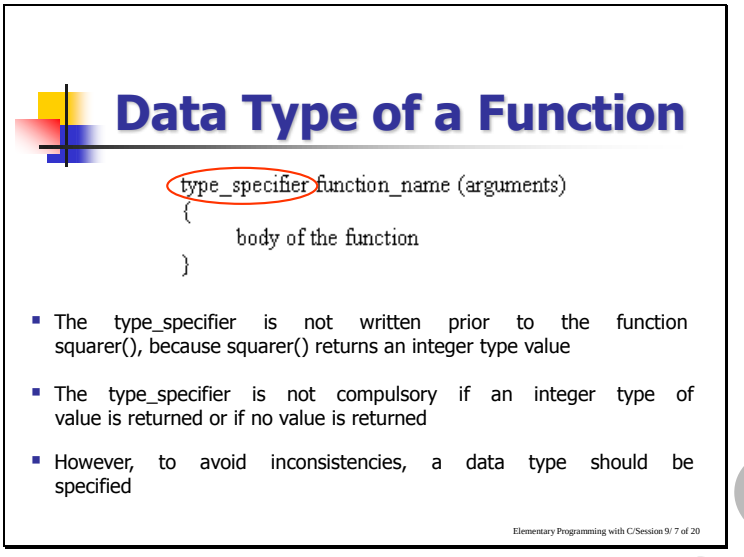
Elementary Programming with C/Session 9/ 5 of 20

Returning from the function

```
squarer (int x)
/* int x; */
{
    int j;
    j = x * x;
    return (j);
}
```

- It transfers the control from the function back to the calling program immediately.
- Whatever is inside the parentheses following the return statement is returned as a value to the calling program.

Elementary Programming with C/Session 9/ 6 of 20



Max Time: 15 Minutes

Subject: The teaching of this session should start with the answers given by the students for the functions. The generic syntax of a function as given in the book should be explained. Emphasis should be given on explaining the type_specifier and arguments of a function. The concepts of arguments of a function could be explained with the example given in the book. The difference between formal and actual arguments needs to be explained. Finally, how to return a value from a function needs to be explained.

Additional Information:

Source: http://www.strath.ac.uk/IT/Docs/Ccourse/section3_9.html

The following additional examples could be used.

Examples: C assumes that every function will return a value. If the programmer wants a return value, this is achieved using the return statement. If no return value is required, none should be used when calling the function.

Here, is a function which raises a double to the power of an unsigned and returns the result.

```
double power(double val, unsigned pow)
{       double ret_val = 1.0;
        unsigned i;

        for(i = 0; i < pow; i++)
                ret_val *= val;

        return(ret_val);
}
```

The function follows a simple algorithm, multiplying the value by itself pow times. A for loop is used to control the number of multiplications and variable ret_val stores the value to be returned.

Careful programming has ensured that the boundary condition is correct too. ie Let us examine the details of this function.

```
double power(double val, unsigned pow)
```

This line begins the function definition. It tells us the type of the return value, the name of the function, and a list of arguments used by the function. The arguments and their types are enclosed in brackets, each pair separated by commas.
The body of the function is bounded by a set of curly brackets. Any variables declared here will be treated as local unless specifically declared as static or extern types.

```
return(ret_val);
```

On reaching a return statement, control of the program returns to the calling function. The bracketed value is the value which is returned from the function. If the final closing curly bracket is reached before any return value, then the function will return automatically, any return value will then be meaningless.

The example function can be called by a line in another function which looks like this,

```
result = power(val, pow);
```

This calls the function power assigning the return value to variable result.
Here is an example of a function which does not return a value.

```
void error_line(int line)
{       fprintf(stderr, "Error in input data: line %d\n", line);
}
```

The definition uses type void which is optional. It shows that no return value is used. Otherwise the function is much the same as the previous example, except that there is no return statement. Some void type functions might use return, but only to force an early exit from the function and not to return any value. This is rather like using break to jump out of a loop.
This function also demonstrates a new feature.

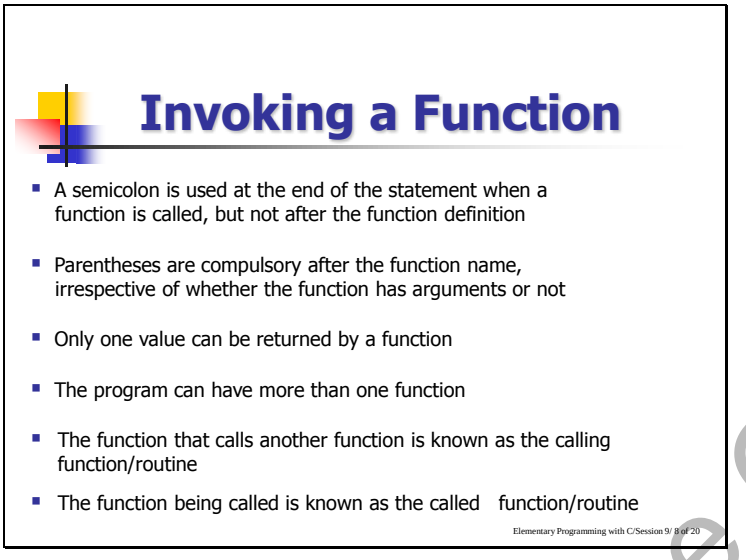
```
fprintf(stderr, "Error in input data: line %d\n", line);
```

This is a variant on the printf statement, fprintf sends its output into a file. In this case, the file is stderr. stderr is a special UNIX file which serves as the channel for error messages. It is usually connected to the console of the computer system, so this is a good way to display error messages from your programs. Messages sent to stderr will appear on screen even if the normal output of the program has been redirected to a file or a printer.

The function would be called as follows:

```
error_line(line_number);
```

Slide-8

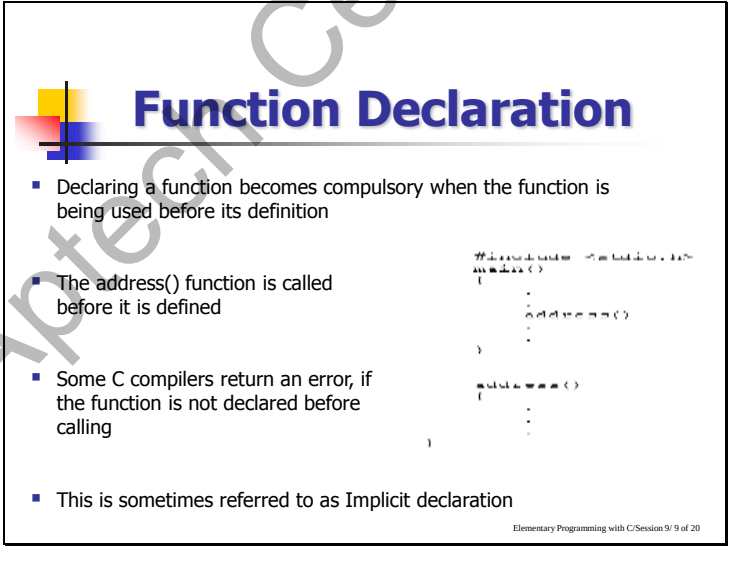


Max Time: 3 Minutes

Subject: This section explains the basic difference between the definition of function and calling a function. The points to be remembered about functions as given in the book need to be emphasized.

The difference between the calling and the called function needs to be emphasized.

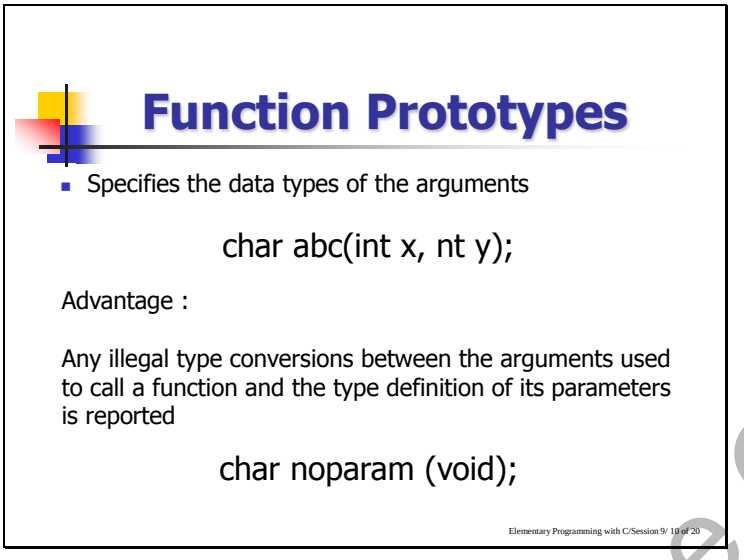
Slide-9



Max Time: 3 Minutes

Subject: Functions are external to the main program. That is the main() itself is a function and hence functions cannot be defined within main(). As a result of this the function need to be declared inside main().

Slide-10



Max Time: 5 Minutes

Subject: Declaring a function informs the C compiler that a function of this name is going to be used in this program. C compilers if informed more about the function are capable of giving warnings during wrong usage. That is, C compilers match the function declaration with the function definition. The number of arguments and the data types of the arguments are compared to throw errors or warnings. To help the C compiler to more informative the declaration of a function needs to be precise and clear. Such a declaration is termed as the function prototype.

Additional Information:

The following additional material can be used to teach the concept.

Source: http://gd.tuwien.ac.at/languages/c/programming-bbrown/c_050.htm
These have been introduced into the C language as a means of provided type checking and parameter checking for function calls. Because C programs are generally split up over a number of different source files which are independently compiled, then linked together to generate a run-time program, it is possible for errors to occur.
Consider the following example.

```
/* source file add.c */
void add_up( int numbers[20] )
{
        ....
}

/* source file mainline.c */
static float values[] = { 10.2, 32.1, 0.006, 31.08 };

main()
{
        float result;
        ...
```

```
            result = add_up( values );
        }
```

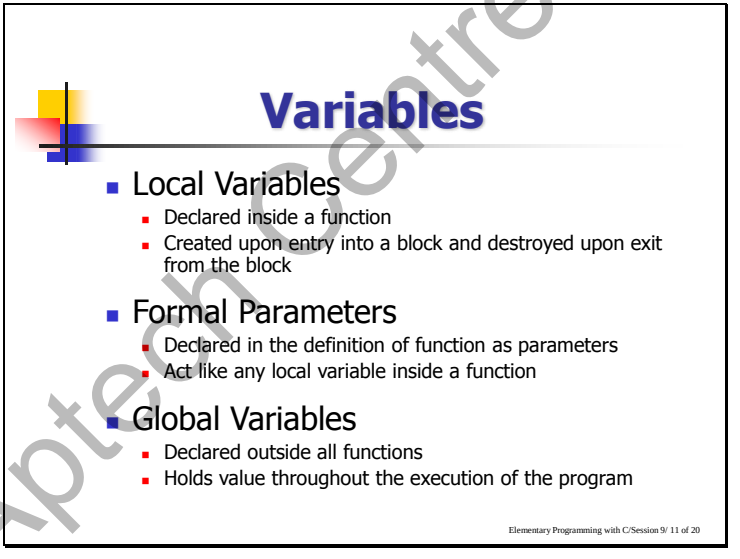
As the two source files are compiled separately, the compiler generates correct code based upon what the programmer has written. When compiling mainline.c, the compiler assumes that the function add_up accepts an array of float variables and returns a float. When the two portions are combined and ran as a unit, the program will definitely not work as intended.

To provide a means of combating these conflicts, ANSI C has function prototyping. Just as data types need to be declared, functions are declared also. The function prototype is,

```
/* source file mainline.c */
void add_up( int numbers[20] );
```

NOTE that the function prototype ends with a semi-colon; in this way we can tell its a declaration of a function type, not the function code. If mainline.c was re-compiled, errors would be generated by the call in the main section which references *add_up()*.
Generally, when developing a large program, a separate file would be used to contain all the function prototypes. This file can then be included by the compiler to enforce type and parameter checking.

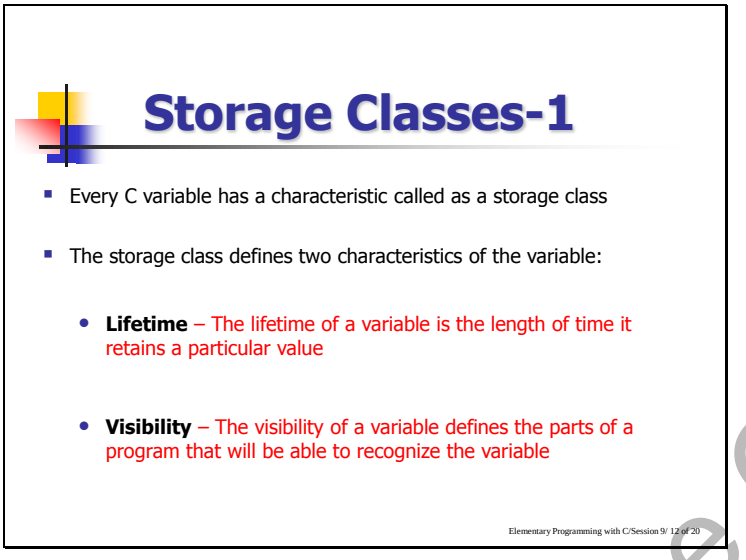
Slide-11

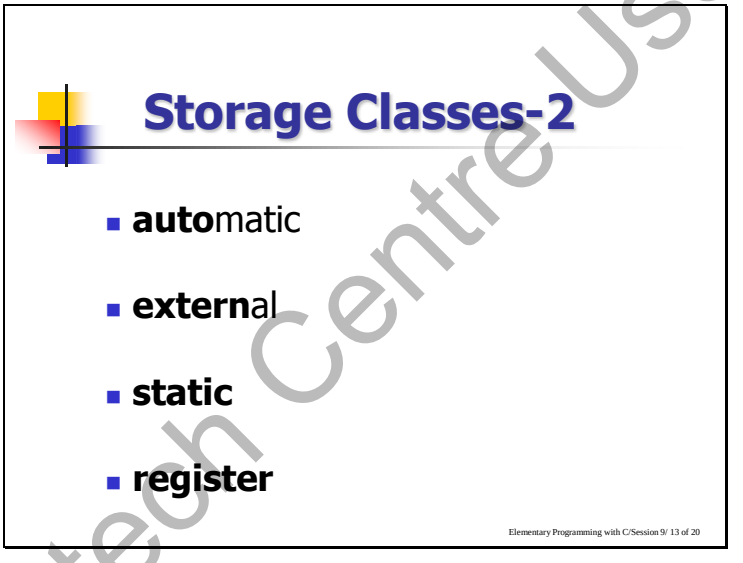


Max Time: 3 Minutes

Subject: As the students will be already aware of variables, you can move to explain the types of variables after reviewing the concept of variables. The differences between local and global variables need to be emphasized. Also the fact that the formal parameters are like local variables inside a function needs to be explained.

Slides-12 and 13





Max Time: 15 Minutes

Subject: The variables are stored in the RAM of the computer. In order to increase the speed of the execution of a program, C allows the storage of some variables in registers. However, we can just hope that the variable will be stored in register. The actual storing and processing depends upon the compatibility and availability of registers. This fact needs to be explained to the students. Explain that the storage class specified also defines the scope of a variable. Point out to the students that the use of extern will be well understood while doing multi-file programs later in this chapter.

The following material can be used to explain the concept of storage classes in addition to the explanation given in the book.

Additional Information:

auto - storage class

auto is the default storage class for local variables.

```
{
    int Count;
    auto int Month;
}
```

The example defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and cant have the unary '&' operator applied to it (as it does not have a memory location).

```
{
  register int  Miles;
}
```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

static - Storage Class

static is the default storage class for global variables. The two variables (**count** and **road**) both have a static storage class.

```
static int Count;
 int Road;

main()
{
  printf("%d\n", Count);
  printf("%d\n", Road);
}
```

'static' can also be defined within a function. If this is done, the variable is initialized at compilation time and retains its value between calls. Since it is initialized at compilation time, the initialization value must be a constant.

```
void Func(void)
{
  static Count=1;
}
```

There is one very important use for 'static'. Consider this bit of code.

```
char *Func(void);

main()
{
  char *Text1;
  Text1 = Func();
}

char *Func(void)
{
  char Text2[10]="martin";
  return(Text2);
}
```

'Func' returns a pointer to the memory location where 'Text2' starts. BUT Text2 has a storage class of auto and will disappear when we exit the function and could be overwritten by something else. The answer is to specify:

```
static char Text2[10]="martin";
```

The storage assigned to 'Text2' will remain reserved for the duration of the program.

extern - storage Class

extern defines a global variable that is visible to ALL object modules. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

Source 1

```
extern int count;
write()
{
printf("count is %d\n", count);
}
```

Source 2

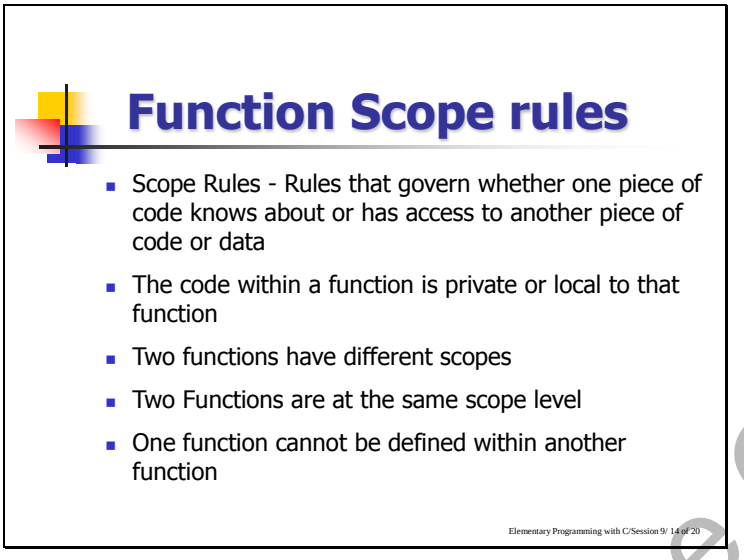
```
int count=5;
main()
{
 write();
}
```

Count in 'source 1' will have a value of 5. If source 1 changes the value of count - source 2 will see the new value. Here are some example source files.

The compile command will look something like.

```
gcc source1.c source2.c -o program
```

Slide-14



Max Time: 5 Minutes

Subject: It is very important to understand the scope of the variables used in a function. It is very essential to determine which variables will be accessible to the function and which function variables will be accessible to the other parts of the programs.

The following additional material could be used by you.

