ES 1036 Programming Fundamentals

Unit 1: Introductory Information

Dr. Quazi M. Rahman, Ph.D, P.Eng, SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Outline

- Why ES1036?
- Goals
- Course Related information
- My recommendations

Why ES1036?

- Computer is everywhere and so the software (A.K.A. computer program)
- Most of us will use software for solving Scientific and/or engineering problems efficiently and accurately
- Some will be software engineers
- Others may have to professionally interact with a software engineer
- Some will even experiment at constructing software

Goals

- Be familiar with the introductory aspects of programming language
- Learn how to write computer programs using C++
- Learn how to use the Object-based features of C++
- Learn how to use problem-solving methods as an engineer/scientist
- Get a good grade!

General Learning Objectives

Knowledge Base	**X**	**Individual Work**		**Ethics and Equity**	
Problem Analysis	**X**	**Team Work**		**Economics and Project Management**	
Investigation		**Communication**	**X**	**Life-Long Learning**	
Design		**Professionalism**			
Engineering Tools	**X**	**Impact on Society**			

Course Info: The Reference book

- Note: All the lab and exam questions will be derived from the materials available in the lecture handouts (check the course site on Owl).
- Introduction to Programming with C++, 2/E, Y. Daniel Liang,

  ISBN-10: 0136097200
  ISBN-13: 9780136097204

  Prentice Hall, 2009.
- One can use the older version too:
  - Introduction to Programming with C++ (Brief Version), Y. Daniel Liang,

    ISBN: 0132320495, Prentice Hall, 2007.

Course Website

- OWL course area is an integral part of this course
- Please go to https://owl.uwo.ca and click on the "Documentation" link. Follow the steps to join the OWL Documentation site. It is important to "join" this site to have access to all the documentation you need, to use the new OWL.
- Log in and explore the ES1036 course materials ASAP
- All lab-assignment submissions should be made through the Owl course site unless informed otherwise
- All grades (except final exam and final grade) will be available on the Owl course site.
- All lecture handouts will be available there.
- Discussion board on the Owl course site will be available for discussing any course related issue in an open forum

Components of Your Grade (100%)

- **Class Participation 5%**
- **Lab Quizzes 10%**
- **Lab Assignments 15%**
- **Midterm Exam 20%**
- **Final Examination 50%**
- **In order to pass the course, a student must obtain a passing grade in Lab Assignment average and in the Final Exam and, the total obtained marks after adding all components need to be at least 50%.**
- **Note: passing grade is 50%**

Class Attendance (5%)

- Class attendance will be recorded on an attendance sheet.

Percentage of Attendance	Grade (Out of 5)
1 – 30	1
31 – 55	2
56 – 72	3
73 – 89	4
90 and above	5

Lab Quizzes (10%)

- Every lab will start with a 15-minute long lab quiz.
  - The quiz questions will be taken from the course components used in the lab-assignment
  - You might have been asked to write (on paper) the lab-code, you just submitted.
- *Two worst quiz grades (including zero) will be taken out of the final quiz average.*

Lab Assignment (15%)

- A new lab assignment will be available in each week
- The assignment will be available at least one week prior to your lab day
- Lab Assignment 0: Week of 21/January/2013;
  - Mandatory; Most Important lab; No quiz during this lab
- Lab Assignment 1: Week of 28/January/2013;
  - First quiz on lab 0 materials
  - Tutorial/Discussion with the TAs on lab 1
- Lab Assignment 2: Week of 04/February/2013;
  - Quiz on lab 1 materials
  - Lab 1 has to be uploaded with in next 30 minutes after the quiz on your lab day
  - Tutorial/Discussion with the TAs on lab 2
- In each of the following weeks, a new lab exercise will be available which will follow the same practice as above

Lab Assignment (15%) contd.

- You may ask for help from your instructor, TAs and your course mates
- Once you understand it, solve it by yourself, and demonstrate it to the TA. The TA will ask some questions, or ask you to write some code to evaluate your lab
- Grade will be assigned based on the lab demonstration and working (submitted) code; **zero grade** will be assigned (for the whole lab) for no demonstration even though you submit the correct code.
- You MUST submit your source code (.cpp files) to Owl course site (electronically) before the end of your lab session; **zero grade** will be assigned (for the whole lab) for no electronic submission even though you demonstrate the code to your TA.
- *Two worst lab grades (including zero grades) will be taken out of the lab average*

Checking for Code Similarity

- All submissions will be checked for similarity using a similarity checking software
- This excludes common (provided) code
- Any significant similarity will be treated as an academic offence and in this case all the similar codes (irrespective of source or destination) will be given zero without any warning.
- Similarity check will be done at the end of the semester.
- What can lead to code similarity
  - Copy code from someone else
  - Ask someone to correct your code
  - Someone uses your code

How to Avoid Code Similarity

- Safeguard your code (some students claimed that there codes were stolen and used! → This is not a valid excuse)
- Ask why your code doesn't work – not how to correct it
- Don't look at someone's code while you are typing
- Don't give your code to someone either to help or to get help
- If you fail to follow any one of the above measures, you may end up getting zero due to similarity, and in this case, no second chance will be given.

Scholastic Offences

- **Please read the definition and penalties of academic offense policy in the Western Academic Calendar and in the course outline.**

Midterm Examination (20%)

- Worth 20%
- Midterm date and time:
  - **TBA**
  - **Please check Owl announcement for the midterm exam date.**
- Probable week: Week of 25/February.

Special Permission

- If you can not write any exam (Midterm/Final/Quiz) or attend any lab in case of illness / religious festival / other reason
  - Get permission from Undergraduate Services Office located in SEB 2097; The website address: http://www.eng.uwo.ca/undergraduate/
- If you cannot attend any exam due to an emergency, call the Undergraduate Services Office (Ph. 519-661-2130, FAX. 519-661-3757) ASAP, and get the next set of instructions from that office.
- Check your course outline (last two pages) for detailed info.

Get your VS Software

- You can make your copy of VS2012 by downloading it on your hard drive or on an external drive or on a USB data key, and it's free. Follow the instructions below to get the software
- For all the students in the faculty of engineering, Microsoft software are available for personal use through MSDNAA. As a first year student, you should have received an email from MSDNAA on this issue.
- Follow the link provided on that email;
  - register yourself for the first time
  - Go to Software link
  - VS2012 will be available under 'Popular' Category
  - Download Microsoft Visual Studio 2012 Ultimate (or Premium or Professional) 32-bit (English) - MSDNAA software
- If you have not received that email, or if you're not an engineering student, **please contact the ITG Help Desk at ext. 88112 or enghelp@uwo.ca.** ITG help desk is located in SEB1010a.
- For any IT related issue please check the following link: http://www.eng.uwo.ca/itgroup/first_year.htm
- MAC users: please get the xcode info from the course homepage on Owl

How to Get Help?

- See the instructor during his office hours (**Mondays, Wednesdays and Fridays: 9am – 10.30am in TEB 361.**) Other times by appointment – please email (Qrahman@eng.uwo.ca).
- Get help from the TA(s) during lab session
- Discussion board on the Owl course site : Any unanswered question on the discussion board will be answered within 24 hours during the week days

Course Instructor

- Dr. Quazi M. Rahman, PhD, P.Eng., SMIEEE
  - Office: TEB 361
  - Email: QRAHMAN@eng.uwo.ca
  - Telephone: x81399
- Office hours
  - **Mondays, Wednesdays and Fridays: 9am – 10.30am in TEB 361.** Other times by appointment – please email (Qrahman@eng.uwo.ca)

Teaching Assistant (TA)

- Know your TA by name and email address (will be available on Owl course site before the lab starts).
- You can get help from your TA
- You can reach your TA over email

ES 1036 Programming Fundamentals

Unit 2: Basic Terminology and problem solving steps

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman , Ph.D, P.Eng, SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Outline

- **Basic terminology for computers**
- **Problem-solving methodology for engineering problems**

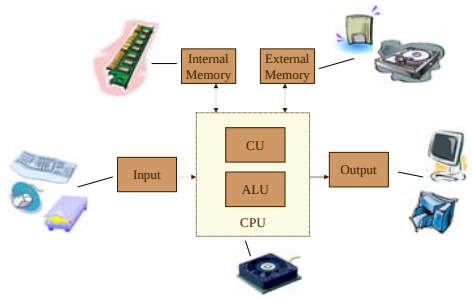
Definitions

- Computer:
  - Programmable machine designed to follow/process instructions at an enormous speed
- Computer Program (Also known as software):
  - A list of instructions directing the computer to perform a task

Definitions, *cont*

- **Hardware:**
  - Ø Refers to the computer equipment, e.g., keyboard, mouse, terminal, hard disk, printer
- **Software:**
  - Ø Refers to the computer programs
  - Ø Example: An operating system is a software that provides a convenient and efficient interface between the user and the hardware
- **Computer system:**
  - Ø The combination of hardware and software

Computer Hardware



Computer Hardware, *cont.*

- CPU (Central Processing Unit): 
  - Control Unit (CU) : Controls the computer resources, also called Processor
  - Arithmetic and Logic Unit (ALU): Performs mathematical operations

Computer Hardware, *cont.*

- Input Devices: 
  - Devices used to input information in to the CPU of the computer
  - Example:
    - keyboard, mouse, scanner, digital camera, disk drive, CD drive

Computer Hardware, *cont.*

- Output Devices : 
  - Devices used to make the computer-processed data/information available at the output
  - Examples:
    - Computer monitor, printer, disk drive, writable CD drive

Computer Hardware, *cont.*

- Internal Memory: 
  - Holds both program instructions and data
  - Volatile – requires constant power to maintain the stored information, erased when computer is turned off
  - Also called primary storage and composed of:
    - Random Access Memory (RAM)
    - Read-Only Memory (ROM)

Computer Hardware, *cont.*

- External Memory : 
  - Non-volatile - data retained even if the computer is turned off
  - It is suitable for long-term storage of information, also called secondary storage
  - Examples: floppy disk, hard drive, flash memory, CD

Review

1) A CPU consists of an ALU, memory, and a processor

   a) True
   ✓ b) False

Review

2) Instructions and data are stored in

   a) the arithmetic logic unit (ALU)
   b) the control unit (processor)
   c) the central processing unit (CPU)
   ✓ d) the memory
   e) the keyboard

Review

3) An operating system is

a) The software that is designed by users

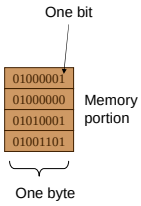
✓ b) A software that provides a convenient and efficient interface between the user and the hardware

c) The set of utilities that allow us to perform common operation

d) A set of software tools

Main Memory Organization

- Main memory is divided into numbered locations each of which contains one **byte** of information
  - Byte is a sequence of **8 bits**
  - Bit stands for **b**inary digi**t** which can have either **0** (OFF, FALSE) or **1** (ON, TRUE) value.
- The location number associated with a byte is called the **address**
- Each location has a unique address
  - *No two memory locations can have the same address*



How Data is Stored?

- Computers use zeros and ones because digital devices have two stable states, which are referred to as zero and one by convention.
- The programmers need not to be concerned about the encoding and decoding of data, which is performed automatically by the system based on the encoding scheme.
- The encoding scheme varies. For example, character 'J' is represented by 01001010 in one byte.
- If computer needs to store a large number that cannot fit into a single byte, it uses more than one adjacent bytes.

Computer Languages

- Computers do not understand human languages, so we need to use computer languages to communicate with computers
- We tell a computer what to do through programs
- Programs are written using programming languages

Computer Languages, *cont.*

- **Machine Language:**
  - A computer hardware understands this language only
  - Used for communication with computer hardware directly
  - Is written using two symbols represented by **0** and **1**
  - Also called **low-level** language, or **binary language**
  - Machine dependable
  - Example:

```
0010 0000 0000 0100
1101 1010 1001 1010
1000 0000 0000 0101
```


Computer Languages, *cont.*

- **Assembly Language:**
  - This language assigns short names to commands (e.g. ADD, MOV) instead of binary instructions as done for machine language
  - Assembly language is easier to understand when compared to machine language
  - Must be translated to machine language
  - The *assembler*, a computer program, translates the assembly language in to machine language.
  - Still processor dependent
  - Example:

```
ADD R1, R2, R3
```

Computer Languages, *cont.*

- **High-level Language:**
  - Uses English-like command and instruction
  - Easier than machine and assembly languages
  - Processor independent
  - Must be translated (compiled) into machine language. This translator is known as compiler.
  - Example: the following statement computes the area of a circle with radius 5

```
area = 5 * 5 * 3.1415;
```


Popular High-Level Languages

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslation)
- BASIC (Beginner All-purpose Symbolic Instructional Code)
- Pascal (named after Blaise Pascal)
- Ada (named after Ada Lovelace)
- C language
- Visual Basic (Basic-like visual language developed by Microsoft)
- Java
- C++ (an object-oriented language, We'll learn the fundamental principles of this language in this course)

Problem-solving methodology for engineering problems

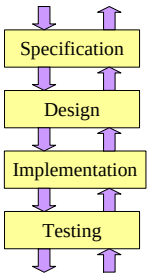
Engineering Problem Solving

- Problem Example:

  Compute the square of a number ranged from 0 to 2,000,000

Engineering Problem Solving

- **Specification**
- **Design**
- **Implementation**
- **Testing**



Specification

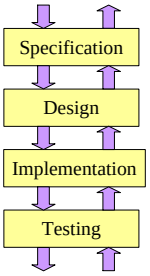
- Developed in response to the "requirements"
- Precise statement of what the system will do
- Include
  - Functional specifications
  - Input/output specifications
  - Performance specifications
- Most important, hardest, most neglected

Specification - example

- Requirements
  - Program to accept any number ranged from 0 to 2,000,000 and calculate and display the square of this number
- **Specifications**
  1. Request user to enter any number
  2. Read the entered number
  3. Check the number, if less than 0 or greater than 2,000,000 display "Invalid" and terminate, otherwise continue
  4. Calculate the square of the number
  5. Display the result
  6. Terminate

Engineering Problem Solving

- **Specification**
- **Design**
- **Implementation**
- **Testing**

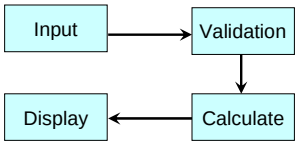


Design

- Architecture
  - Decompose the system into components
- Interfaces
  - How the components talk to each other
- Detailed design
  - How components are designed
  - Each component can be another system

Design - example

- Decompose the system into components
- How the components talk to each other

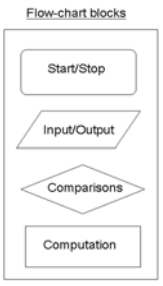


Design, *cont.*

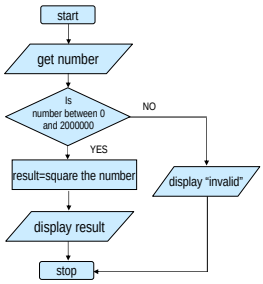
- Algorithm, (after Al Kho-war-iz-mi a 9th century Persian mathematician)
  - An ordered sequence of well-defined instructions that gives an initial state, performs some task and halts in finite time
- Algorithms can be illustrated by one of the following ways:
  - Pseudo-code
    - Kind of structured English for describing algorithms
  - Flow-Charts
    - Often used to represent algorithms graphically

Flow-Charts in Design

- Flowchart uses specific diagrams connected by arrows to represent the control-flow of the algorithm

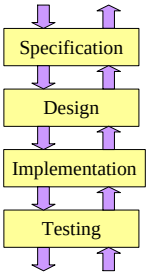


Flow-Charts in Design - example



Engineering Problem Solving

- **Specification**
- **Design**
- **Implementation**
- **Testing**



Implementation

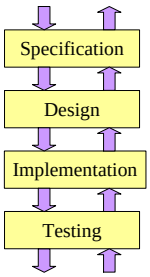
- Easiest part
- There shouldn't be any surprises
- Pick the platform
  - PC, Mac, Palm
- Pick the language
  - C++, Java, C, Ada, C#
- Pick the environment
  - MS Visual Studio, Borland C++ Builder. etc

Implementation - example

```
#include <iostream>
using namespace std;
int main()
{
      double anyNumber;
      cout << "Enter a number:"<<endl;
      cin >> anyNumber;
      if (anyNumber >= 0 && anyNumber < 2000000)
            cout << anyNumber * anyNumber <<endl;
      else
            cout << "Invalid\n";
}
```


Engineering Problem Solving

- **Specification**
- **Design**
- **Implementation**
- **Testing**



Testing

- Second most important part of the process
- Ensures that the implementation fulfils the specification
- A test plan can be generated directly from specifications
- An item in the test plan contains details about:
  - Steps that must be followed,
  - Inputs that must be given and
  - Outputs that must be observed

Testing - example

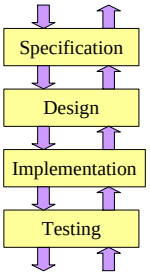
1. Start program
2. Enter '-1'. Observe "Invalid"
3. Enter 'abc'. Observe "Invalid".
4. Enter '16'. Observe "256".
5. Program must exit

```
cout << "Enter a number:";
cin >> x;
if (x >= 0 && x < 2000000)
    cout << x*x <<endl;
else
    cout << "Invalid";
```

Repeat these steps number of times and watch the program outputs

Engineering Problem Solving

- **Specification**
- **Design**
- **Implementation**
- **Testing**



Review

7) A computer program is the implementation of an algorithm

a) True
b) False

Review

8) An algorithm refers to

a) A step-by-step solution to solve a specific problem
b) The collection of instructions that the computer can understand
c) A code that allows us to type in text materials
d) Stepwise refinement
e) A set of math equations to derive the problem solution

Review

9) Assembly language is a high level language

a) True
b) False

Review

10) A high level language is CPU specific (processor/machine dependent)

a) True
b) False

Review

11) Compiling a C++ program generates machine code for a particular CPU

a) True
b) False

Review

12) Writing code is done during

a) Specification
b) Design
c) Implementation
d) Testing

Review

13) In which problem solving steps, syntactical errors are corrected?

a) Specification
b) Design
c) Implementation
d) Testing

Review

14) If you change your code, you must compile it for the change to be effective

a) True
b) False

Review

15) _____ is the physical aspect of the computer that can be seen.

a) Hardware
b) Software
c) Operating system
d) Application program

Review

16) _____ is the brain of a computer.

a) Hardware
b) CPU
c) Memory
d) Disk

Review

17) Why do computers use zeros and ones?

a) because combinations of zeros and ones can represent any numbers and characters.
b) because digital devices have two stable states and those states can be well represented by 0 and 1.
c) because binary numbers are simplest.
d) because binary numbers are the bases upon which all other number systems are built.

Review

18) One byte has _____ bits.

a) 4
b) 8
c) 12
d) 16

Review

19) _____ translates high-level language program into machine language program.

a) An assembler
b) A compiler
c) CPU
d) The operating system

Review

20) _____ is a program that runs on a computer to manage and control a computer's activities and provides interface between the user and the computer hardware.

a) Operating system
b) C++
c) Modem
d) Interpreter
e) Compiler

Homework: Case Study

- **Problem** Your summer surveying job requires you to study some maps that give distances in kilometers and some that use miles. You and your coworkers prefer to deal in metric measurements. Write a program that performs the necessary conversion.

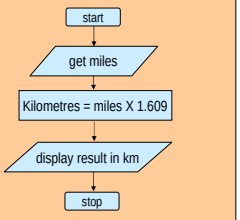
Specifications

- **Requirements:**
  - convert distance measurements (given in miles) to kilometers.
- **Specifications**
  - Problem Input
    miles *distance in miles*
  - Problem Output
    kms *the distance in kilometers*
  - Relevant Formula
    1 mile = 1.609 kilometers

Design

- Formulate the algorithm that solves the problem.
- **Algorithm**
  1. Get the distance in miles.
  2. Convert the distance to kilometers.
  3. Display the distance in kilometers.
- Algorithm Refinement
  2.1 distance in kilometers = 1.609 X the distance in miles

- Flowchart:



Implementation

```
#include <iostream>
using namespace std;
int main( )
{
   const float KM_PER_MILE = 1.609;
   float miles, kms;
   cout << "Enter the distance in miles: ";
   cin >> miles;
   kms = KM_PER_MILE * miles;
   cout << "The distance in kilometers is " <<
   kms << endl;
   return 0;
}
```


Testing

- Test with input data for which one can easily determine the expected results
- E.g.
  10 miles should convert to 16.09 kilometers

Answers to the review-questions

7. A
8. A
9. B
10. B
11. A
12. C
13. C
14. A
15. A
16. B
17. B
18. B
19. B
20. A

ES1036: Programming Fundamentals

Unit 3: C++ Basics

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman, Ph.D, P.Eng., SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Outline

- Program Structure
- Constants and Variables
- C++ Operators
- Standard Input and Output
- Basics Math Functions
- Arithmetic conversions
- Programming Style and Documentation

Program Structure

```
/*-------------------------------------------------------------*/
/* My first C++ program                                        */
/* This program displays a message "Hello world!"              */
/* and terminate                                               */
/*-------------------------------------------------------------*/
#include <iostream>
using namespace std;
int main()
{
      //  Display a message "Hello World!"
      cout << "Hello world!" << endl;
      //  Exit program
      return 0; //  This statement is optional
}
```

Comments

Using directive

main function: The program entry point

C++ statements

Pre-processor directive

Set of curly brackets, enclosing the body of the function

Comments in C++

- Comments allow programmers to put some descriptions inside the code
- Comments are not programming statements and thus are ignored by the complier
- Two types: (check the code in the last slide)
  - *Line comment*: is preceded by two slashes `//` on a line
  - *Paragraph comment*: is enclosed between `/*` and `*/` on one or several lines
- When the compiler see `//`, it ignores all text after `//` on the same line
- When it sees `/*`, it looks for the next `*/` and ignore any text between `/*` and `*/`

Preprocessor Directive

- The line begins with '#' character is known as preprocessor directive.
- Preprocessor is a program that modifies the source code by one of the two following ways:
  - by adding other files and/or
  - by performing various text replacements in the existing code
- It executes automatically before the compilation starts.
  - Examples:
    - **#include <iostream>** Tells the complier to include the **`iostream`** library in this program, which is needed to support standard input and output. (`adding other files`)
    - **#define PI 3.141593** defines PI with the constant value 3.141593 and then replaces all instances of PI in a program with the value 3.141593. (text replacements)
  - Compilation follows immediately after the preprocessing, so none can access the modifications made by the preprocessor

C++ Statements

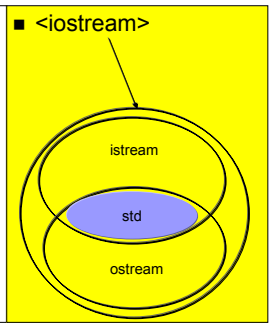
- A statement in C++ is a C++ instruction that ends with a semicolon (might take one or more lines).
- Different types of statement include:
  - Declaration statements
  - Assignment statements
  - Compound statements (also known as block statements)
    - One or more C++ statements enclosed between curly braces '{' and '}'
  - Selection statements

Using directive

- **`using namespace std;`**
  It tells the complier to use the functions from the standard namespace `std`
- Namespaces allow to group entities like variables, objects, functions and classes, under a name. Above, **`std`** is a namespace.
- All the files in the C++ standard library declare all of its entities within the `std` namespace.
- For more info on namespaces see http://www.cplusplus.com/doc/tutorial/namespaces.html

FYI: Graphical representation of <iostream> and std

- <iostream> library
  - istream class
  - ostream class
- Std is a namespace (section) in <iostream> library that contains some preprocessed io stuffs from both istream and ostream classes



The Function `main()`

- Every C++ program executes its instructions from the function called *main*. It is the entry point of the program
- The main function header has the form:
  ```
  int main() or
  int main(void) /*void is a keyword*/
  ```
- All statements in the main function (and in all other functions) must be enclosed in a statement-block that start with left curly brace '**{**' and ends with right curly brace '**}**'
- Every statement in the block must end with a semicolon '**;**'

Note: C++ source programs are case sensitive. It would be wrong, (for example) to replace **`main`** in the program with **`Main`**

The statement `return`

- The statement
  ```
  return 0;
  ```
  is placed at the end of every main function to exit the program
- The value **`0`** indicates that the program has terminated successfully
- This statement is optional but it's a good practice to use this statement in the program

The statement `cout`

- The statement **`cout`** (an ostream object) displays a message and/or data to the standard output (console or monitor)
- **`cout`** stands for console output which has the general form as:
  - `cout << expression << expression;`
- Note: An expression can be any C++ expression such as, string constant (*one or more characters enclosed by double-quotation marks*), character constant (*single character enclosed by single-quotation marks*), identifier (*variable name*), formula (*arithmetic and/or logical expressions*) or function call.
- The **<<** operator, referred to as stream (array of characters) insertion operator, sends a processed stream (string, character, function output etc) to a console.
- A string must be enclosed in double quotation marks
- Example:
  ```
  cout << "Hello world!" << endl;
  ```
  This statement first outputs the string, ***Hello world!***, to the console and then sends **`endl`** (stands for end line) to output a new blank line in the console. The **`endl`** statement takes the cursor to a new line.

Converting C++ programs in to machine language

- Text Editor
- Preprocessor
- Compiler
- Linker
- Loader

Converting C++ in to machine language

- Text Editor
  - The program (the set of instructions), known as source code, is written using the text editor.
  - The source code (human readable instructions) is saved on to the secondary storage (External memory) with an extension **'.cpp'** to let the compiler know that it is written in C++ language.
  - The source code needs to be grammatically (C++ grammar) correct

Converting C++ in to machine language

- Preprocessor is a program that modifies the source code by one of the two following ways:
  - by adding other files and/or
  - by performing various text replacements in the existing code
- It executes automatically before the compilation starts.
  - Examples:
    - **#include <iostream>** Tells the complier to include the **`iostream`** library in this program, which is needed to support standard input and output. (adding other files)
    - **#define PI 3.141593** defines PI with the constant value 3.141593 and then replaces all instances of PI in a program with the value 3.141593. (text replacements)
  - Compilation follows immediately after the preprocessing, so none can access the modifications made by the preprocessor

Converting C++ in to machine language

- Compiler translates the source code into an object code that contains machine readable instructions
  - The object code is saved on to the disk by the compiler with an extension '.obj' along with the same source-code name
  - This file is basically a binary file

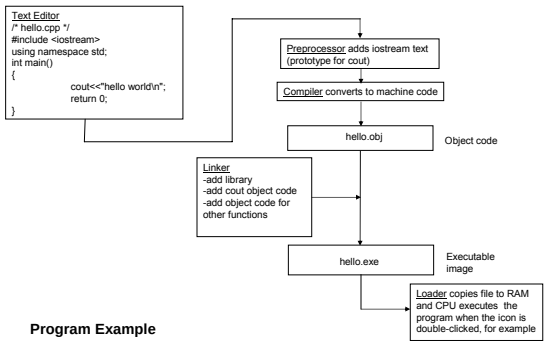
Converting C++ in to machine language

- Linker:
  - scans the library (e.g., <iostream>), the preprocessor directives have included
  - selects the needed function (precompiled) and
  - upon linking them into the object file, produces an executable file with extension **'.exe'** (for UNIX based system it is '.out') and stores it on to the disk.
- Libraries are a collection of functions/objects
- Examples:
  - The linker adds the precompiled (in binary form) codes for cin, cout, etc from <iostream> library.
  - If one separates his/her program into more than one source files, the object code for each source file is added by the linker too.

