C++ PROGRAMMING FUNDAMENTALS

4th Edition

Authors
Arturo L. Jacinto, Jr.
Joy T. de Jesus, Jr.
Brian P. Loya
Cecile Hayag Beltran, M.S.

Series Editors
Avonn C. Nova, MIT
Jaime D.L. Caro, Ph.D.

techFactorsInc

First printing of the fourth edition, 2016
ISBN 978-621-8000-54-4

Published by TechFactors Inc.
Printed in the Philippines

Authors Arturo L. Jacinto, Jr., Joy T. de Jesus, Jr., Brian P. Loya, and Cecile H. Beltran, M.S.
Consultant Bryan G. Dadiz, MIT
Series Editors Avonn C. Nova, MIT and Jaime D.L. Caro, Ph.D.
Cover Design Gilbert Lavides
Content and Editorial Alvin Ramirez, Eireen Camille S. Linang, and John Louie Nepomuceno
Creatives Jiyas Suministrado-Morales, Gilbert Lavides, Loughem Laquindanum, and Christian Sabado
Systems Mark B. Abliter, Kenneth T. Salazar, Allan Nicole C. Celestino, Robie Marc R. Peralta, and Carlo M. Espinoza

Exclusively distributed by TechFactors Inc.
101 V. Luna Road Ext., Sikatuna Village
Diliman, Quezon City
1101 Philippines

Telephone number: (632) 929 6924
E-mail address: info@techfactors.com
Website: www.techfactors.com

FOREWORD

We live in an age of marvels.

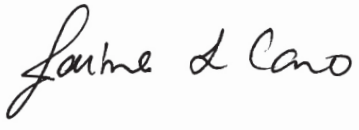
Nowadays, we can move machines with the touch of a button. We can communicate with people halfway across the world in real-time. Innovations occur everyday because of technology.

For adolescents today, technology is already a part of daily living. They use mobile phones, computers, iPods, and other electronic gadgets that assist and expand their awareness of the world. However, they have so much more to learn about the full potential and limitations of the technology within their reach.

In an increasingly computer-dependent world, it is important to be aware of the changes in computing technology, and to be knowledgeable in the various ways that computers can help us in everyday life. This courseware is not only intended to be an instructional manual discussing individual topics, but also to be a means of exploring the continually improving and expanding world of computers – and our society as well.

The Series Editors

Avonn C. Nova, MIT



Jaime D.L. Caro, Ph.D.

ABOUT THE AUTHORS

Arturo L. Jacinto, Jr. took up B.S. Computer Science Major in Information Technology at the De La Salle University. Since then, he has worked as a programmer, systems analyst, project leader, and senior consultant in various corporations. Prior to his inclusion in the TechFactors, Inc. team, he also served as Associate Dean and instructor at an ICT college, teaching computer science subjects. He wrote the first edition of *Programming Fundamentals* and edited the first edition of *ICT and Society*, both published by TFI.

Joy T. de Jesus, Jr. specializes in ATM-based and ATM-related systems. He has served as trainor and content developer of Techfactors Inc., co-authoring the following books: Office Productivity, IT Project Management, Java Programming By Example, Animation in a Flash, and C++ Programming Fundamentals. He studied Computer Science major in Software Technology at De La Salle University and pursued further studies at Mapua Information Technology Center.

Brian P. Loya graduated from UP Diliman with a B.S. in Computer Science. In college, he belonged to several organizations such as UP SIKAT- an acting organization, UP PARSER-the official publication of the Computer Science Department, and UP KUSTURA- an organization for Marikina-based students. He coauthored the book Animation in a Flash, edited the second edition of Java™ Programming by Example, made numerous games for the interactive CDs, and was part of the TechFactors development team.

Cecile Hayag Beltran is a B.S. Computer Science graduate of Pamantasan ng Lungsod ng Maynila and has an M.S. in Information Technology. She has more than 10 years of experience teaching in De La Salle University Dasmariñas and STI College Southwoods. She was an administrator at the Asian Institute of Computer Studies in Sta. Rosa and a Student Services head at the STI College Southwoods.

ABOUT THE SERIES EDITORS

Avonn C. Nova, MIT has more than 10 years of teaching experience in the field of Computer Science and Information Technology. He received his Masters in Information Technology degree at Technological University of the Philippines in 2006 and Bachelor of Science in Computer Science (cum laude) at Cavite State University in 2001. He already earned units for Doctor of Philosophy major in Education Administration at Manila Central University and served the institution as Dean of the College of Computer Studies for more than 5 years where he pioneered the specialization in Biomedical Informatics.

Jaime D. L. Caro, Ph.D. has more than 20 years of experience in education and research in the areas of Computer Science, Information Technology, and Mathematics. He received the degrees of Bachelor of Science major in Mathematics (cum laude) in 1986, Master of Science in Mathematics in 1994, and Doctor of Philosophy in Mathematics in 1996, all from the University of the Philippines, Diliman. He spent a year as a post doctorate research fellow at the University of Oxford from 1997 to 1998. He is presently Assistant Vice President for Development of the University of the Philippines, Program Director of the UP Information Technology Development Center (UP ITDC), and a professor of Computer Science in UP Diliman. He is an honorary member of the Philippine Society of Information Technology Education (PSITE), President of the Computing Society of the Philippines (CSP), and a member of the Technical Panel on IT Education of the Commission on Higher Education (CHED).on Higher Education (CHED). Dr. Caro is a recognized expert on Complexity Theory, Combinatorial Network Theory, Online Communities, and e-Learning.