Additional Information:

Source: http://www.strath.ac.uk/IT/Docs/Ccourse/subsection3_9_1.html

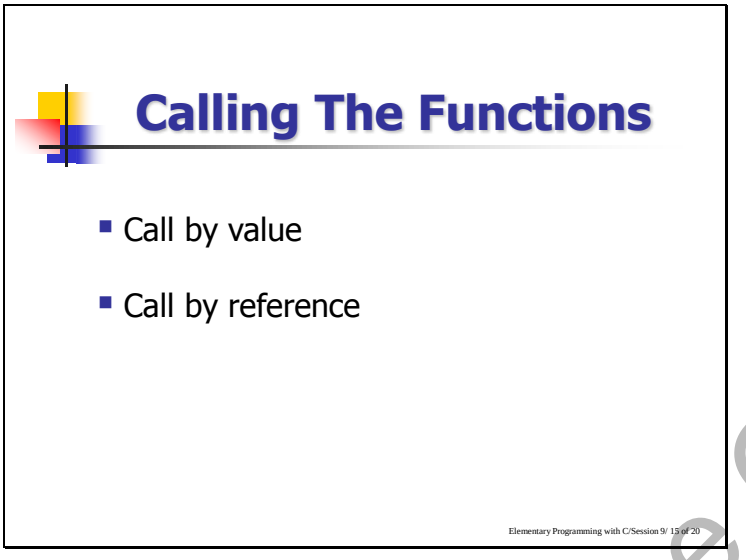
Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

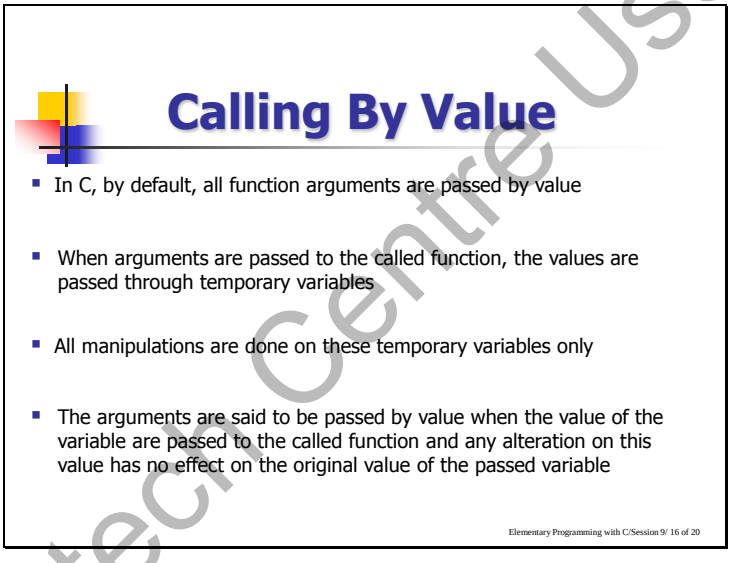
Local variables are declared within a function. They are created anew each time the function is called and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

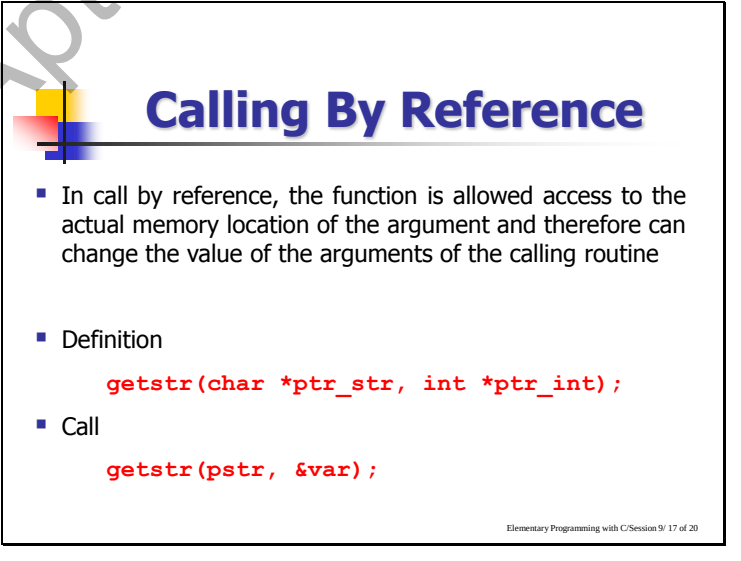
Static variables are slightly different, they don't die on return from the function. Instead their last value is retained and it becomes available when the function is called again.

Global variables don't die on return from a function. Their value is retained, and is available to any other function which accesses them.

Slides-15, 16, and 17







Max Time: 20 Minutes

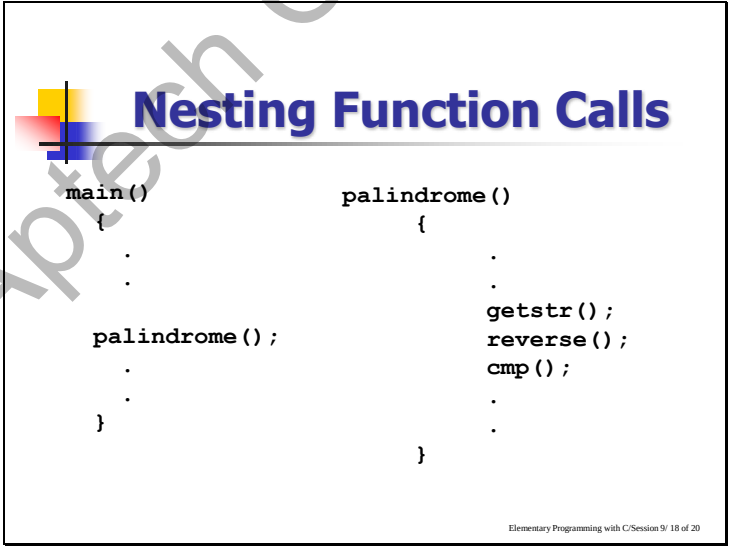
Subject: While passing values from a calling function to the called function the called function can access only the copy of the data. The actual data is retained as it is. However, there are instances when we would require the actual data to be accessed by the called function. This can be done when the data reference is passed to the function.

To do this the address of the data has to be passed to the called function. Once the called function is aware of the address it can pick up the actual value and modify it.

Examples: The following additional example can be used to explain call by value.

```
#include <stdio.h>

void changeit (int x)
{
  x = x*x;
}

void main ()
{
  int n;
  printf("Enter a number: ");
  scanf("%d", &n);
  changeit(n);
  printf("The value of n is now %d\n", n);
}
```

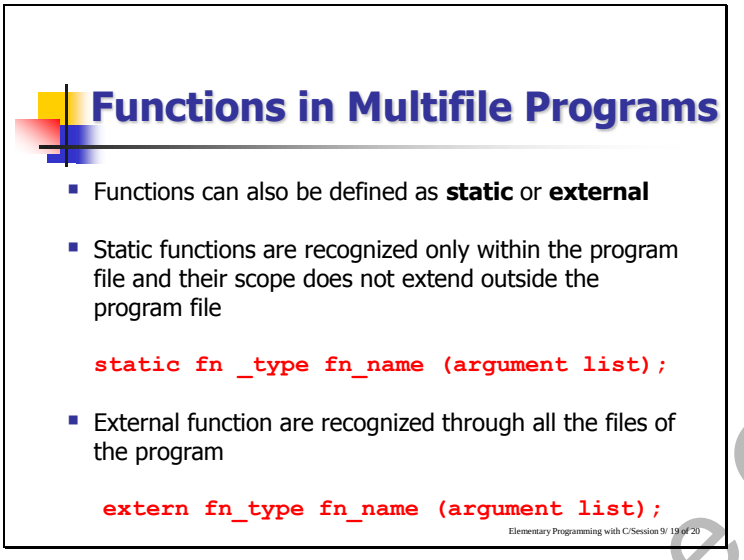
Slide-18



Max Time: 5 Minutes

Subject: In C, a function cannot be defined inside another function. However, a function can be called from another function. This function in turn can call another function. Thus, the function calls are said to be nested.

Slide-19



Max Time: 15 Minutes

Subject: The following additional material could be used by you to explain the concept of multifile programs.

Additional Information:

Source: http://www.strath.ac.uk/IT/Docs/Ccourse/section3_14.html

The main advantages of spreading a program across several files are:

- Teams of programmers can work on programs. Each programmer works on a different file.
- An object oriented style can be used. Each file defines a particular type of object as a datatype and operations on that object as functions. The implementation of the object can be kept private from the rest of the program. This makes for well structured programs which are easy to maintain.
- Files can contain all functions from a related group. For Example all matrix operations. These can then be accessed like a function library.
- Well implemented objects or function definitions can be re-used in other programs, reducing development time.
- In very large programs each major function can occupy a file to itself. Any lower level functions used to implement them can be kept in the same file. Then programmers who call the major function need not be distracted by all the lower level work.
- When changes are made to a file, only that file need be re-compiled to rebuild the program. The UNIX make facility is very useful for rebuilding multifile programs in this way.

How to Divide a Program between Several Files

Where a function is spread over several files, each file will contain one or more functions. One file will include main while the others will contain functions which are called by others. These other files can be treated as a library of functions.

Programmers usually start designing a program by dividing the problem into easily managed sections. Each of these sections might be implemented as one or more functions. All functions from each section will usually live in a single file.

Where objects are implemented as data structures, it is usual to keep all functions which access that object in the same file. The advantages of this are:

- The object can easily be re-used in other programs.
- All related functions are stored together.
- Later changes to the object require only one file to be modified.

Where the file contains the definition of an object or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly.

The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file.

Whenever a function in another file calls a function from our C file, it can define the function by making a #include of the appropriate .h file.

Organization of Data in each File

Any file must have its data organized in a certain order. This will typically be:

1. A preamble consisting of #defined constants, #included header files and typedefs of important datatypes.
2. Declaration of global and external variables. Global variables may also be initialized here.
3. One or more functions.

The order of items is important, since every object must be defined before it can be used. Functions which return values must be defined before they are called. This definition might be one of the following:

- Where the function is defined and called in the same file, a full declaration of the function can be placed ahead of any call to the function.
- If the function is called from a file where it is not defined, a prototype should appear before the call to the function.

A function defined as,

```
float find_max(float a, float b, float c)
 {  /* etc ... ... */
```

would have a prototype of,

```
float find_max(float a, float b, float c);
```

The prototype may occur among the global variables at the start of the source file. Alternatively, it may be declared in a header file which is read in using a #include.

It is important to remember that all C objects should be declared before use.

Compiling Multi-File Programs

This process is rather more involved than compiling a single file program. Imagine a program in three files prog.c, containing main(), func1.c and func2.c. The simplest method of compilation (to produce a runnable file called a.out) would be

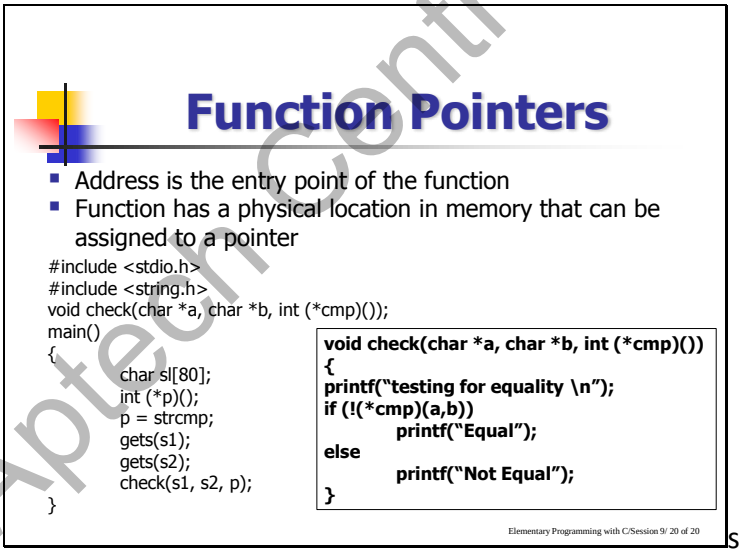
```
cc prog.c func1.c func2.c
```

If we wanted to call the runnable file prog we would have to type,

```
cc prog.c func1.c func2.c -o prog
```

In these examples, each of the .c files is compiled and then they are automatically linked together using a program called the loader ld.

Slide-20



Max Time: 15 Minutes

Subject: This section introduces the concept of function pointers in C. The teaching of this section should start with explaining that functions are also represented in memory like variables and hence pointers could be used to reference functions instead of function name. The same should be explained with the example given in the book.

The following additional examples could be used by you.

Additional Information:

Source: http://oopweb.com/CPP/Documents/FunctionPointers/Volume/CCPP/FPT/em_fpt.html

Examples: Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus, a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.

Define a Function Pointer: Since a function pointer is nothing else than a variable, it must be defined as usual. In the following example, we define two function pointers named *pt2Function* and *pt2Member*. They point to functions, which take one *float* and two *char* and return an *int*.

```
int (*pt2Function)        (float, char, char);
```

Assign an address to a Function Pointer: It's quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. It's optional to use the address operator & infront of the function's name.
Note: You may have got to use the complete name of the member function including class-name and scope-operator (::). Also you have got to ensure, that you are allowed to access the function right in scope where your assignment stands.

```
int DoIt  (float a, char b, char c){ printf("DoIt");
                                     return a+b+c; }

int DoMore(float a,char b,char c){printf("DoMore");
                                  return a-b+c; }

pt2Function = DoMore;     // assignment
pt2Function = &DoIt;      // alternative using address operator
```

Calling a Function using a Function Pointer: In C you have two alternatives of how to call a function using a function pointer: You can just use the name of the function pointer instead of the name of the function or you can explicitly dereference it.

```
int result1 = pt2Function    (12, 'a', 'b');   // C short way
int result2 = (*pt2Function) (12, 'a', 'b');  // C
```

Solutions to Check Your Progress

1. Function
2. Formal Parameters
3. calling routine or calling function
4. Calling Function, Called Function
5. Function Prototype
6. Local Variables
7. Global Variables

8. Scope rules
9. by value
10. Call by reference

9.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions

Session 10 – Strings

Note: This TG maps to Session 17 of the book.

10.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

10.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain string variables and constants
- Explain pointers to strings
- Perform string input/output operations
- Explain the various string functions
- Explain how arrays can be passed as arguments to functions
- Describe how strings can be used as function arguments

Difficulties

The students may find difficulty in understanding the following topic:

- Passing Arrays to Functions
- Passing Strings to Functions

Hence, the mentioned topic should be handled carefully.

10.1.2 Teaching Skills

You should be aware of and well-versed with various string functions in C. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

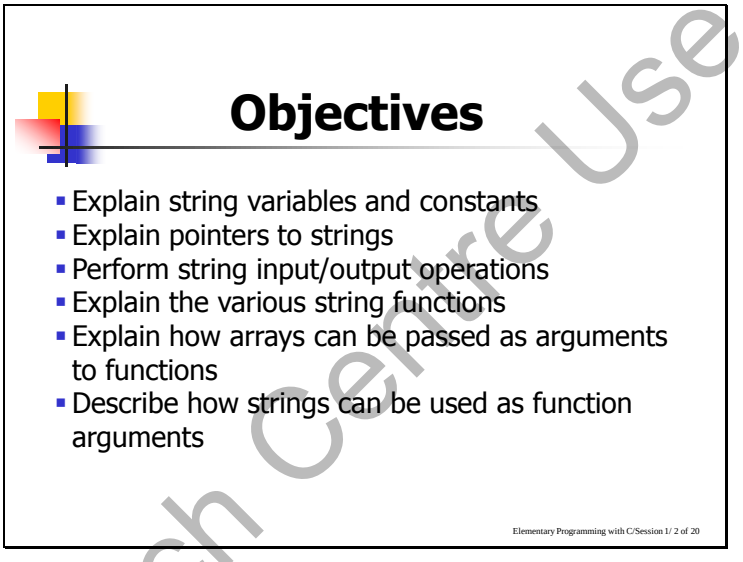
Follow the order as given here for the In-Class activities.

Overview of the Session:

The focus of this session is to introduce the students to the inbuilt string manipulation functions in C.

10.2 In-Class Explanations

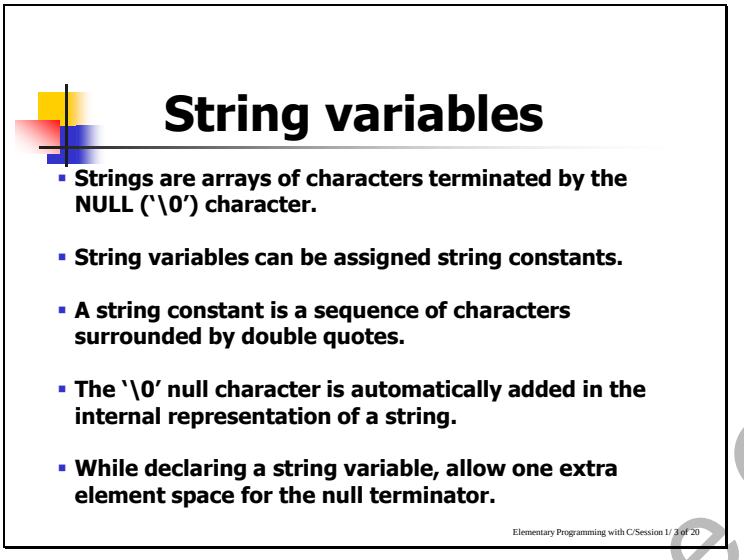
Slide 2

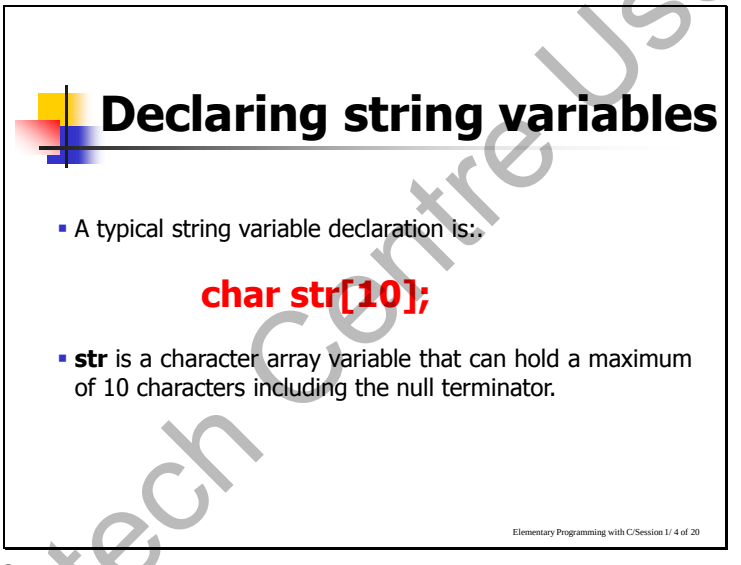


Max Time: 5 Minutes

Subject: List the objectives as given on slide 2 and inform the students what they are going to learn in next 2 hours. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slides-3 and 4

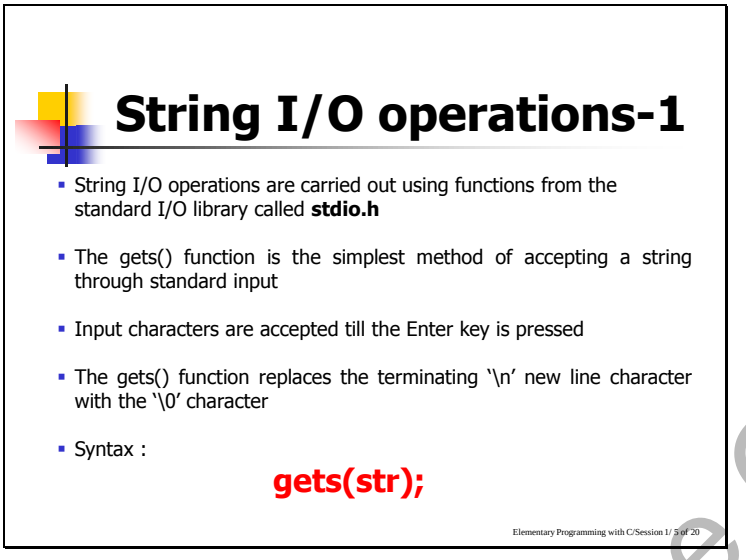


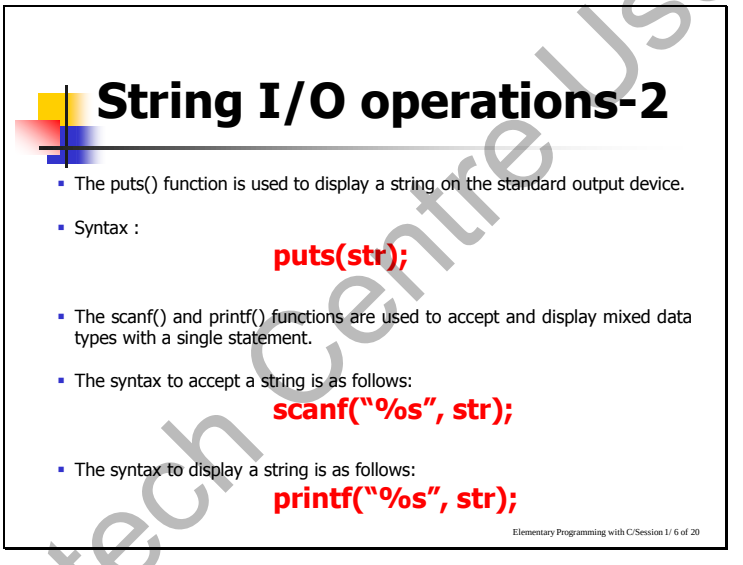


Max Time: 10 Minutes

Subject: The teaching of the session should start with the answers given by the students for their knowledge about strings in general. Followed by this, you should go on to explain the concept of string and string variable in C. Syntax declaration of strings should stress on the fact that strings are stored in C in character arrays. Emphasize the fact that the character array storing a string has the last character as \0 (Null character) also known as NULL terminator.