Converting C++ in to machine language

- Loader: The loader places the executable file on to the internal memory (RAM), from where the CPU executes the program, instruction by instruction

Converting C++ in to machine language



Program Example

Converting C++ in to machine language

- Integrated Development Environment (IDE) program:
  - It manages all the previously discussed steps and combines an editor, compiler, linker and debugger into a single development environment.
  - Advantage: all the pieces are designed to work together; for example, if the compiler detects an error, the system is switched to the editor with the file positioned at the problem line.
- Debugger: The debugger locates the problem in the program during the compilation time (see the webct home page on the debugger)

Converting C++ in to machine language: The errors

- **Syntax errors**: (also called parse errors, compilation error) reported by the compiler once any syntactical error is made. The program will not compile with erroneous syntax.
- **Runtime errors (Execution error) occur when the programming environment detects an operation that is impossible to carry out during program execution.**
- **Logical errors (**or **program bugs) occur when a program doesn't perform the way it is expected to perform.**
- **Linker errors:** reported by the linker

Review

19) Computer can execute the code in _____.

a) machine language
b) assembly language
c) high-level language
d) none of the above

Review

20) The extension of a C++ object file is

a) .java
b) .obj
c) .class
d) .exe

Review

21) The extension of a C++ source code file is

a) .java
b) .obj
c) .class
d) .exe
e) .cpp

C++ Variables (Objects)

- In the programming environment we need to reserve some memory spaces to process any data or information
- This reservations are done using variable (and/or constant) names
- Variables are boxes or placeholders in the memory that can hold things
- Each variable (box) has to be identified by a name and then it has to be declared before using it. The name of a variable (or constant, function etc) is called an "identifier"
- Size of the variable (box) depends on the "type" of things we are planning to store there
- We have to tell the compiler in advance ("declare"),
  - Names of each of the variables (boxes) we want
  - The type of things that will be put in each variable (box)

Identifier characteristics

- An identifier (name of a variable, constant, function, class, struct, namespace etc) may consist of
  - Any letters (A to Z, both capital and small),
  - Any digits (0 to 9), and
  - Two special characters : underscore (_) and dollar ($).
- An identifier cannot start with a digit
- An identifier cannot be a reserved word
- An identifier can be of any length, but your C++ compiler may impose some restriction. Use identifiers of 31 characters or fewer to ensure portability

For example: `radius, A, anyName, _number, SurfaceArea` are legal identifiers, but `2A, yes&No,` and `d+4` are not

Reserved Words

Definition: The words which are reserved for a particular programming language (grammar) are known as reserved words; also known as keywords. Each of these words has a particular meaning to the programming language.

Keywords/Reserved words in C++:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Variable Declaration

- To use a variable, one has to declare it by writing the name of the variable as well as the data type it represents. This is called variable declaration
- Declaring a variable tells the compiler to allocate enough memory space to hold a value of this data type and to associate the identifier with this location
- Here is the syntax (grammar) for declaring a variable:

```
datatype variableName;
```

Example of declaration statements:

```
int x;              //Declare x to be an integer variable
double radius;      //Declare radius to be a double variable
double SurfaceArea; //Declare SurfaceArea to be a double variable
char a;             //Declare a to be a character variable
bool xy;            //Declare xy to be a boolean variable
```

- *Note: It's a good practice to pick up a meaningful name for a variable*

Some built-in Data Types

Data Type	Range of Values	Size
`char`	-128 to 127	1 byte
`int`	-2,147,483,648 to 2,147,483,647	4 bytes
`float`	6 Digits of Precision $\pm 3.402823 \times 10^{\pm 38}$	4 bytes
`double`	15 Digits of Precision $\pm 1.797693 \times 10^{\pm 308}$	8 bytes
`bool`	**true** or **false**	1 byte

Mixing different data types

- We're not supposed to mix numerical data types (int, float, double etc) with the non-numerical (char, string etc) ones
- Try to be consistent in using the data types e.g., `result = (a + b)*2;`
  - Try to use same data type for the variables: result, a and b
- Exception
  - It is OK to mix numerical types i.e. `int, float, double`
    - Example: `double x = 10.25; float y = 12; int z = 4; double r = x + y + z;`
  - Be careful not to loose precision (example?)

Where to Declare a variable?

- Immediately prior to use

```
int main()
{
 cout<<"Add two
     numbers:"<<endl;
 int sum(0), a=4, b=5;
 sum = a + b;
 cout<<sum<<endl;
 return 0;
}
```

- At the beginning

```
 int main()
{
 int sum(0), a=4, b=5;
 cout<<"Add two
    numbers:"<<endl;
 sum = a + b;
 cout<<sum<<endl;
 return 0;
}
```

Assigning a Value to a Variable

- After a variable is declared, one can assign a value to it; this is called variable initialization.
- It's a good and safe practice to initialize a variable (or object) when it is declared
- *Note: If any uninitialized variable is used in any expression, run time error will be generated.* In Class discussion with example?
- Variable value can also be assigned using an assignment statement
- The syntax for assignment statement is:

```
variable = expression;
```

- An *expression* represents a computation involving
  - values,
  - variables, and
  - operators that altogether evaluates to a value
- The equal sign (=) is known as the assignment operator

Examples on variable declaration and initialization

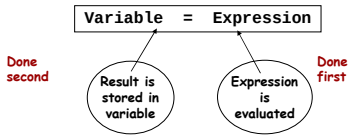
```
// declare x as an integer variable and assign 1 to it
int x =1; //method 1: declaration and initialization
int x(1); //method 2: declaration and initialization
// declare radius as a double type data variable and assign 1 to it
double radius = 1.0; //method 1
double radius = 1; //method 2
double radius (1.0); //method 3
double radius (1); //method 4
/*each of the above statements can also be written using two statements as follows:*/
double radius; //variable declaration
radius = 1; // assigning a value to a declared variable
//declare a variable called SurfaceArea
double SurfaceArea;
//compute surface area by using the declared variable radius
SurfaceArea = 4 * radius * radius * 3.14159; //an assignment statement
//Declare a to be a character variable
char a;
// assign the letter K to the character variable a
a = 'K';
```


Notes on literals

- A *literal* is a constant value that appears directly in the program.
- This is also referred as *hard-coding*
- Examples: in the following statements, 34, 1,000,000, 5.5 and 'A' are literals:

```
int i = 34;
long x = 1000000;
double d = 5.5;
char k = 'A';
```


Assignment Expression



- A variable can appear in both sides of the assignment operator
- For example: `int x = 10; x = x + 1;`
  - First the right hand side term x + 1 is evaluated and then the result is assigned to the variable x on the left hand side
- In the example above, what will be the value of x after the second statement?

Review: C++ Statements

- One C++ instruction that ends with a semicolon
- Can use more than one line
- Examples:
  - Declaration statements
  - Assignment statements (assignment operator '=')

```
int sum(0), a(5),
 b(10);/*declaration
 and
 initialization*/
sum = a + b;

float temp_c(0);
float temp_f(78);
temp_c = (temp_f -
 32)*5/9;
```


Review

22) Name of a variable (or object) is used to:

a) Tell the computer how much memory this variable needs
b) Distinguish one variable from another
c) Identify the type of things that can be written to it
d) Both (a) and (b)

Review

23) Type of a variable (or object) is used to:

a) Tell the computer how much memory this variable needs
b) Identify the type of things that can be written to it
c) Distinguish one variable from another
d) Both (a) and (b)

Review

24) What type of a statement is

```
float w_lb, w_kg;
```

a) A pre-processor directive
b) A selection statement
c) A declaration statement
d) An assignment statement

Review

25) What type of a statement is

```
w_kg = w_lb * 0.454;
```

a) A pre-processor directive
b) A selection statement
c) A declaration statement
d) An assignment statement

C++ Compound/block Statements

- Use **{** and **}** to group any number of statements
- Example: The following block is treated as one statement

```
{
  cout << "Hello";
  return 0;
}
```


Input data with cin

- **cin**
  - is an istream object
  - Inputs data from the standard input (console or keyboard) on to a declared variable
  - uses the **>>** (input) operator, also known as *data extraction operator*
  - **NEVER use endl within cin statement**
- General Form:
  - `cin >> var1>>var2;`
- In above the first value entered from the keyboard will be assigned to var1 while the second one will be assigned to var2
  - These two (or more) variables must be separated by white-space characters; white space characters are space-bar, tab, enter key entries
- The data typed at the keyboard is not actually assigned to the respective variable until the 'Enter' key is pressed.
- A cout statement usually precedes a cin statement to prompt the user to enter data; see the examples given afterward

Named Constants

- A named constant is a location in memory (a box) in which a data value cannot be changed
- Modifying the value of a named constant will result in a syntax error
- So, they must be given a value in its declaration otherwise it will result in a syntax error
- As a matter of style, names are all caps (**not mandatory**)
- Example of Valid constant declarations:

```
const double PI =  3.14159;
const char BLANK = ' ';
const int VOTING_AGE(18);

const float MAX_HOURS(40.0);
MAX_HOURS=MAX_HOURS+1;//Error; why?
```

- Note: Named constants can also be defined using the pre-processor directive which follow the similar grammatical rules as above. E.g.,

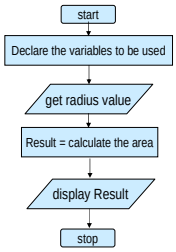
```
#define aaa 2.3
```

No semicolon after the above statement. Also no data type can be used for the define-directive

Example problem using constants and variables

- **Problem**
  Find the surface area a sphere. The radius value of this sphere will be received from the user as an input.

Flow chart



The outline (we can call this 'algorithm')

```
/*************************************************
*  This program finds the surface area           *
*  of a sphere with a given radius               *
*************************************************/
#include <iostream>
using namespace std;
int main()
{
// Step 1. Input the value of the radius of the sphere.

// Step 2. Compute the surface area of the sphere.

// Step 3. Display the values of the radius and
//         the surface area of the sphere.

// exit
return 0;
}
```


The Outline (more specific: approach 1)

```
/*************************************************
*  This program finds the surface area           *
*  of a sphere with a given radius               *
*************************************************/
#include <iostream>
using namespace std;
int main()
{
/*Step 1. Input the value of the radius of the sphere.
        - declare a variable to store the radius
        - ask the user to input the value
        - input the value from the keyboard into the
          declared variable
Step 2. Compute the surface area of the sphere.
        - declare a variable to store the area
        - compute the area and store the result into the
          declared variable
Step 3. Display the values of the radius and
         the surface area of the sphere.

Exit*/
return 0;
}
```


The Outline (more specific: approach 2)

```
/*************************************************
*  This program finds the surface area           *
*  of a sphere with a given radius               *
*************************************************/
#include <iostream>
using namespace std;
int main()
{
/*Prep: Variable and constant declarations
        - declare variables to store the radius and area values.
          Also, declare constant expression, if needed.
Step 1. Input the value of the radius of the sphere.
        - ask the user to input the value
        - input the value from the keyboard into the
          declared variable for radius
Step 2. Compute the surface area of the sphere.
        - compute the area and store the result into the
          declared variable for area
Step 3. Display the values of the radius and
         the surface area of the sphere.

Exit*/
return 0;
}
```

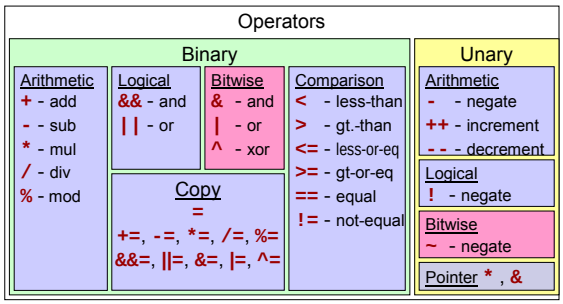


```
/*************************************************
*  This program finds the surface area           *
*  of a sphere with a given radius (approach 2)  *
*************************************************/
#include <iostream>
using namespace std;
int main()
{
// Variable and constant declarations
   double radius, SurfaceArea;
   const double PI = 3.14159;

// Step 1. Input the value of the radius of the sphere.
   cout << "Enter the radius of the sphere ";
   cin  >> radius;

// Step 2. Compute the surface area of the sphere.
   SurfaceArea = 4 * PI * radius * radius;

// Step 3. Display the values of the radius and
//         the surface area of the sphere.
   cout << endl;
   cout << radius << " cm is the value of the radius\n\n";
   cout << "In a sphere of radius " << radius
        << " cm, the surface area is " << SurfaceArea
        << " sq cm ";
   cout << endl << endl;
// exit
return 0;
}
```

C++ Operators

- Operators take one or more input values and produce one output value
  - E.g. **+** , **-** , ***** , **/** , **<** , **>** , **<=** , **>=** , **&&** , **||**
- Operators come in three flavours
  - Unary operators – take one operand (value)
  - Binary operators – take two operands (values)
  - Ternary operators – take three operands (values)

C++ Operator Map



Arithmetic Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Arithmetic Operators

- **a+b**, **a-b**, **a*b**, **a/b**
- Integer division: **11/4** gives 2
- Floating point division:
  - **11.0/4** gives 2.75
- Modulo operator: **%** (**only for integers**)
  - **11%4** gives 3 (output the remainder after integer division of 11/4)
  - **10%5** gives 0 (output the remainder after integer division of 10/5)
  - **5%10** gives 5 (output the remainder after integer division of 5/10)

Example

```
// Find average of three integers
#include <iostream>
using namespace std;

int main()
{
  int number1(0), number2(0), number3(0);
  cout<<"Input three different numbers: ";
  cin>>number1>>number2>>number3;
  double average=0;
  average=(number1+number2+number3)/3.0;
  cout<<"The average is: "<<average<<endl;
  return 0;
}
```


Example: application of % operator and integer division

```
/*Given a number of days (less than 365), compute the number of
  months and weeks */

#include <iostream>
using namespace std;
int main()
{
  int days (0), months(0), weeks (0);
  cout<<"Input number of days: ";
  cin>>days;
  months = days/30;//integer division will discard the fractional part
  days = days%30;//remaining days
  weeks = days/7; //integer division will discard the fractional part
  days = days%7; //remaining days
  cout<<months<<" months, "<<weeks<<" weeks, and
  "<<days<<" days."<<endl;
  return 0;
}
```

Comparison Operators

Operator	Meaning	Example: assume a=1, b=2
>	Greater than	(a > b) returns 0 (false)
<	Less than	(a < b) returns 1 (true)
>=	Greater than or equal to	(a >= b) returns 0 (false)
<=	Less than or equal to	(a <= b) returns 1 (true)
==	Equal (equivalent)	(a == b) returns 0 (false)
!=	Not equal	(a != b) returns 1 (true)

Note: For very precise calculation avoid using == or != operators with floating-point numbers (double or float data types). double/float type numbers are stored in the memory using FEEE 754 floating point numbering system (http://en.wikipedia.org/wiki/IEEE_floating_point) and as a result .57 is stored as: .569999999999 (you can check it in the debug mode of visual studio. Here are the steps: compile the code first and then press F10 key in steps and at the bottom left part of the window, check the content of each variable you're using in the code, in each step, . Once you're done, press F5 key continue). When you multiply this number with 100 it becomes 56.999999999; if you cast this number to integer you'll get 56, NOT 57. To correct this minor error, we can add 0.5 to the number 56.9999999 and then cast it to integer which will solve our problem. Caution: we can't use ceil function which will always round up the number. Anyway, you can read tons of discussion on this issue on the web.

Logical Operators

Operator	Meaning	Example of Use	Truth Value
&&	AND	(exp1) && (exp2)	Returns true (1) if both exp1 and exp 2 are true (1) Returns false (0) if any of exp1 or exp 2 is false (0)
||	OR	(exp1) || (exp2)	Returns true (1) if any of exp1 or exp2 is true (1) Returns false (0) if both of exp1 and exp 2 are false (0)
!	NOT	! (exp1)	Returns true (1) if exp1 is false (0) Returns false (0) if exp1 is true (1)

- These are called "short-circuit" operators
- Logically any expression will result in TRUE or YES or a value of 1 if the expression has a non-zero value

Examples of Logical Operators

```
(-6<0)&&(12>=10)
```
true && true results in true

```
(3.0 >= 2.0) || (3.0 >= 4.0)
```
true || false results in true

```
(3.0 >= 2.0) && (3.0 >= 4.0)
```
true && false results in false

Unary Operators

```
Given:
int a(9);
```

- **Negate: -a gives -9**
- **Logical invert: !a gives 0**
- **Increment: ++**
  - **++a gives 10**
- **Decrement: --**
  - **--a gives 8**

More Unary Expressions

```
Given: int a(9), b;
```

- **Pre-Increment :**
  - `b = ++a;` b is 10 and a is 10
- **Post-Increment:**
  - `b = a++;` b is 9 and a is 10
- **Pre-Decrement :**
  - `b = --a;` b is 8 and a is 8
- **Post-Decrement:**
  - `b = a--;` b is 9 and a is 8

Expressions With Side Effects

- `r = x + y--;` **is equivalent to two step operations**

```
r = x + y; /*Step 1*/
y = y-1; /*Step 2*/
```

- `r = ++x - y;` **is equivalent to two step operations**

```
x = x+1; /*Step 1*/
r = x - y; /*Step 2*/
```

- **Keep it simple**
  - **Not** `r = --x + y++;`

- *Note: completely avoid writing any code that involves reassignment of a variable more than once on the same statement; such as r = –x + r++ where r = –x + r (step 1) and r = r-1 (step 2). Syntactically this is fine but some compilers may not accept it. E.g., X-Code will not give you the correct result.*

Copy Operator (Assignment operator)

- Simple copy: `a = b;`
  - Overwrites the left object (l-value) with the result of the expression on the right (r-value)
  - l-value must be writable (not a const)
- Copy with add
  - `a += b;` is the same as `a = a + b;`
- Other copy-with-subtract, multiplication, division behaves the same
  - `a -= b; a *= b; a /= b;` etc

Operator Precedence

- Use parenthesis if you are not sure!
- Unary (`++(pre-increment)`, `--(pre-decrement)`) (left to right)
- Unary (`!`, `-`) (left to right)
- Arithmetic (`*`, `/`, `%`, `+`, `-`) (left to right) (BODMAS, for review: http://www.mathsisfun.com/operation-order-bodmas.html)
- Comparison (`<` , `<=` , `>` , `>=`, `==`, `!=`) (left to right)
- Logical binary (`&&`, `||`) (left to right; **if no bracket is used && gets evaluated before ||**)
- Assignment (`=` , `*=` , `/=` , `%=` , etc) (right to left)
- Unary (`++(post-increment)`, `--(post-decrement)`),

Note: U Are the best CLAss in the Universe

Numeric Operator Precedence

- **The numeric operators in C++ expression are applied as follows:**
  - **First operation: Operands contained with pre-increment or pre-decrement operators are evaluated first**
  - **Second operation: Operators contained within pairs of round parentheses are evaluated first**
    - **Parentheses (all are round) can be nested and , in thisS case the inner one is evaluated first**
  - **Third operations: (multiplications, division and modulus) If the expression contain several multiplication, division and modulus operations, they are applied from left to right (higher precedence than addition and subtraction)**
  - **Fourth operations: If an expression contains several addition and subtraction, they are applied from left to right**
  - **Fifth operations: Assignment operations**
  - **Last operations: post-increment or post-decrement operations**

Introducing sizeof operator

sizeof() operator (an unary operator) can be used to find out the size of any variable or object in terms of bytes.

```
int x = 123;
cout <<"char byte-size: "<< sizeof(char)<<endl;
cout <<"int byte-size: "<< sizeof(x)<<endl;
cout <<"float byte-size: "<< sizeof(float)<<endl;
cout <<"double byte-size: "<< sizeof(double)<<endl;
```

Output:

```
char byte-size: 1
int byte-size: 4
float byte-size: 4
double byte-size: 8
```


Expressions

- **A series of operators and their inputs**
  - **E.g. `x+3-y+5`**
- **Evaluated from left-to-right**
- **Be careful with division**
- **Pay attention to operator precedence**

$$\frac{x+3}{5}+\frac{4}{y+3}$$

is not the same as

`x + 3 / 5 + 4 / y + 3`

Correct form:

```
(x+3)/5 + 4/(y+3)
```


Expression Examples

- `2*x*x-3*y/5*y+6`
- or `2*x*x-(3*(y/5)*y)+6`

$$\Rightarrow \quad 2x^2-\frac{3y}{5}y+6$$

- `(2*x*x-3*y)/(5*y+6)`

$$\Rightarrow \quad \frac{2x^2-3y}{5y+6}$$

Expression Example

The following expression:

$$z = \frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

is written in C++ syntax as:

```
Z=(3+4*x)/5 - 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```


FYI: Using Standard Utilities

- C++ compilers comes with a vast collection of utilities: i/o, math etc
- These are called the "standard library" files or header files
- You need to:
  - Include proper header files. E.g. `#include<iostream>`
  - Use `using namespace std;` statement
  - Or use `std::` prefix

FYI: Without `using` statement

:: is called scope resolution operator

```
#include <iostream>
// without "using namespace std;"
// entry point
int main()
{
   float w_lb, w_kg;
   std::cout << "Enter weight (lb): ";
   std::cin >> w_lb;
        //Scope Resolution Operator (::)
   w_kg = w_lb * 0.454;
   std::cout << "Weight is " << w_kg << "kg\n";
   return 0;
} // end function main
```


Characters and String

- **`char`**: holds a single character
  example:
  ```
  char alphabet = 'k';
  ```
- **`string`**: holds a sequence of characters
  example:
  ```
  string title = "C++ programming";
  // it needs #include <string> directive
  ```


Character Input

- Reading in a character

```
char alphabet;
cin >> alphabet;
```

//Reads in any non-blank character.
/*Note: Blank characters, also know as white-space characters, include single blank (press the space bar), tab space (press the tab key) and a new line space (press the enter key) */

String Input
(include <string> library file)

- Reading in a string

```
// it needs #include <string> directive
string title;
cin >> title;
/*cin extraction stops reading data as soon as it
  finds any blank (white space) character, so in
  this case we will be able to get just one word
  for each string extraction.*/
// Example ?
  Question: How can we read a sentence from the
  keyboard?
  Answer: In order to get entire lines, we can use
  the built-in function getline(). See lecture
  unit 5 for details.
```

String Operators (#include <string>)

- = operator assigns a value to a string

```
string taste;
taste = "Yummy ";
```

- + operator joins two strings together

```
string s1 = "hot", s2 = "dog";
string food = s1 + s2;
// food = hotdog
```

- += operator for concatenation

```
taste += food;
// taste = Yummy hotdog
```


Backslash Codes with cout

code	Meaning
\b	backspace
\f	form feed (when output to printer)
\n	new line
\r	carriage return
\t	horizontal tab
\"	double quote
\'	single quote

code	Meaning
\0	null
\\	backslash
\v	vertical tab (when output to printer)
\a	alert
\?	question mark
\N	octal constant (N= value from 0 to 7)
\xN	hexadecimal constant (N= Hexadecimal constant)

- These codes are also known as "escape sequence".
- These escape sequences need to be used as string constants (enclosed in a pair of double quotation marks)

Practice problem: Output?

```
#include <iostream>
using namespace std;
int main()
{
  cout<<"\\\n\\\\\n\\\\\\\n";
}
```


Math Functions in <cmath>

- Include <cmath> library at the top of the code to be able to use the following functions:

abs(x)	computes absolute value of x
fabs(x)	*computes absolute value of floating point variable x
sqrt(x)	*computes square root of x, where x >=0
pow(x,y)	**computes x^y
ceil(x)	*nearest integer larger than x
floor(x)	*nearest integer smaller than x
exp(x)	*computes e^x
log(x)	*computes ln x, where x >0
log10(x)	*computes $\log_{10}x$, where x>0

* Compiler error if the argument type is not double or float
** Good when both x and y are int type; if y is double/float, x has to be double/float

Math Functions in <cmath>

sin(x)	sine of x, where x is in radians
cos(x)	cosine of x, where x is in radians
tan(x)	tangent of x, where x is in radians
asin(x)	$\sin^{-1}(x)$, returns angle in radians [-π/2, π/2]
acos(x)	$\cos^{-1}(x)$, returns angle in radians [0,π]
atan(x)	$\tan^{-1}(x)$, returns angle in radians [-π/2, π/2]
atan2(y,x)	$\tan^{-1}(y/x)$, returns angle in radians [-π, π]
sinh(x)	Hyperbolic sine of x
cosh(x)	Hyperbolic cosine of x
tanh(x)	Hyperbolic tan of x


```
/* cos example */
#include <iostream>
#include <cmath>
using namespace std;

int main ()
{
  const double PI = 3.14159;
  double degree, result;
  degree = 45;
  result = cos (degree*PI/180);
  cout << "Cosine of " << degree << " degree is " << result << endl;
  return 0;
}
```

degrees —(degrees*PI/180)→ radians

radians —(radians*180/PI)→ degrees

```
/*********************************************************
*  This program finds the surface area                   *
*  of a sphere with a given radius                       *
*********************************************************/
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
// Variable declarations
   double radius, SurfaceArea;
   const double PI = 3.14159;
// Step 1. Input the value of the radius of the sphere.
   cout << "Enter the radius of the sphere ";
   cin  >> radius;

// Step 2. Compute the surface area of the sphere.
   SurfaceArea = 4 * PI * pow (radius, 2);
// Step 3. Display the values of the radius and the surface
//         area of the sphere.
   cout << endl;
   cout << radius << " cm is the value of the radius\n\n";
   cout << "In a sphere of radius " << radius
        << " cm, the surface area is " << SurfaceArea
        << " sq cm ";
   cout << endl << endl;
// exit
return 0;
}
```


Arithmetic conversions

- Implicit Conversion
- Explicit Conversion

Implicit Conversion (IC)

- In an arithmetic expression, the final result of the expression (with different numerical data types) will be stored using a data type of the operand (present in the expression) having the highest data range in the memory.
- Implicit arithmetic conversion is also called automatic conversion, implicit conversion, coercion, promotion or widening.
- Ex: *int a*, *float b*; (*a* + *b*) will be a *float* type where *int a* gets promoted to *float a*.

Explicit Conversions: Cast

- Explicit arithmetic conversions are called *casts*.
  - For example, *int a* can be explicitly converted to a float variable in any statement that follows the above declaration, by the cast operator as: (*float*) *a*.
- Cast operator: (data_type) variable
- Cast operator is an unary operator
- Cast operator works only for the part of the statement that uses it.
- Example:

```
int x(5);
cout<<x/2<<endl;//output?
cout<<(float)x/2 + x/10<<endl;//output?
cout<<x/2<<endl;//output?
```


Review

26) `a+b/c-d` is the same as

a) a+b/(c-d)
b) a+(b/c)-d
c) (a+b)/c-d
d) (a+b)/(c-d)

Review

27) Which of the following expressions is zero

a) 8%3 - 1
b) 3 - 8%5
c) 9%3 - 1
d) 2 - 5%2

Review

28) Which C++ expression computes the following equation

$$\frac{a(x-y)}{b} + c$$

a) `(a/b)*x-y+c`
b) `a/(b*(x-y))+c`
c) `a/(b*x-y)+c`
d) `(a/b)*(x-y)+c`

Review

29) What is the value of newNum after these two lines?

```
int i(10);
int newNum = 10 * i++;
```

a) 10
b) 100
c) 110
d) 1000

Review

30) The following two C++ statements perform the same operation.

```
regWages = regPay + overTime;
regPay + overTime = regWages;
```

a) True
b) False

Review

31) The following two expressions will result in the same value:

```
a + b * c
b * c + a
```

a) True
b) False

Review

32) The following statement doubles the value stored in answer.

```
answer *= 2;
```

a) True
b) False

Review

33) Which of the following is the correct expression that evaluates to TRUE if the number x is in between 1 and 100 or the number is negative?

```
a) (1 < x < 100) && (x < 0)
b) ((x < 100) && (x > 1)) || (x < 0)
c) ((x < 100) && (x > 1)) && (x < 0)
d) (1 > x > 100) || (x < 0)
```

Review

34) What will be the value of 'c' after the third statement?