TABLE OF CONTENTS

INTRODUCTION

From typing data to determining the weather condition, software programs have definitely optimized the use of computers. Such applications are created and developed through programming languages. They evolved as computers have, that is, from machine languages of 0s and 1s to object-oriented and distributed programming languages. Since information technology is ever-growing, programmers should be able to easily shift from a simple language to a more complex one. This module introduces the students to an object-oriented programming language called "C++."

LEARNING GOALS

By the end of this courseware, the student is expected to:

1. Discover object-oriented programming and its applications.
2. Recognize the different control structures of C++.
3. Create C++ programs based on certain requirements.
4. Demonstrate the ability to modify and debug C++ programs.

LESSON 1

No; she was not made for mean and shabby surroundings, for the squalid compromises of poverty. Her whole being dilated in an atmosphere of luxury; it was the background she required, the only climate she could breathe in. But the luxury of others was not what she wanted. A few years ago it had sufficed her: she had taken her daily meed of pleasure without caring who provided it. Now she was beginning to chafe at the obligations it imposed, to feel herself a mere pensioner on the splendour which had once seemed to belong to her. There were even moments when she was conscious of having to pay her way.

—from *The House of Mirth*, Edith Wharton

Programming Languages and Paradigms

During the Victorian era in England, society placed a great deal of importance on one's class standing. You had to belong to a certain economic bracket or maintain a certain social standing in order to be accepted into the mainstream. Status was important, as well as the perceptions of other people. The class divisions in English society were enforced by tradition, and could be seen in the way people dressed, acted, and spoke.

Language is a good indicator of a person's background. The way we use words and create sentences is largely influenced by the environment we grew up in. But it's not only people that use language—computers use a kind of language as well. Programming languages have also been heavily influenced by the prevailing computing philosophies during the time of their development.

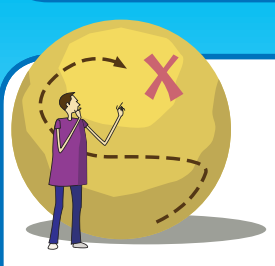
Evolution of Programming Languages

Parallel developments in hardware technology, as well as increasing demands from users, have significantly influenced the development of programming languages. And just like how human languages have developed and gained sophistication over time, they have also undergone a similar evolution as they moved from one generation to the next. By "generation," we mean a particular period in computing history that was characterized by a specific approach in programming.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Identify and differentiate the generations through which programming languages have evolved.
2. Identify and differentiate the paradigms or models that programming languages are categorized in.

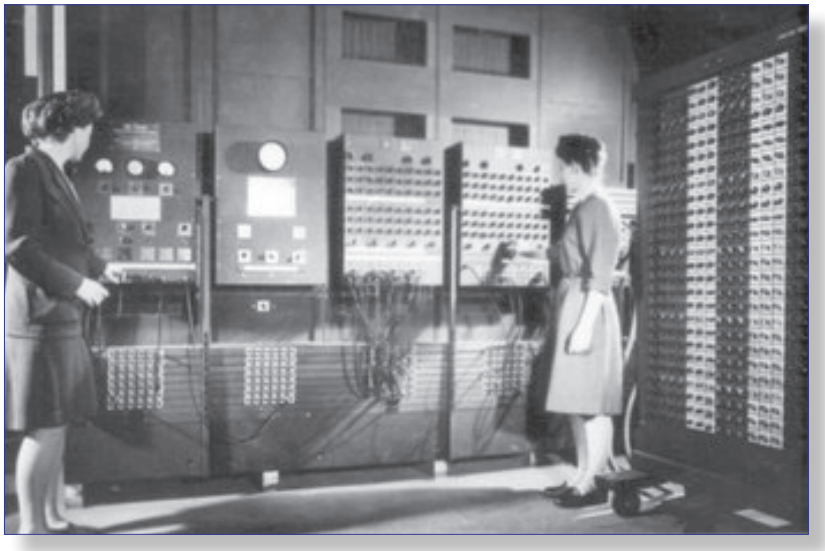


LESSON OUTLINE

1. Evolution of Programming Languages
2. Overview of Programming Paradigms

First Generation: Machine Languages

The late 1940s to early 1950s saw the emergence of first generation languages (**1GL**). They are known as **machine languages** and use a **binary code** that consists of strings of only zeroes (0) and ones (1). These are languages that the computer can understand and execute directly without any need for translation. Programs written in 1GL relatively run quickly and efficiently. But 1GL is very difficult to learn and use; instructions need to be entered through switches on the front panel of the machine. In addition, a 1GL program written for one computer might not be readily transferable to and executable on another computer. This clearly shows that 1GL is **machine-dependent**—machine instruction sets between computers can be different depending on the **processor family** and environment where the computer belongs.