Slides-5 and 6

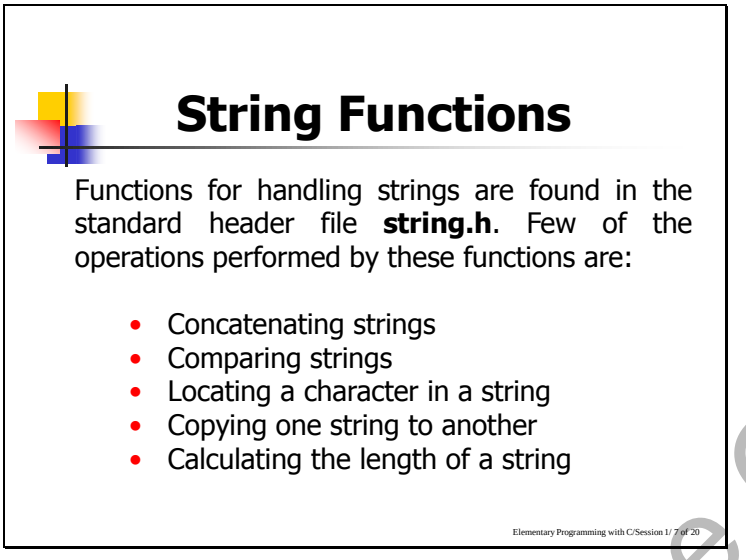




Max Time: 10 Minutes

Subject: The syntax of gets(),puts() should be explained to students very clearly. Input and output of strings using printf() and scanf() should also be discussed. Stress should be laid on the fact that in scanf() we do not use the '&' symbol while inputting the string. Students should be made aware that the declaration of character arrays is needed to store strings. Also the size of the character array has to be one more than the number of characters in the string so as to accommodate the '\0' Null terminator. Examples given in the SG regarding gets(), puts(), printf(), and scanf() should be explained.

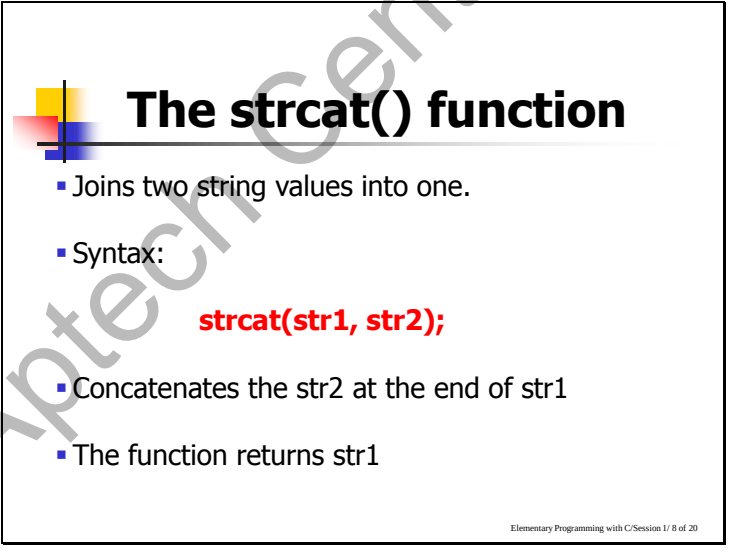
Slide-7



Max Time: 5 Minutes

Subject: Explain the students the usage of the string functions with respect to computer databases (maintained using files). A brief idea of their usage can be conveyed to students.

Slides-8, 9, 10, 11, and 12



The strcmp() function

- Compares two strings and returns an integer value based on the results of the comparison.
- Syntax:

  strcmp(str1, str2);

- The function returns a value:
  - Less than zero if str1<str2
  - Zero if str1 is same as str2
  - Greater than zero if str1>str2

Elementary Programming with C/Session 1/ 9 of 20

The strchr() function

- Determines the occurrence of a character in a string.
- Syntax:

  strchr(str, chr);

- The function returns a value:
  - Pointer to the first occurrence of the character (pointed by **chr**) in the string, **str**
  - NULL if it is not present

Elementary Programming with C/Session 1/ 10 of 20

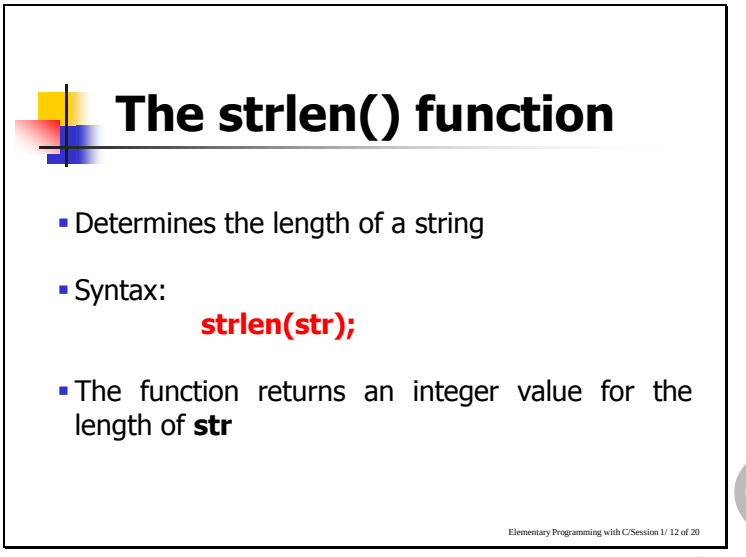
The strcpy() function

- Copies the value in one string onto another
- Syntax:

  strcpy(str1, str2);

- The value of str2 is copied onto str1
- The function returns **str1**

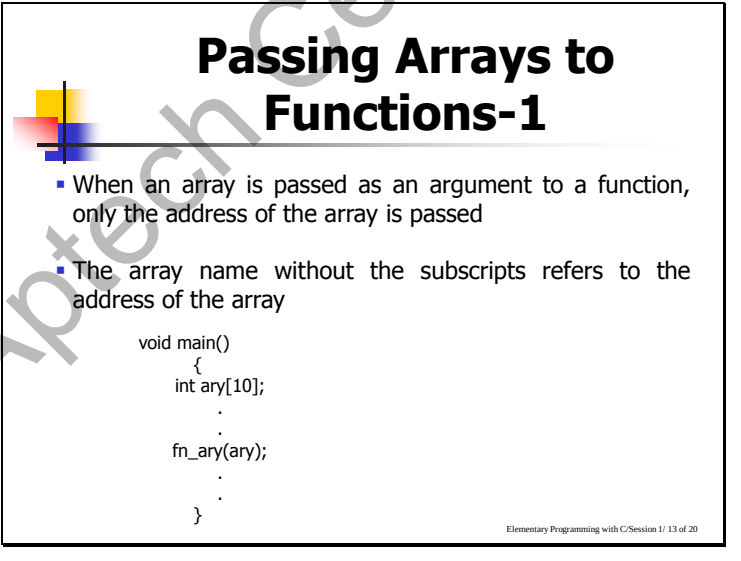
Elementary Programming with C/Session 1/ 11 of 20



Max Time: 30 Minutes

Subject: While explaining the different functions, parameters for each of the functions have to be explained in detail. Examples given in the SG have to be explained step wise while explaining the string functions. While explaining the examples, the names of the students can be used to demonstrate the effect of the string functions if they are passed as parameters.

Slides-13, 14, 15, and 16



Passing Arrays to Functions-2

```
#include<stdio.h>

void  main()
{
int num[5], ctr, sum=0;
int sum_arr(int num_arr[]); /* Function declaration */

  clrscr();

  for(ctr=0;ctr<5;ctr++) /* Accepts numbers into the array */
  {
        printf("\nEnter number %d: ", ctr+1);
        scanf("%d", &num[ctr]);
  }
```


Passing Arrays to Functions-3

```
  sum=sum_arr(num); /* Invokes the function */

  printf("\nThe sum of the array is %d", sum);

  getch();
}

int sum_arr(int num_arr[]) /* Function definition */
{
  int i, total;

  for(i=0,total=0;i<5;i++) /* Calculates the sum */
        total+=num_arr[i];

  return total; /* Returns the sum to main() */
}
```


Passing Arrays to Functions-4

Sample output of the program

Enter number 1: 5

Enter number 2: 10

Enter number 3: 13

Enter number 4: 26

Enter number 5: 21

The sum of the array is 75

Max Time: 30 Minutes

Subject: The slide can be used to emphasize on the fact that when arrays are passed to functions, additional memory is not used to store the passed values. It is the base address of the array which is passed and any manipulation done on the array in the function will be reflected in the array itself. Local variables are not created when arrays are passed. So when an array is passed to a function, it is a call by reference and not value. The example given on slides 14, 15, and 16 should be explained in detail.

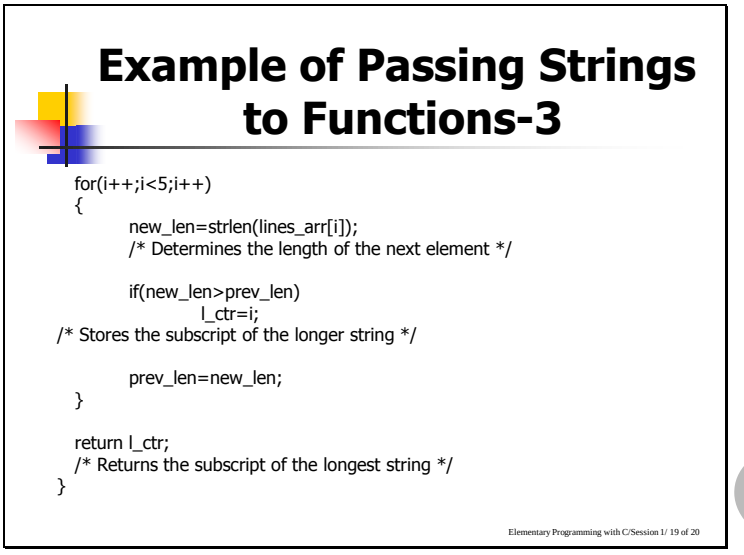
Slides-17, 18, 19, and 20

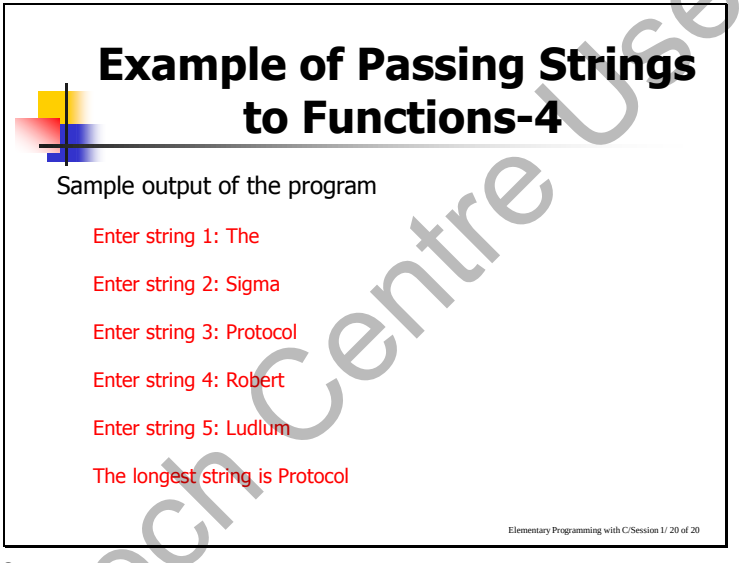
Example of Passing Strings to Functions-1

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

void  main()
{
char lines[5][20];
int ctr, longctr=0;
int longest(char lines_arr[][20]);
/* Function declaration */

   clrscr();
   for(ctr=0;ctr<5;ctr++)
   /* Accepts string values into the array */
   {
         printf("\nEnter string %d: ", ctr+1);
         scanf("%s", lines[ctr]);
   }
```


Example of Passing Strings to Functions-2

```
   longctr=longest(lines);
   /* Passes the array to the function */

   printf("\nThe longest string is %s", lines[longctr]);

   getch();
}

int longest(char lines_arr[][20]) /* Function definition */
{
   int i=0, l_ctr=0, prev_len, new_len;

   prev_len=strlen(lines_arr[i]);
   /* Determines the length of the first element */
```





Max Time: 30 Minutes

Subject: First explain the passing of arrays to functions. The example given in the SG has to be explained in detail. The students can be asked to add more string manipulation functions to this program after the example is explained.

Useful links on the Internet regarding Strings

http://www.howstuffworks.com/c14.htm
http://www.cs.cf.ac.uk/Dave/C/node7.html#SECTION00700000000000000000
http://gd.tuwien.ac.at/languages/c/programming-bbrown/c_030.htm
http://www.eskimo.com/~scs/cclass/notes/sx8.html

Solutions to Check Your Progress

1. NULL ('\0')
2. 14
3. True
4. True
5. string.h
6. False
7. Zero
8. Address

Solutions to Do It Yourself

1.

```
#include <stdio.h>

void  main()
{
   char str1[20], str2[5], flag='y';
   int len1, len2;

   clrscr();

   printf("\nEnter main string : ");
   scanf("%s", str1);
   printf("\nEnter tail string : ");
   scanf("%s", str2);

   len1=strlen(str1);
   len2=strlen(str2);

   for(len1--,len2--;len1>0 && len2>0 && flag=='y';len1--
   ,len2--)
        if(str1[len1]!=str2[len2])
             flag='n';

   if(flag=='y')
        printf("%s occurs at the end of %s", str2, str1);
   else
        printf("%s does not occur at the end of %s", str2,     str1);

   getch();
}
```

2.

```
#include <stdio.h>

void  main()
{
   int num[5], ctr, avg=0;
   int avg_arr(int num_arr[]);

   clrscr();

   for(ctr=0;ctr<5;ctr++)
   {
         printf("\nEnter number %d: ", ctr+1);
         scanf("%d", &num[ctr]);
   }

   avg=avg_arr(num);

   printf("\nThe average of the array is %d", avg);

   getch();
}

int avg_arr(int num_arr[])
{
   int i, total;

   for(i=0,total=0;i<5;i++)
         total+=num_arr[i];

   return total/i;
}
```

10.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 11 – Advanced Data Types and Sorting

Note: This TG maps to Session 19 of the book.

11.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

11.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain structures and their use
- Define structures
- Declare structure variables
- Explain how structure elements are accessed
- Explain how structures are initialized
- Explain how assignment statements are used with structures
- Explain how structures can be passed as arguments to functions
- Use arrays of structures
- Explain the initialization of structure arrays
- Explain pointers to structures
- Explain how structure pointers can be passed as arguments to functions
- Explain the typedef keyword
- Explain array sorting with the Selection sort and Bubble sort methods

Difficulties

The students may find difficulty in understanding the following topic:

- How structures can be passed as arguments to functions
- Arrays of structures
- Pointers to structures
- Sorting

Hence, the mentioned topic should be handled carefully.

11.1.2 Teaching Skills

You should be aware of and well-versed with advanced data types in C, especially structures, and the concept of array sorting for teaching this session. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

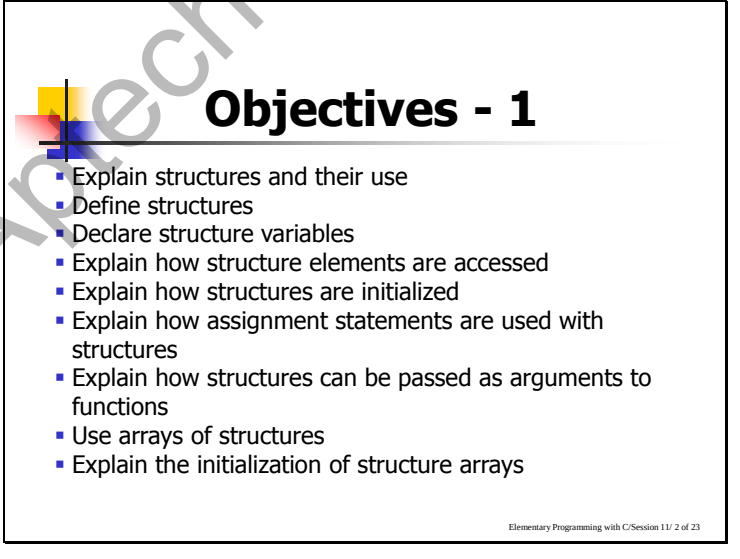
Follow the order as given here for the In-Class activities.

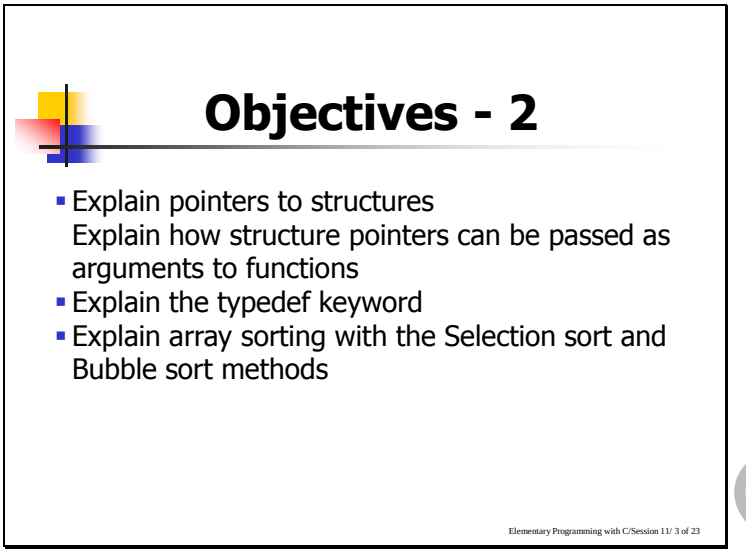
Overview of the Session:

The focus of this session is to introduce the students to the concept of advanced data types and sorting.

11.2 In-Class Explanations

Slides 2 and 3

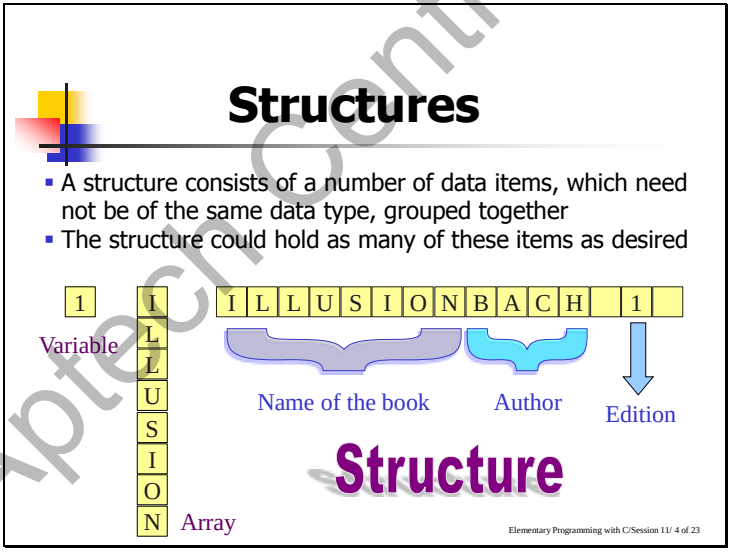


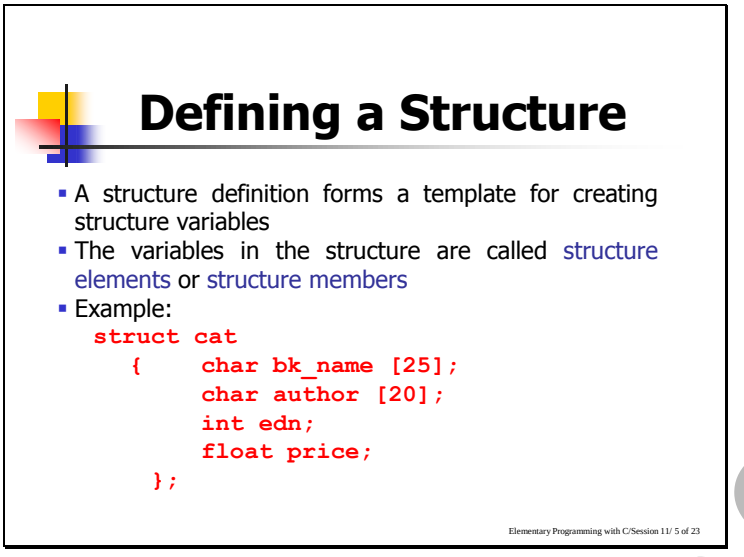


Max Time: 2 Minutes

Subject: Slide 2 and slide 3 should be read out to inform the students what they are going to learn in next 2 hours. By the end of slide 3, students should have an idea of the topics they will be learning in the session.

Slides-4 and 5





Max Time: 10 Minutes

Subject: The teaching of the session should start with asking the students about the normal data types in C. You should then explain the concept of structure and should explain how it is different from other data types of C. Also, clarify the need for this advanced data type. The generic syntax of defining a structure variable as given in the book should be explained.

Additional Information:

The following material could be used to explain structures.

Usually, if you have lots of different but related information, you tend to store it together. For example a file about an employee will probably have their name, age, the hours they worked, salary, etc. All of that is usually stored together in someone's filing cabinet. In programming, if you have lots of related information you do the same thing that is you group it together in an organized fashion. Let's say you have a bunch of employees and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place.
That is where structures come in. C's structures allow you to store multiple variables of ANY type in one place (the structure). Structures can even have arrays or other structures stored in them. They can hold any number of variables and you can make arrays of structures! All that flexibility make structures ideal for databases. Before we get too much further, let's see how structure variables are declared.