```
int a(3), b(2);
float c;
c = (a+b)/2;
```

a) 2.5
b) 2
c) 3
d) 0

Review

35) value of variables 'a' and 'b' after the second statement?

```
int a(2), b(4);
a=++b;
```

a) a is 2 and b is 5
b) a is 4 and b is 4
c) a is 5 and b is 5
d) a is 4 and b is 5

Review

36) value of variable 'a' after the second statement?

```
int a(2);
a = 5==5;
```

a) 5
b) 0
c) 10
d) 1

Review

37) Output ?

```
int a;
cout<<a<<endl;
```

a) 0
b) Any random value
c) Compilation error will be generated
d) Run time error will be generated

Review

38) Output ?

```
int a (2), b = a+2;
double b(4);
cout<<a/b<<endl;
```

a) 0
b) 0.5
c) The code will result in a Compilation error
d) The code will result in a Run time error

Programming Style and Documentation

This part is very important for labs

- Appropriate Comments
- Naming Conventions
- Proper Indentation and Spacing Lines
- Block Styles

Appropriate Comments

- Include a summary at the beginning of the program to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses.
- Include your name, class section, instructor, date, and a brief description at the beginning of the program.

Naming Practices (Read the tutorial posted with lab 1 handout)

- Choose meaningful and descriptive names.
- Variables and function (or method) names:
  - Use lowercase. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name.
    - For example, the variables `interestRate` and `year`, and the function `computePayment`.

Naming Practices, *cont.*

- Class names (*Class will be discussed later*):
  - Capitalize the first letter of each word in the name. For example, the class name `InterestRateCalculation`.
- Constants:
  - Capitalize all letters in constants, and use underscores to connect words. For example, the constant `PI` and MAX_VALUE

Proper Indentation and Spacing

- Indentation
  - Indent two spaces.
- Spacing
  - Use blank line to separate segments of the code.

Block Styles

Use either next-line or end-of-line style for braces.

```
int main()
{//Next line style
      cout<<"This is next-line block style\n";
      return 0;
}
```

```
int main(){//end-of-line style
      cout<<"This is end-of-line block style\n";
      return 0;
}
```


Answers to the review questions

19. A
20. B
21. E
22. B
23. D
24. C
25. D
26. B
27. B
28. D
29. B
30. B
31. A
32. A
33. B
34. B
35. C
36. D
37. D
38. C

ES 1036 Programming Fundamentals

C++ Control Structures

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman, Ph.D, P.Eng., SMIEEE,
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Flow of Control

- The order in which statements are executed, is called flow of control.
- There are three basic control structures
  - Sequence – execute statements in sequence (example: all the program examples shown earlier)
  - Selection – choose which statements to execute,
  - Repetition- repeat certain statements.

Selection: the `if` Statement

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

single statement executed if *expression* is true

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

if *expression* and its following statement can be written in one line

```
if(expression)
{
      statement 1;
      statement 2;
      …
      statement n;
}
```

statements inside {} are executed if *expression* is true The {} make all the statements appear as a single statement

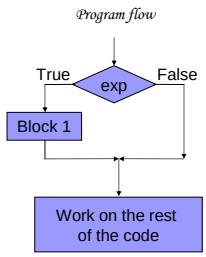
If expression is always evaluated as either true (1) or false (0)

Expressions in if-statements

- If-expression is always evaluated as either true (1) or false (0)
- In general, the expression in the if statement can use two types of operators :
  - Comparison
    - Relational (>, <, >=, <=)
    - Equality (==, !=)
  - Logical (&&, ||, !)
- It can also use other operators
- The expression can be a combination of more than one nested expression or it can be just a single variable without any operator (see an example later)

Nested code structure: a code structure where any code-block resides inside another code-block is known as nested code structure. See the example on slide #9.

The if statement - examples




```
#include<iostream>
using namespace std;
int main()
{
   int x(5),k(1);
   if (x>0 && k>=1)
      k++;
   if (x<0 || k==1)
   {
      x=x+3;
      k++;
   }
  cout<<"x = "<<x<<" and k =
"<<k<<endl;
}
```


The **if** statement - example

```
#include<iostream>
using namespace std;
int main() {
  int x(0), y(6);
  if (x) {// an expression can be a single variable without operator
     cout << "x has a non-zero value" << endl;
  }
  if (1) {// an expression can be a constant value
     cout << "The above if is always true\n";
  }
  cout<< "It's over\n";
  return 0;
}
```

The `if` statement - examples

```
int i=5;
if ((i>0) && (i<10))
   cout << "i is an integer between 0 and 10" << endl;
cout<< "This statement is outside the if block" << endl;
```

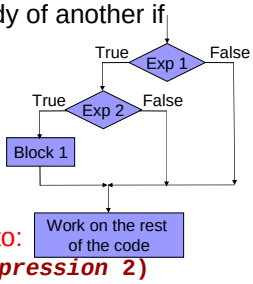
```
int a = 5, b = 1;
if (a>b) cout<<"a is greater\n"; cout<<"This statement is
not a part of the if-statement\n";
```

```
int sum=0, a(1), count=0;
if (a < 5){
   ++count;
   sum += a;
}
cout<<sum<<" and "<<count;
```


Nested if statement

- Since the if-statement itself is a statement it can appear in the body of another if statement

```
if(expression 1)
   if(expression 2)
     statement;
```



The above is equivalent to:

```
if(expression 1 && expression 2)
   statement;
```


Example on Nested if statement

```
/*Problem: If the user-entered choice is Y and user-entered
   number is 5, print the entered number on the screen*/
#include<iostream>
using namespace std;
int main()
{
  char choice('y'); double anyNumber(0);
  cout<<"Enter choice: "; cin>>choice;
  choice = toupper(choice);
  cout<<"Enter any number: "; cin>>anyNumber;
  if(choice == 'Y')
  {
      if(anyNumber == 5)
      {
            cout<<"Output: "<<anyNumber<<endl;
      }
  }
  cout<<"goodbye"<<endl;
  return 0;
}
```

Revisiting the previous Example

```
/*Problem: If the user-entered choice is Y and user-entered
   number is 5, print the entered number on the screen*/

#include<iostream>
using namespace std;
int main()
{
  char choice('y'); double anyNumber(0);
  cout<<"Enter choice: "; cin>>choice;
  choice = toupper(choice);
  cout<<"Enter any number: "; cin>>anyNumber;
  if(choice == 'Y' && anyNumber == 5)
  {
      cout<<"Output: "<<anyNumber<<endl;
  }
  cout<<"goodbye"<<endl;
  return 0;
}
```


Note on toupper() and tolower()

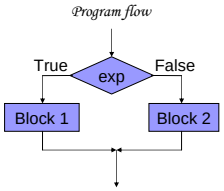
- int toupper (int c);
  - This function converts the argument *c* to its uppercase equivalent if *c* is a lowercase letter; otherwise it returns the same letter
- int tolower (int C);
  - This function converts the argument *C* to its lowercase equivalent if *c* is an uppercase letter; otherwise it returns the same letter
- *No especial library file needs to be included to work on these two functions.*

The `if-else` statement

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

Considered as one C++ statement

```
if(expression)
{
  statement block 1;
}
else
{
  statement block 2;
}
```



The `if-else` statement - example

```
/*Find out whether the entered number is even or
odd. (It's another application of % operator)*/

#include <iostream>
using namespace std;
int main()
{
    // Prompt the user to enter an integer
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    if (number % 2 == 0)
         cout << number << " is even.";
    else
         cout << number << " is odd.";
return 0;
}
```


Program example

- Problem: Prompt the user to enter a number between 3 and 30 inclusive and print the entered number on the screen. If the number is outside the above range print "out of range".

```
#include <iostream>
using namespace std;

int main ()
{
   cout<<"Enter an integer: ";
   int x; cin>>x;
   if((x>=3)&&(x<=30))
       cout<<"You've entered "<<x<<endl;
   else
       cout<<"Out of range\n";
   cout<<"Good Bye!\n";
}
```

Homework: Draw the flowchart for the above problem


```
/*Example Problem: Write a program that prompts the user
to enter two integers and finds the greater one*/
#include<iostream>
using namespace std;
int main() {
   // define two integers
   int x , y;
   // step 1:
   // prompt the user to enter the values of x and y
   cout << "Enter the value of x and y: ";
   cin >> x >> y;
   // step 2:
   // print out a message based on the x and y values
   if (x > y) {
       cout << "x is greater than y" << endl;
   }
   else {
       cout << "y is greater or equal to x" << endl;
   }
   return 0;
}
```

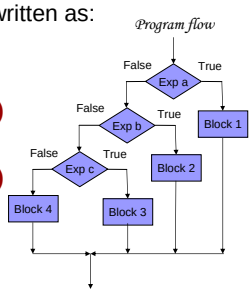
Homework: Draw the flowchart for the above problem

```
/* Example Problem: How to compare two floating point
numbers? See the note at the bottom of slide #55,
lecture Unit 3 */

#include <iostream>
using namespace std;
#include <cmath>
const double tolerance = 0.01;
int main()
{
  double num1(6.32), num2 = 632000/100000.1;
  // below, check it for yourself
  cout<<632000/100000.1<<endl;
  cout<<fabs(num1-num2)<<endl;

  //now compare formally
  if(fabs(num1-num2)<=tolerance)
     cout<<"Equal numbers\n";
  else
     cout<<"Not equal numbers\n";
}
```


`if-else-if` Statement

- The if-else-if statement is written as:

```
if (expression a)
   statement block 1;
else if (expression b)
   statement block 2;
else if (expression c)
   statement block 3;
else
   statement block 4;
```



- This is the most widely used if-structure

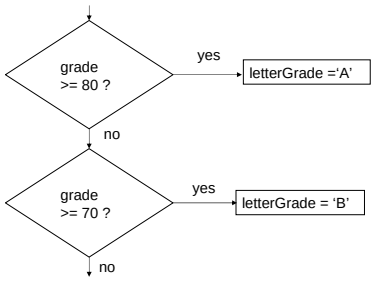
Grading Program Example

- Consider the following table for assigning marks for a course. Assumption: user will not enter any mark greater than 100 or less than 0.

Number Range	Letter Grade
80 or greater	A
70-79	B
60-69	C
50-59	D
Less than 50	F

Grading program

- Flow chart




```
#include <iostream>
using namespace std;
int main ()
{
      int grade;
      char letterGrade;
      cin >> grade;
      if(grade >= 80)
            letterGrade = 'A';
      else if (grade >= 70)
            letterGrade = 'B';
      else if (grade >= 60)
            letterGrade = 'C';
      else if (grade >= 50)
            letterGrade = 'D';
      else
            letterGrade = 'F';
      cout << "Your grade is " << letterGrade << endl;
      return 0;
}
```

Homework; Modify the code so that you can display A+ when the grade is more than 89.

Caution!

- **Adding a semicolon at the end of an if clause is a common mistake.**

Logic Error

```
if (radius >=   0);
{
  area = radius * radius * PI;
  cout <<  "The area "
    << " is "  << area;
}
```

Equivalent

Empty Body

```
if (radius >=   0) { ;
{
  area = radius * radius * PI;
  cout <<  "The area "
    <<  " is "  <<  area;
}
```

- **This mistake is hard to find, because it is NOT a compilation error, it is a logical error.**
- **This error often occurs when you use the next-line block style.**

Caution!

- **Using assignment operator (=) in place of an equality operator (==) in an if-expression is a common mistake.**
- **Sometimes this mistake may result in a compilation error which will be easy to detect.**
  - **E.g.** `if (num%2 = 0){//some code}`

    The reason for the above error: the left-hand side of an assignment-operator has to be a writable memory location.
- **Sometime this mistake is hard to find, because it may result in a logical error.**
  - **E.g.** `if (num = 0){//some code}`

    The reason for the above logical error: In the if-expression the value 0 is being assigned to the variable *num*.

Caution!

What happens if the expression,
`if(0 <= n && n <= 100)` is written as:
`if(0 <= n <= 100) which is grammatically correct`

The operator <= associates from left to right:
Suppose n = 230

```
if(0 <= 230 <= 100)➔
if(1<= 100) ➔
if(1)➔true
```

True (1)

Logical error! Very hard to find!

Note: Use curly braces for clarity

```
if(x > y)
    if(y < z)
        k++;
    else
        m++;
else
    j++;
```

Better with braces

```
if(x > y)
{
    if(y < z)
        k++;
    else
        m++;
}
else
    j++;
```

Review

37) What are the values of j, k and m?

```
int x=9, y=7, z=2, k=0, m=0, j=0;
if(x > y)
      if(y>z && y>k)
            k++;
      else
            m++;
else
      j++;
```

a) j is 0, k is 1, m is 0
b) j is 1, k is 0, m is 0
c) j is 0, k is 0, m is 1

Review

38) The following `cout` statement will be executed:

```
int s = 0;
if (s = 0)
     cout << "S is zero";
```

a) True
b) False

Practice problem

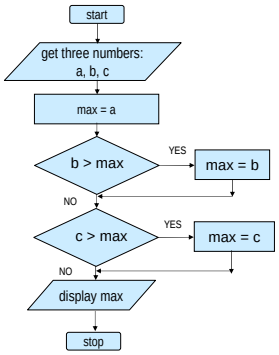
Write a program that prompts the user to enter three integers and finds the maximum of the three

Step 1: Draw the flow-chart.

Programming Tip: Whenever solving any problem that requires to find the maximum (or minimum) number from a set of numbers, declare a suitable variable (max or min) and assign the first number (from the set) to it. Then, compare all the other numbers one at a time with this variable value and refresh this value accordingly.

Note: *Any problem can be solved in many different ways. Try to solve the above problem before looking at the solution available later*

Flow-chart for Finding max from three entered numbers




```
//Solution
#include<iostream>
using namespace std;
int main() {
    // define three integers
    int a , b , c;
    int max;
    // step1: prompt the user to enter the values of a , b and c
    cout << "Enter the value of a , b, and c: ";
    cin >> a >> b >> c;

    // step2: determine which integer has the maximum value
    // then save it into max
    max = a;
    if (b > max) {
        max = b;
    }
    if (c > max) {//using else-if will result in a logical error
        max = c;
    }

    // step 3: Display the maximum value
    cout << "The maximum integer is " << max << endl;
    return 0;
}
```


Practice Problem

Write a program that lets the user enter a year and checks whether it is a leap year.

Note: A year is a leap-year if (http://www.mathsisfun.com/definitions/leap-year.html)

- **it is divisible by 4 but not by 100 or**
- **if it is divisible by 400.**

So you can use the following Boolean expression

```
(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

Note: *Try to solve the above problem before looking at the solution on the next slide*

```
//Solution 1 for leap-year problem
#include <iostream>
using namespace std;
int main()
{
  cout << "Enter a year: ";
  int year;
  cin >> year;

  // Check if the year is a leap year
  bool isLeapYear =
    (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

  // Display the result in a message dialog box
  if (isLeapYear)
    cout << year << " is a leap year.";
  else
    cout << year << " is a not leap year.";
}
```



```
//Solution 2 for leap-year problem


#include <iostream>
using namespace std;
int main()
{
  cout << "Enter a year: ";
  int year;
  cin >> year;

  // Check if the year is a leap year and display the result
  if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
      cout << year << " is a leap year.\n";
  else
    cout << year << " is a not leap year.\n";
}
```


Selection: `switch` Statement

- *Switch-expression* must result an integer or a character and always be inside a bracket
- The keyword case must be followed by a *constant value* (value1, ..., and valueN) and this constant must have the same data type as the value of the switch-expression.
- The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression.

```
switch (switch-expression) {
  case value1: statement(s)1;
       break;
  case value2: statement(s)2;
       break;
  …
  case valueN: statement(s)N;
       break;
  default: statement(s);
}
```

Note that value1, ..., and valueN are constants meaning that they cannot contain variables, such as 1 + x.

`switch` Statement *cont.*

- The keyword **`break`** is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed along with the chosen one(s).
- The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.
- No block structure is required inside the case statement, if you don't declare any variable inside the case; otherwise block structure is required.
- The case labels can be in any order

```
switch (switch-expression) {
  case value1: statement(s)1;
     break;
  case value2: statement(s)2;
     break;
  …
  case valueN: statement(s)N;
     break;
  default: statement(s);
}
```


Review: the `switch` statement

```
switch(expression)
{
   case constant :
      statement(s);
      break;
   case constant :
      statement(s);
      break;
// default is optional
   default:
      statement(s);
}
```

```
switch(n)
{
   case 1:
      cout << "one";
      break;
   case 2:
      cout << "two";
      break;
   default:
      cout << "error";
}
```


Example: Enter month number to print the name of the month

```
#include <iostream>
using namespace std;
int main()
{
   cout<<"Enter month\'s number: ";
   int month_number; cin>>month_number;
   switch(month_number)
   {
   case 1:
       cout<<"January\n";
       break;
   case 2:
       cout<<"February\n";
       break;
       //other cases up to 12
   default:
       cout<<"Invalid month number\n";
   }
}
```

Practice!

Modify the following code using **switch** statement: (solve it and then check the solution on the next slide)

```
#include <iostream>
using namespace std;
int main()
{
      int rank(0);
      cout<<"Enter rank: "; cin>>rank;
      if (rank==1 || rank==2)
            cout << "Lower division \n";
      else if (rank==3 || rank==4)
            cout << "Upper division \n";
      else if (rank==5)
            cout << "Graduate student \n";
      else
            cout << "Invalid rank \n";
}
```


Practice Solution!

```
#include <iostream>
using namespace std;
int main()
{
      int rank(0); cout<<"Enter rank: "; cin>>rank;
      switch(rank)
      {
      case 1: case 2:
            cout << "Lower division \n";
            break;
      case 3: case 4:
            cout << "Upper division \n";
            break;
      case 5:
            cout << "Graduate student \n";
            break;
      default:
            cout << "Invalid rank \n";
      }//end switch (rank)
}
```


Practice: Switch structure with character value

```
#include <iostream>
using namespace std;
int main()
{
      char choice('a'); cout<<"Enter choice: "; cin>> choice;
      choice = toupper(choice);
      switch(choice)
      {
      case 'A':
            cout << "Choice A has been entered\n";
            break;
      case 'B':
            cout << "Choice B has been entered\n";
            break;
      case 'C':
            cout << "Choice C has been entered\n";
            break;
      default:
            cout << "Invalid choice \n";
      }//end switch (choice)
}
```

Caution: curly bracket for each case

```
#include <iostream>
using namespace std;
int main()
{
      int k=0, m = 15; cin>>k;
      switch(k)
      {
      case 1:
            {
              int y = 10;
              cout<<y<<endl; break;
            }
      case 2:
            cout<<m<<endl; break;
      default:
            cout<<"do nothing\n";
      }
}
```

Set of curly brackets are required inside the case (as shown) if any variable is declared in there; otherwise the code will not compile.

Review

39) In C++ when a relational expression is false, it has the value _____.

a) 1
b) 0
c) -1
d) of any negative number
e) None of these

Review

40) The _____ statement will execute a group of statements if the test expression is true, or it'll execute a different group of statements if the expression is false.

a) if
b) if/else
c) switch
d) None of the above

Review

41) A trailing _____ placed at the end of "if/else if" statement provides a default action when none of the "if/else if"s have true expressions.

a) if
b) break
c) else
d) exit
e) None of these

Review

42) The following test expression in the *if* statement checks if the variable child is in the range between 3 and 12 inclusive.

```
if (child >= 3 || child <= 12)
        {statement(s);}
```

a) True
b) False

Review

43) If a switch statement has no _____ statements, the program "falls through" all of the statements below the one with the matching case expression.

a) break
b) exit
c) switch
d) else
e) None of the above

Review

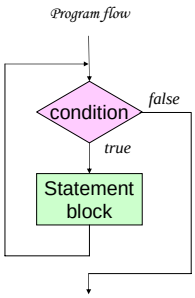
44) A switch statement branches to a particular block of code depending on the value of a floating-point variable or constant.

a) True
b) False

Repetition structures

- while
- do-while
- for

Repetition: While Loop




```
while (condition)
  statement;

while (condition)
{
  statement block;
}
```

Basic characteristics of *while* Loops

- Loop repetition condition is tested before the body of the loop.
- Body of the loop will be executed 0 or more times.
- Doesn't require braces ({ }) if the body of the loop contains only one statement, but it is a good practice to have them (always).
- Useful when the exact number of repetitions is unknown

Practice!

```
#include <iostream>
using namespace std;
int main()
{
      int n=4;
      while(n>0)
      {
            cout << n << endl;
            n--;
      }
      cout << "value of n outside while is " << n
            << endl;
      return 0;
}
```


Typical Use of *while* Loops

- Infinite loop: if the loop repetition condition is always true, an endless loop will result.

```
…
while(1){
 cout<<"Help, I'm stuck in a while loop!\n";
}
…
```

Example with *infinite* loop

- sometimes Infinite loops indicate an error in a program.

```
int main(void)
{
   int a = 5;
   while(a < 10) {
      cout<<"a = "<< a<<endl;
   }
   cout <<"bye";
   return 0;
}
```


Typical Use of *while* Loops

- "Validation" loop: the loop repetitively checks whether the user has input a valid data (set by the loop test expression) or not
- When the user enters a valid input, the looping ends and the program continues from the statement that follows the loop structure.

Example on *Validation* loop

- Problem: Prompt the user to enter a number between 3 and 30 inclusive, and after validating the number, print the entered number on the screen.
- Discussion: How to write the validation loop-statement correctly?

```
#include <iostream>
using namespace std;
int main ()
{
   cout<<"Enter an integer: "; int x; cin>>x;
   while(!((x>=3)&&(x<=30)))//same as: while((x<3)||(x>30))
   {
       cout<<"You've entered an invalid number\n";
       cout<<"Enter an integer between 3 and 30: ";
       cin>>x;
   }
   cout<<"You've entered "<<x<<endl;
}
```

Note: for validating any data input, write the expression based on the requirement and, logically invert the whole expression inside the Loop-condition

Example on *Validation* loop

- Validating characters: the following code validates that the user enters y or n as his/her choice

```
#include <iostream>
using namespace std;
int main()
{
    char choice ('y');
    cout<< "Enter choice: ";
    cin>>choice; choice = tolower(choice);
    while(!(choice=='y' || choice =='n')) //same as: while(choice!='y' && choice !='n')

    {
          cout<< "Invalid choice! enter again: ";
          cin>>choice; choice = tolower(choice);
    }

    cout<<"you entered " <<choice<< "\n";
}
```


Caution! Inputting characters for an integer variable while checking a loop condition

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string month, days;
    cout<<"Enter a month number ( 1 = January, etc.): ";
    cin>>month;
    int myint = atoi(month.c_str());
    while( myint<  1 || myint > 12 )
    {
          cout<<"Month number is out of range, please enter a valid month number
    between 1 and 12: ";
          cin>>month;
          myint = atoi(month.c_str());
    }
}
```

While checking a loop condition for a possible integer input, if we input any character accidentally, the output will become unstable. To solve this problem this particular code can be used. Please note that this slide will not be included in the exam

atoi() function available in iostream library converts a string to an integer but atoi needs a character array to do so. In the string libray the c_str() method does this work.

Homework on Validation loop

- Write the code for the problem given in slide #18 after removing the assumption there.
- Prompt the user to enter an integer between 3 and 30 inclusive and validate this input.
- Prompt the user to enter an integer between 3 and 30 inclusive which can be divisible by 2 and 7. Validate this input and print the validated integer on to the screen
- Validate that an user-entered number is an odd number

Typical Use of *while* Loops

- Sentinel loop: A chosen value (based on the program's requirement), called the sentinel, is used to signal an end to data entry.

Sentinel loop – example 1

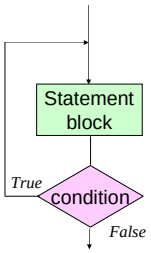
```
/*write a code to enter integer values from the keyboard and find the sum of
those numbers before the user enters zero. The program should let the user
input the integer numbers until the user enters zero. Once the user enters
zero, the program will display the sum of all the entered number.*/
// Here the sentinel has been chosen to be zero
#include <iostream>
using namespace std;
int main()
{
   cout << "Enter an int value (the program exits if the input is 0): ";
   int data;
   cin >> data;

   // Keep reading data until the input is 0
   int sum = 0;
   while (data != 0)
   {
        sum += data;
        // Read the next data
        cout << "Enter an int value (the program exits if the input is 0): ";
        cin >> data;
   }
   cout << "The sum is " << sum << endl;
   return 0;
}
```


Sentinel loop – example 2

- Sentinel loop: A letter can be chosen as the sentinel

```
int main(void)
{
   int number; char again = 'y';
   while(again == 'y') {
      cout<<"Enter an integer: ";
      cin>>number;
      cout<<" The square of " <<number<<" is "
          << number*number;
      cout<<"\ntry again?(y/n) ";
      cin>>again; again = tolower(again);
      } /* end while loop */
   cout<<"good-bye\n";
   return 0;
}
```

Repetition: The `do/while` statement




```
do
{
  statement;
}
while (condition);

do
{
  statement block
} while (condition);
```


Basic characteristics of *do while* Loops

- Loop repetition condition is tested after the body of the loop.
- Body of the loop will be executed 1 or more times.
- Doesn't require braces ({ }) if the body of the loop contains only one statement, but it is a good practice to have them (always).
- Useful when at least one repetition of the loop is required.

Validation Example with *do-while*

Problem: Prompt the user to enter a number between 3 and 30 inclusive, and after validating the number, print the entered number on the screen.

```
#include <iostream>
using namespace std;
int main ()
{
  int x;
  do
  {
    cout<<"Enter an integer between 3 and 30: ";
    cin>>x;
  }while (!((x>=3)&&(x<=30)));
  cout<<"You've entered "<<x<<endl;
}
```

Validation Example(2) with *do-while*

Problem: revisiting the previous problem to make it more user friendly.

```
#include <iostream>
using namespace std;
int main ()
{
  int x;
  do
  {
    cout<<"Enter an integer between 3 and 30: ";
    cin>>x;
    if(!((x>=3)&&(x<=30)))
       cout<<"Invalid Entry! ";
  }while (!((x>=3)&&(x<=30)));
  cout<<"You've entered "<<x<<endl;
}
```


More on Validation with do-while

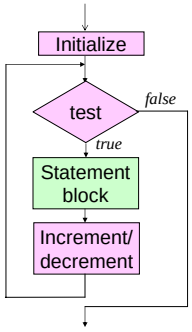
- Example: Write a program to display a menu that contains three numbered operations and prompt the user to select a correct operation-number.


```
#include <iostream>
using namespace std;
int main()
{
    int choice;

    cout << "***        Main Menu       ***" << endl;
    cout << "    (1) Operation one"   << endl;
    cout << "    (2) Operation two"   << endl;
    cout << "    (3) Operation three" << endl;
    cout << "*******************************" << endl << endl;

    // Prompt the user to enter a number between 1 and 3
    do
    {
       cout << "Enter any number between 1 and 3 " ;
       cin >> choice;
    }while (!(choice >= 1 && choice <= 3)); // data validation
    system("cls"); // clears the screen
    cout << "Performing operation " << choice << endl;
    return 0;
}
```

Evaluating a series expression using loops

- Always initialize a variable that will hold on to the result (for summation the initialized value will be zero and for multiplication it will be 1)
- Find out the following:
  - Start point
  - step size of the series
  - End point
- Find the expression based on the step size
- In each loop update the variable, which was declared and initialized in the first step
- Print the result outside the loop
- Use any loop (while, do-while or for) of your choice
- Note: When a division-operation is present in the series expression, be very careful about data type where casting might be required for logically correct output.

Repetition: The `for` statement




```
for(init; test; inc/dec)
    statement;
for(init; test; inc/dec)
{
    statement;
    statement;
}
```


Basic characteristics of *for* Loops

- Loop repetition condition is tested before the body of the loop.
- Body of the loop will be executed 0 or more times.
- Doesn't require braces ({ }) if the body of the loop contains only one statement.
- Useful when exact number of repetition is known

The `for` statement: Examples

```
//sum integers from 1 to 10
int sum =0;
for(int i=1; i<=10; i++)
     sum = sum + i;

//sum odd integers from 1 to 10
int sum =0;
for(int i=1; i<=10; i+=2)
     sum = sum + i;
```


Practice!

Determine the number of times that each of the following `for` loops is executed.