Electronic Numerical Integrator Analyzer and Computer (ENIAC)
Picture from Wikipedia, the free encyclopedia on the Internet

CONNECT

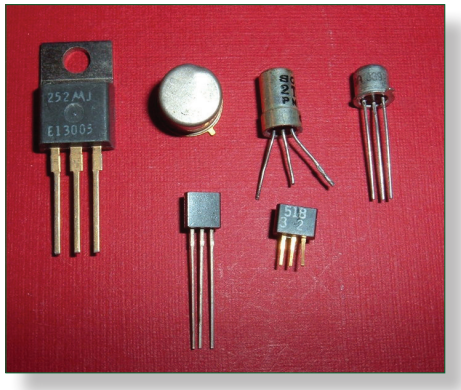
Computer hardware evolved through similar generations. The developments in hardware technology also influenced the evolution of programming languages.

1st Generation → vacuum tubes
2nd Generation → transistors
3rd Generation → integrated circuits (ICs), small-scale integration (SSI), medium-scale integration (MSI)
4th Generation → large-scale integration (LSI), very large-scale integration (VLSI)
5th Generation → parallel processing architectures
6th Generation → massively parallel architectures, parallel/vector architectures

Second Generation: Assembly Languages

In the early to mid-1950s, second generation languages (**2GL**) came out. They usually refer to some form of **symbolic** or **assembly language**—instead of zeroes and ones, assembly languages use mnemonics or very short words for commands. Compared to 1GL, 2GL is a little less difficult for programmers to learn and use. However, 2GL programs need to be converted into machine language first by an **assembler** before they can be run. This involves mapping the assembly language code into the target computer's binary machine code. Thus, 2GL is also machine-dependent. Fortunately, not much overhead is introduced during this translation process and the converted machine language program still runs fast and efficiently. Yet it is still a challenge for programmers to write large applications effectively through 2GL.

1GL and 2GL are both **low-level languages**. They are designed to facilitate fast and efficient execution in computers, without much regard for the convenience of the programmers who use them. In those days, computers did not have much computing power to spare. So programs had to be direct to the point—that is, programs were written to make it as easy as possible for computers to run them (the burden was clearly on the side of the programmers who had to go down to the level of the machine in order to speak its language).



Assorted Component Transistors
Picture from Wikipedia, the free encyclopedia on the Internet

Third Generation: High-Level Languages

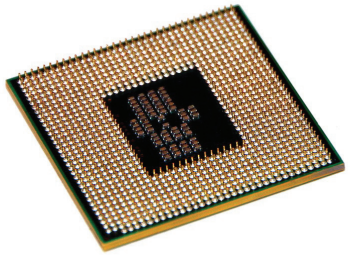
Third generation languages (**3GL**), also called **high-level programming languages**, began to unfold during the mid- to late-1950s. With developments in computer hardware during this period showing a trend towards a rapid increase in computing power, developments in programming languages started to shift towards a high-level language, which is closer to human language. Finally, the programmers' convenience and concerns began to matter.

Most modern computer languages belong here. 3GL introduced the use of **data structures** and **control structures**, which are high-level abstractions of programming concepts that previously had to be coded directly in low-level languages. Most 3GL programs also support the concept of **structured decomposition**—that is, breaking down a program into smaller modules or subprograms that can be reused. All of these factors contributed to making 3GL programming an easier and faster experience.

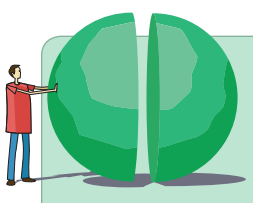
Unlike low-level languages, 3GL is largely machine-independent or **portable**. A 3GL program written for one computer does not need to be rewritten for another computer that

belongs to a different processor family and environment. However, the **source code** may need to be *recompiled* first using the language **compiler** for the other computer.

Examples of third generation languages include Fortran, Algol60, Algol68, Basic, Pascal, C, and Ada.



Microprocessor



CONNECT

Compilation is the process that translates the source code written by the programmer into an **object code** that is directly executable by the computer. However, the compiled object code might not be as optimal compared to an equivalent program written directly in machine language. Translating a high-level language to a low-level language introduces overhead code into the translated low-level language program. This is because high-level data structures and control structures correspond to a significant amount of machine language code.

Fourth Generation: Declarative Languages

Despite the improved capabilities of 3GL, programming continued to be a slow, frustrating, and error-prone process. The amount of programming work soon exceeded the amount of time available for the programmers to do it. This situation only worsened over time and soon brought about the first "programming crisis" in the early 1970s. This, in turn, led to the development of the next generation of programming languages.

Fourth generation languages (**4GL**) are more advanced than traditional high-level programming languages. Commands are usually English-like and do not require traditional **input-process-output** logic. They are also referred to as **non-procedural specification languages**—a programmer who writes 4GL programs concentrates more on what needs to be done (the result/output) rather than how to do it (the steps/process). This generation also includes recent programming languages where many functions and commands are embedded in graphical interfaces and can be activated by simple click-and-drag actions with the mouse.