```
struct employee
{
        int age;
        float salary;
        char fName[15];
        char lName[20];
};
```

To declare a structure you must start with the keyword struct. Next, type the structure's name. After an open curly bracket you can fill in any variable declarations you want. In this case we want a record

for employees, so we have an integer for their age, a float for their salary, and two strings for their first and last name. When you have all the variables you want, end with a closing curly bracket, and a semicolon. Note this does not actually create any variables, at this point there has been no memory allocated, and there is no 'employee' structure for you to work with. All we have done so far is to declare a type of structure, later we will use this structure type to make actual variables based on this type. First however, let's look at some of the terminology associated with structures.

The structure name is often referred to as its *tag*. The variables declared inside a structure are called its *fields*. Sometimes structure fields are also called *data members*. That's all there is to it.

The following material could be used to explain the abilities and limitation of structures. It also gives a review of structure.

Source: http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial/structs.html

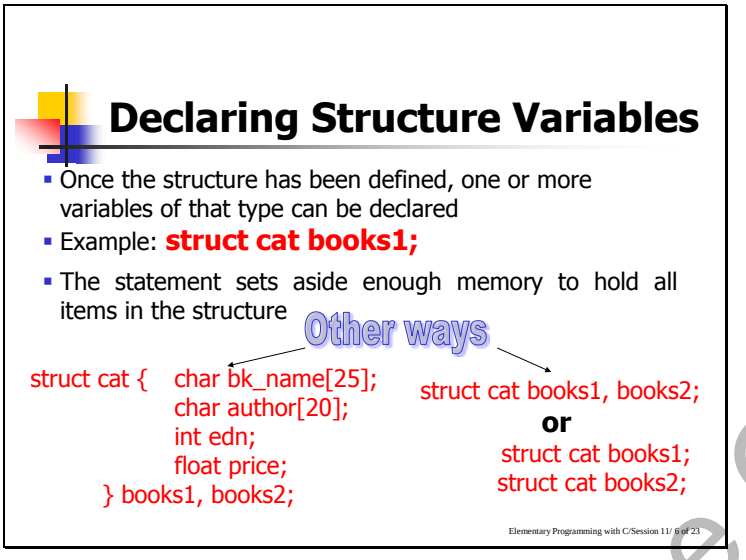
Structures: Abilities and Limitations

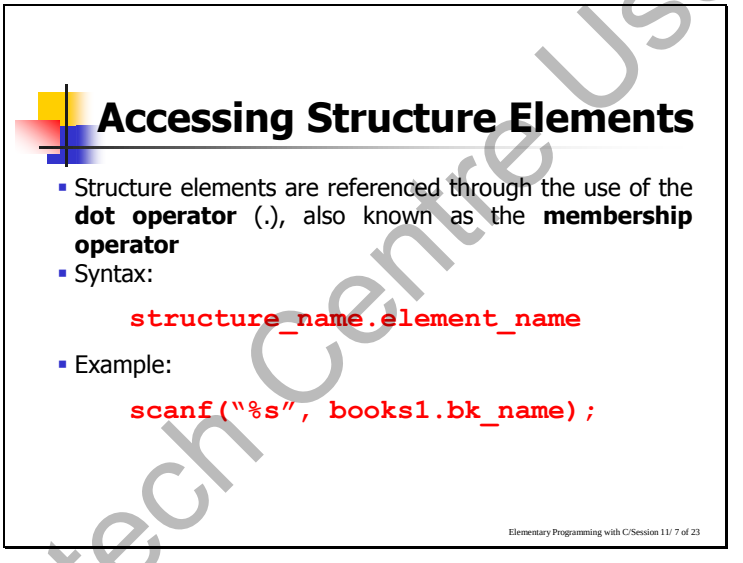
- You can create arrays of structs.
- Structs can be copied or assigned.
- The & operator may be used with structs to show addresses.
- Structs can be passed into functions. Structs can also be returned from functions.
- Structs **cannot** be compared!

Structures Review

- Structures can store non-homogenous data types into a single collection, much like an array does for common data (except it isn't accessed in the same manner).
- Pointers to structs have a special infix operator: for dereferencing the pointer.
- typedef can help you clear your code up and can help save some keystrokes.
- Enumerated types allow you to have a series of constants much like a series of #define statements.

Slides-6 and 7





Max Time: 15 Minutes

Subject: The teaching of this session should start with the answers given by the students for the questions asked on structures. The generic syntax of how to declare a structure and how to access a structure element as given in the book should be explained. Emphases should be given on how to use the dot operator.

Additional Information:

The following material could be used to explain how to declare a structure and how to access structure elements.

Declaring Structure Variables:

The given declaration done in the previous topic does not actually create any variables. To do that, we have to do something like this:

```
struct employee Jill;
//or
struct employee Jack, Jill, Bill;
```

Simply start with the *struct* keyword, then the tag of the structure you want variables of, and finally the actual variable name(s). As you can see, the tag, gets used as the variable type, just like int, char, or float. It's kind of like declaring your own new variable type (which will actually be discussed later). The result of the second declaration is that there are now three variables of type employee. Each employee has their own unique set of data fields. By the way, to calculate the size, in bytes, of a structure just add up the size of its data members. Our employee structure has a four byte integer and float, one fifteen byte string and one twenty byte string. All totaled, each employee variable is forty three bytes! You can verify this by writing a little program that uses the size of keyword.

More ways of Declaring Structure Variables:

There are other (probably more convenient) ways to declare variables of structures than the one I mentioned. One way is to list the variable names after the closing curly bracket of the structure definition, but before the semicolon. Like this:

```
struct employee
{
        int age;
        float salary;
        char fName[15];
        char lName[20];
}Jack, Jill, Jane;
```

This method yields the exact same results as typing struct employee Jack, Jill, Bill; as we did earlier. If you know you will only create certain variables of a certain structure type, you can do what we just did, and leave out the structure tag. That would look like this:

```
struct
{
        int age;
        float salary;
        char fName[15];
        char lName[20];
}Jack, Jill, Jane;
```

We still have three variables called Jack, Jill, and Jane, but we can't declare anymore, and it would be tricky to pass them as function parameters. I don't ever use this method. The last (and probably the best) way to create structures is to actually declare your own type. This is done by using the keyword typedef. It works like this:

```
typedef struct
```

```
{
        int age;
        float salary;
        char fName[15];
        char lName[20];
}employee;
```

NOTE: In a source file, type definitions usually go up with prototypes, and those #define things, above the main function.
Now, you actually told the compiler that employee is a variable type, just like int or char. Notice though, that the structure tag is now at the end of the definition. You can now do this to declare variables:

```
        //given the code above
        employee Jill, Jack, Jane;
```

You would use the type employee just as you would any other variable type. This is how I will declare structures in most future examples.

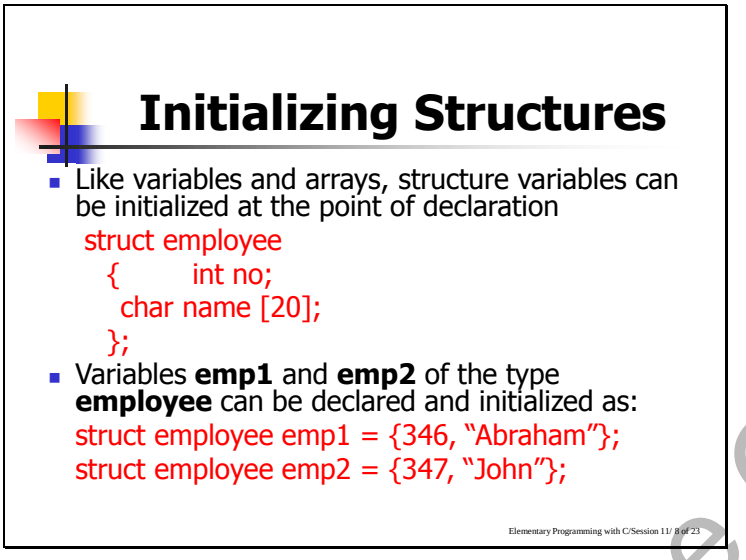
Accessing Structure Elements:

The fields of structures can be accessed with the . operator. It is called the *dot* operator. It works like this:

```
//Assume we already declared an employee variable called Jill
        Jill.age = 24;
        Jill.salary = 10000.50;
        strcpy(Jill.fName, "Jill");
        strcpy(Jill.lName, "Smith");
        printf("%s %s is %d years old.", Jill.fName, Jill.lName, Jill.age);
```

The code mentioned fills all the fields of the variable Jill, then prints some of it out. When using the dot operator, you can use the fields just like you would any other variable. You can assign those values, use them in calculations, pass them as parameters, or anything else you can think of.

Slide-8



Max Time: 5 Minutes

Subject: This section explains how to initialize a structure variable. At this stage, the individual member elements of a structure variable are assigned some values. The elements are simply listed inside a pair of braces, with each element separated by a comma.

Additional Information:

The following material could be used to explain how to initialize structures.

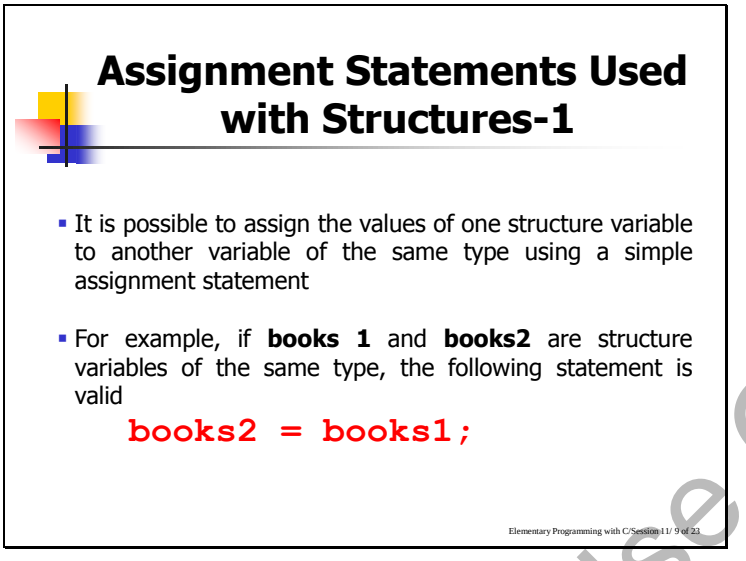
- Structure variable can be initialized when defined by using values of the proper type for each member inside braces.
- Similar to array initialization, but value types can vary.
- The following creates structure definition for CBOX and allocates 2 variables. *blockx* is not initialized, but *block2* is.

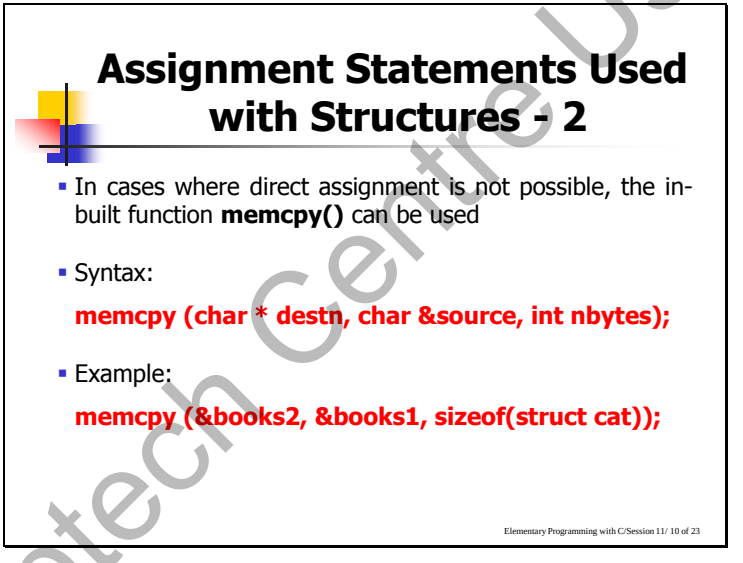
```
struct CBOX
{
   double lng;
   double wid;
   double hgt;
   char  *color;
}  blockx,
block2 = { 10.5, 17.2, 12.0, "red" };
```

The following allocates 2 more variables, both initialized

```
struct CBOX slab1 =
            { 12.5,7.75,8.0, "red"};

struct CBOX blockhead =
{
            10.5,       /* length */
            17.3,       /* width */
            9.75,       /* height */
            "yellow"    /* color */
}
```

Slides-9 and 10





Max Time: 5 Minutes

Subject: The example given in the book could be used to explain about assignment statements in structures. Emphasize on both the ways of assignment as given in the book.

Slide-11

Structures within Structures

- It is possible to have one structure within another structure. A structure cannot be nested within itself

```
struct issue
{
   char borrower [20];
   char dt_of_issue[8];
   struct cat books;
}issl;
```

- To access the elements of the structure the format will be similar to the one used with normal structures,

```
issl.borrower
```

- To access elements of the structure cat, which is a part of another structure issue,

```
issl.books.author
```

Elementary Programming with C/Session 11/ 11 of 23

Max Time: 10 Minutes

Subject: The example given in the book could be used to explain about structures within structures. Emphasize on the fact that structure cannot be nested within itself.

Additional Information:

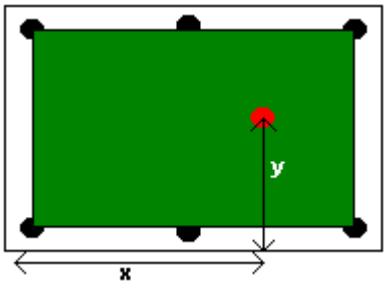
The following material could be used to explain structures within structures.

It is perfectly possible to define one structure as part of another. Here's an example - a snooker table. Again, I know less about snooker than I do about Monopoly, so forgive me if I get some of the details wrong! A snooker table has a length and a width. It also has six pockets, one at each corner and one in the middle of the longest sides. I believe there are 10 red balls and one each of the following colors: white, black, green, yellow, brown, blue, and pink, each worth different numbers of points. Each ball is defined by its position on the table, or it could be off the table (in somebody's pint) or down one of the pockets.

Right. Let's start by defining a position as a structure. This is defined as an x-co-ordinate (horizontal position) and a y-co-ordinate.

struct position
{ float x,y; };

I've put the whole body of the structure all on one line.



Now, we move up one step and define a ball. This has a position, a color, a point value, and a status (1 meaning on the table, 0 meaning in a pocket and -1 meaning off the table for some reason).

```
struct ball
  { position p;
    int color;  // 0=black, 1=white etc.
```

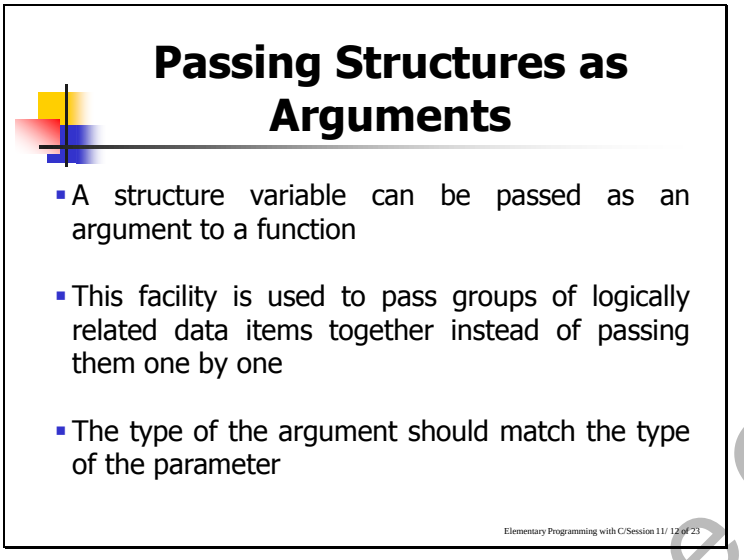
```
    int points;
    int status;
  };
```

This contains a reference to the previous structure that we defined. Finally, the table itself is defined as having 17 balls and 6 pockets. Each pocket is defined solely by its position.