```
for (int k=3; k<=10; k++)
   cout<<"do nothing\n";
```

```
for (int k=3; k<=10; ++k)
{
   cout<<"do nothing\n";
   k=k+2;
}
```

```
for (int count=-2; count<=10; count++)
   cout<<"do nothing\n";
```


for Loop in series summation

$$y = \sum_{i=1}^{4} x_i = x_1 + x_2 + x_3 + x_4$$

```
int main(void)
{
   int i, y = 0, x;
   cout<<"Add 4 integers and display the
  total:\n\n";
   for (i = 1; i < 5; i++)
   {
      cout<<"Enter integer number "<< i;
      cin>>x;
      y += x;
   }
   cout<<"\nThe total equals: "<<y; return 0;
}
```

Typical Uses of Loops: Maximum or Minimum

```
int main(void)
{
   int smallest, number, i;
   cout<<"Enter an integer: ";
   cin>>number; smallest = number;
   for(i = 0; i < 4; i++)
   {
      cout<<"Enter another integer: ";
      cin>>number;
      if(number <= smallest)
         smallest = number;
    } /* end for loop */
   cout<<"\nThe smallest number is: "<<smallest;
return 0;
}/*modity this code so that it can find both smallest
  and the largest numbers*/
```


for Loop in series multiplication

$$y = \prod_{i=1}^{4} x_i = x_1 \times x_2 \times x_3 \times x_4$$

```
int main(void)
{
   int i, product = 1, number;
   for (i = 1; i < 5; i++)
      {
      cout<<"Enter integer number "<< i;
      cin>>number;
      product *= number;
    }
   cout<<"\nThe result equals: "<< product;
   return 0;
}
```


Practice problems with series operations (try to solve with while, do-while and for structures)

- Write a C++ program to calculate the value of **e** for an integer value of **n** (n>=0) where **e** is given by the following series (factorial (!)operation has been demonstrated later):

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots\ldots + \frac{1}{n!}$$

- Write a C++ program to numerically calculate the value of y given by,

$$y = x^2 + px + 2p^2$$

where,

$$-\infty < x < \infty;$$

$$p = \prod_{i=1}^{N} \frac{1}{i^2} = 1 \text{ x } \frac{1}{4} \text{ x } \frac{1}{9} \text{ x } \ldots\ldots \text{ x } \frac{1}{N^2};\ \text{(N is a positive non-zero integer)}$$

Typical Uses of Loops: Nested Loops

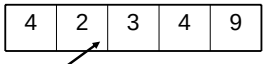
```
int main (void)
{
   int i, j;
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
      {
       cout<<"outer loop i = "<<i
           <<", inner loop j = "<< j;
       cout<<endl;
      }
   }
   return 0;
}
```


Output

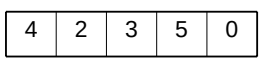
outer loop i = 0, inner loop j = 0
outer loop i = 0, inner loop j = 1
outer loop i = 1, inner loop j = 0
outer loop i = 1, inner loop j = 1

Nested Loop Analogy

- A nested loop is somewhat analogous to a an odometer or counter.



The "outer loop" increments by one each time the "inner loop" completes a full cycle.



Another example on nested loop

```
//This program prints a square with
//a preselected chracter
#include <iostream>
using namespace std;
int main()
{
  char a = '&';
  int square_size = 4, row (0), col (0);
  for(row=0; row<square_size; row++)
  {
      for(col=0; col<square_size; col++)
      {
            cout<<a;
      }
  cout<<endl;
  }
}
```

Output:
&&&&
&&&&
&&&&
&&&&

Another Example with Nested loop: 3 bit numbers

```
int main (void)
{
   int d1, d2, d3;
   for (d3 = 0; d3 < 2; d3++)
    {
      for (d2 = 0; d2 < 2; d2++)
         {
         for (d1 = 0; d1 < 2; d1++)
             {
           cout<<"d3= “<<d3<<"d2= "<<d2<<"d1= "<<d1<<"\n";
             }
       }
  }
   return 0;
}
```


Output: 3 bit numbers

d3=0 d2=0 d1=0
d3=0 d2=0 d1=1
d3=0 d2=1 d1=0
d3=0 d2=1 d1=1
d3=1 d2=0 d1=0
d3=1 d2=0 d1=1
d3=1 d2=1 d1=0
d3=1 d2=1 d1=1

Home work Question: Modify the code in the previous slide so that it can print the above table in the reverse order (the bottom one should be printed first and so on)

Practice!

```
//This while loop calculates n!
int nfact=1, n;
cout << “enter positive integer “;
cin >> n;
int m = n;

while(n > 0)
{
     nfact = nfact*n;
     n--;
}
cout << m << “! = “ << nfact;
..
```

Write a **for** loop to replace the while loop

Solution

```
//This for loop calculates n!
int nfact, n;

cout << "enter positive integer ";
cin >> n;

for (nfact=1; n > 0 ; n--)
{
   nfact = nfact*n;
}

cout << "The factorial is " << nfact << endl;
..
```


How to validate an integer input

```
#include <iostream>
using namespace std;
#include <string>
int main ()
{
   double base;
   cout<<"Enter an integer: ";
   cin>>base;
   int k = (int)base;
   while(k!=base)
   {
      cout<<"Invalid entry! enter an integer: ";
      cin>>base;
      k = (int)(base);
   }
   cout<<“The integer value you entered is: "<<k<<endl;
}
```

How to validate an integer input

```
#include <iostream>
using namespace std;
#include <string>
int main ()
{
  string a;
  cout<<"Enter an integer: ";
  cin>>a;
  double p = atof(a.c_str());
/*atof() function converts the entered string to a floating point
number. This function returns zero for any character or string
input*/
 while(a!="0" && p==0 || (int)p<p)
 {
  cout<<"invalid entry! enter again: ";
  cin>>a;
  p = atof(a.c_str());
 }
 int k = (int)p;
 cout<<"The validated integer input is: "<<k<<endl;
}
```


Practice!

- Problem: Write a program that uses nested-for loops to print a multiplication table.

Solution

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  cout << "       Multiplication Table\n\n";
  // Display the number title
  cout << "  | ";
  for (int j = 1; j <= 9; j++)
    cout << setw(3) << j;
  cout << "\n";
  cout << "-----------------------------\n";
  // Print table body
  for (int i = 1; i <= 9; i++) {
    cout << i << " | ";
    for (int j = 1; j <= 9; j++) {
      // Display the product and align properly
      cout << setw(3) << i * j;
    }
    cout << "\n";
  }
  return 0;
}
```

Practice!

- **Problem: Write a program that prompts the user to enter an integer from 1 to 15 and displays a pyramid. For example, if the input integer is 6, the output is shown below.**

```
          1
        2 1 2
      3 2 1 2 3
    4 3 2 1 2 3 4
  5 4 3 2 1 2 3 4 5
6 5 4 3 2 1 2 3 4 5 6
```



```
//The solution
#include <iostream>
using namespace std;
int main()
{
  cout<<"Enter the size of the pyramid: ";
  int n; cin>>n;
  for(int i=1; i<=n; i++)//loop for ith row; i is the row no.
  {
    //printing spaces in each row
    for(int j=n; j>i; j--)
        cout<<" ";
    //printing numbers from 'row number' to 1 in each row
    for(int k=i; k>=1; k--)
        cout<<k;
    //printing numbers from 2 to 'row number' in each row
    for(int m=2; m<=i; m++)
        cout<<m;
    cout<<endl;//printing a new line after each row
   }
}
```


Home work problem

- Draw a flow-chart and then write the corresponding code with the following specs:
  - Prompt the user to enter an integer value n greater than 10; validate that number
  - Now ask the user to enter these n different numbers from the key board and do the following:
    - Validate that the entered numbers are in between 100 and 200 inclusive.
    - Calculate the average of all the odd numbers entered and
    - Calculate the average of all the even numbers entered.
  - Your code should print the following:

    You've entered <x> odd numbers and <n-x> even numbers; the average for the odd numbers is <odd_average> and the average for the even numbers is <even_average>

    Note: <> Represents the corresponding values.

Note

- The initial-action in a for loop can be a list of zero or more comma-separated expressions.
- The action-after-each-iteration in a for loop can be a list of zero or more comma-separated statements.
- Therefore, the following two for loops are correct. They are rarely used in practice, however.

```
for (int i = 1; i < 100; cout << (i++));


for (int i = 0, j = 0; (i + j < 10); i++, j++)
{
  // Do something
}
```


Note

- If the loop-repetition-condition in a for loop is omitted, it is implicitly true.
- Thus the statement given below in (a), which is an infinite loop, is correct.
- Nevertheless, it is better to use the equivalent loop shown in (b) to avoid confusion:

This is better

```
for ( ; ; )
{
   // do something
}
```

(a)

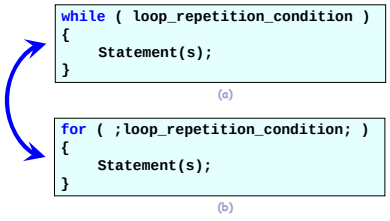
Equivalent

```
while (true)
{
   // do something
}
```

(b)

Which Loop to Use?

- The three forms of loop statements, while, do-while, and for, are expressively equivalent; that is, you can write a loop in any of these three forms.
- For example, a while loop in (a) in the following figure can always be converted into the following for loop in (b):



```
while ( loop_repetition_condition )
{
    Statement(s);
}
```

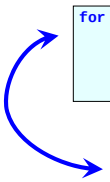
(a)

```
for ( ;loop_repetition_condition; )
{
    Statement(s);
}
```

(b)

Which Loop to Use? *cont.*

- A for loop in (a) in the following figure can generally be converted into the following while loop in (b) except in certain special cases



```
for (initial_action; loop_repetition_condition;
    action_after_each_iteration)
    {
       //loop body
    }
```

(a)

```
initial_action;
while ( loop_repetition_condition )
{
    //loop body
    action_after_each_iteration;
}
```

(b)

Recommendations

- **Use the one that is most understanding and comfortable for you.**
- **In general, a for loop may be used if the number of repetitions is known**
  - **For example, when you need to print a message 100 times.**
- **A while loop may be used if the number of repetitions is not known,**
  - **For example, reading the numbers until the input is 0.**
- **A do-while loop can be used to replace a while loop if the loop body has to be executed before testing the continuation condition.**

`break` and `continue`

- **`break;`**
  - terminates loop completely
  - Program execution continues with the statement that follows the loop
- **`continue;`**
  - It is used to skip remaining statements in a loop and start a new iteration

Practice! What is the output?

```
#include<iostream>
using namespace std;
int main()
{
   for (int i=0; i<10; i++)
   {
       if (i%2)
         continue;
       cout << i << endl;
   }//end for
   cout<<"done\n";
   return 0;
}//end main
```

```
0
2
4
6
8
done
```


Practice! What is the output?

```
#include<iostream>
using namespace std;
int main()
{
   for (int i=0; i<10; i++)
   {
       if (i%2)
          break;
       cout << i << endl;
   }//end for
   cout<<"done\n";
   return 0;
}//end main
```

```
0
done
```


Application of continue: find the average of *n* different positive numbers

```
#include <iostream>
using namespace std;
int main()
{
  cout<<"How many positive numbers you want to add: ";
  int n; cin>>n;
  double sum = 0, x;
  for(int i = 1; i<=n; i++)
  {
      cout<<"Enter data "<<i<<": ";
      cin>>x;
      if(x<0) //validating with `if` inside a for loop
      {
            cout<<"Invalid Entry! Try again.\n";
            i--;
            continue;
      }
      sum +=x;
  }
  cout<<"The average is: "<<sum/n<<endl;
}
```

Application of break: break the loop when the user enters 'over'

```
#include <iostream>
using namespace std;
#include <string>
int main()
{
   string name, maxName; double grade(1000), max(10);
   int counter=0;
   do{
       cout<<"Name: "; cin>>name;
       if(name=="over")
              break;
       cout<<"Grade: "; cin>>grade; counter++;
       if(counter == 1){
              max = grade; maxName = name;
       }
       else if(grade>max){
              max = grade; maxName = name;
       }
   }while(1);
   cout<<maxName<<" got the highest marks ("<<max<<").\n";
}/*modify the code to find the student scored lowest mark, the class average, total students etc.*/
```


Practice: Counting the positive numbers divisible by 5 but not by 15

```
#include <iostream>
using namespace std;
int main()
{
   int num = 0, counter=0;
   do{
       cout<<"Enter number: "; cin>>num;
       if(num<=0)
              break;
       if(num%5==0 && num%15!=0)
              counter++;
   }while(1);
   cout<<"You have entered "<<counter<<" numbers which are
   divisible by 5 but not by 15.\n";
}
```


Practice: validation using for loop: validate that an entered character is in between 'a' and 'd' inclusive

```
#include <iostream>
using namespace std;
int main()
{
   char choice('i');
   cout<<"Enter a chracter between a and d inclusive: ";
   cin>>choice;
   for( ;!(choice>='a' && choice<='d'); )
   {
       cout<<"Invalid chracter! ";
       cout<<"Enter a chracter between a and d inclusive: ";
       cin>>choice;
   }
   cout<<"You entered "<<choice<<".\n";
}
```

Note: In alphabets (a to z) are stored as ASCII equivalent integer numbers in the memory. These integer values are such that A<B<C …..<Z<a<b<c…..<z; in other words, uppercase A gets the minimum value and lowercase z gets the maximum value

Review

45) What is the output?

```
int x(5), y(6);
if (x > y)
   cout << “x is greater\n”;
else
    cout << “y is greater\n”;
```

a) x is greater
b) y is greater
c) x is greater
   y is greater
d) No output

Review

46) Assume x is 0. What is the output of the following statement?

```
if (x > 0)
   cout << (“x is greater than 0”);
else if (x < 0)
   cout << (“x is less than 0”);
else
   cout << (“x equals 0”);
```

a) x is greater than 0
b) x is less than 0
c) x equals 0
d) No output

Review

47) Which of the following statements will display “True!" given the following declaration

```
int i(100), j(0);
```

```
a) if (!(j<1)) cout << “True!\n";
b) if (i>0 || j>50) cout << “True!\n";
c) if (i<3) cout << “True!\n";
d) if ((j<i) && (i<=10)) cout << “True!\n";
```

Review

48) How many times the following code prints "Welcome to C++"?

```
int count = 0;
while (count < 10)
{
  cout << "Welcome to C++";
  count++;
}
```

a) 8
b) 9
c) 10
d) 11
e) 0

Review

49) How many times the following code prints "Welcome to C++"?

```
int count = 0;
while (count++ < 10)
{
  cout << "Welcome to C++";
}
```

a) 8
b) 9
c) 10
d) 11
e) 0

Review

50) How many times the following code prints "Welcome to C++"?

```
int count = 0;
do
{
   cout << "Welcome to C++";
} while (count++ < 10);
```

a) 8
b) 9
c) 10
d) 11
e) 0

Review

51) How many times the following code prints "Welcome to C++"?

```
int count = 0;
do
{
   cout << "Welcome to C++";
} while (++count < 10);
```

a) 8
b) 9
c) 10
d) 11
e) 0

Review

52) Analyze the following code.

```
int x = 1;
while (0 < x) && (x < 100)
   cout << x++;
```

a) The loop runs for ever.
b) The code does not compile because the loop body is not in the braces.
c) The code does not compile because (0 < x) && (x < 100) is not enclosed in a pair of parenthesis
d) The numbers 1 to 99 are displayed.
e) The numbers 2 to 100 are displayed.

Review

53) The following code will result in an infinite loop?

```
int balance = 10;
while (balance >= 1)
{
   if (balance < 9) continue;
   balance = balance - 9;
}
```

a) True
b) False

Review

54) What will be the value of the variable 'balance' after the following code is executed?

```
int balance = 10;
while (balance >= 1) {
   if (balance < 9) break;
       balance = balance - 9;
}
```

a) -1
b) 0
c) 1
d) 2

Review

55) What is the output of the following code segment?

```
for (int i = 0; i < 15; i++)
{
if (i % 4 == 1)
   cout << i << " ";
}
```

a) 1 3 5 7 9 11 13 15
b) 1 5 9 13
c) 1 5 9 13 16
d) 1 3 5 7 9 11 13
e) 1 4 8 12

Review

55) What is the output of the following code segment?

```
for (int i = 0; i < 15; i++)
{
if (i % 4 = 1)
   cout << i << " ";
}
```

a) 1 3 5 7 9 11 13 15
b) 1 5 9 13
c) 1 5 9 13 16
d) 1 3 5 7 9 11 13
e) The code will not compile

Answer Key

37. A
38. B
39. B
40. B
41. C
42. B
43. A
44. B
45. B
46. C
47. B
48. C
49. C
50. D
51. C
52. C
53. A
54. C
55. B
56. E

ES 1036 Programming Fundamentals

Modular Programming with Functions

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman, Ph.D, P.Eng., SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Outline

- **C++ functions**
  - **Function characteristics**
  - **Function definition**
  - **Function call**
- **Scope and storage of Objects**
  - **Local and Global scope**
  - **Automatic and Static storage**

C++ functions

Introducing Function with an example of a predefined function

```
#include  <iostream>
using namespace std;
#include <cmath>
int main ()
{
   double anyNumber(4), y(0);
   /*Note: in the following statement, the function sqrt has been called with an
   argument anyNumber. This function returns a value to the variable y.*/
   y = sqrt(anyNumber);
   /*In the following statement pow function is called with two arguments in the
   cout statement and the result is returned on the screen*/
   cout<<pow(y,3.0)<<endl;
   /*Question 1: Where have we defined these functions?
         Both sqrt and pow functions are predefined functions which have been
         defined in the <cmath> library

   Question 2: How did we define these functions?
         This is our learning goal in this section*/
}
```


Introducing Function: definition and types

- **Definition: A function is a collection of statements that are grouped together to perform an operation.**

Function Types:

- **Pre-defined**
  - **Stored in the standard libraries (examples: <cmath> etc.)**
  - **Functions are already defined**
  - **User only needs to know how to use it**
    - **Function name, return type, number & type of parameters**
  - **E.g. `double sin(double angle)`**
- **User defined**
  - **User writes the entire function definition (name, return type, number & type of parameters and body)**

Example on Pre-defined Functions

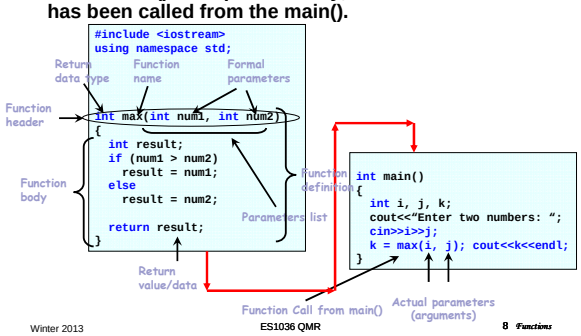
Function	Description	Example
abs(x)	Returns the absolute value of the argument	abs(-2) is 2
ceil(x)	x is rounded up to its nearest integer and returns this integer	ceil(2.1) is 3 ceil(-2.1) is -2
floor(x)	x is rounded down to its nearest integer and returns this integer	floor(2.1) is 2 floor(-2.1) is -3
exp(x)	Returns the exponential function of x	exp(1) is 2.71828
pow(x, y)	Returns a raised to power b (x^y)	pow(2.0, 3) is 8
log(x)	Returns the natural logarithm of x	log(2.71828) is 1.0
log10(x)	Returns the base-10 logarithm of x	log10(10.0) is 1
sqrt(x)	Returns the square root of x	sqrt(4.0) is 2
sin(x)	Returns the sine of x. x represents an angle in radians	sin(3.14159 / 2) is 1 sin(3.14159) is 0
cos(x)	Returns the cosine of x. x represents an angle in radians	cos(3.14159 / 2) is 0 cos(3.14159) is -1
tan(x)	Returns the tangent of x. x represents an angle in radians	tan(3.14159 / 4) is 1 tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double	fmod(2.4, 1.3) is 1.1

Introducing Function: User defined function

- A user-defined function is defined outside the main() function (either above main() or below main()).
- It follows certain grammatical (c++) rules discussed later
- In any program, the 'first' user-defined function-call is generated from the main() function.
  - After that, any user defined function can be called either from the main() function or from inside any other user-defined function, which has already been called from the main().

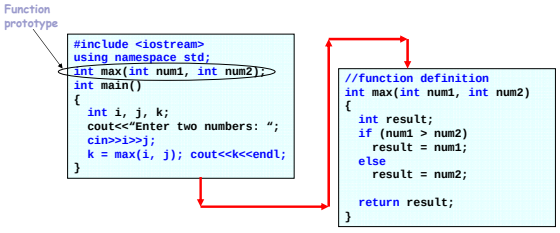
Introducing Function definition

- **Below, a function named max has been defined above main() as separate entity, and this function has been called from the main().**



Function definition can be put below main()

- **Below, the function named max has been defined underneath main() and it has been called from the main(). In this case, we need to use the function prototype (function declaration) above main().**



Terminology of Functions

- Function Prototype
  - Function declaration with its signature
- Function Call
- Function Arguments
  - Used in function call
- Function Definition
  - Function header
  - Function body
- Formal Parameters
  - used in function definition
- Function Header/ Signature
  - Return type
  - Function name
  - Parameter list
    - Type
    - Order
    - Number

Introducing Functions, *cont.*

- A function will always have a return data type, name and a formal parameter list; e.g.,

```
int gradePAverage(string name, int numberOfSubjects);
```

- Return data type:
  - A function may (or may not) return a value back to the calling function
  - When a function returns a value, the returnValueType is the data type of the value the function returns. In the above function prototype, int is the data type of the returned value of the function.
  - If the function does not return a value, the returnValueType is the keyword void.
- Function name
  - The function name can not be a keyword and it has to be a valid C++ Identifier (name).
- Formal parameter list
  - The variables defined after the function name enclosed by round bracket are known as formal parameters.
  - This is a Comma separated list of parameter types and names
  - If no parameter needs to be passed in the function the list can be left empty or the keyword void can be used there (`e.g. int main(void)`)

Introducing Functions, *cont.*

- **Function header/signature is the combination of the return type, function name and the parameter list.**
- **Function Definition contains Function header and Function body**
- **When a function is called (invoked), no value (such as void) or different types of values can be passed (pass by value) to a function. These values which are passed to a function are referred as actual parameters or arguments.**
  - The arguments must agree with the formal parameters in order, number and data type, but identifiers (variable names) in the function header and in the argument list can be different

Function prototype

- A function prototype is a function declaration
- Function Prototype contains
  - Return-type, function Name, and parameter list, e.g.,

```
int gradePAverage(string name, int numberOfSubjects);
```

  - The above prototype can also be written (without using the names of the parameters) as,

```
int gradePAverage(string, int); /* this is not a recommended
   approach */
```

- A function declaration/ prototype can be made either
  - outside and above the main function (recommended approach) or
  - Just before the function is called
- Exception: In the program (or code) if the function definition is placed above the main() function definition, function declaration (function prototype) becomes optional!

Why Functions?

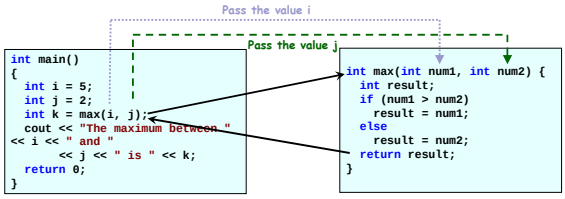
- Complex program can be broken down into sub tasks, each of which is easy to manage and implement
- This is called Modular programming approach
- The advantages to using functions, instead of writing the entire solution in main:
  - Multiple programmers can be involved
  - Testing/Debugging/Maintenance becomes easy
  - Code duplication is reduced
  - Information hiding: Hide the implementation from the user.