In general, 4GL is intended to reduce the following:

a. programming effort
b. time it takes to develop software
c. cost of software development

These are not always achievable using 4GL and, sometimes, the effort results in inelegant and unmaintainable code. In the interest of simplifying programming, some of the power and flexibility available from lower-level languages have also been sacrificed. But given the right problem, the use of an appropriate 4GL can be very successful and produces impressive results.

Ordinary Personal Computer

Examples of fourth generation languages include Standard ML, Lisp, Haskel, SQL, Oracle Designer & Developer, Informix 4GL, Progress 4GL, and Visual Basic.

Fifth Generation: AI

Artificially-Intelligent Being

Thought to be the wave of the future during the 1990s, fifth generation languages (**5GL**) are basically an outgrowth of **artificial intelligence** (**AI**) research from the 1980s. Conventional languages then rely on algorithms defined by the programmer to solve problems; the approach is to build a program that implements a specific algorithm devised by the programmer to address a specific problem. On the other hand, 5GLs operate on the concept of solving problems based on constraints or rules that have been declared in the program. The focus is on making the computer program solve the problem for you. The programmer no longer needs to formulate a specific algorithm to solve the problem. This strategy makes 5GL well-suited for AI applications, expert systems, and neural networks.

CONNECT

A **neural network** is a computer architecture modeled after the human brain's network of neurons. It imitates the brain's ability to adapt and learn from past patterns. An **expert system** is an application that uses a knowledge base of human expertise, heuristics, and an inference engine to suggest solutions to problems in a particular subject.

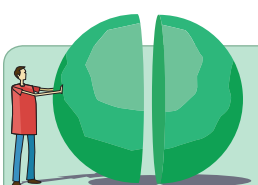
Most constraint-based languages, logic programming languages, and some declarative languages belong to 5GL. Prolog, OPS5, Mercury, and Wolf are the best known fifth generation languages.

Overview of Programming Paradigms

A programming language provides us with the means for specifying computations and telling computers what to do. However, not all programming languages follow the same approach or model for computing. There are different ways of thinking about computation. Depending on the nature of the problem and what needs to be done, one style may be much more effective than the other.

Imperative Programming

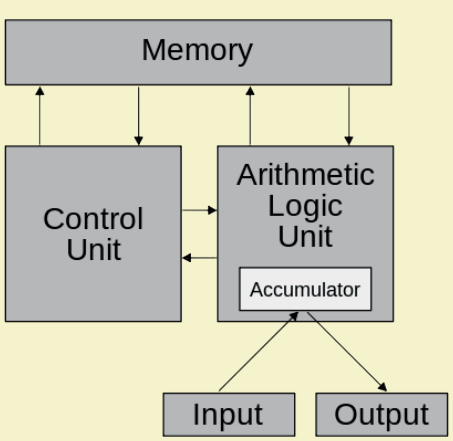
Imperative programming is the oldest and most traditional model of computing. In a regular, non-programming context, the word "imperative" refers to a mood expressing an essential order or command. Here, it describes computation in terms of a program state and the actions that change that state. Imperative languages evolved from machine and assembly languages, whose characteristics reflect the principles of the **von Neumann architecture**. The program consists of explicit commands or instructions to be executed, performing operations on data and modifying values of program variables and the external environment.



CONNECT

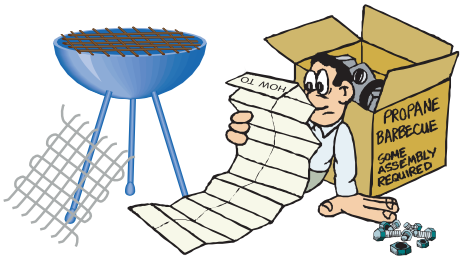
Von Neumann Architecture

John von Neumann is a Hungarian-American mathematician who created the first practical stored-program computer architecture in the mid-1940s. It is comprised of the five classical components (input, output, processor, memory, and datapath).



The processor is divided into an arithmetic logic unit (ALU) and control unit—a method of organization that persists until now. Within the processor, the ALU datapath facilitates the transfer of data for computations. The registers are fast memory modules from/to which data can be read/written to support the computations. Within the ALU, an accumulator supports increment of values corresponding to variables such as loop indices.

An imperative programming language is "how"-oriented. It specifies how the computation is to take place. Therefore, an imperative program is a sequence of commands or instructions for the computer to follow.



Examples of imperative programming languages include Ada, Fortran, Basic, Algol, Pascal, Cobol, Modula, and C.

Functional Programming

Functional programming is an expression-oriented programming paradigm. Computations are specified through mathematical functions that evaluate input expressions and convert them into output values. The emphasis is on the evaluation of expressions rather than the execution of commands. The expressions are formed by using functions to combine basic values.

Functional programming languages are "what"-oriented. They describe the problem to be solved rather than specify the actual steps required for the solution (imperative programming). We should also note that some Functional Programming languages are in postfix notation which means that the operation is at the beginning of the statement.

Example in Scheme:
The following statement
+ 9 0
results to
9

Examples of functional programming languages include Lisp, Scheme, FP, Standard ML, and Haskell.