```
struct table
  { ball balls[17];  // First ball is balls[0]
    position pockets[6];
    float length,width;
  };

table snooker;
```

The structures within structures are accessed using multiple full stops. The table length is set using snooker.length = 2.4; and the pocket positions are set using snooker.pockets[0].x = 0; and snooker.pockets[0].y = 0;. The ball positions are set using even more full stops: snooker.balls[4].p.x = 1.466; and snooker.balls[4].p.y = 0.782;.

Slide-12



Max Time: 15 Minutes

Subject: The example given in the book could be used to explain about how to pass structure as an argument to a function.

Additional Information:

The following material could be used to explain how to pass structures as arguments.

As I just said, if you typedef a structure, you can use it just like any other variable type. This holds even for parameter passing, and return values. In the prototype, and formal parameter list just use the tag followed by the variable name. Let's assume we had a function to display an employees data called PrintInfo. It takes one parameter (a structure of type employee) and returns nothing. Here is some code to pass an employee structure to a function:

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

typedef struct
{
        int age;
        float salary;
        char fName[15];
        char lName[20];
}employee;

void PrintInfo ( employee );

int main()
{
        employee Jill;

        Jill.age = 24;
        Jill.salary = 10000.50;
```

```
        strcpy(Jill.fName, "Jill");
        strcpy(Jill.lName, "Smith");

        PrintInfo( Jill );

    return 0;
}

void PrintInfo( employee emp )
{
    printf("%s %s is %d years old.", emp.fName, emp.lName,      emp.age);
    printf("\n%s's salary is %.2f.\n", emp.lName, emp.salary);
}
```

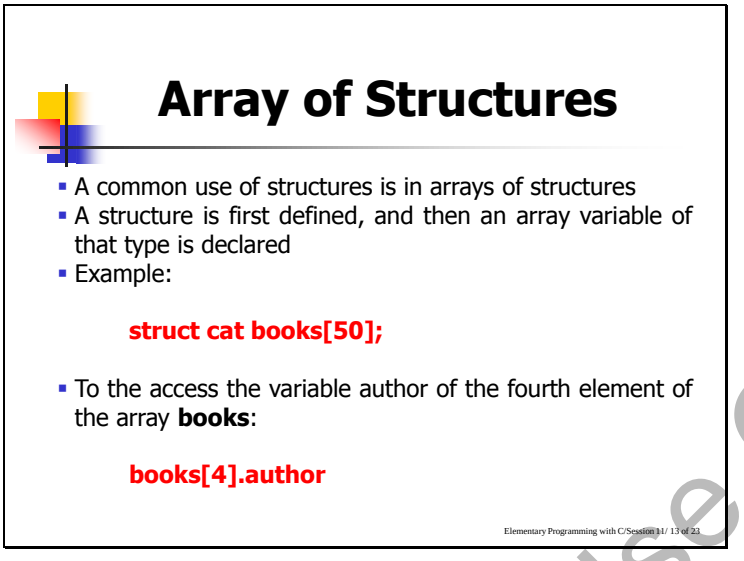
As you can see, passing a struct is just like passing a normal variable. Note that the typedef must come before the prototype, or else the compiler will not know what the prototype is talking about in its parameter list.
Returning structures is also the same as returning other types. Just use the return keyword, and have a variable catch the results. Here is a snippet where a function increases an employee's salary. It takes one parameter (an employee variable) and returns a structure of type employee.

```
.
.
        Jill = RaiseSalary( Jill );
.
.

employee RaiseSalary( employee emp )
{
        emp.salary = emp.salary * 1.1;     //A ten percent raise

        return emp;
}
```

The RaiseSalary function is called with Jill as its parameter. Her salary is increased by 10% then the modified structure is returned.
If you do not typedef you structure, you can still do all the same stuff, you just have to use the struct keyword in front of employee every time it is used. For example a function heading might look like this:

```
        void PrintInfo( struct employee emp )
```

Slide-13



Max Time: 10 Minutes

Subject: The syntax for declaring an array of structures given in the book should be explained. You should emphasize that the structure should be defined before declaring the array variable.

Comparison between Structures and Arrays:

- Same like arrays, structures are single, named items which on closer examination have internal fields which can store values.
- In case of structure the members can be of different types and lengths which is not in case of Arrays.
- In case of structures the order of the members is irrelevant where they are to be thought of as simply grouped together for programming convenience. This is not true with arrays.
- You can only access structure members by naming them, not by indexing.
- Same as in case of arrays, we can use the & operator in order to obtain the address of the start of the structure. This is a convenient way of passing structures to and from functions.

Additional Information:

The following material could be used to explain the arrays of structures.

Arrays of structures:

Again, using arrays of structures is similar to using arrays of anything else. No matter which way you declare your structure variables, just use the normal array notation (square brackets enclosing the number of elements) after the variable name. So if you already defined (but did not typedefed) a structure employee, and you have 20 employees, you can do this:

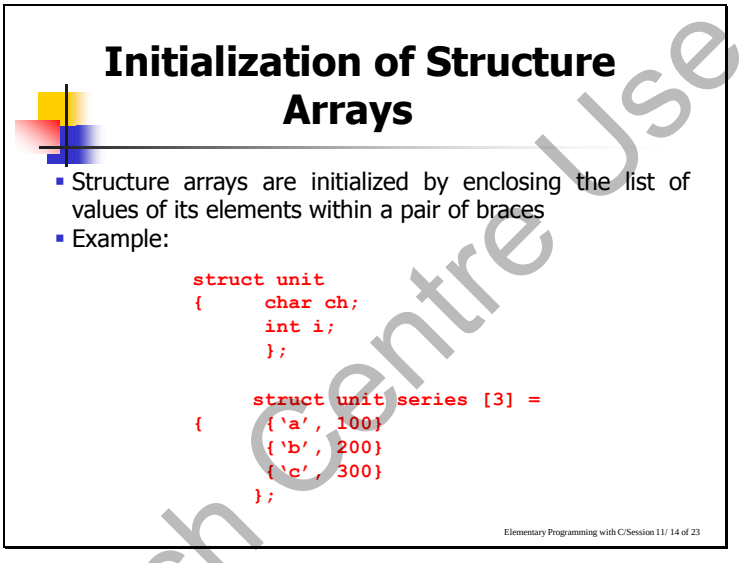
```
struct employee workers[20];
```

The mentioned code would give you an array of twenty employee structures. To use those structures, just use the array notation, like you normally would to select a specific employee, then use the dot operator. Like so:

```
for( i=0; i<20; i++)
       worker[i].salary = 24000.00;
```

That's all there is to it. Use the brackets like you would for any variable then the dot operator to access the data members. So, to sum up, structures are pretty much like regular variables, except that you need the dot operator sometimes. Next, we will look at a pretty simple database program based on arrays of structures.

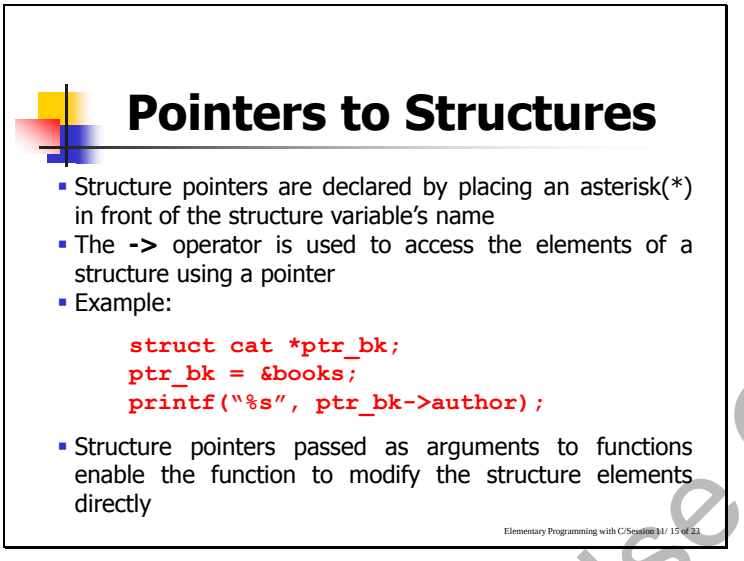
Slide-14



Max Time: 5 Minutes

Subject: The syntax for initializing an array of structures given in the book should be explained.

Slide-15



Max Time: 10 Minutes

Subject: The syntax for declaring pointers to structures given in the book should be explained. Emphasize that the **->** operator is used to access the elements of a structure using a pointer.

Additional Information:

The following material could be used to explain the concept of pointers to structures.

Source : http://oopweb.com/CPP/Documents/FromTheGroundUp/Volume/5/3.html

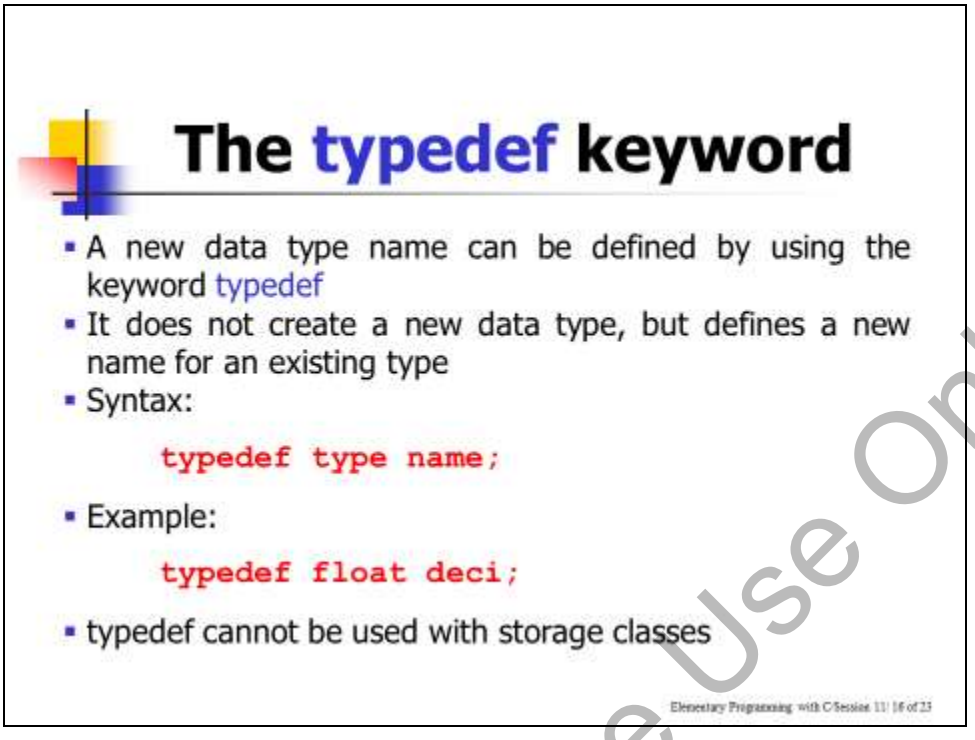
Before we move on, take a look at some examples and we'll step through them so you can be sure you're learning all this. Here we go,

```
#include <iostream.h>

struct food
{
   int rating;       /* Rating, out of ten */
   double weight;    /* Weight, in Pounds */
   int ttc;          /* Time To Cook */
   int calories;     /* Seconds it takes to cook */
};

void main(void)
{
   food good;    /* A can of Jolt Cola */
   food *bad;    /* A school "lunch"   */

   bad = new food;  /* Create the school "food" */

   good.rating = 10;  /* Fill in Jolt Cola's stats */
```

```
   good.weight = .2;
   good.ttc    = 0;
   good.calories = 150;

   bad->rating = 1; /* Fill in School "Food"'s stats */
   bad->weight = 5.0;
   bad->ttc    = 30;
   bad->calories = 1400;

   if (good.rating > bad->rating)
      cout << "I like Jolt Cola more than that icky pseudo-food." << endl;
   else
     cout << "I like watching my food glow!" << endl;
}
```

Here's the breakdown: First, a structure named food is defined, detailing a bunch of variables I'd use to describe a food. Don't forget that semicolon at the end of the structure's definition! Inside the main procedure, a variable good is defined of type food, and a variable bad is defined to be of type pointer-to-food. Right afterwards, bad is filled is given something to point at. Note that saying "bad = new food" is just like you've done in the past, nothing's changed. Next, the variables are all given values. Finally, a comparison of good.rating and bad->rating is done; if good.rating is higher (in this code, it is), it spits out a message saying so. If not, is spits out a message saying how much the program likes school lunch.

Slide-16



Max Time: 10 Minutes

Subject: The syntax for typedef keyword given in the book should be explained.

Additional Information:

The following material could be used to explain typedef keyword.

You can define your own types by using typedef. These types can either be entirely new types, such as record structures, or aliases for existing types. It's common practice for mainframe shops to use typedef to define types such as WORD or BYTE, presumably to make C more familiar to assembler programmers. Be aware that merely calling an integer WORD won't make it fullword align if you're using a noalign compiler option.

It's considered good practice to type typedefs in UPPERCASE, or at least put the first character in uppercase.

```
typedef char * STRING;
typedef int WORD;
typedef unsigned char BYTE;
typedef float AMOUNT;
typedef struct salesrec SalesRec, *SalesPtr;

STRING cptr;
WORD count, *iptr;
BYTE flag = 'N';
AMOUNT price = 0.0;
```

```
SalesRec order;
SalesPtr order_p;
```

The following material could be also used to explain typedef keyword.

***typedef* - define your own data types**

- *typedef* allows you to give new names to basic types or name complex types
- Once defined, new type can be used in declaration just like basic types
- *typedef*'s typically placed in header files, provide standardized and extended types
- *typedef* used to define portable types for same source code in various environments, such as *INT* which is 32 bits in any environment
- Format
  typedef *typespec typnm1*, *typnm2*;
- Standard is that all defined types should use all upper case names
- *typespec* can be basic type or complex with *struct*, *union,* or *enum*
- Using *typedef* to define structures, unions, and enumerations simplifies defining variables of that type later.
- *typedef* used more frequently than *struct*, *union,* or *enum* tags
- Using *typedef* to give names to basic data types

```
typedef unsigned short  USHORT;
typedef signed long     LONG;
typedef signed long     INT;
typedef double          FLOAT;

...

void main()
{
   USHORT x, y;
   INT    totx, i, j;
   FLOAT  pctincr = .015;
   LONG   burgers_sold, mo_sales[12];
   ...

}
```

The new type definitions are used just like a basic C type in declarations.

- Using *typedef* to name complex types

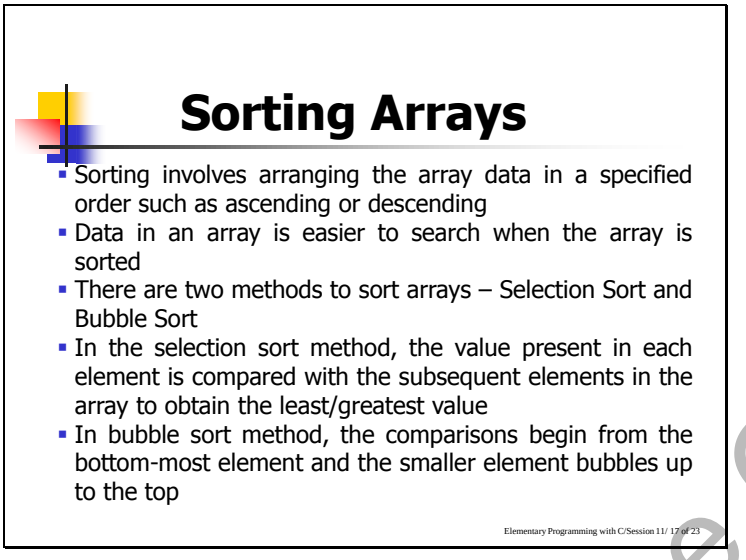
```
typedef struct
{
   short month;
   short day;
   short year;
}  GREGDATE;

typedef enum
{ CORN, WHEAT, SOYBEANS } CROPS;

typedef union
{
   float fval;
```

```
   char  cval;
   long  lval;
}  MIXEDUP;

...

int i, j;
GREGDATE orddate, duedate;
CROPS back40;
MIXEDUP whacko, x, zippy;

...

back40 = WHEAT;

whacko.fval = 7.081;

duedate.year = 1998;
```

- Can use * indirection operator in front of type name to declare type that is a pointer Declaration of variable then does NOT use the *

```
typedef struct
{
   short month;
   short day;
   short year;
}  GREGDATE, *PGREGDATE;

...
int i, j;
GREGDATE orddate, duedate;
PGREGDATE pdate;  /* ptr to type
                     GREGDATE */

...

pdate = &duedate

...

pdate->month = 11;
pdate->day = 30;
pdate->year = 1997;
```

Slide-17



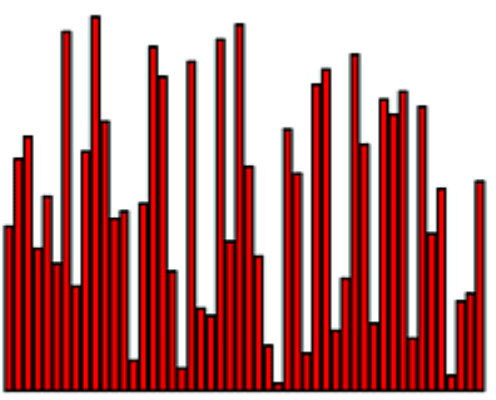
Max Time: 5 Minutes

Subject: Explain the students the purpose of sorting. Then discuss sorting of arrays. Once this is done, then move ahead with explaining various algorithms for doing this.

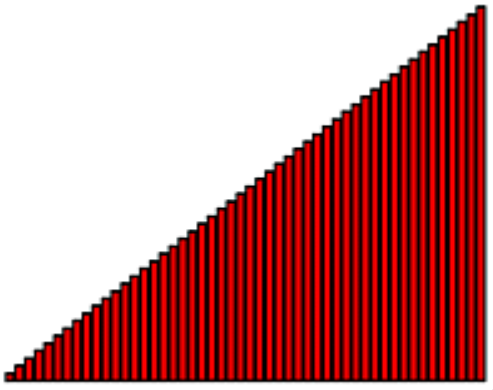
Additional Information:
The following material could be used to explain sorting of arrays process.

Definition: Sorting is to arrange things in order. Or more formally, “sorting is the problem of taking an arbitrary permutation of n items and rearranging them into the total order”. We need a comparison function to order two items.