Calling (Invoking) Functions

- This program calls a Function `max` to return the greater of the two `int` values (look at the function prototype)

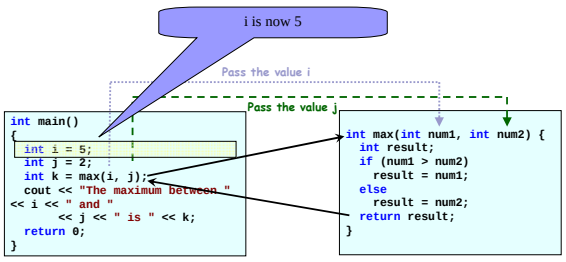
```
#include <iostream>
using namespace std;
int max(int num1, int num2); /*function protptype*/
/** Returns the max between two numbers */
int main()
{
  int i = 5;
  int j = 2;
  int k = max(i, j);
  cout << "The maximum between " << i << " and "
       << j << " is " << k;
  return 0;
}
/*function definition*/
int max(int num1, int num2) {
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result;
}
```


Calling Functions, *cont.*

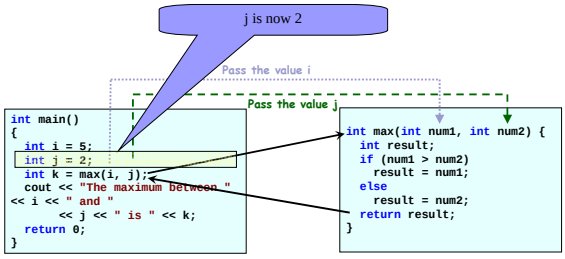


Trace Function Invocation

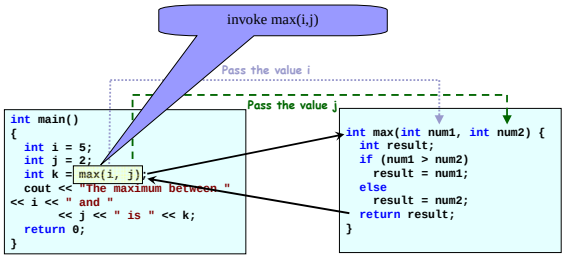
Draw the memory diagram as you trace the code



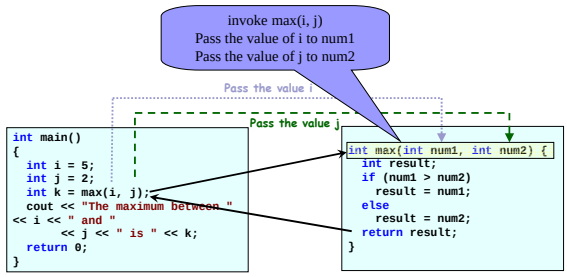
Trace Function Invocation



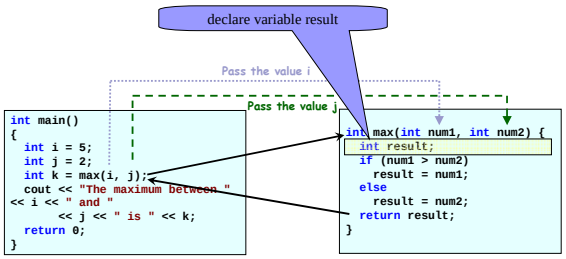
Trace Function Invocation



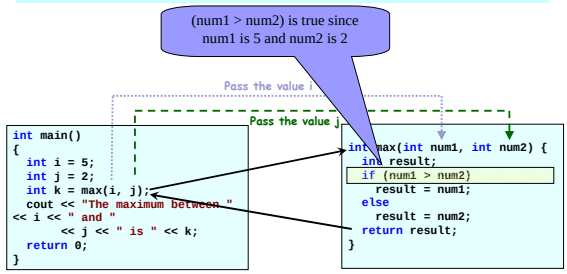
Trace Function Invocation



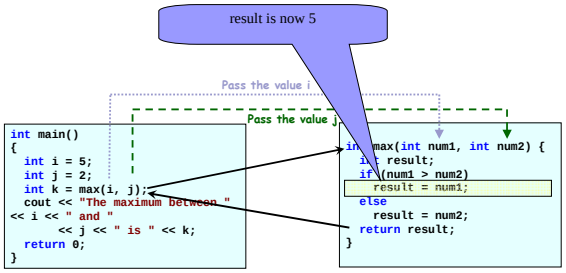
Trace Function Invocation



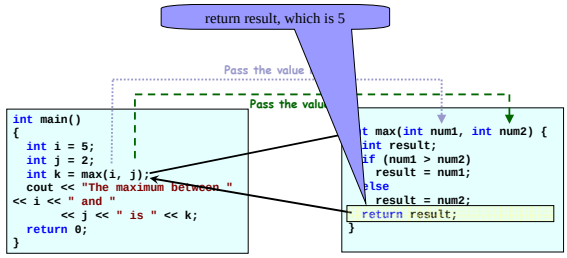
Trace Function Invocation



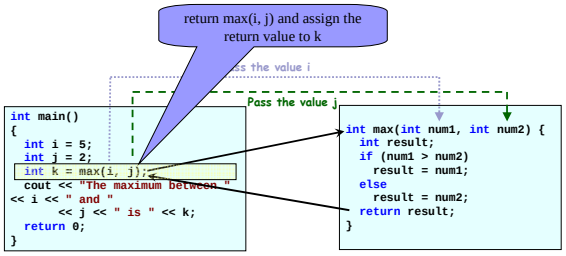
Trace Function Invocation



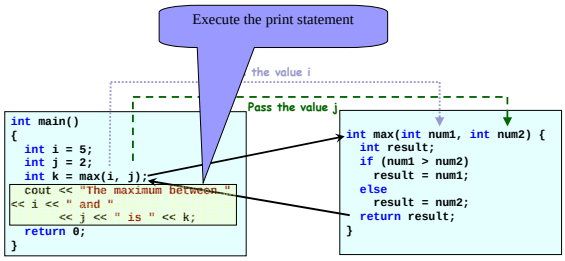
Trace Function Invocation



Trace Function Invocation

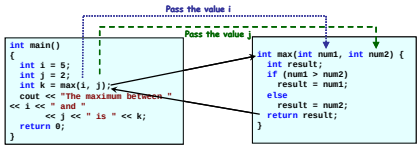


Trace Function Invocation



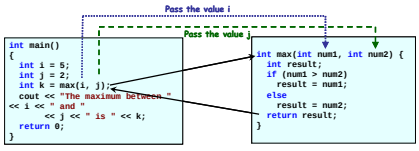
Using a Function

- "Call" (or "Invoke") a function when you want to use it
- When you "call" (or "invoke") a function, flow of control is transferred to its body
- Values of parameters are passed along
- Formal parameters must agree with arguments in order, number and data type, but parameter names can be different
- If you call a function from a cout-statement, the function call will have the higher precedence than the cout statement.



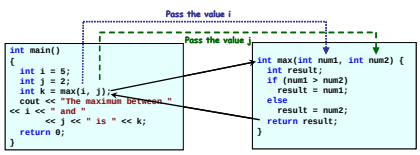
Using a Function, *cont.*

- At the end of body (or with an explicit `return` statement) flow of control is transferred back
- A function can return a single object to the calling program
  - The function header declares the type of object to be returned
  - A return statement is required in the body of the function



Parameter Passing

- Pass by value
  - Default behavior in C++
  - Formal parameter receives a copy of the argument
  - Changes to the formal parameter content do not affect the original object value that was passed via argument
- Pass by reference
  - Changing formal parameter content will change the orginal object value that was passed via argument!



void Functions

- A `void` function does not return a value to the calling program
- `return` statement is optional
- If a return statement is used, it has the following form
  - `return;`

Write a program to read 4 different integers and displays 4 lines of stars '*', each representing the values of these integers.

The solution in the main()

```
#include<iostream>
using namespace std;
int main() {
   int a , b , c , d;
   cin >> a >> b >> c >> d;
   //Display the first line
   for (int i=0; i<a; i++)
       cout << "*" ;
   cout << endl;
   //Display the second line
   for (int i=0; i<b; i++)
       cout << "*" ;
   cout << endl;
   //Display the third line
   for (int i=0; i<c; i++)
       cout << "*" ;
   cout << endl;
   //Display the fourth line
   for (int i=0; i<d; i++)
       cout << "*" ;
   cout << endl;
   return 0;
}
```


Solution with user defined Functions

```
#include<iostream>
using namespace std;

void print_stars (int length)
{
   for (int i=0; i<length; i++)
       cout << "*" ;
   cout << endl;
}
int main() {
   int a , b , c , d;
    cout<<"Enter four integers: ";
   cin >> a >> b >> c >> d;
   //Display the first line
   print_stars (a);
   //Display the second line
   print_stars (b);
   //Display the third line
   print_stars (c);
   //Display the fourth line
   print_stars (d);
   return 0;
}
```


Another Example with void type return

```
//output formatted date
void print_date(int mo, int day, int year)
{
   string month;
   switch(mo)
   {
      case 1:  month = "January"; break;
      case 2:  month = "February"; break;
       …
      case 12: month = "December"; break;
      default: month = "January";
   cout << month << ' ' << day << ',' << year
        << endl;
   return;   //return is optional
} //end print date
```

Homework: Write the main functin and call the print_date() function from the main() with appropriate parameters

Example: defining and calling a function

Problem: Write a function double addNnumbers(int n_numbers) **to add n numbers**

```
#include <iostream>
using namespace std;
double addNnumbers(int n_numbers);
int main()
{
   int n;
   cout<<"How many numbers do you
   want to add? ";
   cin>>n;
   cout<<"The result is
   "<<addNnumbers(n)<<endl;
   /*//The above line can be
   replaced by the following
   statements
   double result=addNnumbers(n);
   cout<<"The result is
   "<<result<<endl;*/
   return(0);
}//endmain
```

```
double addNnumbers(int n_numbers)
{
   double x(0),y;
   for(int i=1; i<=n_numbers; i++)
   {
      cout<<"Input number "<<i<<":
   ";
      cin>>y;
      x+=y;
   }
   return x;
}
//Watch out for function
   call in the main function
```


Example: defining and calling a function

Problem: Write a function void addNnumbers(int n_numbers) **to add n numbers**

```
#include <iostream>
using namespace std;
void addNnumbers(int n_numbers);
int main()
{
   int n;
   cout<<"How many numbers do you
   want to add? ";
   cin>>n;
   addNnumbers(n);
   return(0);
}//endmain

//Watch out for function
   call
```

```
void addNnumbers(int n_numbers)
{
   double x(0),y;
   for(int i=1; i<=n_numbers; i++)
   {
      cout<<"Input number "<<i<<":
   ";
      cin>>y;
      x+=y;
   }
  cout<<"The result is "<<x<<endl;
  return;
}
```

Example: defining and calling a function

Problem: Write a function double addNnumbers(void) **to add n numbers**

```
#include <iostream>
using namespace std;
double addNnumbers();
int main()
{
   cout<<"The result is
   "<<addNnumbers()
   <<endl;
   return(0);
}//endmain

//Watch out for
  function call
```

```
double addNnumbers()
{
  double x(0),y;
  int n;
  cout<<"How many numbers do you want to
   add? ";
  cin>>n;
  for(int i=1; i<=n; i++)
   {
      cout<<"Input number "<<i<<": ";
      cin>>y;
      x+=y;
   }
 return x;
}
```


Example: defining and calling a function

Problem: Write a function void addNnumbers(void) **to add n numbers**

```
#include <iostream>
using namespace std;
void addNnumbers();
int main()
{
   addNnumbers();
   cout<<endl;
   return(0);
}//endmain
```

```
void addNnumbers()
{
  double x(0),y;
  int n;
  cout<<"How many number do you want to add? ";
  cin>>n;
  for(int i=1; i<=n; i++)
   {
      cout<<"Input number "<<i<<": ";
      cin>>y;
      x+=y;
   }
  cout<<"The result is "<<x<<endl;
  return;
}
```


Example: Calling a user defined function from inside another user defined function

```
/*Here the function
  multiplyBy2 is called
  from inside the
  function add3Numbers*/

#include <iostream>
using namespace std;
int add3Numbers();
int multiplyBy2(int a);
int main()
{
  int x = add3Numbers();

  cout << "The result
  is: "<<x<<endl;
}
```

```
int add3Numbers()
{
  int sum=0, n;
  for(int i=1; i<=3; i++)
  {
      cout<<"Enter the next
  number: ";
      cin>>n;
      sum=sum+multiplyBy2(n);
  }
  return sum;
}
int multiplyBy2(int a)
{
  return (2*a);
}
```


Function overloading

- If two or more functions have the same name but different parameter lists, we call these functions as overloaded functions, and the procedure is known as function overloading.
- In this case, the return-data type can be same or different.
- Example:

```
void myFunction(int);
int myFunction(int, double);
double myFunction(string);
void myFunction(void);
```


Function overloading example

```
#include  <iostream>
using namespace std;

//Function definitions
void myFunction(int x) {cout<<x<<endl;}
int myFunction(int x, double y){return x%(int)y;}
double myFunction(char k){cout<<endl<<k; return 10;}
void myFunction(void){cout<<"\nDo nothing\n";}

int main ()
{
  //Function calls
  myFunction(5);
  cout<<myFunction(4, 5.5);
  cout<<myFunction('c');
  myFunction();
}
```


Ambiguous Invocation in overloaded functions

- Sometimes there may be two or more possible matches for an invocation of an overloaded function
- In this case, the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*.
- Ambiguous invocation results in a compilation error.

Example: Ambiguous Invocation

```
#include <iostream>
using namespace std;
int maxNumber(int num1, double num2);
double maxNumber(double num1, int num2);
int main()
{
  cout << maxNumber(1, 2) << endl;//error
  return 0;
}
```

```
int maxNumber(int num1, double num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
double maxNumber(double num1, int
   num2)
{
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```


Default Arguments

- C++ allows us to define functions with default parameter values.
- The default values are passed to the parameters when a function is invoked without the arguments.
- In this case all function parameters have to have default values
- Note that the function definition has to be placed before main function without using any function declaration.

```
#include <iostream>
using namespace std;
void printArea(double radius = 1)
{
  double area = radius * radius *
   3.14159;
  cout << "area is " << area <<
   endl;
}
int main()
{
  printArea();
  printArea(4);
  return 0;
}
```

```
//output:
area is 3.14159
area is 50.2654
```


Default Arguments continues..

- Also, note that if there is any function declaration, the declaration should contain the default value while the function header should not contain any default value. See the example.
- If there is more than one parameter in the parameter-list that contains default parameters, these default parameters should be placed at the end of the list. (In class example)

```
#include <iostream>
using namespace std;
void printArea(double radius = 1);

int main()
{
  printArea();
  printArea(4);
  return 0;
}
void printArea(double radius)
{
  double area = radius * radius *
   3.14159;
  cout << "area is " << area << endl;
}
```

```
//output:
area is 3.14159
area is 50.2654
```


Pre-defined Function: `int rand(void)`

- **rand function is a standard library function, (available in C-standard library <stdlib.h>; for C++, we don't need to add any library other than <iostream>) which returns a pseudo-random number between 0 and RAND_MAX, when it is invoked.**
- **RAND_MAX is a symbolic constant defined in the standard library. It has a value of at least 32767 (2^{15}-1; when the processor uses two bytes = 16 bits of memory for the number), depending on the specific type of computer being used**

Generate a random number using rand()

- **The statement int x = rand(); assigns a random number between 0 and RAND_MAX to x.**
- **A common way to generate a random number between 0 and some integer is to use the modulus operator. To generate a random number between 0 and 5 inclusive, use rand()%6.**
- **If random numbers starting at a value other than zero are to be generated, the formula is modified by adding the starting value. For example, the roll of a die will yield a 1,2,3,4,5, or 6. To simulate the roll of a die, use 1 + rand() % 6**
- **In general, to generate numbers from min to max inclusive, use min + rand() % (max – min + 1)**

Program example with rand()

```
#include <iostream>
using namespace std;
int main(void)
{
  int i;
  for(i = 0; i < 5; i++)
      cout<<rand()<<endl;
}
```

Output
First run:
41
18467
6334
26500
19169
Second run:
41
18467
6334
26500
19169

Note: same output for both runs

Predefined function: `void srand (unsigned int)`

- The C-library function (in C++ we don't need to add any library function other than <iostream>) srand is used to get different random numbers each time the program is run.
- The srand function is said to "seed" the function rand.
- It takes an unsigned integer argument. A different sequence of numbers can be generated by using different seed values.

Program example with srand()

```
#include <iostream>
using namespace std;
int main(void)
{
  int i;
  int seed;
  cout<<"Enter a seed value
   for srand: ";
  cin>>seed;
  srand(seed);
  for(i = 0; i < 5; i++)
      cout<<rand()<<endl;
}
```

Output
First run:
Enter a seed value for srand:
45
185
32645
5926
3942
14259

Second run:
Enter a seed value for srand:
6
58
6673
30119
15745
5206

Predefined function: `time_t time(time_t*)`

- Incude <ctime> library
- The time function is commonly used to enter seed values automatically.
- The time function returns the time of day in seconds.
- It is actually the number of seconds since 1 January 1970.
- The return value changes in each second.
- Using "NULL" as the parameter causes the time function to return the value of the seed.

Program example with srand() and time()

```
#include <iostream>
#include <ctime>
using namespace std;
int main(void)
{
  int i;
  srand((unsigned int)time(NULL));
  for(i = 0; i < 5; i++)
      cout<<rand()%5<<endl;
}
```


Example: Factorial - n!

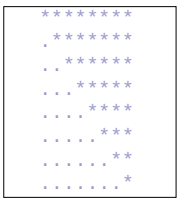
- n! = n*(n-1)*(n-2)*…*1
- n is a positive integer
- 0! is 1 by definition
- Lets define a function that
  - Takes one integer parameter , say n
  - Calculate n!, say nFact
  - Returns nFact

Example: Factorial - n!

```
int factorial (int n)
{
   int nFact = 1;
   while (n > 1)
   {
      nFact = nFact * n;
      n--;
   }
   return (nFact);
}
```

Example Problem

- Given a number *n*, display a *n*x*n* square as follows



Solution with for-loops

```
#include <iostream>
using namespace std;
int main()
{
   int n, i, j, k;
   cout<<"Input n: ";
   cin>>n;
   cout<<endl<<endl;
   for(i=1; i<=n; i++)
   {
      for(j=1; j<i; j++)
       cout<<".";
      for(k=j; k<=n; k++)
       cout<<"*";
      cout<<endl;
   }
   cout<<endl<<endl;
   return 0;
}
```


Solution with function

```
#include<iostream>
using namespace std;
int main() {
   void PrintChar(int m, char c);
   int n;
   cin >> n;
   for (int i=0 ; i < n ; i++)
   {
       PrintChar (i,'.');
       PrintChar (n-i, '*');
       cout << endl;
   }
   return 0;
}
void PrintChar(int m, char c)
{
   // Display the value of c m times
   for (int i=0 ; i<m; i++)
   {
       cout << c ;
   }
}
```

```
GetInput (the value of n)
Loop with i, n times
{
   PrintChars(i,'.');
   PrintChars( n-i, '*');
   Print newline
}
```

```
PrintChar(int m, char c)
{
   loop m times
   {
       cout << char;
   }
}
```


Read a line using getline() function (for lab only)

- Syntax:

  void getline(cin, string_variable_name);
  - Note
    - There are more to the above syntax; the above is good enough for our course.
    - The statement *cin.ignore();* will always be required before using the getline function, if any cin operation is carried out earlier in the same programming block. If that's not the case, we don't need to use cin.ignore() statement.
    - *cin.ignore();* statement skips the newline ('\n') character in the input buffer ()

Program example-1 with getline() (for lab only)

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favourite sports? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

Note: This code does not need cin.ignore() statement because it contains all getline function calls (no cin operation prior to any getline function call)

Program example-2 with getline() (for lab only)

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  int ID; string mystr;
  cout << "What's your ID number? ";
  cin>>ID;
  cout << "What's your name? ";
  cin.ignore();
  getline (cin, mystr);
  cout << "Hello " << mystr << "! Your ID is:
   "<<ID<<".\n";
  return 0;
}
```

Note: comment-out the cin.ignore() statement and run the code to see the output

Program example-3 with getline() (for lab only)

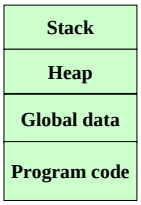
```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  int ID; string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "What's your ID number? ";
  cin>>ID;
  cout << "Hello " << mystr << "! Your ID is:
   "<<ID<<".\n";
  return 0;
}
```

Note: This code does not require cin.ignore() because getline function is called before any other cin statement.

Scope and Storage of Objects

Data storage and Memory diagram

- Four sections of the memory areas store four different types data
- Stack
  - Stores Local variables and formal parameters
  - Managed by the compiler
- Heap
  - Stores dynamic variables
  - Managed by storage allocator
- Global Data
  - Stores global variables
- Program code
  - Stores Other program data



Scope of an Object

```
#include <iostream>
using namespace std;
int main()
{
  int n(10);
  while (n)
  {
    int i(0);
    i = i + n--;
  }
  cout << i; // error! Why?
  return 0;
}//end main
```

Scope of **i**

Scope of **n**

Scope of an Object

- Scope refers to the region of the program in which an object can be legally used
  - Local scope
  - Global scope

```
#include <iostream>
int main()
{
  int n(10);
  while (n)
  {
    int i(0);
    i = i + n--;
  }
  std::cout << i; // error! Why?
  return 0;
}//end main
```

Scope of **i**

Scope of **n**

Local scope

- **Local scope, also called Block scope: is the scope of an object that is declared inside a block (this includes function parameters)**
- **Extends from the point of declaration to the end of the block**
- **Can be accessed only within the block that declares it**
- **This includes:**
  - **An object declared within '{' and '}'**
  - **An object declared within a function**
  - **Formal Parameters of a function**
  - **Any parameter declared in the for-header has the scope of that for-structure**

Global scope

- **Global scope, also called File scope: is the scope of an object that is declared outside of all namespaces, functions (including main function) and classes**
- **Extends from point of declaration to the end of the entire file containing the program code**
- **Can be accessed from any part within the program file**

Global Scope

```
#include <iostream>
using namespace std;
int n(9);
void add_to_n(int k)
{
  n = n + k;
}
int main()
{
  int i(5);
  cout << n-- << endl; // prints 9
  add_to_n(i);
  cout << n << endl; // output?
  return 0;
}//end main
```

Scope of **n**

Scope of a Function

- **Functions are objects too!**
- **They are declared outside `main` function**
- **They have global scope!**
- **But only from point of declaration onwards**
- **Use prototypes to get around this limitation**

Scope of `add_to_n`

```
#include <iostream>
int n(10);
void add_to_n(int k)
{
  n = n + k;
}
int main()
{
  int i(5);
  std::cout << n << std::endl; // prints 10
  add_to_n(i);
  std::cout << n << std::endl; // what?
  return 0;
}//end main
```

Scope of **`add_to_n`**

Storage Class of an Object

- **Storage class refers to the lifetime of an object**
- **The lifetime of an object is the time during program execution in which an identifier actually has memory allocated to it**
  - **Automatic Storage Class**
  - **Static Storage Class**

Automatic Storage Class

- **Default for local variables, their storage is created (allocated) when control enters the function or a block**
- **Local variables are "alive" while function (or a block) is executing**
- **Their storage is destroyed (deallocated) when function (or a block) exits**

Static Storage Class

- Any object with global scope, its lifetime is the lifetime of the entire program
- Any object with the static qualifier, its storage remains allocated throughout execution of the entire program
- Initialized once when their declarations are encountered; by default this initialized value is zero for numerical (int, double or float) data types and it is NULL (‘ ‘) for non-numerical (e.g., character) type data.
- Retain their values when the function returns to its caller
- It does not affect the scope of the variable
- All numerical global variables are initialized with zero value by default

What is the Output?

Local storage

Static Storage

```
// Storage Class

#include <iostream>
using namespace std;
void donothing();
int main()
{
   donothing();
   donothing();
   donothing();
   return(0);
}//endmain

void donothing()
{
   int x(0);
   static int y;/*it
is automatically
initialized to zero*/
   x++;
   y++;
   cout <<“x is ”<< x
     << “, y is ” << y
     << endl;
   return;
}//end donothing
```

Global Scope

Block Scope

Output?

```
// Storage Class

#include <iostream>
using namespace std;
void donothing();
int main()
{
   donothing();
   donothing();
   donothing();
   return(0);
}//endmain

void donothing()
{
   int x(0);
   static int y(0);
   y = 0;
   x++;
   y++;
   cout <<“x is ”<< x
     << “, y is ” << y
     << endl;
   return;
}//end donothing
```

Global Scope

Block Scope

Static Storage

Local storage

Output?

```
// Storage Class
#include <iostream>
using namespace std;
char c; /*c gets
automatically initialized to ‘ ‘ */
void donothing();
int main()
{
   donothing();
   donothing();
   donothing();
   return(0);
}//endmain

void donothing()
{
   int x(0);
   static int y; /*y
gets automatically
initialized to zero*/
   x++;
   y++;
   cout <<“x is”<<c<<x
     <<“, y is”<<c<< y
     << endl;
   return;
}//end donothing
```

Global Scope

Block Scope

Static Storage

Local storage

Example with static modifier

```
#include <iostream>
using namespace std;
int f1(int);
int f2(int);
int main()
{
    int x(3); cout<<x<<endl;
    x = f1(x); cout<<x<<endl;
    x = f2(x); cout<<x<<endl;
    cout<<x<<endl;
}
int f1(int x)
{
    static int y=2;
    return ++y+x;
}
int f2(int x)
{
    static int y=2;
    return x+y;
}
```

- In this example, the static variable y in function f1 and the static variable y in function f2 are different from each other because of their scopes.
- Both of these two variables will be given different spaces in the RAM with two different flags and there will be no conflict between these two.

Application example on static modifier

- Multiplying numbers using calculators:

```
#include <iostream>
using namespace std;

double product(double);


double product(double c)
{
    static double k = 1;
    k = k*c;
    return k;
}
```

```
int main()
{
  cout<<"Calculating product:\n";
  double x=1;
  double z=0;
  while(x)
  {
    cout<<"Enter the next number to multiply;
     enter zero to end: ";
    cin>>x;
    if(x==0)
       break;
   z=product(x);
   cout<<"The result is: "<<z<<endl;
  }
  cout<<"The final result is: "<<z<<endl;
}
```

Output?

```
#include <iostream>
using namespace std;
int i;
int main()
{
   int i = 1;
   cout << i << endl;
   {
      cout << i << endl;
      int i=2;
      cout << i << endl;
   }
   cout << i << endl;
   cout<< ::i<<endl;
   return 0;
}
```


FYI: Accessing the Global variable

- A global variable can be accessed anywhere in the program from the point of declaration
- If a local variable name is same as a global variable name, one can access the global variable using ::globalVariable
- The :: operator is known as the unary scope resolution operator

FYI: Output?

```
#include <iostream>
using namespace std;
int x;
int main()
{
   int x(3);
   static int z;
   cout<<x<<endl;
   cout<<::x<<endl;
   for(int x=2;x<4;x++)
      cout<<"loop inside: "<<(++z+x+::x++)<<endl;
   cout<<x<<endl;
}
```


Scope and storage: Some tips

- It is easy to declare a variable globally once and use it in all functions without re-declaring it. However, this is a bad practice because it could lead to errors that are hard to debug.
- It is commonly acceptable to declare a local variable with the same name multiple times in different no-nesting blocks in a function (see the next slide for program example)
- It is not a good practice to declare a local variable twice in nested blocks (see the example code in the previous slide). The program can compile and run, but it is easy to make mistakes.

Example: using the same local variable name in different non-nesting blocks

```
#include <iostream>
using namespace std;
int main()
{
      cout<<"Loop 1"<<endl;
      for(int i=2;i<4;i++)
            cout<<"loop 1: "<<i<<endl;
      cout<<"Loop 2"<<endl;
      for(int i=4;i>2;i--)
            cout<<"loop 2: "<<i<<endl;
}
```


Review

56) The parameter list contains all objects (variables) used by the function?

a) True
b) False

Review

57) Which of the following is a valid definition for a function called "cube" ?

a) `function cube(double x){ ... }`
b) `double cube(double x){ ... }`
c) `double cube(x){ ... }`
d) `cube(double x){ ... }`

Review

58) Suppose your function does not return any value, which of the following keywords can be used as a return type?

a) float
b) unsigned short
c) int
d) double
e) void

Review

59) The signature of a function consists of _____.

a) parameter list
b) function name
c) function name and parameter list
d) return type, function name, and parameter list

Review

60) When you invoke a function with a parameter, the value of the argument is passed to the parameter. This is referred to as _____.

a) function invocation
b) call by name
c) call by reference
d) call by value

Review

61) What will be the output of the function call nPrint('a', 4)?

```
void nPrint(char ch, int n)
{
  while (n > 0)
  {
    cout << ch;
    n--;
  }
}
```

a) invalid call
b) aaa
c) aaaaa
d) aaaa

Review

62) Breaking a program in to a manageable sized functions is called

a) modular programming
b) break up programming
c) high-level programming
d) functional programming
e) None of the above

Review

63) When a function is called, flow of control moves to the function's prototype.

a) True
b) False

Review

64) A _____ variable is defined inside the body of a function and is not accessible outside that function.

a) reference
b) counter
c) local
d) global
e) None of the above

Review

65) Functions are ideal for use in menu-driven programs. When a user selects a menu item, the program can _____ the appropriate function to carry out the user's choice.

a) define
b) declare
c) call
d) return
e) None of the above

Review

66) A function is executed when it is

a) defined
b) declared
c) prototyped
d) called
e) None of the above

Review

67) The group of the C++ statements in which an object can be used is its

a) Storage class
b) Scope
c) Declaration
d) Formal argument

Review

68) What does storage class say about an object

a) Where the object can be used
b) When the object gets created and destroyed
c) Both a and b

Review

69) An object can have both block scope and static storage class

a) **True**
b) **False**

Review

70) An object can have both global scope and local storage class

a) **True**
b) **False**

Review

71) Which combination of storage class and scope is not possible

a) **Block scope, local storage class**
b) **Block scope, static storage class**
c) **File scope, local storage class**
d) **File scope, static storage class**

Review

72) If an object has local scope, it can only have local storage class

a) **True**
b) **False**

Answer Key

56. B
57. B
58. E
59. D
60. D
61. D
62. A
63. B
64. C
65. C
66. D
67. B
68. B
69. A
70. B
71. C
72. B

ES 1036 Programming Fundamentals

Arrays

Dr. Quazi M. Rahman, Ph.D, P.Eng., SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Outline

- **What is an array?**
- **how to declare an array**
- **why an array is necessary in programming**
- **how to access array elements using indexed variables**
- **initializing the values in an array**
- **declaring and creating multidimensional arrays**

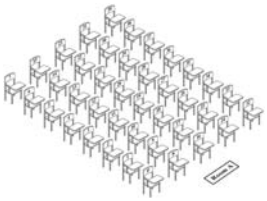
Array?