Logic Programming

In the logical paradigm, programs are written as logical statements that describe the properties the solution must have. The program is viewed as a logical theory and computation is basically the search for proof. It may also be stated that **logic programming** is based on the concept of logical deduction in symbolic logic, or the manipulation of symbols. Thus, it has very high potential for AI applications.

Logic programming languages could also be thought of as being "what"-oriented. A logic program does not specify how to compute the solution, but rather consists of a declarative description of the problem as a set of rules. Solutions are then inferred from these rules.

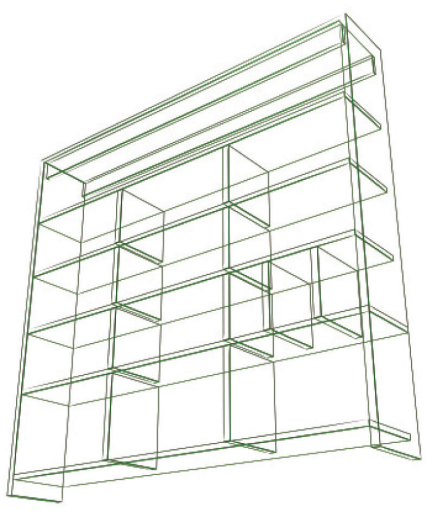
Examples of logic programming languages include Prolog, Lambda Prolog, CLP, and PICAT.

Object-Oriented Programming

In **object-oriented programming**, data structures are viewed as objects, and programmers create relationships between these objects. A group of objects that have the same properties, operations, and behaviors is called a **class**. By reusing classes developed for other applications, new applications can be developed faster and with improved reliability and consistency of design.

We could also look at the object-oriented paradigm as an attempt to model a real-world system. A software system is thus modeled as a set of real-world objects that interact with each other. An object-oriented program is the specification of a set of objects: their attributes and the way they interact with each other. **Computation** then refers to the transformation of the state of these objects.

Examples of object-oriented programming languages include Smalltalk, Simula, Java, and C++.



Concurrent and Distributed Programming

Concurrent or **parallel programming** is a computer programming technique that allows for the execution of two or more operations at the same time. In a non-programming context, "concurrent" means "happening or occurring at the same time." The term "parallel" suggests that the processes run side-by-side yet independently of each other.

Conventional paradigms deal with programming operations in linear fashion—that is, from start to finish and in one continuous execution. The idea in concurrent programming is to split a single task into several smaller *subtasks* that can run relatively independently of each other. This means that the subtasks do not require any input from other subtasks they are running simultaneously with. Their individual results can then be combined later on to form a single coherent solution.

Parallel programming is effective mainly for programming problems that can be readily broken down into independent subtasks, such as certain mathematical problems like factorization. It can be implemented on a single computer, or across a number of computers. When a multi-processor or multi-computer platform is used to implement this, we use the term **distributed programming**. Multi-processor machines achieve better performance by taking advantage of this kind of programming technique.

Most commercial languages have concurrent capabilities (Parallel Pascal, Java, Concurrent ML, etc.), but they offer only a limited implementation of the full potential of the concurrent programming paradigm.

SUMMARY

Programming languages have gone a long way since the early days of low-level languages. In those days, programs had to be easy for machines to execute but were difficult for programmers to understand and use. As hardware computing power escalated and the science of programming gained sophistication, developments in programming languages also brought it closer to the level of human languages. The trend, even to this day, is to make programming a faster and easier undertaking, and at the same time make programs more "intelligent." Computing philosophies or paradigms heavily influenced the design and implementation of most of these programming languages. Each paradigm views computing in a different way and has its own approach towards problem-solving. But certain programming languages are better suited for certain categories of programming solutions.

WORD BANK

Assembler – a program that translates assembly language programs into machine code

Binary – having only two possible values (e.g., 0 or 1, Yes or No, On or Off)

Compilation – the process of translating a high-level language program into machine code

Distributed Programming – concurrent programming implemented on a multi-processor or multi-computer platform

Functional Programming – computations are specified through mathematical functions that evaluate input expressions into output values

Imperative Programming – a programming paradigm based on explicit commands or instructions to be executed, performing operations on data and modifying values of program variables and the external environment

Input-Process-Output – refers to the process of gathering input from the computer user, and the delivery of output after the encoded data are processed by the computer

Logic Programming – a programming paradigm based on logical deduction; a declarative description of the problem as a set of rules is provided, from which the solutions are then inferred

Object Code – the machine code that results from the compilation of the source code

Object-Oriented Programming – a programming approach where not only the data type of a data structure is defined, but also the types of operations or functions that can be applied to the data structure

Portable – describes a program that can be readily transferred from one computer to another without the need to rewrite the source code (i.e., platform-independent)

Processor Family – a grouping terminology used to distinguish a particular set of processors being manufactured by companies which share some common features in design and/or architecture

Programming Paradigm – a model or way of thinking about computing

Source Code – the code written by the programmer before it is compiled into object code

Structured Decomposition – the process of breaking down a program into smaller modules or subprograms that can be reused

PERFORMANCE TASK

Create a group composed of five members to do group research work on C++. Research the history of C++, determine the generation C++ was developed in and determine the type of programming paradigm(s) C++ can be classified in.