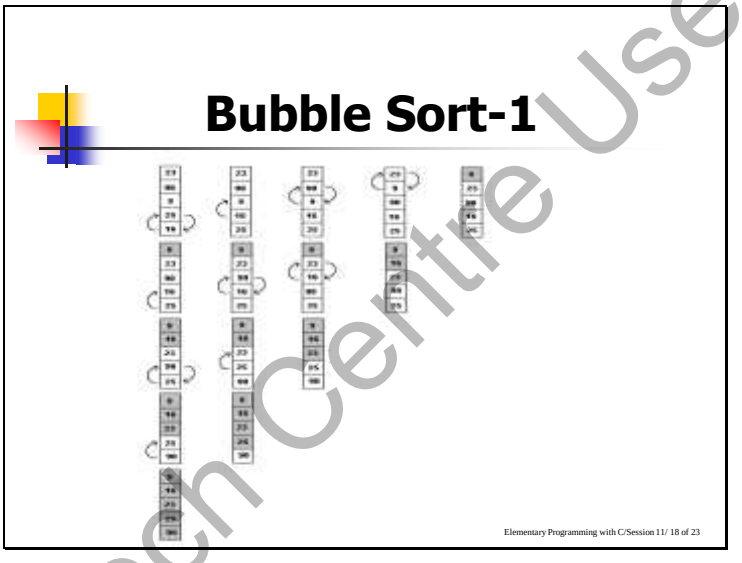
Input

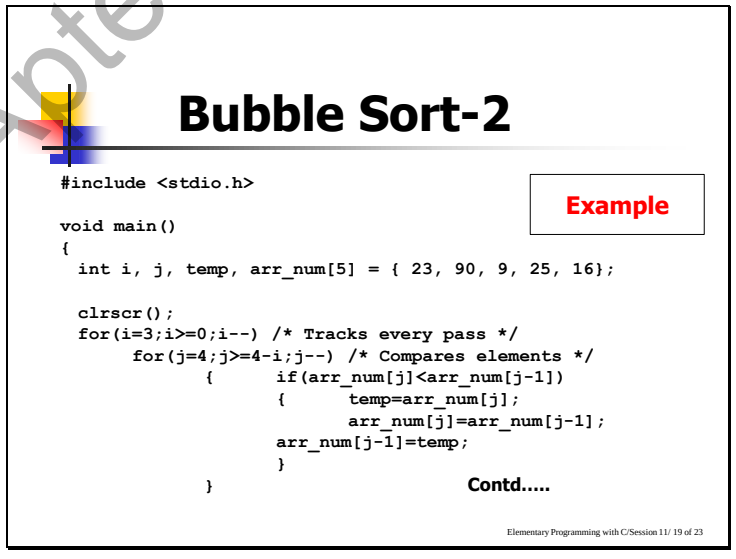


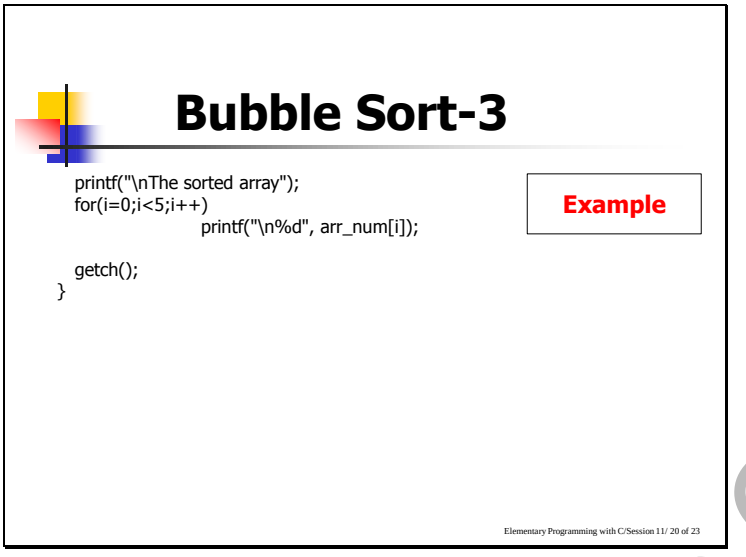
output



Slides-18, 19, and 20







Max Time: 15 Minutes

Subject: Explain the students the method of bubble sorting. Then discuss the example of bubble sorting given in the book.

Additional Information:

The following material could be used to explain bubble sorting method for sorting arrays.

Array sorting - 1 dimensional array:

I can say that I definitely learnt something really cool in this last week. I was given the assignment of reading our IIS log files and constructing management reports from the data. It didn't take me long to realize that working with data without the benefit of SQL is long winded.

Each file was the equivalent of one day so I decided that I'd place all of the data for each file into its own Day object. So I created a Class accordingly. This enabled me to have many properties for each day, such as the Date, Number of Hits, Number of Visitors, Bytes Sent, and Bytes Recieved.

After a considerable amount of data munging I ended up with a Dictionary object full of Day Objects with the dates as the Keys.

Now came the tough part. I had to work out how to sort my data. For example, some people would like their reports shown in Date ascending order and some would like it shown in Hits ascending order. I'd never sorted an array before never mind a Dictionary full of Objects.

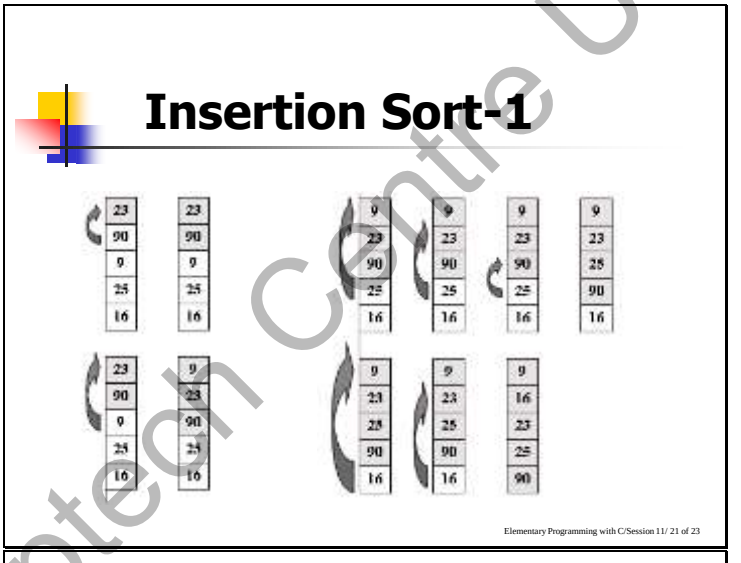
Let's start at the beginning. What are we trying to achieve here?

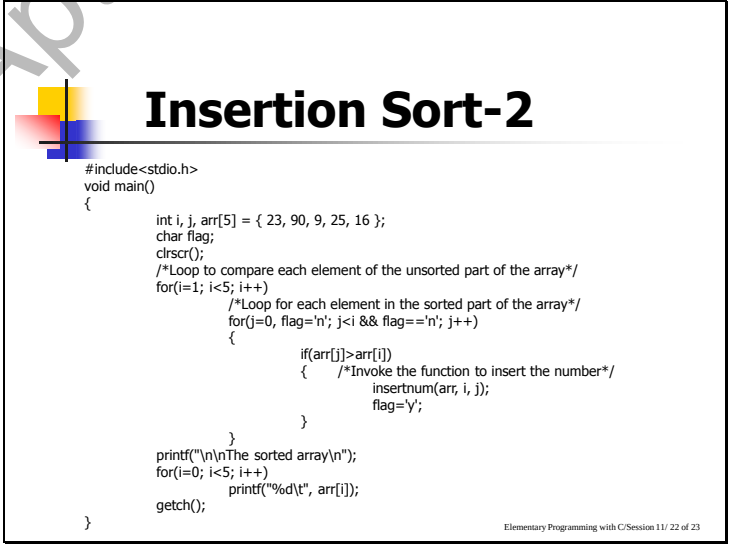
```
Starting:
    7 3 5 6 2
                Use all 5 elements:
                7 > 3, so swap: 3 7 5 6 2
                7 > 5, so swap: 3 5 7 6 2
                7 > 6, so swap: 3 5 6 7 2
                7 > 2, so swap: 3 5 6 2 7
        3 5 6 2 [7]
                Now use first 4 elements:
                3 not > 5, no swap
                5 not > 6, no swap
```

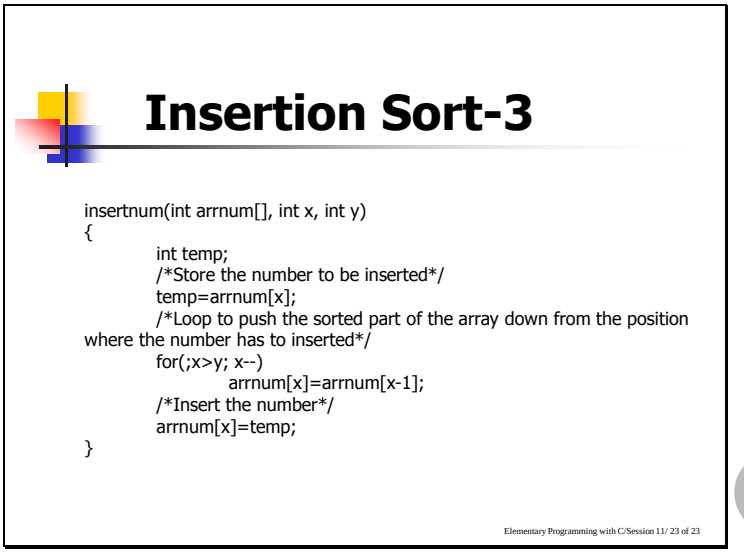
```
                6 > 2, so swap: 3 5 2 6 [7]
3 5 2 [6 7]
                3 elements:
                3 not > 5, no swap
                5 > 2, so swap: 3 2 5 [6 7]
3 2 [5 6 7]
                2 elements:
                3 > 2, so swap: 2 3 [5 6 7]
2 [3 5 6 7]
                1 element:
                nothing to do
[2 3 5 6 7]
```

This method is called a **Bubble Sort**. The keyword 'bubble' in bubble sort indicates that elements are bubbling to the top of the array.

Slides-21, 22, and 23







Max Time: 15 Minutes

Subject: Explain the students the method of insertion sorting. Then discuss the example of insertion sorting given in the book.

Additional Information:

The following material could be used to explain insertion sorting method for sorting arrays.

Sorting is one of the most important operations performed by computers. In the days of magnetic tape storage before modern data-bases, it was almost certainly the *most* common operation performed by computers as most 'database' updating was done by sorting transactions and merging them with a master file. It's still important for presentation of data extracted from databases: most people prefer to get reports sorted into some relevant order before wading through pages of data!

There are a large number of variations of one basic strategy for sorting. It's the same strategy that you use for sorting your bridge hand. You pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

```
/* Insertion sort for integers */

void insertion( int a[], int n ) {
/* Pre-condition: a contains n items to be sorted */
    int i, j, v;
    /* Initially, the first item is considered 'sorted' */
    /* i divides a into a sorted region, x<i, and an
       unsorted one, x >= i */
    for(i=1;i<n;i++) {
        /* Select the item at the beginning of the
           as yet unsorted section */
        v = a[i];
        /* Work backwards through the array, finding where v
           should go */
        j = i;
```

```
        /* If this element is greater than v,
              move it up one */
        while ( a[j-1] > v ) {
          a[j] = a[j-1]; j = j-1;
          if ( j <= 0 ) break;
          }
        /* Stopped when a[j-1] <= v, so put v at position j */
        a[j] = v;
        }
    }
```

Solutions to Check Your Progress

1. Structure
2. Dot operator
3. T
4. F
5. typedef
6. Adjacent
7. T

Solutions to Do It Yourself

1.

```
#include <stdio.h>
#include <string.h>
struct stud_marks{    char stud_name[20];
                      int marks;};

void main()
{
      int i, j, temp;
      char temp_name[20];
      struct stud_marks st_mk[5] ={ {"Diana", 65},
                                    {"John", 85},
                                    {"Kate", 73},
                                    {"Ben", 67},
                                    {"Lionel", 91}};

      clrscr();

      for(i=0;i<4;i++)
            for(j=i+1;j<5;j++)
            {     if(st_mk[i].marks<st_mk[j].marks)
                  {     temp=st_mk[i].marks;
                        strcpy(temp_name, st_mk[i].stud_name);
                        st_mk[i].marks=st_mk[j].marks;
                        strcpy(st_mk[i].stud_name,
                        st_mk[j].stud_name);
                        st_mk[j].marks=temp;
                        strcpy(st_mk[j].stud_name,temp_name);
                  }
```

```
            }

      printf("\nThe top 3 scores");
      for(i=0;i<3;i++)
            printf("\n%s scored %d", st_mk[i].stud_name,
            st_mk[i].marks);

      getch();
}
```

2.

```
#include <stdio.h>

struct stud_marks{    char stud_name[20];
                      int marks;};

void main()
{
      int i, j, temp;
      char temp_name[20];
      struct stud_marks st_mk[5] ={ {"Diana", 65},
                                    {"John", 85},
                                    {"Kate", 73},
                                    {"Ben", 67},
                                    {"Lionel", 91}};

      clrscr();

      for(i=0;i<5;i++)
            for(j=i+1;j<5;j++)
            {     if(st_mk[i].marks<st_mk[j].marks)
                  {     temp=st_mk[i].marks;
                        strcpy(temp_name, st_mk[i].stud_name);
                        st_mk[i].marks=st_mk[j].marks;
                        strcpy(st_mk[i].stud_name,
                        st_mk[j].stud_name);
                        st_mk[j].marks=temp;
                        strcpy(st_mk[j].stud_name,temp_name);
                  }
            }

      printf("\nThe top 3 scores");
      for(i=0;i<3;i++)
            printf("\n%s scored %d", st_mk[i].stud_name,
            st_mk[i].marks);

      getch();
}
```

11.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.

Session 12 – File Handling

Note: This TG maps to Session 21 of the book.

12.1 Pre-Class Activities

Before you commence the session, you should familiarize yourself with the topics of the current session in depth. Prepare a question or two, which will be a key point to relate the current session objectives.

12.1.1 Objectives

By the end of this session, the learners will be able to:

- Explain streams and files
- Discuss text streams and binary streams
- Explain the various file functions
- Explain file pointer
- Discuss current active pointer
- Explain command-line arguments

Difficulties

The students may find difficulty in understanding the following topic:

- File pointer
- File position
- Command Line Arguments

Hence, the mentioned topic should be handled carefully.

12.1.2 Teaching Skills

You should be well-versed with various file handling concepts in C for teaching this session. You will teach the concepts in the theory class and can use the programs that are given in the session to support the concepts.

You should teach the concepts in the theory class using slides and LCD projectors.

Tips:

It is recommended that you test the understanding of the students by asking questions in between the class.

In-Class Activities:

Follow the order as given here for the In-Class activities.

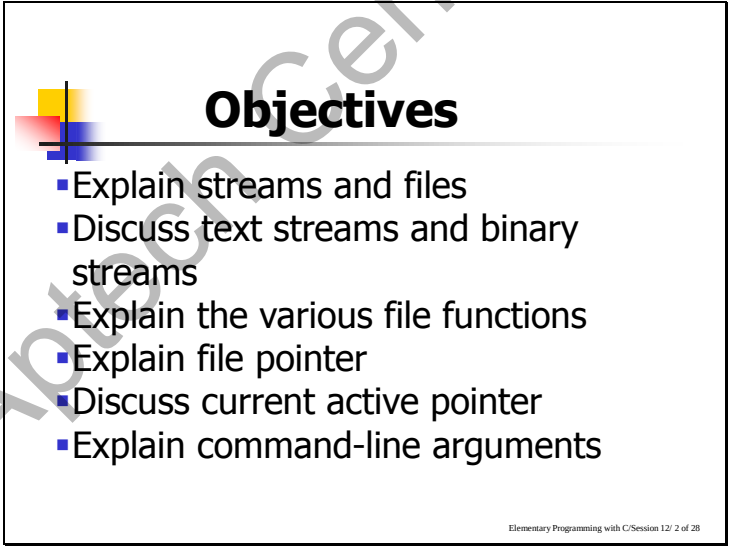
Overview of the Session:

The focus of this session is to introduce the students to the concept of file handling in C.

The session begins with explaining streams and files and proceeds to describe text streams and binary streams. The session explores the various file functions and the concept of file pointer and command-line arguments.

12.2 In-Class Explanations

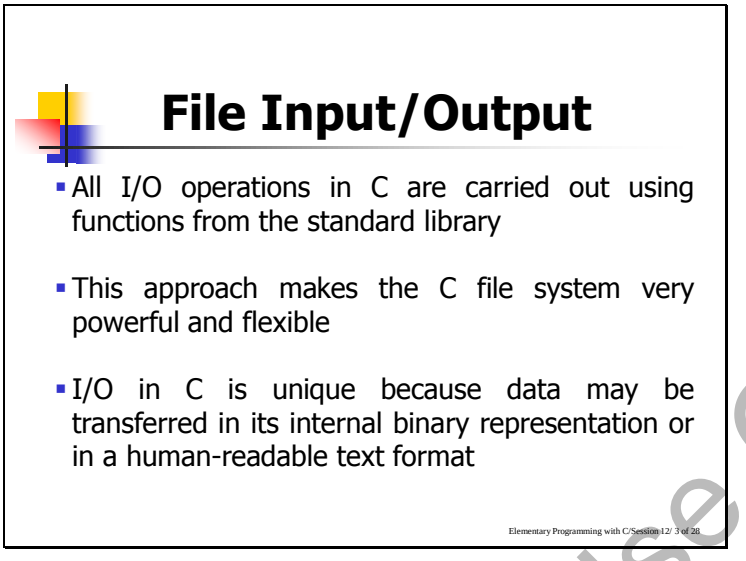
Slide 2

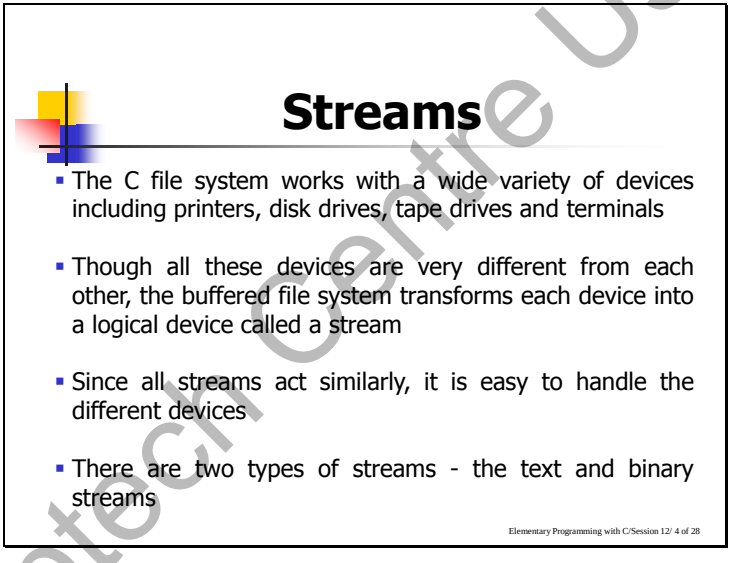


Max Time: 2 Minutes

Subject: List the objectives as given on Slide 2 and inform students what they are going to learn in next 2 hours. By the end of slide 2, students should have an idea of the topics they will be learning in the session.

Slides-3 and 4

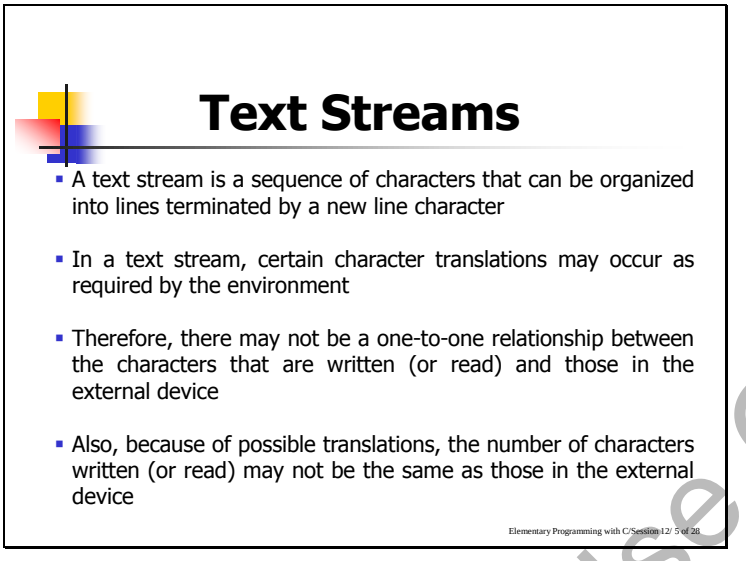


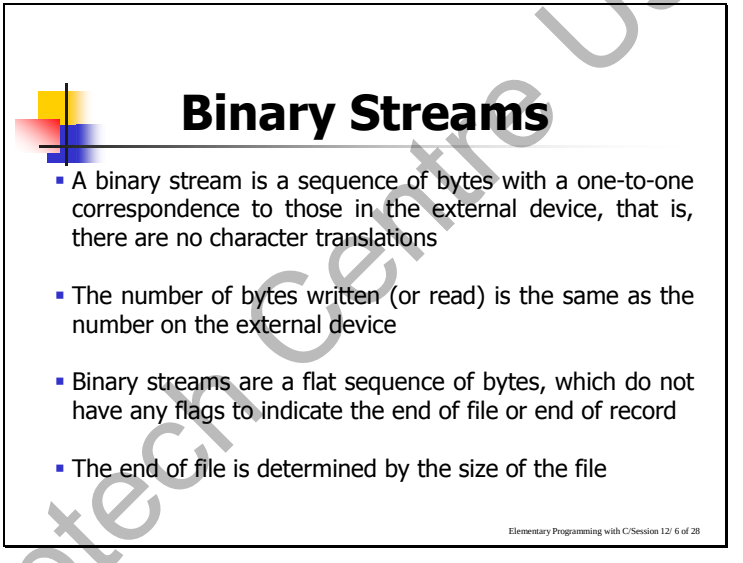


Max Time: 5 Minutes

Subject: The teaching of the session should start with the basic file operations such as save, open, delete, close. You should communicate to the students about the power of C in file handling. Explain clearly that file handling operations are done through streams.