- An **array** is a set of consecutive memory locations used to store **same types of data**.
- Each item in an array is called an **element**
- The number of elements in an array is called the **dimension** or the **size** of the array.
- An array shares a common identifier and data type; e.g., `double myArray[10];`

Memory Address (in bytes)	Memory Content	Variable Name
1000	2	myArray[0]
1008	4.5	myArray[1]
1016	2.1	myArray[2]
1024	3.5	myArray[3]
1032	8	myArray[4]
1040	5.6	myArray[5]
1048	0.2229	myArray[6]
1056	9.5	myArray[7]
1064	10	myArray[8]
1072	2.5	myArray[9]

Array: An Analogy

- Consider a classroom with 40 desks in fixed position
  - Desks are wasted if less than 40 students
  - Not enough desks if more than 40 students
- An array is like the classroom
  - Each desk represents an array location



Array declaration

Syntax: `dataType arrayIdentifier[arraySize];`

Example:

```
double myArray[10];
```

- C++ requires that the array size must be a constant expression. This is called static array declaration. For example, the following code segment is illegal:

```
int size = 4; // not a constant expression
double myArray[size]; // Wrong
```

Q. From which statement will the error be generated?

- But it would be OK, if size is a constant expression as follows:

```
const int size = 4; // a constant expression
double myArray[size]; // Correct
```


Indexing Array elements

- Individual elements of an array are specified using offsets referred to as index or subscripts.
- The index of the first element of an array always has a subscript value of zero, e.g., myArray[0] holds the first element of the array myArray[10].
- The index of the last element of an array always has a subscript value of [dimension-1]. For the array 'myArray[10]' the last element goes to myArray [9].

Memory Address (in bytes)	Memory Content	Variable Name
1000	2	myArray[0]
1008	4.5	myArray[1]
1016	2.1	myArray[2]
1024	3.5	myArray[3]
1032	8	myArray[4]
1040	5.6	myArray[5]
1048	0.2229	myArray[6]
1056	9.5	myArray[7]
1064	10	myArray[8]
1072	2.5	myArray[9]

Arithmetic with indexed array elements

- Each element in the array is represented using the following syntax, known as an indexed variable: `arrayName[index];`
- After an array is created, an indexed variable can be used in the same way as a regular variable. Index can be an integer expression.
- For example, the following code segment adds the values of myArray[0] and myArray[1], and assign the new value to myArray[2].

```
int i = 0;
myArray[2] = myArray[0] + myArray[i+1];
```

Memory Address (in bytes)	Memory Content	Variable Name
1000	2	myArray[0]
1008	4.5	myArray[1]
1016	6.5	myArray[2]
1024	3.5	myArray[3]
1032	8	myArray[4]
1040	5.6	myArray[5]
1048	0.2229	myArray[6]
1056	9.5	myArray[7]
1064	10	myArray[8]
1072	2.5	myArray[9]

Why arrays?

- **Write a C++ program to read four integers and display them in reverse order**

```
#include <iostream>
using namespace std;
int main()
{
   int a, b, c, d; cout<<"Enter four integers: ";
   cin >> a >> b >> c >> d;
   cout<<"The integers in reverse order..\n";
   cout << d << " " << c << " " << b << " " << a << endl;
   return 0;
}
```

  - **What if we need to read 100 different integers and display them in reverse order?**

Let's use Arrays

```
#include <iostream>
using namespace std;
int main()
{
   int a[100]; // array to hold 100 values all of type integer
               // get 100 integer values

   for (int i=0; i<100 ; i++)
      cin >> a[i]; // it would be better if we read them
                   // from a file

   // Display them in the reverse order
   for (int i=99; i>=0 ; i--)
      cout << a[i] << " ";
   cout << endl;
   return 0;
}
```


No Boundary Checking

- C++ does not check array's boundary.
- So, accessing array elements using subscripts outside the declared boundary does not cause syntax errors.
- E.g., array declaration `double myArray[10];` sets the boundary from 0 to 9 and in this case accessing `myArray[-1]` and `myArray[10]` will not cause any syntax error.
- But the operating system might report a memory access violation, which is a run time error.
  - **If it does not result in memory access violation it may result in logical error.**

Array Initialization

- Initialization list : Declare, create, initialize in one step

```
dataType arrayName[arraySize] = {value0, value1, ..., valuek};//this is also known as Shorthand method
double myArray[4] = {1.9, 2.9, 3.4, 3.5};
```

- The above initialization can be made using indexed array elements of a declared array.

```
double myArray[4];//declaration
myArray[0] = 1.9;
myArray[1] = 2.9;
myArray[2] = 3.4;
myArray[3] = 3.5;
```

Initialization; one can use a Loop to initialize these elements

  - **Question: Declare an array `int anyArray[6]` and initialize each element using a loop with integer multiple of 12 starting from 12 (i.e., 12, 24, 36 etc).**

CAUTION

- **While using the initialization list (shorthand method), one has to declare, create, and initialize the array (all) in one statement.**
- **Splitting it would cause a syntax error.**
- **For example, the following code segment is syntactically incorrect; a compilation error will be generated in the second statement:**

```
double myArray[4];
myArray = {1.9, 2.9, 3.4, 3.5};
```

Implicit Size

- **C++ allows us to omit the array size when declaring and creating an array using an initialization list.**
- **For example, the following declaration is fine:**

```
double myArray[] = {1.9, 2.9, 3.4, 3.5};
```

- **C++ automatically figures out how many elements are in the array.**

More on Initialization

- C++ allows one to initialize a part of the array.
  - `double myArray[4] = {1.9, 2.9};`
- The other two elements will be set to zero. (For numerical type array the missing elements get zero values, and for non-numerical type array the missing elements get NULL values).
- An array cannot have more initializing values than its size
  - `int odd[3]={3, 5, 7, 9};` //is wrong!
- But it can have less as shown in the above example
  - `char myArray[3]={'q', '5'};` //is acceptable!
  - `int bigarray[3000]={1};` //is acceptable!
- Note that if an array is declared, but not initialized, all its elements will contain useless data items like all other local variables.

Initializing Character Arrays

```
char city[] =  {'L','o','n','d','o','n'};//size = 6
```

- **Any string is considered as a character array and so a character array can be initialized with a single string by using the following syntax. In this case, no curly braces are used.**

```
char city[] =  "London";//Array size is 7
```

- **The above two statements are equivalent, however, in the second one (the string) C++ adds the character '\0' , called the 'end of string character' or NULL character, to indicate the end of the string, as shown below:**

'L'	'o'	'n'	'd'	'o'	'n'	'\0'
city[0]	city[1]	city[2]	city[3]	city[4]	city[5]	city[6]

How to convert a string to its uppercase (or lowercase)

```
#include <iostream>
using namespace std;
#include <string>
int main ()   {
  string k = "hello world!";
  int i = 0;
/* stringName.length() gives the length of
  the string called stringName */
  while (i<k.length())
{
     k[i] = toupper(k[i]);
     i++;
  }
cout<<k<<endl;
}  //homework; rewrite the code for lowercase conversion
```


How to convert a string to its uppercase (another one)

```
#include <iostream>
using namespace std;
int main()
{
  char k[10]={};
  cout<<"Enter a string: ";
  cin>>k;
  //for(int i = 0; i<sizeof(k); i++)
  for(int i = 0; k[i] != '\0'; i++)
     k[i] = toupper(k[i]);
  cout<<k<<endl;
}/* Disadvantage: size needs to be fixed first, and no white space
   chracter inside the string can be accommodated ! */
```


Initializing string Arrays

- **One has to include `<string>` library**
- **Examples:**

```
string city[] =  {"London"};//Array size is 1
string x[3] =  {"London", "e"};//Array size?
string x[] =  {"London", "e", ","};//Array size?
```

Example : Initializing arrays with random values

- The following loop initializes the array myArray with random values between 0 and 99:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  myArray[i] = rand() % 100;
}
```

- Review: rand() returns a random number between 0 and RAND_MAX; RAND_MAX is a platform dependent constant. In visual C++ it is 32767.
- Review: For rand() function, we don't need to add any special library file.

Example : Printing arrays

- To print an array, one has to print each element in the array using a loop:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  cout << myArray[i] << " ";
}
```


Example : Printing Character Array

- For a character array, it can be printed using one print statement. For example, the following code displays London:

```
char city[] = "London";
cout << city;
```

- But the following code displays London followed by some special characters:

```
char city[] = {'L','o','n','d','o','n'};
cout << city;
```

- To take care of this problem one has to add a null character ('\0') at the end of the array (as shown below) or use a loop to print one character at a time:

```
char city[] = {'L','o','n','d','o','n', '\0'};
cout << city;
```


Copying Arrays

- Can we copy one array to another by using a syntax like this?

```
list = myArray; // This is not allowed in C++.
```

- We need to copy each individual element from one array to the other using loop:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  list[i] = myArray[i];
}
```

- *Also, you can do it by assigning each individual element from the source to destination array. But it's not a recommended approach since it takes lots of assignment expressions for a big array.*

Example 1: Summing All Elements

- The following code segment sums all the elements of an array and displays the result:

```
int main(){
      double total = 0, myArray[3]={3, 2,1};
      for (int i = 0; i < 3; i++)
      {
            total += myArray[i];
      }
      cout<<"Total = "<<total<<endl;
      return 0;
}
```


Example 2: Finding the Largest Element

- Use a variable named max to store the largest element. Initialize max with myArray[0] if myArray[0] has already been initialized.
  - What do you need to do if myArray is not initialized at the first place
- To find the largest element in the array myArray, compare each element in myArray with max, update max if the element is greater than max.

```
int main(){
      double myArray[3] = {4.5,3,6}, max = myArray[0];
      for (int i = 1; i < 3; i++)
      {
            if (myArray[i] > max)
                  max = myArray[i];
      }
      cout<<"The max value is: "<<max<<endl;
}//Question: Modify the code to find out the smallest element
```

Example 3: Populate (enter values) an array and print the elements

```
#include <iostream>
using namespace std;
const int N = 5;
int main(void)
{
   int number[N], i;
    cout<<"Populate the array\n";
   for ( i = 0; i < N; i++ ) {
      cout<<"Enter element "<<i<<": ";
      cin>>number[i];
      }
   cout<<"Printing the array elements:\n";
   for ( i = 0; i < N; i++ ) {
      cout<<"number["<<i<<"] = "<<number[i]<<endl;
   }
   return 0;
}
```


Example 4: Smallest, Largest, Range and Mean

```
const int N = 12;
int main(void)
{
   int number[N] = {  2,  7, 85, 12,  7,  0,  9,  2, 64,
  16,  8,  -1};
   int i, smallest, largest, range, sum; float mean;
   smallest = largest = sum = number[0];
   for( i = 1; i < N; i++) {
      sum += number[i];
      if ( number[i] < smallest )
         smallest = number [i];
      else if (number[i] > largest )
         largest = number [i];
      }/* end for loop */
   range = largest - smallest; mean = (float)sum / N;
   cout<<"The smallest number is "<<smallest<<endl;
   cout<<"The largest number is "<<largest<<endl;
   cout<<"The range is "<<range<<endl;
   cout<<"The mean is "<<mean<<endl;
   return 0;
}
```


Array Name Vs Array Element

- **Array elements have the same base type**
  - **E.g. In double myArray[10]; all the elements myArray[0] , myArray[1] , … , myArray[9] etc are double type**
- **Array name itself is a memory address of the first element of the array**
  - **E.g. the object myArray holds the starting memory address of the array myArray[10] ;**
  - **cout<<myArray<<endl; will print 1000.**

Memory Address (in bytes)	Memory Content	Variable Name
1000	2	myArray[0]
1008	4.5	myArray[1]
1016	2.1	myArray[2]
1024	3.5	myArray[3]
1032	8	myArray[4]
1040	5.6	myArray[5]
1048	0.2229	myArray[6]
1056	9.5	myArray[7]
1064	10	myArray[8]
1072	2.5	myArray[9]

Passing Arrays to Functions

- Just as one can pass single values to a function, one can also pass an entire array to a function. In this case, the name of the array is passed to the function.
- The following code demonstrates how to declare and invoke this type of functions.
- Draw the memory diagram.

```
#include <iostream>
using namespace std;
void printArray(int list[], int arraySize);
int main()
{
  int numbers[5] = {1, 4, 3, 6, 8};
  printArray(numbers, 5);//watch out for the call
  cout<<numbers[2]<<endl;
}
void printArray(int list[], int arraySize)
{
  for (int i = 0; i < arraySize; i++)
    cout << ++list[i] <<  " ";
}
```


Pass-by-Reference

- Passing an array to a function means that the starting address (instead of value) of the array is passed to the formal parameter; this is called pass by reference (address) where only the name of the array is passed
- The parameter, inside the function, references to the same array that is passed to the function.
- No new array is created in this process. The new array name becomes the alias of the old array name.
- In general, when passing an array to a function, one should also pass its size via another argument
- The function definition header may use square brackets with or without indicating the size of the array inside the brackets.
- Be extremely careful about not writing past the end of an array!


```
//Output?
#include <iostream>
using namespace std;
void myFunction(int anyNumber, int anyArray[]);
int main()
{
  int x = 1; // x represents an int value
  int y[10]; // y represents an array of 10 int values
  y[0] = 1; // Initialize y[0]

  myFunction (x, y);// Invoke myFunction with arguments x and
y

  cout << "x is " << x << endl;
  cout << "y[0] is " << y[0] << endl;
 return 0;
}
void myFunction (int anyNumber, int anyArray[])
{
  anyNumber = 10; // Assign a new value to number
  anyArray[0]= 25; // Assign a new value to numbers[0]
}
```

```
//Output?
#include <iostream>
using namespace std;
void printArray( int someArray[], int size);
void addArray( int someArray[], int size);
int main()
{
 const int size = 5; int myArray[size] = {8,2,3,9,10};
 addArray(myArray, size); printArray(myArray, size);
}
void addArray(int InputArray[], int size)
{
  for (int i = 0; i < size ; ++i)
      InputArray[i] = InputArray[i] + 2;
}
void printArray( int InputArray[], int size )
{
  for (int i = 0; i < size ; ++ i)
      cout << "Array value [" << i << "] = " <<
  InputArray[i] << endl;
}
```


Example-code on searching an element

```
//Problem: search the given array for a value -7
#include <iostream>
using namespace std;
int findValue(int [], int, int);
int main()
{
  const int size = 5; int myArray[size] = {5, -3, 6, -7, 4};
  int valueToBeFound = -7;
  int indexFound = findValue(myArray, size, valueToBeFound);
  if(indexFound != -1)
   cout<<"The value is found in index "<<indexFound<<endl;
  else
   cout<<"Not found!\n";
}
int findValue(int ma[], int sz, int vtf){
  for(int i = 0; i<sz; i++)
  {
   if(ma[i] == vtf)
       return i;
  }
  return -1;
}
```


Example-code on sorting an array

```
//Problem: sort an array in descending order
#include <iostream>
using namespace std;
int main()
{
  const int sz = 5;int myArray[sz] = {5, -3, 6, -7, 4};
  for(int i = sz-1; i>0; i--)
    for(int j = 0; j<i; j++)
    {
      if(myArray[j]<myArray[j+1])
      {//swapping the array elements
         int temp = myArray[j+1];
         myArray[j+1] = myArray[j];
         myArray[j] = temp;
      }
     }
  cout<<"[";
  for(int i = 0; i<sz; i++)
    cout<<myArray[i]<<", ";
  cout<<"\b\b]\n";//\b stands for backspace
}
```


Practice Problem

- Rewrite the code given in slide #16 in a function with prototype: **`string convertToUpper (string k),`** and call this function to convert any string to uppercase string.
- Write a C++ program that can find and display the largest, smallest and average values of 50 integers ranging from 4 to 32 inclusive. The following specifications need to be implemented
  - Populate the array by prompting the user for validated inputs.
  - Define a function **`int largestNumber(int anArray[], int size);`** This function should return the largest number stored in the array.
  - Define another function **`double average(int anArray[], int size);`** This function should return the average of all the array values.
  - Define another function **`void inclrementValues(int anArray[], int size);`** This function should increment all the array values by 1 as long as the incremented array values are with in the abovementioned range otherwise leave those as is.
  - Call the function **`double average(int anArray[], int size);`** This call should return the average of the array values.
  - Print the largest, average values (before and after the increment operations) from the main function

FYI: const Parameters

- Passing arrays by reference makes sense for performance reasons.
- If an array is passed by value, all its elements must be copied into a new array. For large arrays, it could take some time and additional memory space.
- However, passing arrays by reference could lead to logical errors if your function changes the array contents accidentally.
- To prevent it from happening, one can put the const keyword before the array parameter to tell the compiler that the array contents cannot be changed.
- The compiler will report errors if the code in the function attempts to modify the array (see the code in the following slide).

FYI: const Parameters

```
#include <iostream>
using namespace std;

void p(const int list[], int arraySize)
{
  // Modify array accidentally
  list[0] = 100; // Compilation error!
}
int main()
{
  int numbers[5] = {1, 4, 3, 6, 8};
  cout<<++numbers[4]<<endl;
  p(numbers, 5);
  return 0;
}
```

Multidimensional arrays

- Multidimensional arrays with two dimensions
  - Called two dimensional or 2-D arrays
  - Represent tables of values with rows and columns
  - Elements referenced with two subscripts ([x][y])
  - In general, an array with m rows and n columns is called an m-by-n array
- Multidimensional arrays can have more than two dimensions

Two-dimensional Arrays

- **General form**

```
dataType arrayName[rowSize][columnSize];
```

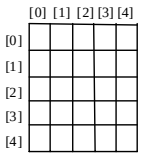
Example:

```
int my2DArray[5][5];
```

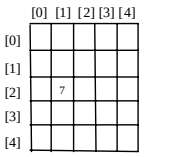

2D Arrays with initializer-list

- `int myArray[2][3]={{2, 5, 15}, {6, 21, 9}};`
  - In the above declaration row contains 2, 5 and 15 for its three columns (column 0, 1 and 2 respectively) and the second row contains 6, 21 and 9 for its three columns.
- `int myArray[2][3]={{2, 5},{6, 21, 9}};`
  - In the above declaration first row contains 2, 5 and 0 for its three columns and the second row contains 6, 21 and 9 for its three columns.
- `int myArray[2][3]={2, 5, 6, 21};`
  - In the above declaration first row contains first three values 2, 5 and 6 for its three columns and the second row contains second three values 21, 0 and 0 for its three columns.
  - In general, for this list, using no size values for the row and column will make the compiler confused. Although it does not result in any compilation error but this kind of coding should be avoided.

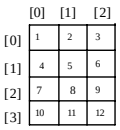
Two-dimensional Array Illustration



`int my2DArray[5][5];`



`my2DArray [2][1] = 7;`




```
int array[][] = {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```


Initializing Arrays with Random Values

- The following loop initializes the array with random values between 0 and 99:

```
const int r = 10, c = 10;
int matrix [r][c] = {0};
for (int row = 0; row < rowSize; row++)
{
  for (int column = 0; column < columnSize; column++)
  {
    matrix[row][column] = rand() % 100;
  }
}
```


Printing 2D Array Values

- To print a two-dimensional array, you have to print each element in the array using a nested loop.

```
for (int row = 0; row < rowSize; row++)
{
  for (int column = 0; column < columnSize; column++)
  {
    cout << matrix[row][column] << " ";
  }
  cout << endl;
}
```

Creating a function with a 2D array parameter

- In this case, the array parameter has to have the column size written inside the square braces.
- Using the row size inside the square bracket is optional.
- e.g., see the function prototype below:

```
void array2D(int matrix[][3], int rSize, int cSize);
```

- As before, call by reference uses the array name only.


```
//Passing an 2D array to a function
#include <iostream>
#include <cmath>
using namespace std;
void array2D(int matrix[][3], int rowSize, int
   columnSize);
int main(void)
{
  int const rs=3, cs = 3;
  int twoDArray[rs][cs]={1,2,3,4,5,6,7};
  cout<<"The array before the Call:"<<endl;
  for (int row = 0; row < rs; row++)
  {
     for (int column = 0; column < cs; column++)
     {
        cout<<twoDArray[row][column]<<" ";
     }
    cout<<endl;
  }
  //Calling the function below
 array2D(twoDArray,rs,cs);
  cout<<"The array after the Call:"<<endl;
  for (int row = 0; row < rs; row++)
  {
     for (int column = 0; column < cs; column++)
     {
        cout<<twoDArray[row][column]<<" ";
     }
     cout<<endl;
  }
}
```

```
void array2D(int matrix [ ][3], int rS, int cS)
{
     for (int r = 0; r < rS; r++)
     {
          for (int c = 0; c < cS; c++)
          {
          matrix[r][c] = matrix[r][c]+2;
          }
     }
}
```

```
//Output
The array before the Call:
1 2 3
4 5 6
7 0 0
The array after the Call:
3 4 5
6 7 8
9 2 2
```


Review

73) What are the valid indices for the array shown below?

```
int myArray[25];
```

a) 0 to 24
b) 1 to 24
c) 0 to 25
d) 1 to 25

Review

74) Given an array named `scores` with 25 elements, what is the correct way to access the 25th element?

```
a)scores[last]
b)scores(24)
c)scores[24]
d)scores[25]
```


Review

75) When an array is passed to a function, it is actually _____ the array that is passed.

a) a copy of all the values in
b) the data type and size of
c) the value stored in the first element of
d) the memory address of the first element of
e) None of the above

Review

76) An array of 10 integers named `myArray` can have its contents displayed with the statement

```
a)cout << myArray[ ];
b)cout << myArray;
c)cout << myArray[10];
d)cout << myArray[0-9];
e)None of the above
```

Review

77) What is the size of the array in the following statement:

```
char city[] =  "London";
```

a) 5
b) 6
c) 7
d) None of the above

Review

78) 0utput?

```
#include <iostream>
using namespace std;
void add (int x[], int
n)
{
 for(int i=1;i<n;i++)
       x[0]+=x[i];
}
```

```
int main()
 {
   int test[3]={3,2,4};
   add(test, 3);
   cout<<test[0]<<endl;
   return(0);
 }
```

a) 3
b) 9
c) 12
d) None of the above

Review

79. Based on the following array declarations, which of the following statements is TRUE

```
char myArray1[] = {'c', 'o', 'o', 'l'};
char myArray2[] = "cool";
```

a) They both have the same size of 4
b) They both have the same size of 5
c) They have different sizes
d) One of the above declarations will result in a compilation error
e) One of the above declarations will result in a run time error

Review

80. For a static (non-dynamic) type array, which of the following statements is FALSE?

a) Every element in an array has the same data type
b) The array size is fixed after it is created
c) The array size used to declare an array must be a constant expression
d) The array elements are initialized by default values when an array is created locally
e) None of the above

Review

81. What will be the output of the following code segment?

```
int myArray[4][3] = {{1, 2}, {3}, {4,
5, 6}, {7, 8}};
cout<<myArray<<endl;
```

a) 1
b) 0
c) The code will result in a compilation error
d) The code will result in a runtime error
e) None of the above

Answer Key

73. A
74. C
75. D
76. E
77. C
78. B
79. C
80. D
81. E

ES 1036 Programming Fundamentals

An Introduction to Pointers

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Dr. Quazi M. Rahman, P.Eng., SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

Outline

- Introducing Pointers
- Pointer arithmetic
- Pointers and functions
- Dynamic Memory Allocation
- References Vs Pointers
- Pointers and arrays

Pointers

- A pointer is a variable
- It contains an address of another variable
- A pointer variable is declared with asterisk (*) sign followed by the name of the variable (e.g., `int *ptr`). In this case, the asterisk sign is known as pointer operator.
- If an already declared pointer variable uses the * sign in front of it, the * sign is called as indirection or dereferencing operator.

Declaring pointer variables

- General form -
  - `type *name1;` **OR**
  - `type* name1;`
- Notes:
  - When declaring more than one pointer variable, the * must precede each variable e.g.,
    `type *name1, *name2, *name3;`
  - The 'type' is any variable (int , char etc) or object (user defined) type
  - The 'name' has to be valid identifier

Example

```
int a,b;   /* integer variables */
int *p;   /* p is a pointer to an int */
```

- The statements shown above could be represented in memory as follows:

Arbitrary Address	Expected Content	Variable name
ac04	some integer value	a
ac00	some integer value	b
ab00	address of an integer	p

Initializing (or assigning values to) a pointer variable

- Before using any pointer-variable it has to be initialized
  - with the address of an existing-variable (which has already been declared) using address of (&) operator, [AKA reference operator], or
  - with the address of another existing pointer variable, or
  - with NULL (all uppercase) or 0. Assigning NULL or 0 to a pointer indicates that it is not pointing anywhere. It only checks if a pointer has been assigned a value or not.

Example with Address-of Operator

- The address-of operator **&** in front of any variable produces the address of that variable.
- Example:

```
int x=75;
cout << "x is " << x;
cout << "\nAddress of x is " << &x;
```

x is 75
Address of x is 0012ff50

Address	Content	Variable name
0012ff50	75	x

Example: Pointer Assignment / Initialization

Example:

```
int a, b; /* integer variable declaration */
a = b = 7;   /* integer assignment */
int *p; /*Line 3: pointer declaration*/
p = &a;   /* Line 4: pointer assignment */
```

Very important note:

```
/*lines 3 and 4 above can be replaced by the
following statement provided that the variable (or
object) 'a' has already been declared*/
int *p = &a;
```

- Note: The address operator & returns the address of a variable. E.g.,

```
int x;
cout<<&x<<endl; //prints the address of variable x
```


Corresponding memory diagram:

```
#include <iostream>
using namespace std;
int main()
{
  int a, b;
  a = b = 7;
  int *p;
   p = &a;
}
```

Address	Content	Variable-name
ac04	7	a
ac00	7	b
ab00	ac04	p

Note: ***p is pointing to variable (or object) 'a'; a's address is held in p. In the above memory diagram, addresses are chosen arbitrarily***

Dereferencing Pointer Variables

- **Accessing a value of a variable using a pointer is called dereferencing a pointer and in this case, the dereferencing operator * (asterisk sign) is used to serve the purpose.**

```
[Draw the memory diagram for the following program]
int main(){
  int a = 7;
  int *p; p = &a;
  cout<<a<<endl; // direct reference; output?

  cout<<*p<<endl; /* indirect reference: go to the memory
  location of the variable whose address is held in the
  pointer p and work on the value stored in that memory
  location based on the instructions presented in the
  statement*/output?
  *p=*p+5; // indirect reference;
  cout<<a<<endl; // direct reference; output?
  a = *p - 7;
  cout<<*p<<endl; // indirect reference; output?
}
```


Corresponding Memory Diagram:

Address	Content	Variable name
ac04	~~7~~ ~~12~~ 5	a
ab00	ac04	p

Note: The table above displays the memory diagram for the example program shown in the pervious slide)

The Base Type

- **In the declaration**

```
int *a;     // the base type of p is int
double *q;  // the base type of q is double
```

- **Base type: The type of the variable whose address is held by the pointer.**
- **Since any pointer stores an address (or zero), the size of all the pointer variable (contents of the pointers) are always the same (4 bytes‡)**
  - `cout<<sizeof(a)<<endl;  //output?`
  - `cout<<sizeof(q)<<endl;  //output?`

‡compiler dependent

Base Type continued

- A pointer and the object (or variable) that 'pointer is pointing to' must be of same type; otherwise it'll result in a compilation error
- Example:

```
int a(5), *ptr=&a; //valid statement
int a(5); double *ptr=&a; //error; why?
```

- Base type also affects pointer arithmetic

Very Important: Declaring pointer variable, and dereferencing operator

- As a beginner one might get confused with the use of asterisk (*) sign in the pointer domain; let's clarify this confusion:
- When the asterisk sign is used with the data type (e.g., `double *p`), the compiler interprets this asterisk as a pointer operator in pointer declaration.
- When the asterisk sign is used with the name [not data-type] of the pointer only (e.g., *p = 10), the asterisk is interpreted as dereferencing operator.