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

1. Considering how difficult they were to learn and use, what was the motivation for developing and using low-level languages?

   _____
   _____
   _____

2. How do you describe the general trend in the development of programming languages through the four generations?

   _____
   _____
   _____

3. What is the von Neumann architecture?

   _____
   _____
   _____

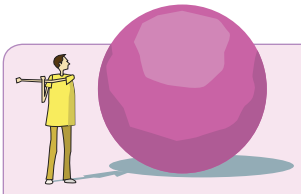
SKILLS WORKOUT

Classify the following languages according to programming paradigm.

Algol, Basic, C++, CLP, Cobol, Concurrent ML, Fortran, Haskell, Lisp, Modula, Parallel Pascal, Prolog, Scheme, Simula, Smalltalk

Imperative	Functional	Logic	Object-Oriented	Concurrent/ Distributed

CUT THIS PAGE



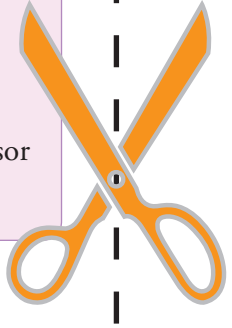
SKILLS WARM-UP

A. On the blanks provided, write the generation corresponding to the programming languages and technologies listed below.

___ 1. portable programming languages
___ 2. declarative languages
___ 3. assembly language
___ 4. machine language
___ 5. high-level programming languages
___ 6. artificial intelligence
___ 7. vector architecture
___ 8. small-scale ICs
___ 9. medium-scale ICs
___ 10. large-scale ICs
___ 11. very large-scale ICs
___ 12. compilers
___ 13. vacuum tubes
___ 14. transistors
___ 15. parallel processing architecture

B. On the space provided below, identify the terms being described by each item.

_____ 1. a programming language that uses code which consists of strings of only zeroes and ones
_____ 2. a program that translates assembly language programs into machine code
_____ 3. breaking down a program into smaller modules or subprograms which can be reused
_____ 4. human-readable code written by the programmer
_____ 5. a directly executable translation of the code written by the programmer
_____ 6. the process of translating the code written by the programmer into a code that can be directly executed by the computer
_____ 7. a type of computer architecture modeled after the human brain's network of neurons
_____ 8. an application which uses a knowledge base of human expertise, heuristics, and an inference engine to suggest solutions to a problem
_____ 9. a type of computer architecture comprised of five components, namely the input, output, processor, memory, and data path components
_____ 10. a programming paradigm based on explicit commands to be executed
_____ 11. the programming paradigm in which computations are specified through mathematical functions
_____ 12. a programming paradigm in which the solutions are inferred from a declarative description of the problem stated as a set of rules
_____ 13. a programming approach where both the data type of a data structure, and the operations that can be applied on these, are defined
_____ 14. a programming technique that allows the execution of two or more operations at the same time
_____ 15. parallel or concurrent programming implemented on a multi-processor or multi-computer system



PERFORMANCE TASK

Create a group composed of five members to do some group research work and discuss the different programming generations and paradigms. Look for additional information about what encourages the development of each of the different programming paradigms.

Also, try to find out what the dominant or common types of software programs written in each generation are, and in what fields these programs were used. Present the results of your research and discussions in class for additional insight. Use the space provided for your notes.

LESSON 2

He robbed him of a great deal of his natural force, and so do all those who try to turn books written in verse into another language, for, with all the pains they take and all the cleverness they show, they never can reach the level of the originals as they were first produced.

—from *Don Quixote*, Miguel de Cervantes

C++ Overview

In the second volume of *Don Quixote*, the quixotic Don finds himself in the pages of a world-famous book about—guess who—himself and his legendary adventures with his loyal squire Sancho Panza. But, according to him, a lot of entries in the book are not accurate (as if he is not mad enough to know if any of the events in his life was real!). Now this, in reality, is the author Miguel de Cervantes saying that his contemporaries' attempts to copy or improve upon his novel still fall short of what he achieved in the first volume.

C++ Roots

Not so long ago, in the late 1980s, AT&T developed the **C** programming language as an **operating system** (OS) for the PDP-11 series of computers (C eventually became the UNIX OS). It was the most popular programming language for commercial software development then, so it was improved upon to provide the features and functions that facilitate object-oriented programming (**OOP**). Spanking-new as it was, though, most OOP programs were very sluggish and unproductive. So it was **C++** that ushered in object-oriented techniques while preserving the efficiency of C.

CONNECT

A good C++ program has the elements of both OOP and classic programming approaches.