Slides-5 and 6

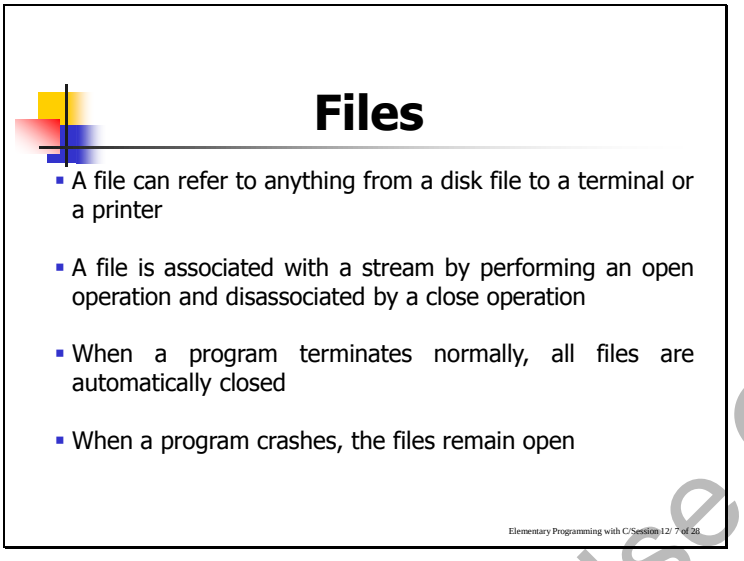


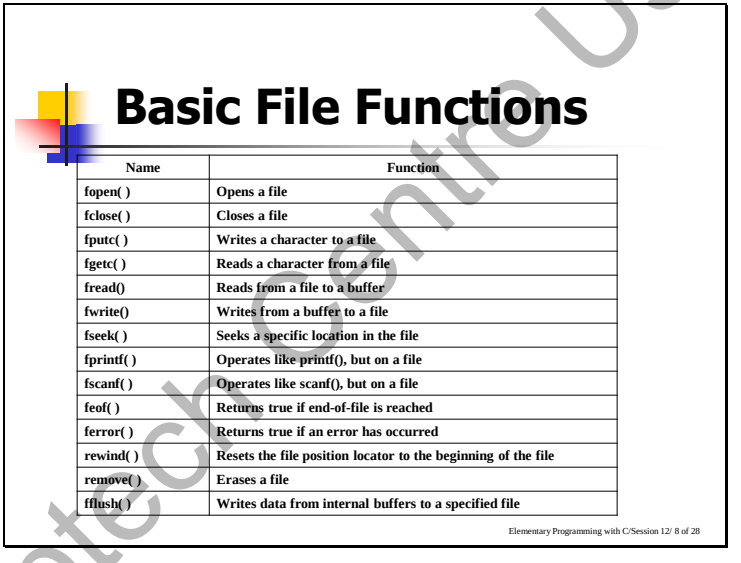


Max Time: 5 Minutes

Subject: Text stream and binary stream should be explained to students and also make them clear the difference between Text streams and Binary streams.

Slides-7 and 8

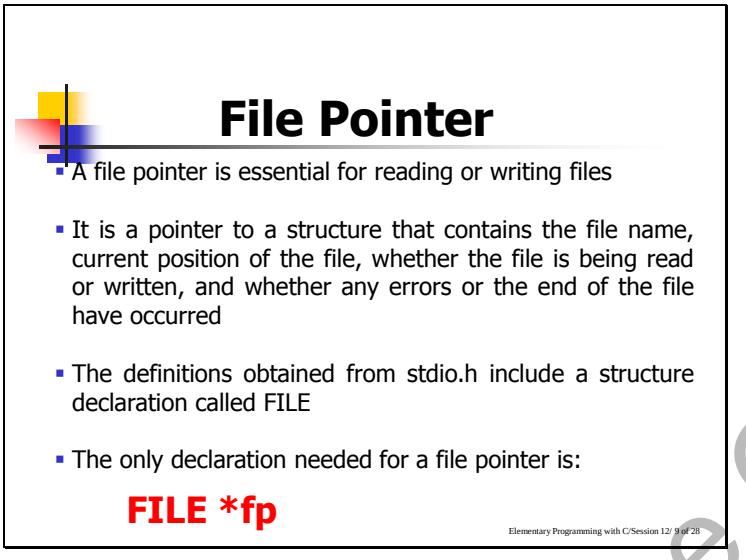




Max Time: 10 Minutes

Subject: You have to explain the file with the example given in the topic. Then the basic file functions have to be taught. Tell the students that the functions are included in the header file **stdio.h.** Also explain clearly about the BOF and EOF concepts.

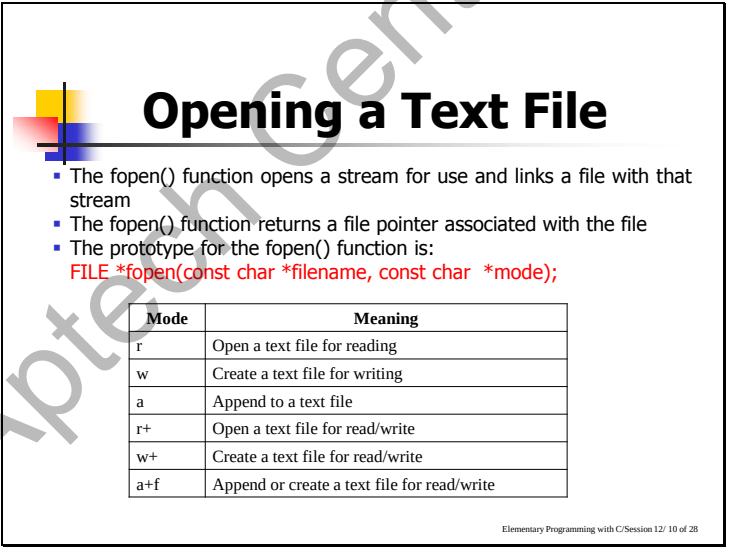
Slide-9

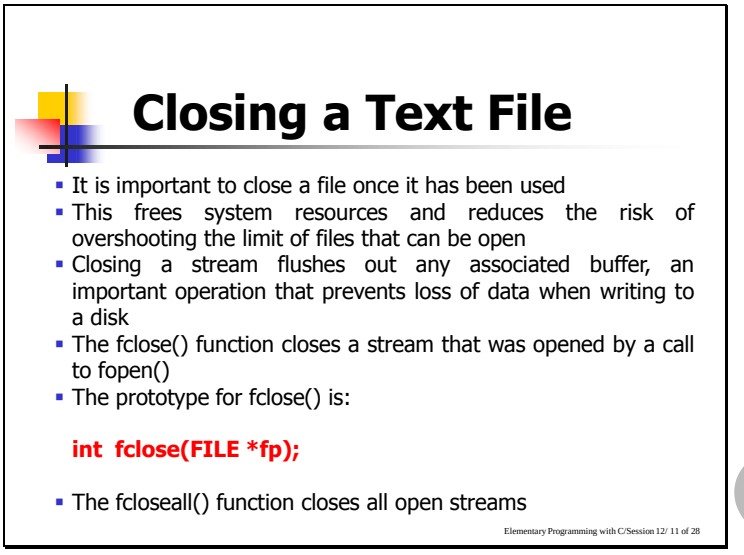


Max Time: 5 Minutes

Subject: Tell the students that the file pointer is essential for reading or writing files. Also explain that FILE is a structure used for file handling operations.

Slides-10 and 11





Max Time: 7 Minutes

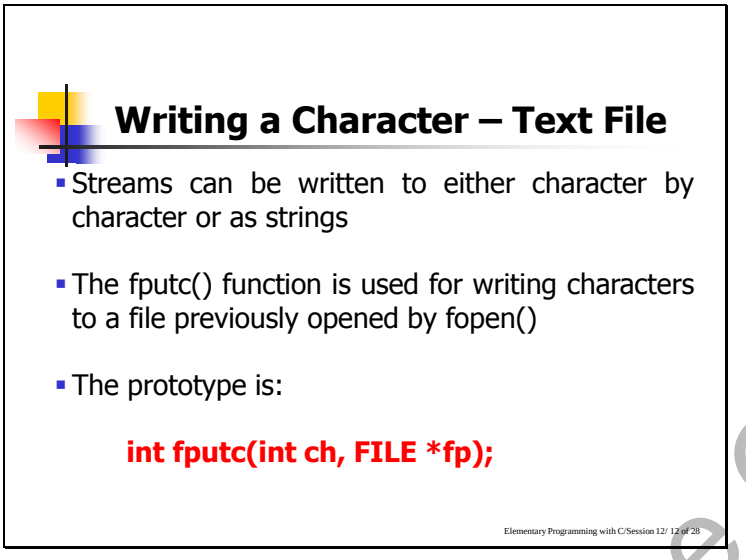
Subject: Introduce the students fopen() and fclose() function . The students should be clear about the use of those functions. Also explain the return values from fopen() and flcose() functions.

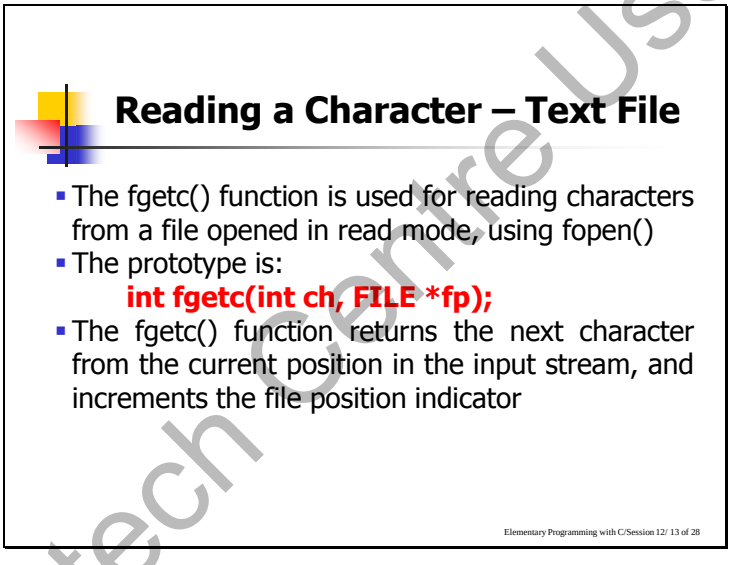
Additional Information:

The program given here opens a file called 'test.txt' and writes the text 'fopen example'.

```
/*fopen example */
#include <stdio.h>
int main ()
{
  FILE *pFile;
  pFile = fopen ("c:\\test.txt","wt");
  if (pFile!=NULL)
  {
    fputs ("fopen example",pFile);
    fclose (pFile);
  }
  return 0;
}
```

Slides-12 and 13





Max Time: 9 Minutes

Subject: Explain clearly the use of fputc() function and fgetc() function. Explain them streams can be written to either character by character or as strings. In this session, we are discussing about writing character to streams. Also explain the syntax of both the functions.

Additional Information:

This example writes inputted text to a file it creates, called 'sentence.txt' and then displays the entered text by reading from it:

```
#include <stdio.h>

int main() {
  FILE *file;
  char sentence[50];
  int i;
```

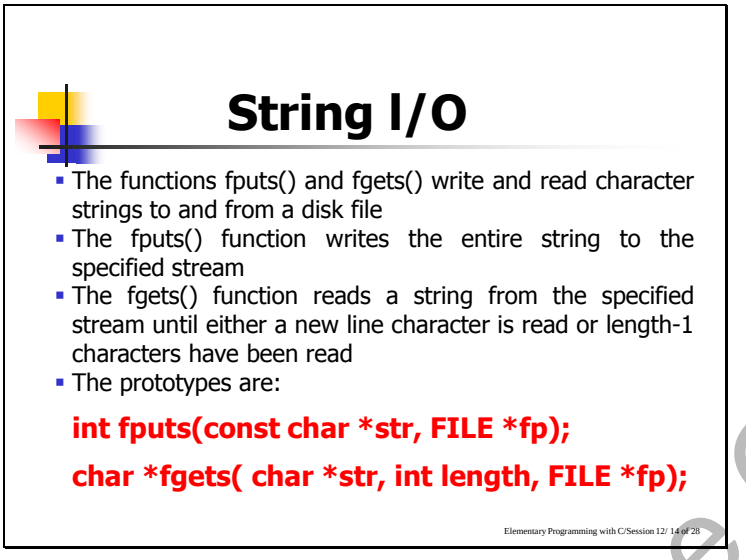
```
  file = fopen("c:\\test.txt", "w+");
  /* we create a file for reading and writing */

  if(file==NULL) {
    printf("Error: can't create file.\n");
    return 1;
  }
  else {
    printf("File created successfully.\n");

    printf("Enter a sentence less than 50 characters: ");
    gets(sentence);

    for(i=0 ; sentence[i] ; i++) {
      fputc(sentence[i], file);
    }

    rewind(file); /* reset the file pointer's position */

    printf("Contents of the file: \n\n");

    while(!feof(file)) {
      printf("%c", fgetc(file));
    }

    printf("\n");
    fclose(file);
    return 0;
  }
}
```

Output depends on what you entered.

First of all, I stored the input sentence in a char array, since we're writing to a file one character at a time it'd be useful to detect for the null character. Recall that the null character, \0, returns 0, so putting sentence[i] in the condition part of the for loop iterates until the null character is met.

Then we call rewind, which takes the file pointer to the beginning of the file, so we can read from it. In the while loop, we print the contents a character at a time, until we reach the end of the file - determined by using the feof function.

Slide-14



Max Time: 5 Minutes

Subject: The mentioned slide can be used to explain the use of fputs() and fgets() function. Tell them that fputs() is used to write a string to the specified stream and fgets() is used to read a string from the specified stream until either a new line character is read or length -1 characters have been read.

Additional Information:

fputs requires two arguments: a char * for the string you'd like to write and a FILE pointer. It returns 0 only if the string is successfully written.

```
#include <stdio.h>

int main() {
  FILE *file;
  char *sentence;

  file = fopen("c:\\test1.txt", "a+");
  /* open or create a file for appending */

  if(file==NULL) {
    printf("Error: can't create file.\n");
    return 1;
  }
  else {
    printf("File created successfully.\n");

    printf("Enter a sentence: ");
    gets(sentence);

    fputs(sentence, file);
    fputs("\n", file);

    rewind(file);

    printf("Contents of the file: \n\n");

    /*while(fgets(sentence, 10, file)!=NULL) {
```

```
      printf("%s", sentence);
    }*/

    while(!feof(file)) {
      printf("%s", fgets(sentence, 10, file));
    }

    printf("\n");
    fclose(file);
    return 0;
  }
}
```

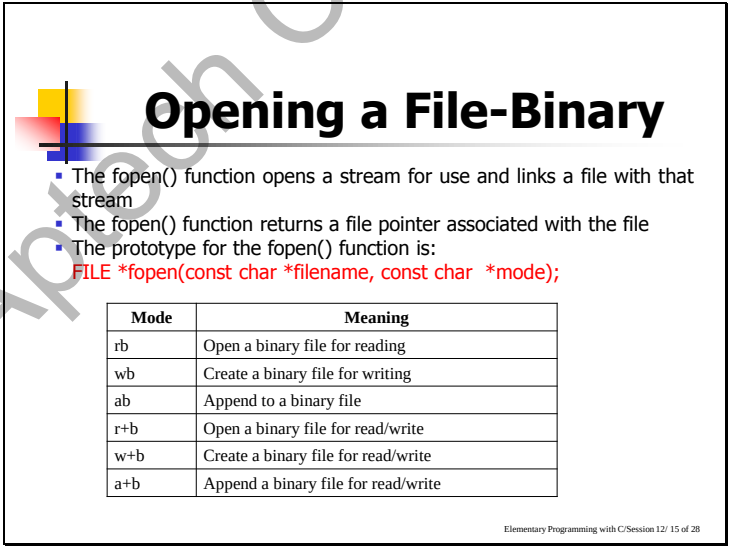
Output depends on what you entered.

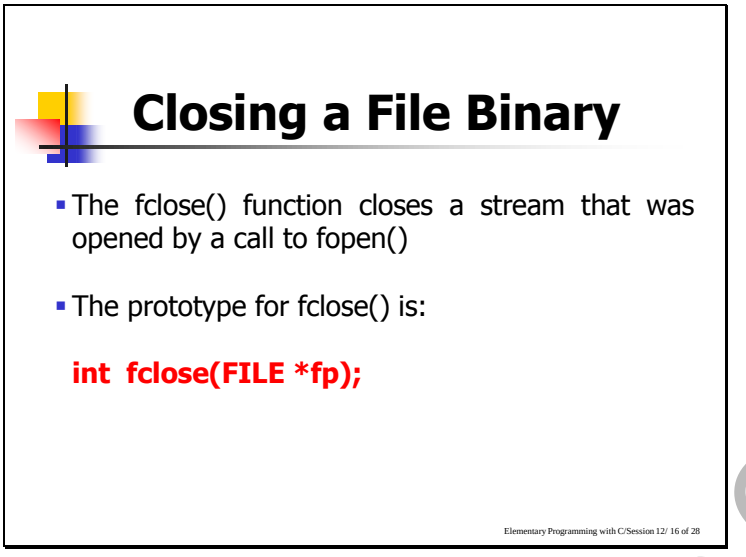
This example demonstrates how to append a string to a file. After entering a sentence, we use fputs to add the sentence to the end of the text file (appending means 'adding to the end'). We use another fputs to append the newline character immediately afterwards - this isn't necessary but it does make the text file look neater.)

Then to read the file from the beginning, use the rewind function.

The while loop simply prints out all the lines of the text file, but if you run the example on your machine, you might get (null) displayed as the last line. This is because we reach the end of the file before the while loop checks the `!feof(file)` condition. I personally prefer the check to see if `fgets` returns a NULL pointer - see the commented while loop.

Slides-15 and 16

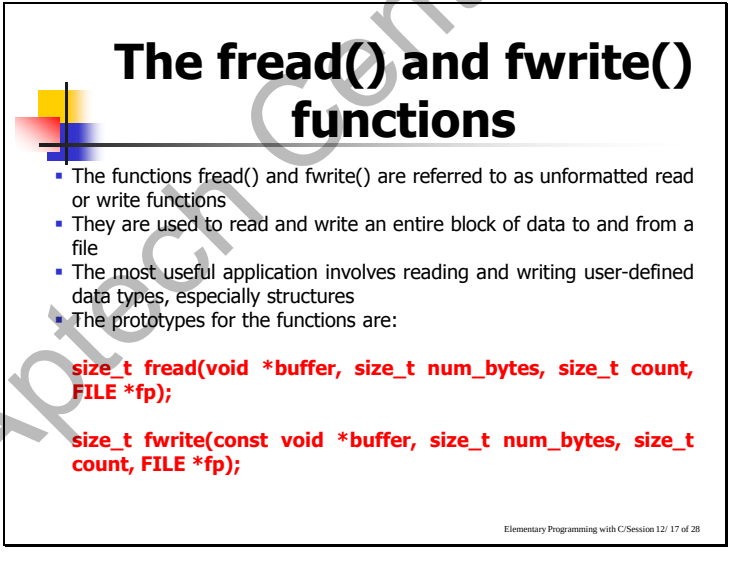




Max Time: 7 Minutes

Subject: Explain the students how to use fopen and fclose functions and their syntax. The different modes used with fopen should be explained in detail.