NULL pointer

- Other than an address of a variable, there is only one value that can be assigned to any pointer; This is **NULL** (all capital; it holds a zero value) or **0**
- In this context it is known as the null address, and it does not point to anything that can be referenced.
- Dereferencing the null address will result in a **runtime error** (NOT compilation error)

Example

```
int *iPtr=0; //valid
char *s=NULL; //valid, NULL has to be all capital
double *dPtr=NULL; //valid
cout<<*iptr<<endl; //runtime error!
```

0	iPtr
0	s
0	dPtr

Review: Program example with srand() and time()

```
//Predefined function: time_t time(time_t*)
#include <iostream>
#include <ctime>
using namespace std;
int main(void)
{
  int i;
  srand((unsigned int)time(NULL));
  for(i = 0; i < 5; i++)
      cout<<rand()%5<<endl;
}
```


Program example with string and character pointers

```
#include <iostream>
#include <string>
using namespace std;
void main()
{
  string anyString = "Life is beautiful";
  string *sPtr = &anyString;
  char anyChar = '!', *cPtr = &anyChar;
  cout<<*sPtr<<*cPtr<<endl;
}
```

Functions with pointers

- A pointer can be used in the formal parameter list of a function.

  E.g., `void swap(int *x, int *y);`
- In this call the function is called by reference, because the pointer in the formal parameter list can only be activated (initialized to be used) by a variable that contains an address of an existing variable
- When the above function is called (from the main() or any other function), the associated addresses must be passed to the function through the argument list. E.g.,

  `int a(5), b(6); swap(&a, &b);`
- A pointer parameter must be explicitly de-referenced to access the contents at that address


```
//Call by reference using pointers(method 1)

#include <iostream>
using namespace std;
void swap_byPTR(int *x, int *y);
int main()
{
     int a(5), b(6);
     swap_byPTR(&a,&b);
     cout<<a<<" and  "<<b<<endl;
}
void swap_byPTR(int *x, int *y)
/*after the call: int *x=&a and int *y=&b */
{
     int temp; temp=*x; *x=*y; *y=temp;
     cout<<*x<<" and  "<<*y<<endl;
}
```



```
//Call by reference using pointers(method 2)

#include <iostream>
using namespace std;
void swap_byPTR(int *x, int *y);
int main()
{
     int a(5), b(6), *p = &a, *q = &b;
     swap_byPTR(p,q);
     cout<<a<<" and  "<<b<<endl;
}
void swap_byPTR(int *x, int *y)
/*after the call: int *x=p (&a) and int *y=q (&b) */
{
     int temp; temp=*x; *x=*y; *y=temp;
     cout<<*x<<" and  "<<*y<<endl;
}
```


Review

99) The _____, also known as the address operator, returns the memory address of a variable.

a) asterisk (*)
b) ampersand (&)
c) percent sign (%)
d) exclamation point (!)

Review

100) What is the Output?

```
int x(5), *iPtr=0;
cout<<*iPtr<<endl;
```

a) 5
b) 0
c) The code will result in a compiler error
d) The code will result in runtime error

Review

101) What is the Output?

```
int x(5), *iPtr=&x;
*iPtr = 0;
cout<<x<<endl;
```

a) 5
b) 0
c) The code will result in a compiler error
d) The code will result in a runtime error

Review

102) What is the Output?

```
int x(15); double *iPtr=&x;
*iPtr = 10;
cout<<x<<endl;
```

a) 15
b) 10
c) The code will result in a compiler error
d) The code will result in a runtime error

Review

102) What is the Output?

```
int x(15); int *iPtr=&x;
*iPtr = 10;
cout<<x<<endl;
```

a) 15
b) 10
c) The code will result in a compiler error
d) The code will result in a runtime error

Pointer Arithmetic

- Four arithmetic operations are supported
  - `+`, `-`, `++`, `--`
  - `Only integers` can be used in these operations
  - `Note:` Two pointers can be subtracted from each other but can not be added to each other
- Arithmetic operation is performed relative to the base type of the variable (int, double etc) or object
- When applied to pointers: `++` means increment pointer to point to the next object and `--` means decrement pointer to point to the previous object
  - Example 1: (Lets assume `p` contains 1000)

```
int x(2),*p=&x; cout<<++p<<endl; //output: 1004
```

  - Example 2: (Lets assume `p` contains 1008)

```
double x(2),*p=&x; cout<<--p<<endl; //output: 1000
```

  - Example 3: (Lets assume `p` contains 1000)

```
double x(2),*p=&x; cout<<(p+1)<<endl; //output: 1008
```


Assignment and equality (equivalent) operators with pointer variables

- Both assignment and equality (equivalent to) operators can be used with pointer variables
- Example:

```
int x(10), *xp, *ip;
xp = &x;
ip = xp;
cout<<*xp;
cout<<*ip;
if(xp==ip)
  cout<<"Equal pointers\n";
else if(*xp==*ip)
   cout<<"Equal values\n";
else if(xp==NULL)
  cout<<"Null pointer\n";
else
  cout<<"bye\n";
/*Draw the memory diagram to find the output*/
```


Review

104) What is the Output?

Assume: &x is 3004 and an integer variable requires 4 bytes of memory space

```
int x(10), *xp, *ip;
xp = &x;
ip = xp;
cout << *ip << endl;
```

a) Any random number
b) 3004
c) 10
d) none of the above

Practice problem with pointers!

If the first cout statement prints 'iPtr is FF20' what are the remaining outputs?

```
int q=6;
int *iPtr = &q;
cout << "iPtr is " << iPtr << endl;
cout << "*iPtr is " << *iPtr << endl;
cout << "++*iPtr is " << ++*iPtr << endl;
cout << "q is " << q << endl;
cout << "iPtr is " << iPtr << endl;
cout << "*iPtr++ is " << *iPtr++ << endl;
          //Watch out! operator binding
cout << "iPtr is " << iPtr << endl;
cout << "q is " << q << endl;
```

Result of Practice

```
iPtr is FF20
*iPtr is 6
++*iPtr is 7
q is 7
iPtr is FF20
*iPtr++ is 7
iPtr is FF24
q is 7
```


Review

105) What is the output?
Assume: &b is 3004 in bytes and the size of double is 8 bytes

```
double b(6.0);
double *a=&b;
a++;
cout << a << endl;
```

a) 7.0
b) 3004
c) 6.0
d) 3012
e) 3005

Review

106) What is the output?
Assume: &b is 3004 in bytes

```
int b =6, c;
int *a=&b;
c = *a++;
cout << a << endl;
```

a) 6
b) 7
c) 3008
d) 3004

FYI: Pointer-variable as a return data-type

```
#include <iostream>
using namespace std;
int* myFunction(int, int);
int main()
{
  int *k = myFunction(4,5); cout<<*k<<endl;
  //Same as:
  int p = *myFunction(4,5); cout<<p<<endl;
}
int* myFunction(int a, int b)
{
  int k= a+b;
  int *ptr = &k;
  return ptr;
}
```


Review

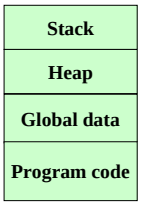
```
107. //Output?
#include <iostream>
using namespace std;
void myFunction(int *);
int main()
{
  int k(5),*iptr = &k;
  myFunction(iptr);
  cout<<*iptr<<endl;
}
void myFunction(int *zz)
{
   *zz = *zz+2;
}
```

Draw the memory diagram to find the output

a) 5
b) 7
c) The code will not compile
d) None of the above

Data storage and Memory diagram

- Four sections of the memory areas store four different types data
- Stack
  - Stores Local variables and formal parameters
  - Managed by the compiler
- Heap
  - Stores dynamic variables
  - Managed by storage allocator
- Global Data
  - Stores global variables
- Program code
  - Stores Other program data



Pointer application: Dynamic Memory Allocation

- Dynamically allocated memory is determined at runtime (not compile time, when the compiler is building the project)
- It allows a program to create as many or as few variables as required, offering greater flexibility
- Dynamic allocation is often used to support data structures such as **stacks, queues, linked lists** and **binary trees**. (application of pointers)
- Dynamic memory is finite.
- Dynamically allocated memory may be freed during execution.

Operator new

- Creates a new dynamic variable/object of the specified type (int, double, Circle, Complex etc)
- Returns the address of a memory location of the new variable and assigns it to a pointer (e.g., `int *ptr = new int;`)
- If allocation fails, new terminates the program (or returns **NULL** on some compilers)

Example

```
int *iPtr = new int;
/*4 bytes are allocated in the heap section of the
  memory. iPtr gets the address of that allocated
  space. In other words, it points to new integer
  variable*/

int size = 0//here, size is a variable
cout<<"Input array size: "; cin>>size; //user enters
  20
double *dPtr = new double[size];
/*Here, the size does not have to be a constant. An
  array of 20 double spaces (160 bytes) are
  allocated in the heap and the address of the first
  location is assigned to the dptr. In this case the
  name of the pointer becomes the array identifier
  (array name)*/

dPtr = new double[50];
/*dPtr now points to a new array of 50
elements. Previous array is abandoned.*/
```


new - continued

- Always check after each request.

```
int *p;
p = new int;
if(p == NULL)
   cout<<"no memory is available\n";
```

The following is not included in the exam

- Also, one can use assert function available in **<cassert>** library:

```
/*if assertion is false, program ends and
an error message is issued */
  assert(p != NULL);
```


Initializing Dynamically Allocated Memory

- To initialize a dynamically allocated variable, specify the initial value inside round bracket after the data-type.
- Example:

```
int *ptr;
ptr = new int(100);
```

- new variable pointed to by **ptr** has a value of 100

Operator delete

- The **delete** operator frees up the memory produced by **new**

  e.g., `int *ptr=new int; delete ptr;`
- While **delete** uses a pointer variable, it does not destroy the pointer variable, it only frees up the storage to which the variable is pointing.
- Deleting dynamically allocated array:

```
int *ptr=new int [5]; delete [] ptr;
```

Example on delete

```
int *ptr;
ptr = new int (100);
cout << *ptr <<endl; // output ?
delete ptr;       // free the memory
ptr = new int [100];
/* when an array is created in the heap
  using DMA, the pointer becomes the array
  identifier for that array.*/
ptr[0]=10;
cout << ptr[0]<<endl;//output ?
delete [] ptr; // free the memory array
```


Arrays of Pointers

- You may define arrays of pointers like any other data type in C++

```
int num=8;
```

- declare an array of 10 pointers to 10 integer variables

```
int *iPtrs[10];
```

- first pointer element is assigned a value

```
iPtrs[0] = &num;
```

- output the value of num

```
cout << *iPtrs[0];
```


Example: Arrays of Pointers

```
#include <iostream>
using namespace std;
const int size=5;
int main()
{
  int x[size]={3,6,1,8,9},*y[size],i;
  cout<<"Assigning all the pointers with
address values\n";
  for(i=0; i<size; i++)
     y[i]=&x[i]; //same as: y[i]=(x+i);
  cout<<"..done..\n"
  cout<<"Printing by dereferencing..\n";
  for(i=0; i<size; i++)
     cout<<*y[i]<<endl;
}
```


Review

108) Following declaration defines

```
double *b[5];
```

a) **an array of five pointers that point to double**
b) **an array of five double type data elements**
c) **a double pointer and an array of double type data elements**
d) **a double type data element and an array of pointers**

References Vs. Pointers

- Same idea, except with different syntax
- References are simpler, but restricted
  - Can only point to one object during its lifetime (i.e. must be initialized at the time of declaration)
  - Cannot do any arithmetic
  - Used as an *alias* to another object
- Pointers are powerful and more complex
  - Can point to different objects during its lifetime
  - Can do "pointer arithmetic"
  - Uses additional syntax such as `->` (used in classes)

Syntax of References vs Pointers

- Declaring references

```
int a;
int &b = a;//means &b=&a
```

Grammar: reference variable must be initialized at the declaration statement

- After the second statement above, `b` becomes an alias of `a`
- Using references

```
int a;
int &b = a;
b = 5;
cout << b++ << endl;
cout<< a << endl;
```

- The following will result in a syntactical error:

```
int a;
int &b;//compilation error!
b = a;
int &c = &a // compilation error!
```

- Arrays of references are illegal!

Review:

- Declaring pointers

```
int *ptr_a;
int b, *iptr;
```

- Using pointers

```
iptr = &b; // assign
*iptr = 5; // use
cout << b << *iptr;
cout << iptr;
```

```
//Call by Reference using referenced variables
#include <iostream>
using namespace std;
void swap_byREF(int &x, int &y);
int main()
{
      int a(5), b(6);
      swap_byREF(a,b);
      cout<<a<<" and  "<<b<<endl;
}
void swap_byREF(int &x, int &y)/*after the call: int &x =a and
int &y=b; x and y become the aliases of a and b */
{
      int temp; temp=x; x=y; y=temp;
      cout<<x<<" and  "<<y<<endl;
}
```



```
//Review: Call by Value
#include <iostream>
using namespace std;
void swap_byVAL(int x, int y);
int main()
{
     int a(5), b(6);
     swap_byVAL(a,b);
     cout<<a<<" and  "<<b<<endl;
}
void swap_byVAL(int x, int y)
{
     int temp; temp=x; x=y; y=temp;
     cout<<x<<" and  "<<y<<endl;
}
```


Tips on Function call and function header

- Always go back to the grammar in declaring and initializing (on the same statement) a variable /object /pointer variable /reference variable.
- In this case, left hand side (of the assignment operator) will be used as a function parameter in the function header while the right hand side will be used as an argument of the function call for the corresponding function parameter.
- Example: call by value
  - Grammar:
    int x = 5;
    int k = x; //declaration and initialization grammar
  - Function header: void MyFunction (int k) {//definition}
  - Funciton call: MyFunction(x);

Tips on Function call and function header continues

- Always go back to the grammar in declaring and initializing (on the same statement) a variable /object /pointer variable /reference variable.
- In this case, left hand side (of the assignment operator) will be used as a function parameter in the function header while the right hand side will be used as an argument of the function call for the corresponding function parameter.
- Example: call by reference with pointer variables
  - Grammar:
    int x = 5;
    int *k = &x; //declaration and initialization grammar
  - Function header: void MyFunction (int *k) {//definition}
  - Funciton call: MyFunction(&x);// also, MyFunction(k);

Tips on Function call and function header continues

- Always go back to the grammar in declaring and initializing (on the same statement) a variable /object /pointer variable /reference variable.
- In this case, left hand side (of the assignment operator) will be used as a function parameter in the function header while the right hand side will be used as an argument of the function call for the corresponding function parameter.
- Example: call by reference with reference variables
  - Grammar:
    int x = 5;
    int &k = x; //declaration and initialization grammar
  - Function header: void MyFunction (int &k) {//definition}
  - Funciton call: MyFunction(x);

Tips on Function call and function header continues

- Function calls with arrays: Always use the name of the array (you need to pass to a function) in the function argument, and use an array identifier in the function header.
- Example: call by reference with arrays
  - Grammar:
    char x[5] = {'a', 'e', 'i', 'o', 'u'};
  - Function header: void MyFunction (char k[5]) {//definition}
  - Funciton call: MyFunction(x);

Caution: Common Pointer Problems

- Using non-initialized pointers

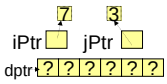
```
int *iPtr;
*iPtr = 100;
```

  - `iPtr` has not been initialized. The value 100 will be assigned to some memory location.
  - This will result a run-time error (debug error); not compiler error

Example program: DMA and functions

```
#include <iostream>
using namespace std;
void myFunction(double *, int sz);
int main()
{  int *iPtr, *jPtr, i, size(6);
   iPtr = new int; jPtr = new int(3);
   double *dPtr; dPtr = new double[size];
   *iPtr = 7;

   cout <<*iPtr << ',' << *jPtr << endl;

   myFunction(dPtr, size);

   for(i=0; i<size; i++)
     cout << (*dPtr)++ << ' ';
   cout << endl;

   for(i=0; i<size; i++)
     cout << dPtr[i] << ' ';
   return 0;
}
void myFunction(double *zz, int sz)
{
   for(int i=0; i<sz; i++)
     zz[i] = 5;//same as *(zz+i)=5
}
```

Draw the memory diagram to find the output



OUTPUT
7 , 3
5 6 7 8 9 10
11 5 5 5 5 5

Review

109) Variables cannot be created when a program is already running

a) True
b) False

Review

110) The delete operator should only be used on pointers that

a) have not yet been used
b) have been correctly initialized
c) points to storage allocated by the *new* operator
d) are appropriately de-referenced

Review

111) Which of the following statements correctly releases a dynamically-allocated array pointed to by p?

```
a)delete p;
b)p delete[ ];
c)delete [ ] p;
d)delete array p;
```


Review

112) Which of the following statements creates an array of 15 integer variables

```
a)int *st(15);
b)int *st = new int;
c)int *st = new int [15];
d)int *st[15] = new int;
```

Review

113. What will be the output of the following code segment?

```
double *dptr = new double[5];
dptr[0]=10;
dptr = new double[5];
cout<<dptr[0]<<endl;
```

a)0;
b)10;
c)5;
d)None of the above;

Pointers and Arrays

- The name of an array is the address of the first element
- That is: The name of the array is the pointer to the first element.
- When an array is declared, space for all elements in the array is allocated,
- However when a pointer variable is declared only space sufficient to hold an address is allocated

- Example:

```
int num[4] = {5,9,4,8}, *p;
cout << num << endl;
cout << num[3] << endl;
p = num; //same as p = &num[0];
cout << *p << endl;
p++;
cout << *p <<endl;//same as: cout<<p[0]<<endl;

/*Draw the memory diagram to find the output*/
```


Array, pointer and function

Draw the memory diagram to find the output

```
#include <iostream>
using namespace std;
void myFunction(int *);
const int sz = 4;
int main()
{
  int num[sz] = {5,9,4,8};
  cout<<"The array: ";
  for(int i = 0; i<sz; i++)
      cout<<num[i]<<" ";
  cout<<"\nThe Modified array: ";
  myFunction(num);
  for(int i = 0; i<sz; i++)
      cout<<num[i]<<" ";
  cout<<"\n\n\n";
}
```

```
void myFunction(int *x)
{
   for(int i = 0; i<sz; i++)
       x[i] = x[i]+ 2;
}
```


Array, reference variable and function

Draw the memory diagram to find the output

```
#include <iostream>
using namespace std;
void myFunction(int &, int);
int main()
{
  int k[3] = {3, 2, 6};
  myFunction(k[0], 3);
  cout<<k[0]<<endl;
}
void myFunction(int &zz, int sz)
{
   int *x = &zz;//line 1
   for(int i=1; i<sz; i++)
     x[0] = x[0]+x[i];//line 2
}
```

2nd method:
-Remove Line 1
-Replace line 2 by:
*(&zz)= *(&zz)+*(&zz+i);

Pointers and Arrays *contd.*

- Arrays and pointers may often be used interchangeably
- Example:

```
int num[4] = {5,9,4,8}, *p;
cout << num << endl;
cout<<*(num+1)<< endl;
cout << num[3] << endl;
p = num; //same as p = &num[0];
cout << *p << endl;
p++;
cout << *p << endl;
cout << *(p+1) << endl; //same as: cout<<p[1]<<endl;

/*Draw the memory diagram to find the output*/
```


Pointers and Arrays *contd.*

- The array name is not a pointer variable; it's a constant pointer, which always points to the first element of the array and its value (the address value) can NOT be changed.

- Example:

```
int num[4] = {5,9,4,8}, *p;
cout << num << endl; //num is a constant pointer
cout<<*(num+1)<< endl;
cout<<*(num++)<< endl;/*Error! Num's value can not be
  reassigned since it's a constant pointer*/
p = num; //same as p = &num[0];
cout << *p << endl;
p++;
cout << *p << endl;
cout << *(p+1) << endl;

/*Draw the memory diagram to find the output*/
```

Pointers and Arrays *contd.*

- Example: `//output?`

```
int num[4] = {5,9,4,8}, *p;
cout << num << endl;
cout<<*(num+1)<< endl;
p = num; //same as p = &num[0];
cout << *p << endl;
p++;
cout << *p << endl;
cout << *(p+1) << endl;
cout << *p << endl;
p = &num[3];
cout<<*p<<endl;
cout<<*(--p)<<endl;

/*Draw the memory diagram to find the output*/
```


FYI: Pointers and 2D-Arrays

```
//output? (not included in the exam; FYI only)
#include <iostream>
using namespace std;
int main()
{
  int myArray[4][3] = {{1, 2}, {3}, {4, 5,
  6}, {7, 8}};
  cout<<*(myArray[0]+7)<<endl; //5
}
```

Note: for 2d arrays we need to have the row number with the array for the pointer to work on.

Review

114. `int b[6]={4, 6, 7, 2, 4, 6};`
`b[5]=7;` is same as

a) *b = 7;
b) *b+4 = 7;
c) (*b)+5 = 7;
d) *(b+5) = 7;

Review

115) Assuming that `arr` is an array identifier, the statement (assume all the declaration and initialization are done appropriately)

`sum += *arr;`

a) is illegal in C++
b) will always result in a compiler error
c) adds the value stored in arr[0] to sum and assign the result to the variable sum
d) adds the address of the pointer arr to sum

Review

116) If `arr` is an array identifier and `k` is an integer, the expression `arr[k]` is equivalent to

a)*(arr + k)
b)*arr + k
c)&arr[k]
d)arr + k

Review

117. output?

(not included in the exam; FYI only)

```
#include <iostream>
using namespace std;
int main()
{
   int myArray[4][3] = {{1, 2}, {3}, {4, 5, 6}, {7, 8}};
   cout<<*(myArray[2])+3 <<endl;
}
```

a)0
b)3
c)4
d)7

Initializing a pointer with the address of another pointer (not included in the exam)

- A pointer variable, pointing to another pointer variable, is represented by double asterisks before its name (e.g. `int **ptr`).
- This variable is initialized with the address of the pointer variable it is pointing to.

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int number(4);
   int *PtrNum=&number, **ptrPtrNum;
   ptrPtrNum=&PtrNum;
   cout<<**ptrPtrNum<<endl;
}
/*Let's draw the memory diagram and find out the
   Output*/
```


Answer Key

99. B
100. D
101. B
102. C
103. B
104. C
105. D
106. C
107. B
108. A
109. B
110. C
111. C
112. C
113. D
114. D
115. C
116. A
117. D

ES 1036 Programming Fundamentals

Struct, Class and Object

Dr. Quazi M. Rahman, PhD, PEng, SMIEEE
Office Location: TEB 361
Email: QRAHMAN@eng.uwo.ca
Phone: 519-661-2111 x81399

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure" ~Colin Powell

Outline

- Struct
  - Definition
  - Struct declaration and Objects
  - Program example
- Class and Object
- Designing a Class
  - Class declaration
  - Class implementation
  - Using classes
- Constructors/Destructors
- Accessor/mutator
- Object oriented programming (OOP)
- Principles of OOP
- Program examples

Struct

- A ***struct*** is a mechanism that allows a programmer to define a new data type.
  - This new data type contains a collection of either same or different data items.
- Syntax:

```
struct valid_data_type_name{
        /*member data variables are declared here. We can not initialize any data member here (except for the static const data members); it will result in a syntax error if we try to do that*/
        /* Since the struct is declared outside (and above) the main function (global scope), the data members of the struct get initialized with the default values*/
};
```

- Example: below, three different data types (string, int and float) are collected together to form a new data type called Student

```
struct Student{
   string name;
   int ID;
   float score;
};
```

Watch out for the semicolon!

Objects

- An ***object*** is a variable of a defined ***struct*** type, also referred to as an ***instance*** of a ***struct***.
- Example: If 'Student' is a struct (a new data-type) then objects can be declared as:
  - Student Adam;
  - Student Eve;
- In the above example, the objects Eve and Adam contains all the characteristics (data members) declared in the struct Student, which can be accessed via member access operator (dot operator) discussed later.

4 Struct, Class and Objects

Struct and objects == Data type and variable name

- The relationship between struct and object is equivalent to the relationship between data-type and variable-name.
- Declaring an object of struct **`Student`** follows similar approach of declaring a variable of **`int`** type data as shown below
  - **`Student x;`** //**`x`** is the name of an object of **`Student`** type
  - **`int y;`** // **`y`** is a name of a variable of **`int`** type

5 Struct, Class and Objects

struct in a Program

```
#include <string>
#include <iostream>
using namespace std;

//create your data type
struct Student{
   string name;//data member name
   int ID;//data member ID
   float score;//data member score
};
//Use your new data type
int main()
{
  Student Q, R; //declaring two Student type objects Q and R
  Q.ID=1111; //assigning a value to the ID data-member for Q
  Q.name="Quazi"; //assigning a value to the name data-member for Q
  Q.score = 100; //assigning a value to the score data-member for Q
  cout<<"Enter R's ID: ";
  cin>>R.ID; /*getting a value for the ID data-member of R from the
   keyboard*/
  cout<<"Enter R's Name: ";
  cin>>R.name; // enter one word only
  cout<<"Enter R's score: ";
  cin>>R.score;
  cout<<"The Student "<<Q.name<<" scored "<<Q.score<<endl;
  cout<<"The Student "<<R.name<<" scored "<<R.score<<endl;
}
```

Accessing data members of a struct

- After an object (variable) of Student data type is declared, its data members can be accessed using the dot operator (.), also known as the *member access operator*:
- Member access operator is a period placed between the object's name (here, Q and R are the object names) and a member name (e.g., ID, name, score).
- Example: Q.ID refers to the data member ID for the object Q.