When C++ was conceptualized by Bjarne Stroustrup, the "C factor" was taken into account: people who don't have knowledge of C should still be able to use C++. You C (pun intended), even a budding programmer can immediately start using C++ since it has a different problem-solving approach. On one hand, C applies: (a) a procedural approach, where the program is

LESSON OUTCOMES

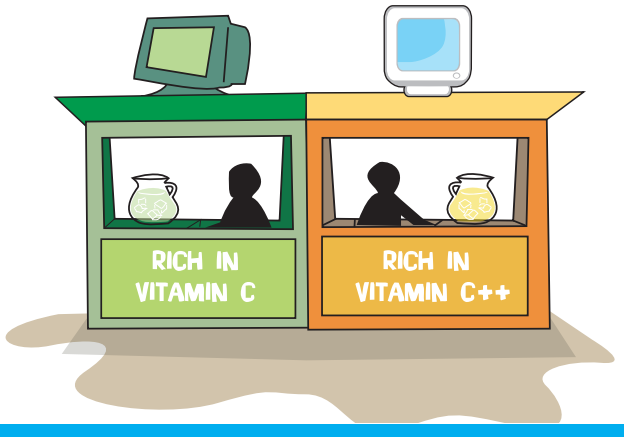
At the end of this lesson, the student will be able to:

1. Discover how C++ originated from C.
2. Examine the underlying concepts of OOP.

LESSON OUTLINE

1. C++ Roots
2. What is OOP?
3. OOP Concepts

performed step-by-step in a series, and (b) a structured approach, where the program with a complicated engine is divided into several small subsets that are each performed separately, later building up to be the whole set. On the other hand, C++ applies both of these, plus OOP. C++ is more than just an updated C; in fact, it provides more functionality and usability than the original C language.



What Is OOP?

Object-oriented programming (OOP) was conceived to model real-world objects. In OOP, the data in a program are viewed as objects with their own attributes and functions. The key to understanding OOP is being able to grasp the concepts behind a "class" and an "object."

An **object** refers to an entity that can be viewed as either concrete or abstract. Examples of concrete objects are cars, books, computers, people, and buildings. Some examples of abstract objects are subject, idea, course, triangle, and number (these are intangible objects).

Now, a **class** is a representation or an "abstraction" of an object. It's like having a plan design—for instance, to describe how a house (the object) will be built or structured, you need a blueprint (the class). It contains the state of the house, its parts, divisions, and purposes. Let's look at another example by drawing attention to ourselves. As humans, we share certain things in common such as what are given in the table below:

Attributes	Behavior
Height	Eat
Weight	Sleep
Hair Color	Drink
Eye Color	Wake-Up
Skin Tone	Talk

We can say that humans are a class of their own. Our physical attributes can be considered our properties and the actions that we can perform are the methods.

In programming, after classes are created, a program is written that uses these classes to solve the problem at hand. A class can be reused in other programs as well by simply referring to them instead of doing them all over again, which makes your other programs more manageable. By reusing classes developed for other applications, new applications can be developed faster and with improved reliability and consistency of design.

So how can we use classes in OOP? Once again let's look at ourselves and those around us. We are surrounded by people. We have our family, our friends, classmates, teachers, and a whole lot more. We can say that we are all human. We are all the same, and yet we are also different. We have different names, heights, weights, etc., but we do the same actions like eating, sleeping, drinking, etc. Each person can be considered an object. And this leads us to define an object as an instance of a class. Classes then are the templates used for objects to be created.

In computer programming, seeing things as objects has added benefits such as:

- The code for an object can be written independently from other objects. This is known as modularity.
- Having objects promotes code re-use. If an object has already been created, it can be shared in other programs that require a similar object. In your own program, you can also have several objects that have the same class.
- It is easier to debug objects since the code is written separately from the rest of the program.

OOP Concepts

Encapsulation

Well-defined classes created in C++, when used by others in their own program, are said to be **encapsulated**. When somebody else uses a certain class and invokes it in a function in his program, he doesn't need to know exactly how it was done, but rather, he needs to know how to use it effectively. Once a function is written, the only concerns left are the data it needs and the output it produces because the statements are already grouped or "encapsulated."

For instance, when a person uses a computer to program something, that person doesn't need to know how the computer operates. The whole process of how the computer functions is well-hidden—the user simply has to use it properly.

Inheritance

In programming, using a predefined class leads to the **inheritance** of its original functions, plus added new ones. To put it simply, instead of creating a program in order to simulate an existing program, one can simply invoke that program and extend it by adding new functionalities in order to save time and effort.

For instance, taking a good look at cellular phones today, one might notice that they function in the same way the original cell phones do: receive calls on the go. However, more functions have been added, like text messaging, music, camera, and Web surfing capabilities. Yet these phones were not created from scratch. By simply utilizing the technology from the original designs and improving on some aspects, the new cell phones simply inherited and retained the designs and functions of the original cell phones, only with added features and improvements.

Polymorphism

"Poly" means many and "morph" means form. So, by adding the two terms, "polymorph" means many forms—and this is exactly what **polymorphism** means: an instance that takes on many types of forms. There may be several instances of different functions, but they may have been derived from only one and the same function but with a different name. By doing so, one can simply make use of that function many times just by changing the instance name.

Going back to our previous example, amidst the different types of cell phones available today, a newer model may still be able to communicate with an older one since a new cell phone still follows the same design as an old one, though with a slightly different format and functionality.

SUMMARY

In this chapter, a short history of C++ was discussed. Then, an explanation of OOP was given, since it is the backbone of C++ and it is what sets C++ apart from C. A simple program was also featured to show that even a beginner can create a C++ program.