Slide-17



Max Time: 7 Minutes

Subject: Explain the use of fread() and fwrite() functions. Parameters of the functions should be taught clearly and should be noted that all students are clear with it.

Additional Information:
This example is saved as file.c - it opens file.c for reading and copies the content into file2.c. Afterwards, it'll open file2.c and display the source code on the screen, including the comments! Notice in the example there are two FILE pointers - one for the reading, the other for the writing.

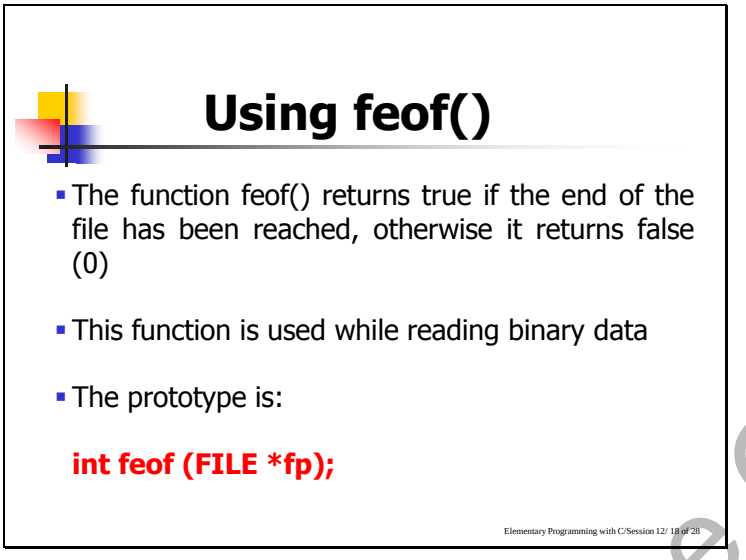
```
#include <stdio.h>
```

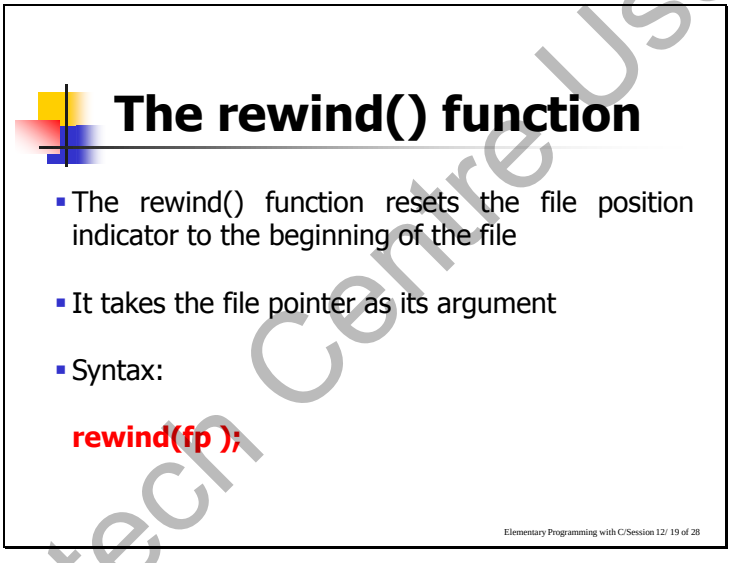
```
int main() {
  FILE *sourceFile;
  FILE *destinationFile;
  char *buffer;
  int n;

  sourceFile = fopen("file.c", "r");
  destinationFile = fopen("file2.c", "w");

  if(sourceFile==NULL) {
    printf("Error: can't access file.c.\n");
    return 1;
  }
  else if(destinationFile==NULL) {
    printf("Error: can't create file for writing.\n");
    return 1;
  }
  else {
    n = fread(buffer, 1, 1000, sourceFile);/*grab all the text */
    fwrite(buffer, 1, n, destinationFile);/*put it in file2.c */
    fclose(sourceFile);
    fclose(destinationFile);

    destinationFile = fopen("file2.c", "r");
    n = fread(buffer, 1, 1000, destinationFile);/*read file2.c*/
    printf("%s", buffer);                    /*display it all */

    fclose(destinationFile);
    return 0;
  }
}
```

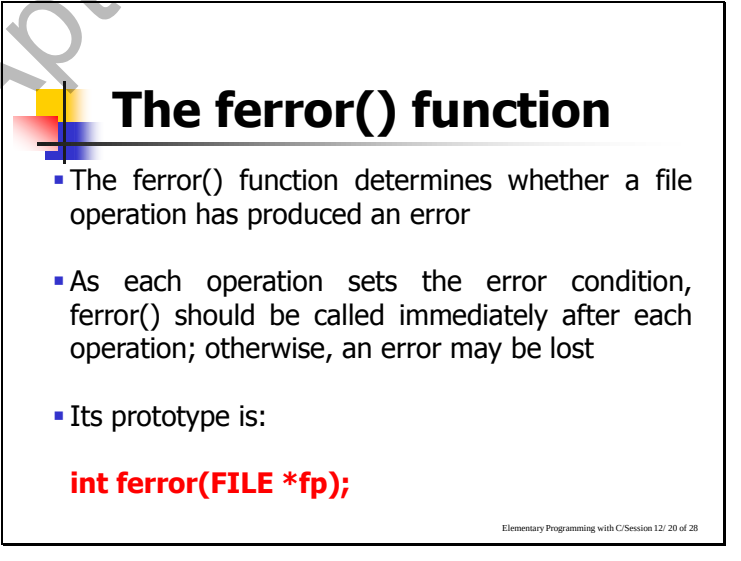
Output is the source code given.

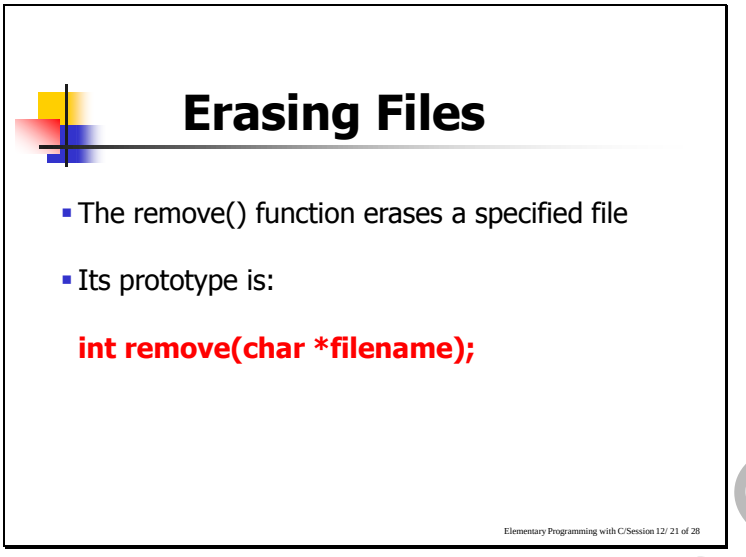
One important thing when making a direct copy from one file to another is to pass in a big enough number in fread so that all the characters are read. Then assign the return value of fread into n so that we know how many characters to write when calling fwrite.

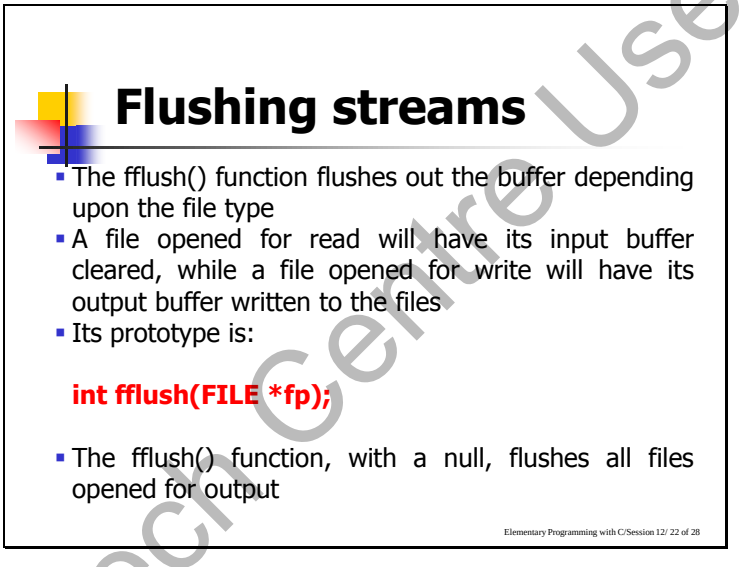
Slides-18, 19, 20, 21, and 22











Max Time: 25 Minutes

Subject: Slide 18 shows the syntax and use of feof() function. Slide 19 demonstrates the rewind() function. Tell the students that rewind() function is used to resets the file position indicator to the beginning of the file. The remaining two slides can be used to explain the ferror() functions and remove() function. Explain that a value '0' when using the remove function() is that the file is removed successfully. Explain the students that flush() function can be used to flush the buffer. Explain briefly what a buffer is.

Additional Information:

To demonstrate the use of fflush, we ask twice the user to input some words in a sentence. Each time the program reads the first word with **scanf** and flushes the rest. The next time the user is prompt the buffer will be cleared so we will be able to obtain the first word of the new sentence.

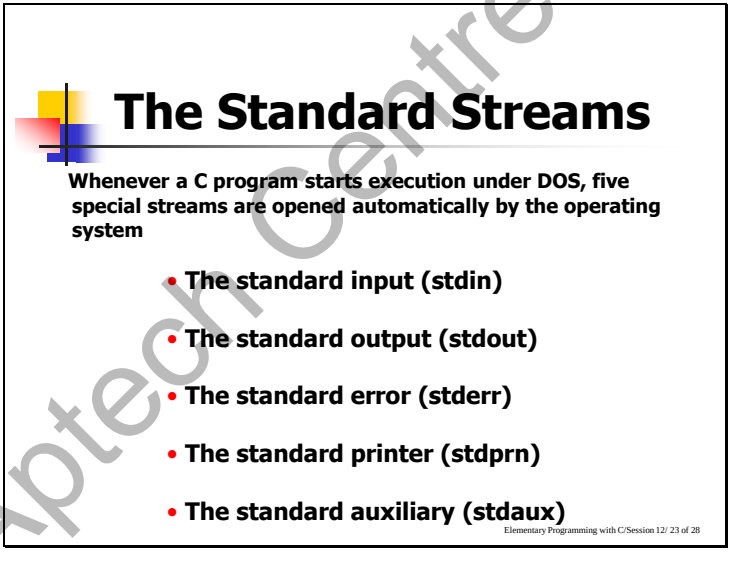
```
/* fflush example */
#include <stdio.h>
int main()
```

```
{
   int n;
   char string[80];
   for ( n=0 ; n<2 ; n++ )
   {
     printf( "Enter some words: " );
     scanf( "%s", string );
     printf( "The first word you entered is: %s\n", string );
     fflush ( stdin );
   }
   return 0;
}
```

Output :

```
Enter some words: Testing this program
The first word you entered is: Testing
Enter some words: It seems to work...
The first word you entered is: It
```

Slides-23, 24, 25, and 26



Current Active Pointer

- A pointer is maintained in the FILE structure to keep track of the position where I/O operations take place
- Whenever a character is read from or written to the stream, the current active pointer (known as curp) is advanced
- The current location of the current active pointer can be found with the help of the ftell() function
- The prototype is:

```
long int ftell(FILE *fp);
```

Elementary Programming with C/Session 12/ 24 of 28

Setting Current Position-1

- The fseek() function repositions the curp by the specified number bytes from the start, the current position or the end of the stream depending upon the position specified in the fseek() function
- The prototype is:

```
int fseek (FILE *fp, long int offset, int origin);
```

Elementary Programming with C/Session 12/ 25 of 28

Setting Current Position-2

- The origin indicates the starting position of the search and has values as follows:

Origin	File Location
SEEK_SET or 0	Beginning of file
SEEK_CUR or 1	Current file pointer position
SEEK_END or 2	End of file

Elementary Programming with C/Session 12/ 26 of 28

Max Time: 23 Minutes

Subject: The first slide in this session specifies the various streams associated with the operating system. Explain the concept of a file pointer. Explain clearly about the various file pointer functions and their use in C programs. The students should be clear about the various parameters used in fseek() function.

Additional Information:

```
/* fseek example */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  pFile = fopen ("c:\\myfile.txt","w");
  fputs ("This is an apple.",pFile);
  fseek (pFile,9,SEEK_SET);
  fputs (" sam",pFile);
  fclose (pFile);
  return 0;
}
```

Example for ftell() function:

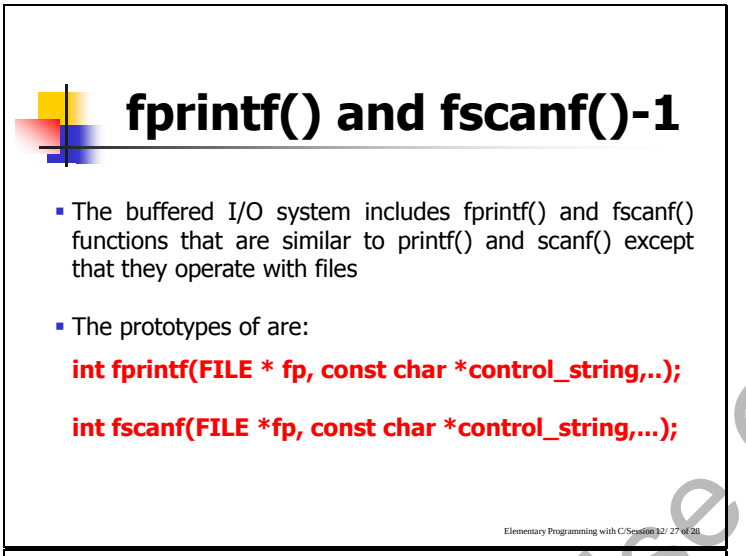
```
/* ftell example : getting size of a file */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  long size;

  pFile = fopen ("myfile.txt","rb");
  if (pFile==NULL) perror ("Error opening file");
  else
  {
    fseek (pFile, 0, SEEK_END);
    size=ftell (pFile);
    fclose (pFile);
    printf ("Size of myfile.txt: %ld bytes.\n",size);
  }
  return 0;
}
```

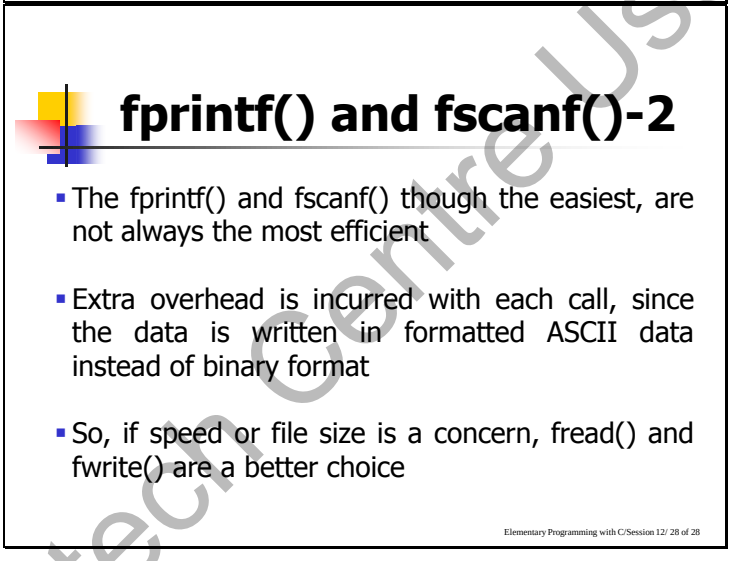
This program opens **example.txt** for reading and calculates its size.

Output:

```
Size of example.txt: 735 bytes
```

Slides-27 and 28





Max Time: 10 Minutes

Subject: This session deals with fprintf() and fscanf() methods. Teach them the difference between printf(), scanf(), and fprintf(), and fscanf(). Tell them the drawbacks when using these functions.

Additional Information:

```
/*
  Read in a text file and output another text file
  that has the contents of the first text file with
  line numbers

*/

#include <stdio.h>

int main()
{

    char buffer[100];// a variable for the strings we read in
    int counter = 1; // used for line numbers

    FILE *inputfile; // a pointer to the input file
    FILE *outputfile;       // a pointer to the output file
    inputfile = fopen("c:\\test1.txt", "r");// open test1.txt

    //Before proceeding, ensure the input file opened properly
    if(inputfile != NULL)
    {
       outputfile = fopen("c:\\test2.txt", "w");//open text2.txt
       // While there are lines in input file, read them in,
       // append a number
       // and output the results to outputfile
       while (fgets(buffer, 100, inputfile))
           {
              fprintf(outputfile, "%d. %s", counter, buffer);
              counter++;
           }

       // close the files
       fclose(inputfile);
       fclose(outputfile);
    }
    else
       printf("Error opening input file.\n");
       return 0;
}
```


Solutions to Check Your Progress

1. Text and Binary
2. False
3. fopen()
4. fputc().

5. True
6. rewind()
7. File pointer
8. binary
9. ftell()

Solutions to Do It Yourself

1.

```
#include<stdio.h>

main()
{
   FILE *fp;
   char ch;
   int i;

   clrscr();
   if((fp=fopen("newfile", "w+"))==NULL)
   {     printf("\nCould not open file");
         exit(1);
   }

   printf("\nEnter the text for the file (type x to
   terminate):\n");
   ch = getche();
   while(ch!='x')
   {     fputc(ch, fp) ;
         ch = getche();
   }

   printf("\n\nPrinting in reverse\n");
   fseek(fp, -1L, 2);
   ch=fgetc(fp);
   putchar(ch);
   while((fseek(fp, -2L, 1))==0)
   {     ch=fgetc(fp);
         putchar(ch);
   }

   fseek(fp, 1L,1);

   getch();
   fclose(fp);
}
```

2.

```
#include<stdio.h>

main()
```

```
{
   FILE *fp1, *fp2;
   char ch;
   int i;

   clrscr();
   if((fp1=fopen("file1", "w+"))==NULL)
   {     printf("\nCould not open file FILE1");
         exit(1);
   }
   if((fp2=fopen("file2", "w+"))==NULL)
   {     printf("\nCould not open FILE2");
         exit(1);
   }

   printf("\nEnter the text for the file (type x to
   terminate):\n");
   ch = getche();
   while(ch!='x')
   {     fputc(ch, fp1) ;
         ch = getche();
   }

   rewind(fp1);
   while((ch=fgetc(fp1))!=EOF)
   {     if(   ch!='a' && ch != 'A' &&
               ch!='e' && ch != 'E' &&
               ch!='i' && ch != 'I' &&
               ch!='o' && ch != 'O' &&
               ch!='u' && ch != 'U' )
               fputc(ch, fp2);
   }

   printf("\n\nPrinting FILE2\n");
   rewind(fp2);
   while((ch=fgetc(fp2))!=EOF)
         putchar(ch);

   getch();
   fclose(fp1);
   fclose(fp2);
}
```

12.3 Post Class Activities for Faculty

You should familiarize yourself with the topics of the next session. You should also explore and identify the Online Varsity accessories and components that are offered with the next session.

Tips: You can also check the Articles/Blogs/Expert Videos uploaded on the OnlineVarsity site to gain additional information related to the topics covered in the next session. You can also connect to online tutors on the OnlineVarsity site to ask queries related to the sessions.