```
int main()
{
  Student Q, R;
  Q.ID=1111;
  Q.name="Quazi Rahman";
  Q.score = 100;
  cout<<"Enter R's ID: ";
  cin>>R.ID;
  cout<<"Enter R's Name: ";
  cin>>R.name;
  cout<<"Enter R's score: ";
  cin>>R.score;

   cout<<"The Student "<<Q.name
   <<" scored "<<Q.score <<endl;

   cout<<"The Student ID "<<
   R.name <<" scored  "<< R.score
   <<endl;
}
```


Why struct?

- struct provides the option to store collections of related data items that may have same or different data types.
- Difference between struct and array: arrays are useful data structures for storing a collection of data elements of the same data type.

Important info on objects

- An object (instance of a struct) can be used
  - in an array,
  - in a function (both as a return-type and as a function parameter in the formal parameter list),
  - in a function call
  - in pointer, reference and dynamic memory applications
- The objects can use assignment (=) operator.

Object initialization

```
#include <iostream>
#include <string>
using namespace std;
struct Student{

   /* Note: We can not initialize any data member inside struct declaration (except
   for the static const data members); it will result in a syntax error if we try to do that*/

   string name;//data member name
   int ID;//data member ID
   float score;//data member score
};

int main()
{
  Student Q = {"quazi", 9999, 99}; /*declaration and initialization of the
   Student type object Q*/
}
```


Example: Array of objects

```
#include <iostream>
#include <string>
using namespace std;
struct Student{
   string name;
   int ID;
   float score;
};
int main()
{
   Student myStudent[3];
   cout<<"Let's populate the array\n";
   for(int i=0; i<3; i++)
   {
        cout<<"Enter the first name: ";
        cin>> myStudent[i].name;
        cout<<"Enter ID for "<<myStudent[i].name<<": ";
        cin>>myStudent[i].ID;
        cout<<"Enter score for "<<myStudent[i].name<<": ";
        cin>>myStudent[i].score;
        cout<<endl;
   }
   cout<<"Let's print the class list\n";
   //Complete the code. This is your home work!
}
```


Example: Struct and function

```
/*Problem: Write a function that accepts a Student type object (as given
   below) and returns a Student type object*/
#include <iostream>
#include <string>
using namespace std;
struct Student{
   string name; //data member name
   int ID; //data member ID
   float score; //data member score
};
Student dataEntry(Student x);/*function prototype; it should be placed
   after struct declaration (In class discussion)*/
int main()
{
  Student Q;
  Q=dataEntry(Q); //Function call
  cout<<"You've entered: "<<Q.name<<", "<<Q.ID<<" and "<<Q.score<<endl;
}
Student dataEntry(Student x)
{
   x.name = "Quazi Rahman";
   x.ID = 1111;
   x.score = 100;
   return x;
}
```

In class Review: A function can not return more than one object or variable

Example: Object and pointer

```
/*Note: The code below demonstrates two ways to access members of
   objects when a pointer is used:  The expressions x->ID=1111
   and (*x).ID = 1111 are equivalent */
#include <iostream>
#include <string>
using namespace std;
struct Student{
   string name; //data member name
   int ID; //data member ID
   float score; //data member score
};
void dataEntry(Student *x); //function prototype
int main()
{
  Student Q;
  dataEntry(&Q); //Function call
  cout<<"You've entered: "<<Q.name<<", "<<Q.ID<<" and "<<Q.score<<endl;
}
void dataEntry(Student *x)
{
   (*x).name = "Quazi Rahman";
   x->ID = 1111; //same as (*x).ID = 1111;
   (*x).score = 100;
}/*When de-referencing a pointer-object, don't forget to enclose it with
   round bracket. E.g., (*x).score = 100; */
```


Example: Array of objects and DMA

```
#include <iostream>
#include <string>
using namespace std;
struct Student{
   string name;
   int ID;
   float score;
};
int main()
{
   cout<<"Enter the size of the array: ";
   int sz; cin>>sz;
   Student *sPtr = new Student[sz];
   cout<<"Let's populate the array\n";
   for(int i=0; i<sz; i++)
   {
        cout<<"Enter the name: ";
        cin>>sPtr[i].name;
        cout<<"Enter ID for "<<sPtr[i].name<<": ";
        cin>>sPtr[i].ID;
        cout<<"Enter score for "<<sPtr[i].name<<": ";
        cin>>sPtr[i].score;
        cout<<endl;
   }
   cout<<"Let's print the class list\n";
   //Complete the code. This is your home work!
}
```


Example: Reference objects and functions

```
#include <iostream>
#include <string>
using namespace std;
struct Student{
   string name; //data member name
   int ID; //data member ID
   float score; //data member score
};
void dataEntry(Student &x); //function prototype
int main()
{
  Student Q;
  dataEntry(Q);//Function call
  cout<<"You've entered: "<<Q.name<<", "<<Q.ID<<" and
  "<<Q.score<<endl;
}
void dataEntry(Student &x)
{
   x.name = "Quazi Rahman";
   x.ID = 1111;
   x.score = 100;
}
```


Homework problem

- Check the old final exam question (available on Owl) on struct and, practice those.

Classes

- A ***class*** (same as struct) is a mechanism that allows a programmer to define new data types by following the object oriented programming principles.
- A ***class*** can be used
  - to add functionality to an existing data type or
  - to create a new data type.
- A ***class*** definition combines data and functionality.

A ***class*** is a mechanism that allows a programmer to define new data types

Class: an analogy

Person
Name
Date of Birth
Address
Weight
Height
. . .

Walk
Jump
Sleep
Speak
Think
Eat
. . .

Class name (Identity)

Data

Functions

Person, the class

Class declaration syntax

- Syntax of a Class:
  class valid_data_type_name {
  /* data member s and function members are declared here. We can not initialize any data member here (except for the static const data members); it will result in a syntax error if we try to do that. The data members get the default values*/
  Member (data or function) access privilege label:
  data member declarations;
  function member declarations;
  };
- Example: below, three different data types (string, int and float) and a function (printData) are collected together to form a new data type called Student

```
class Student{
  public:
      string name;
      int ID;
      float score;

      void printData();
};
```

Open discussion: Differences between struct and class declarations?

Class definition (in-line)

- When a class contains function members, we need to define those functions before we use the class to create a new data type.
- In general, defining the member functions of a class is known as 'Class definition'.
- The member functions can be defined within the class declaration segment, and in this case, it is called in-line declaration as shown below:

```
class Student{
  public:
    string name;
    int ID;
    float score;

    void printData()
    {
        cout<<"Name: "<<name<<endl;
        cout<<"ID: "<<ID<<endl;
        cout<<"Marks: "<<score<<endl;
    }
};
```


Class definition outside the class-declaration block

- The member functions can be defined outside the class declaration segment, and in this case, scope resolution operator is required as shown below:

```
class Student{
  public:
    string name;
    int ID;
    float score;
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

- The syntax for the member function definition (outside the class declaration) using scope resolution operator (::) follows:

```
return_data_type class_name :: function_name (parameter list){
    definition;
}
```


Objects in classes

- An ***object*** is a variable of a defined ***class*** type, also referred to as an ***instance*** of a ***class***.
- Example: If 'Student' is a class (a new data-type) then objects can be declared as:
  - Student Adam;
  - Student Eve;
- In the above example, the objects Eve and Adam contains all the characteristics (data and functions) declared in the class Student, which can be accessed via member access operator (dot operator) same as struct.
- Note that ostream is a class and cout is an instance (object) of it

An ***object*** is a ***class*** type variable

Class and object: an analogy

Person
Name
Date of Birth
Address
Weight
Height
. . .

Walk
Jump
Sleep
Speak
Think
Eat
. . .

Class name (Identity)
Data
Functions

Person, the class

objects
Adam
Eve
Mike
Steve

Class and objects == Data type and variable name

- The relationship between class and object is equivalent to the relationship between data-type and variable-name.
- Declaring an object of class **`Student`** follows similar approach of declaring a variable of **`int`** type data as shown below
  - **`Student x;`** //**x** is the name of an object of **`Student`** type
  - **`int y;`** // **y** is a name of a variable of **`int`** type

Example: Using a class

```
# include <iostream>
#include <string>
using namespace std;
//Class declaration
class Student{
public:
    string name;
    int ID;
    float score;
    void printData() const;
};
//Member Function definition
void Student::printData() const
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
/*  The driver function (main function) goes
    after the function definition */
int main ()
{
    Student x; //Student type object x
    cout<<"Enter Name: ";
    getline(cin, x.name);
    cout<<"Enter ID: "; cin>>x.ID;
    cout<<"Enter Marks: "; cin>>x.score;
    x.printData();
}
```

Note: main() function in a class environment is known as Driver function or Client function.
const *keyword is used to indicate that this member method can not modify any data member (or any other member method) inside its definition;* ***const*** *should be placed at the end of any function that is not intended to modify the calling object.*

Access Privileges in a Class

- The following keywords specify the accessibility of the class members
  - **`public`** data or function members can be accessed from the main() function by the 'user of the class'
  - **`private`** data or function members can not be accessed directly from the main function by the 'user of the class' but these can be accessed indirectly by any **public** member functions
  - **`protected`** data or function members have intermediate access privilege (**not included in this course**)
- Can be listed in any order in a class
- Can appear multiple times in a class declaration
- If not specified, the default is private

Example with private data members

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
void Student::setName(string n) {
    name = n;
}
void Student::setID(int i) {
    ID = i;
}
void Student::setScore(float s) {
    score = s;
}
int main ()
{
    Student x;
    float mark; cout<<"enter marks: ";
    cin>>mark;
    x.setName("Quaintain Osborn");
    x.setID(9999);
    x.setScore(mark);
    x.printData();
}
```


Example 'on error' with private data members

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
void Student::setName(string n)
{
    name = n;
}
void Student::setID(int i)
{
    ID = i;
}
void Student::setScore(float s)
{
    score = s;
}
int main ()
{
    Student x;
    float mark; cout<<"enter marks: ";
    cin>>mark;
    x.score = mark; //Error?!
}
```

Note: The above compilation error is generated because the driver function is trying to access the private data member directly.

Private data members need to be accessed via public function member

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
void Student::setName(string n) {
    name = n;
}
void Student::setID(int i) {
    ID = i;
}
void Student::setScore(float s) {
    score = s;
}
int main ()
{
    Student x;
    float mark; cout<<"enter marks: ";
    cin>>mark;
    x.score = mark; //Error?!
}
```

correction:

Replace the above erroneous statement with the following:

```
x.setScore(mark);
```


More on accessing Private data members

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    float getScore(void);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
void Student::setName(string n) {
    name = n;
}
void Student::setID(int i) {
    ID = i;
}
void Student::setScore(float s) {
    score = s;
}
float Student::getScore(void) {
    return score;
}
int main ()
{
    Student x;
    float mark; cout<<"enter marks: ";
    cin>>mark;
    x.setScore(mark);
    cout<<"Marks: "<<x.getScore()<<endl;
}
```

Homework:

Create two other public methods (functions) (declare and define) so that you can call those from the driver function to print the name and id of the Student.

Accessor and Mutator functions

- To **access private data members** from the main function indirectly, the class declaration contains two types of public functions known as accessor and mutator functions.
- Colloquially, a get-function is referred to as a *getter* (or *accessor*), and a set-function is referred to as a *setter* (or *mutator*).
- A get function has the following generic signature:
  returnType get*PropertyName*();
- A set function has the following generic signature:
  void set*PropertyName*(*dataType propertyValue*);
- Review the examples (with set and get functions) in the previous slides again.
- *Note: The terms set and get with the function names are used for user friendliness. Any valid name can be used instead.*

ES1036 QMR

Final Thoughts on Private Members

- Private members (data/function) can NOT be accessed from the main() function
- Compiler error will be generated if we attempt to access private member(s) using dot operator, e.g.,
  `cout<<x.score<<endl;` will result compiler error since `score` is a private data member
- Private members can only be accessed via public member functions; e.g.,

```
cout << "The student scored "<<
x.getScore()<< endl;
```

- In class declaration if no access privilege label (public:, private: etc) is used, all the members are considered as private members (by default)

Initializing objects using `Constructors`

- The objects in a program can be initialized using a special function, called constructor; the initialization is done inside the definition of the constructor function.
- This special public member function must have the same name as the class itself.
- Constructors do not have a return type, not even void.
- Constructors play the role of initializing objects. A constructor function is automatically called when an object is created.
- Like any other function, constructors can be overloaded (i.e., multiple functions with the same name but different parameter list), making it easy to construct objects with different initial data values.

Example: Student class using 'constructor with no-argument'

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    Student();
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
Student::Student() {
    name = "Any name";
    ID = 1111; score =100;
}
void Student::setName(string n) {
    name = n;
}
void Student::setID(int i) {ID = i;}
void Student::setScore(float s) {
    score = s;
}
int main ()
{
    Student x; x.printData();
    float mark; cout<<"enter marks: ";
    cin>>mark;
    x.setName("Quaintain Osborn");
    x.setID(9999);
    x.setScore(mark);
    x.printData();
}
```


More on Constructors

- A constructor without parameters (e.g., Student ()) is called *no-argument constructor*.
- A class can be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class by the system. This constructor is known as *default constructor*.
- A class may contain one or more constructors with parameters with different parameter lists. These constructors are known as constructors with arguments.
- If the code contains a constructor with argument and does not contain the no-argument constructor, the default constructor does not get created automatically. In this case, creating an object (in the driver function) with no argument will result in a compilation error.
  - The above error can be eliminated by creating both types of constructors.

Example: Student class using 'constructor with argument'

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    Student();
    Student(string n);
    Student (int i, float s);
    Student (float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
    cout<<endl;
}
```

```
Student :: Student() {
    name = "Any name";
    ID = 111; score =100;
}
Student :: Student(string n){
    name = n; ID = 999; score = 85;
}
Student:: Student (int i, float s){
    score = s; ID = i; name = "RRR";
}
Student::Student(float s) {
    score = s; ID = 555; name = "MMM";
}
int main ()
{
    Student x, y("AAA"), z(98.6), m(123, 87.5);
    x.printData();
    y.printData();
    z.printData();
    m.printData();
}
```

Note: The set and get functions have been removed for space constraint.

Destructor Functions

```
#include <iostream>
using namespace std;
//class declaration
class Student{
private:
   string name;
   int ID;
   float score;
public:
   Student();
   ~Student();
   void setName(string n);
   void setID(int i);
   void setScore(float s);
   void printData();
};
```

Destructor

- The destructor is a special function that frees up the memory (used by any object), and releases resources (such as open files) when any object is destroyed.
- A constructor is invoked when an object is created and a destructor is invoked when the object is destroyed.
- The name of the destructor contains the character ~(tilde) followed by the class name [e.g., **`~Student();`**]
- It has no return type, no argument and can not return a value [e.g., **`~Student();`**]
- A destructor function is automatically called when an object is released.

More on Destructor

- A destructor function is automatically called for each object and this call is generated on the last-in first-out basis.
  - Last-in first-out: The object created at last will be released at first, the object created at second-last position will be released at second, and so on so forth (see the next slide).
- Every class has a default destructor if the destructor is not explicitly defined.
- Sometimes, it is desirable to implement destructors to perform customized operations (see the next slide).
- Each class can have only one destructor (although a class can have more than one constructor)
- No modifier (such as const) can be used with a destructor
- It's not mandatory to use destructor in the class but it's a good memory management practice

Example: Student class with destructor

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    Student();
    ~Student();
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
Student::Student() {
    name = "Any name";
    ID = 1111; score =100;
}
void Student::setName(string n) {name = n;}
void Student::setID(int i) {ID = i;}
void Student::setScore(float s) {score = s;}
Student::~Student() {
    cout<<"The object named "<<name<<"
    has been released from the memory\n";
}
int main ()
{
    Student x, y, z;
    y.setName("Adam");
    z.setName("Nikolas Piscar");
    cout<<"Good bye\n";
}
```

Note: A destructor function is automatically called when an object is released / destroyed; it uses last-in first-out approach

Important info on class

- The data type created by class can be used
  - in an array,
  - in a function (both as a return-type and as a function parameter in the formal parameter list),
  - in a function call
  - in pointer, reference and dynamic memory applications
- The objects can use assignment (=) operator.

Example: objects using assignment operator

```
//Output?
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    Student();
    ~Student();
    void setName(string n);
    void setID(int i);
    void setScore(float s);
    void printData();
};
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
```

```
Student::Student() {
    name = "Any name";
    ID = 111; score =100;
}
void Student::setName(string n) {name = n;}
void Student::setID(int i) {ID = i;}
void Student::setScore(float s) {score = s;}
Student::~Student() {
    cout<<"The object named "<<name<<"
    has been released from the memory\n";
}
int main ()
{
    Student x, y, z;
    y.setName("Adam"); y.setID(222);
    y.setScore(99);
    z.setName("Nikolas Piscar");
    x=y; /* In this case each of the member
    values of x will be replaced by the
    corresponding member value of y. */
    x.printData();
    cout<<"Good bye\n";
}
```


Separating Declaration from Implementation

- C++ allows one to separate class declaration from implementation and save those in two different files with same file names but different file extensions, inside the same project.
- The class declaration that contains a lists of all the data fields, constructor/destructor prototypes, and the function prototypes, is stored in a header file.
  - This header file must have the same name as the class name, and an extension '.h'. Example: for a class called 'Student' the header file name should be Student.h.
- The class implementation/definition, which implements/defines the constructors/destructors and the other member functions, is stored in a cpp file.
  - This file should have a same name as the class name with an extension '.cpp'. Example: for a class called 'Student' the implementation/definition file name should be Student.cpp.
- A third file is created with an extension .cpp for storing the driver function (the main function). This file name can have any valid name with extension .cpp.
- The overall procedure insures good file management technique. Visual studio creates the above files automatically (HOW? Please see the lab handout)

Example: Three separate files

//Student.h file

```
# include <iostream>
#include <string>
using namespace std;
class Student{
private:
    string name;
    int ID;
    float score;
public:
    Student();
    void printData();
    //other function prototypes
};
```

//Student.cpp file

```
#include "Student.h" //watch out!
void Student::printData()
{
    cout<<"Name: "<<name<<endl;
    cout<<"ID: "<<ID<<endl;
    cout<<"Marks: "<<score<<endl;
}
Student::Student() {
    name = "Any name";
    ID = 111; score =100;
}
    //other function definitions
```

//myMain.cpp file

```
#include "Student.h" //watch
    out!
int main ()
{
    Student x, y, z;
    x.printData();
    cout<<"Good bye\n";
}
```

No angle bracket

Review Program example

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  int ID; string mystr;
  cout << "What's your ID number? ";
  cin>>ID;
  cout << "What's your name? ";
  cin.ignore();
  getline (cin, mystr);
  cout << "Hello " << mystr << "! Your ID is:
  "<<ID<<".\n";
  return 0;
}
```

Question: in the statement cin.ignore(); what is cin? What is ignore()?

Review Program example

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string month, days;
    cout<<"Enter a month number ( 1 = January, etc.): ";
    cin>>month;
    int myint = atoi(month.c_str());
    while( myint<  1 || myint > 12 )
    {
        cout<<"Month number is out of range, please enter a valid month number
    between 1 and 12: ";
        cin>>month;
        myint = atoi(month.c_str());
    }
} Question: what is string? What is <iostream>
Question: in the statement month.c_str(); what is month? What is c_str()?
```


Review Program example

```
#include <iostream>
using namespace std;
#include <string>
int main ()   {
  string k = "hello world!";
  int i = 0;
/* stringName.length() gives the length of
  the string called stringName */
  while (i<k.length())
{
     k[i] = toupper(k[i]);
     i++;
  }
cout<<k<<endl;
}  //Question: What is k in k.length()? What is length()?
```


Note: General Naming convention of Classes/Structs and members

- When declaring a class, capitalize the first letter of each word in a class name; for example: Circle, Rectangle, and MyClass.
- The data members are given the name after m_; e.g., int m_radius;
- The class names in the C++ library are named in lowercase.

Object Oriented Programming

- Allows creating "objects"
  - Use objects to do things
  - Create more complex objects with others
- A program becomes a "cluster of interacting objects"

Why Object oriented approach?

- Appropriate for programs that model the real world
- Accelerate system development
- Simplify systems integration and standardization
- Suitable for building huge applications

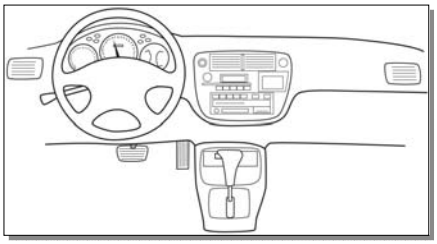
OOP Language Principles

- Encapsulation
- Inheritance (Not included in ES1036)
- Polymorphism (Not included in ES1036)

Encapsulation

- Hides the fine detail of the inner workings of the class
  - The implementation is hidden
  - Often called "information hiding"
- Part of the class is visible
  - The necessary controls for the class are left visible
  - The class interface is made visible
  - The programmer is given only enough information to use the class

Encapsulation



An automobile's controls are visible to the driver, but its inner workings are hidden.

Review

86) A Constructor is a special member function that

a) is called every time an object is created
b) is designed to initialize member variables
c) user can call whenever they want
d) Both A and B

Review

87) A Destructor is a special member function that

a) is called every time an object is destroyed
b) user can call whenever they want
c) Both A and B

Review

88) There can be more than one constructor in a given class

a) True
b) False

Review

89) Public methods in a class can access private members of objects of the same class

a) True
b) False

Review

90) Private member functions of a class can access private members of other classes

a) True
b) False

Review

91) What are the access privileges of data members in a class

a) They all have to be private
b) They can be private or public
c) They all have to be public
d) None of the above

Review

92) Analyze the following code:

```
#include <iostream>
using namespace std;

class A
{
private:
  int s;
public:
  A(int newS){
     s=newS;
  }

  void print(){
     cout << s << endl;
  }
};

int main()
{
  A a;
  a.print();
}
```

a) The program has a compilation error because class A does not have a no-argument constructor.
b) The program has a compilation error because s is private.
c) The program compiles and runs fine and prints nothing.

sizeof operator with objects

sizeof() operator can be used to find out the size of an object in terms of bytes.

Example: Let's assume that we have created a class with two double type data members. Based on the following statements the output will be 16.

```
myComplex C1;
cout <<"Object C1 byte-size: "<< sizeof(C1)<<endl;
```

Output:
`Object C1 byte-size: 16` /*since myComplex object c1 has two double type member variables, the size will be 16 */

Review

93) **When passing a parameter by value, the parameter is copied to a new object**

a) True
b) False

Review

94) **When passing a parameter 'by value' to a function, changes to this parameter inside the function affects the original object**

a) True
b) False

Review

95) **When passing an object by reference during a function call, changes to this object inside the called function affects the original object in the calling function**

a) True
b) False

Review

96) **Output?**

```
#include <iostream>
using namespace std;
class A
{
public:
  int x;
  int y;
  int z;

  A()
  {
       x(1); y(2); z(3);
  }
};

int main()
{
  A a;
  cout << a.x << " " << a.y << " " << a.z;
  return 0;
}
```

a) 2 2 2
b) 1 1 1
c) 3 3 3
d) 1 1 2
e) 1 2 3

Review

97) **The _____ operator can be used to assign one object to another.**

a) equality or equivalent-to (==)
b) assignment (=)
c) address (&)
d) None of the above

Review

98) **When you <u>design</u> a class, you decide on the name of the class**

a) True
b) False

Review

99) When you use a class, you decide on the names of the instances (i.e., the object) of that class

a) True

b) False

Review Example: a class called Circle

```
/* This code designs a class called 'Circle' which,
   in the driver function, can create any Circle type
   object and find the area of that object.*/

//Header file: Circle.h
#include <iostream>
using namespace std;
class Circle
{
public:
  Circle();
  Circle(double);
  double getArea();//a get function
  double getRadius(); //a get function
  void setRadius(double); //a set function
private:
  double radius; //a private data member
};
```



```
//Implementation/definition file: Circle.cpp
#include "Circle.h"
// Construct a circle object
Circle::Circle(){
  radius = 1;
}
// Construct a circle object with the given parameter
Circle::Circle(double newRadius){
  radius = newRadius;
}
// Return the area of this circle
double Circle::getArea(){
  return radius * radius * 3.14159;
}
// Return the radius of this circle
double Circle::getRadius(){
  return radius;
}
// Set a new radius
void Circle::setRadius(double newRadius){
  if (newRadius>=0)
    radius = newRadius;
  else
   radius = 0;
}
```


Client/Driver file: myMain.cpp

```
#include "Circle.h"
int main()
{
  Circle circleObject1;
  Circle circleOject2(5.0);
  cout << "The area of the circle of radius "<<
   circleObject1.getRadius() << " is " << circleObject1.getArea()
   << endl;

  cout << "The area of the circle of radius "<<
   circleObject2.getRadius() << " is " << circleObject2.getArea()
   << endl;
  // Modify circle radius

  circleObject2.setRadius(100);

  cout << "The area of the circle of radius "<<
   circleObject2.getRadius() << " is " << circleObject2.getArea()
   << endl;
  return 0;
}
```


Review Example: Designing a Class Called myComplex

- Create a Class called myComplex which can be used as a complex number adder
  - Initialize a complex number
  - Add a complex number with the initialized one
  - Display the complex number
- Design Issues:
  - Desired operations (add and display) should be carried out by member functions (also known as [AKA] member methods)
  - Required data members (real and imaginary coefficients) need to be used to store object state
  - Access privileges (public, private) of the member methods and data members need to be outlined clearly

Requirement: Class Definition

- A class definition has two parts
  - Class declaration
    - Defines name of class, data members, and prototypes for the member functions
    - Defines access privileges
    - Contained in ***class_name.h*** file
  - Class implementation
    - Implements the member functions
    - Important: Member functions can access all the data members directly, without the requirement of passing the data members through the formal parameter list
    - Contained in ***class_name.cpp*** file
    - Scope resolution operator (::)
      - The prefix :: is used for all member functions
      - It tells the compiler that the function is a member of a class

Step 1: Class declaration (The header file)

```
//Take the advantage of visual studio (see the lab handout)
//create a MyComplex.h file (not project) to define the class
#include <iostream>
using namespace std;
class MyComplex{
private:
      double m_a, m_b;
public:
      MyComplex();
      MyComplex(double x, double y);
      ~MyComplex();
      void display();
      void add(double x, double y);
};
```

No argument constructor function (no return type)

Constructor function with parameters (no return type)

Destructor function (no return type)

Watch out for the semicolon!

Step 2: Class implementation

```
//Take the advantage of visual studio (see lab-7 info)
/*create a MyComplex.cpp (same name as the header file) file,
not project, to define the member functions*/
#include "MyComplex.h"
//default constructor definition
MyComplex::MyComplex()
{
      m_a=0; m_b=0;
}
//second definition for the default constructor that executes
//faster; WE CAN CHOOSE ANY ONE
MyComplex::MyComplex(): m_a(0), m_b(0)
/*Watch out! it won't take equal sign; compilation error will
be generated if equal sign is used*/
{}
//definition of the constructor with parameters
MyComplex::MyComplex(double x, double y)
{
      m_a=x; m_b=y;
}
```

Quotation mark, not angle bracket!

Step 2: Class implementation CONTINUES

```
//The destructor() function
MyComplex::~MyComplex()
{
      cout<<"I'm the destroyer for "<<m_a
<<"+i"<<m_b..\n";/* writing any message inside this
definition is optional*/
}
/*display() function; we don't need to pass any member
data (m_a or m_b) into any member function*/
void MyComplex::display()
{
      cout<<"The complex number is: ";
      cout<<m_a<<" + i"<<m_b<<endl<<endl;
}
//add() function
void MyComplex::add(double x, double y)
{
      m_a+=x; m_b+=y;
}
```


Step 3: Use the Class MyComplex

```
/*Create a project anyname.cpp and add the
MyComplex.h and MyComplex.cpp files to it and
then implement the anyname.cpp as follows*/
#include "MComplex.h"
int main()
{
      MyComplex a, b(1,2);/*Two MyComplex type
objects a and b; constructor's are called here: which
constructor is called for a (or for b)?*/
      a.display(); /*dot (.) operator lets the
object access its member function*/
      b.display();
      a.add(2,3);
      a.display();
}//destructor for each object is called
//What will be the output?
```


Use the class MyComplex and Demonstrate the Destructor operation

```
/*Question: Run this code and find
out which object gets released first
by the destructor function*/
#include <iostream>
using namespace std;
class MyComplex{
private:
   double m_a, m_b;
public:
   MyComplex();
   MyComplex(double x,
   double y);
   ~MyComplex();
   void add(double x, double
   y);
   void display();
};
MyComplex::MyComplex()
{m_a=10; m_b=-4;}
```

```
MyComplex:: MyComplex(double x, double
   y){m_a=x; m_b=y;}
}
MyComplex ::~ MyComplex()
{
   cout<<"I'm the destroyer for "<<m_a<<"
   and "<<m_b<<endl;
}
void MyComplex ::add(double x, double y)
{ m_a+=x; m_b+=y;}
void MyComplex ::display()
{
   cout<<"The complex number is: ";
   cout<<m_a<<" + i"<<m_b<<endl<<endl;
}
int main()
{
   MyComplex b(1,2),a,c;
   a.add(2,3);
   a.display();
}
```

- Note: The object which is declared (created) first is released at last and vice versa

Answer key

86. D
87. A
88. A
89. A
90. B
91. B
92. A
93. A
94. B
95. A
96. E
97. B
98. A
99. A