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

1. Describe OOP.

_____

_____

_____

2. Differentiate objects from classes.

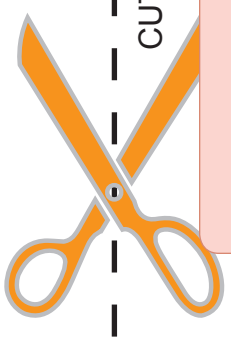
_____

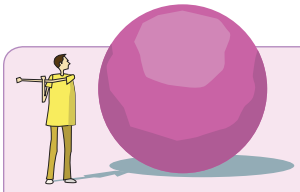
_____

_____

3. Provide 3 real world examples of classes and give a corresponding object of each class.

CUT THIS PAGE





SKILLS WARM-UP

On the blanks provided, identify the term or individual described below.

_____ 1. the procedural programming language C++ is based on

_____ 2. the person who conceptualized C++

_____ 3. a programming paradigm conceived to model real-world objects

_____ 4. a group of objects having the same properties, operations, and behaviors

_____ 5. the process of combining data and function to form a class or an object

_____ 6. a feature of OOP that allows the creation of a class based on another class

_____ 7. a feature of OOP that allows an instance of a class to take on many forms

_____ 8. an instance of a class

_____ 9. a subprogram that performs a specific task

_____ 10. a benefit of using objects wherein it can be written independently from other objects

WORD BANK

C – procedural/structured programming language that uses the top-down design (i.e., the programming approach which breaks a task into smaller subtasks)

C++ – OOP language that is based on the C language

Class – group of objects having the same properties, operations, and behaviors

Data – numerical or other information in a form that may be processed by a computer

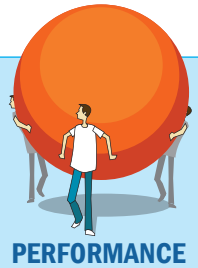
Encapsulation – process of combining data and function to form a class or object

Function – subprogram that does a specific task

Inheritance – creation of one class from another class

Object – instance of a class

Polymorphism – an instance taking many forms



PERFORMANCE TASK

In this chapter, we have learned that C++ can apply procedural, structured, and object-oriented approach in coding. Create a group composed of 5 members to research other procedural/structured and object-oriented languages. Look for another procedural/structural programming language aside from C++ or C and compare that language with C++. Discuss the advantages and disadvantages C++ has over the other language. Also, look for another object-oriented programming language and do the same comparison with C++. Summarize the results of your research and discuss the results in class for additional insight. Write down your notes on the space provided.

LESSON 3

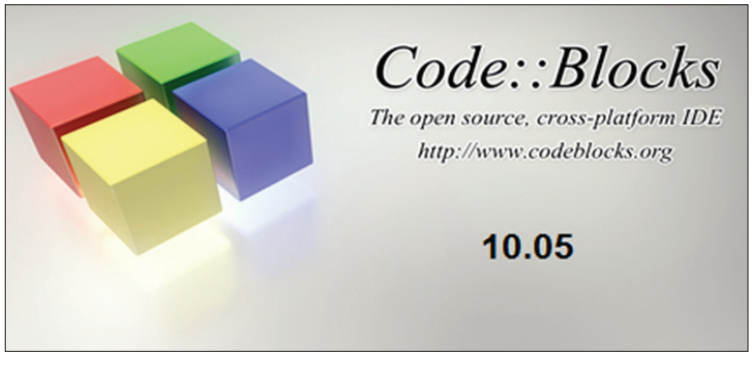
He could see a little white house in the middle of a big lawn. There were vines on the porches, and it must have been early in the evening, for the fireflies were beginning to twinkle over the lawn. And the grass had just been cut, for the air was sweet with the smell of it. A woman, standing on the steps under the vines, was calling "Jules, Jules, it is time to come in, little son!"

But Jules, in his white dress and shoulder-knots of blue ribbon, was toddling across the lawn after a firefly.

—from *The Gate of the Giant Scissors*, Annie Fellows Johnston

Integrated Development Environment

Before you go chase fireflies, there are still a few more things you have to learn about C++. Like the graphical user interface (GUI) in office applications, the integrated development environment (IDE) makes programming easier. In this lesson, we will learn about the Code::Blocks IDE which we will use to program our C++ applications.



What Is Code::Blocks IDE?

Code::Blocks is a free, open source, cross-platform IDE that supports multiple compilers. A compiler turns the code that you have written to an executable program that the computer can run.

Code::Blocks is geared toward the development of C and C++ programs but it can also be used for other programming languages, provided that the respective SDK (Software Development Kit) for that language is installed. Code::Blocks is available for Windows, Linux, and Mac OS X Operating Systems, wherein one of the goals is to provide a unified look across all three OSes.

Being open source, support and development of Code::Blocks may not be as robust as retail C/C++ IDEs such as Turbo C++ or Visual C++. However, Code::Blocks has a strong user base support and there should be no shortage of additional tutorials written by the user community. Being open source, Code::Blocks

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Be familiar with the integrated development environment of C++.
2. Explore the functions of menus and submenus found in IDE.

LESSON OUTLINE

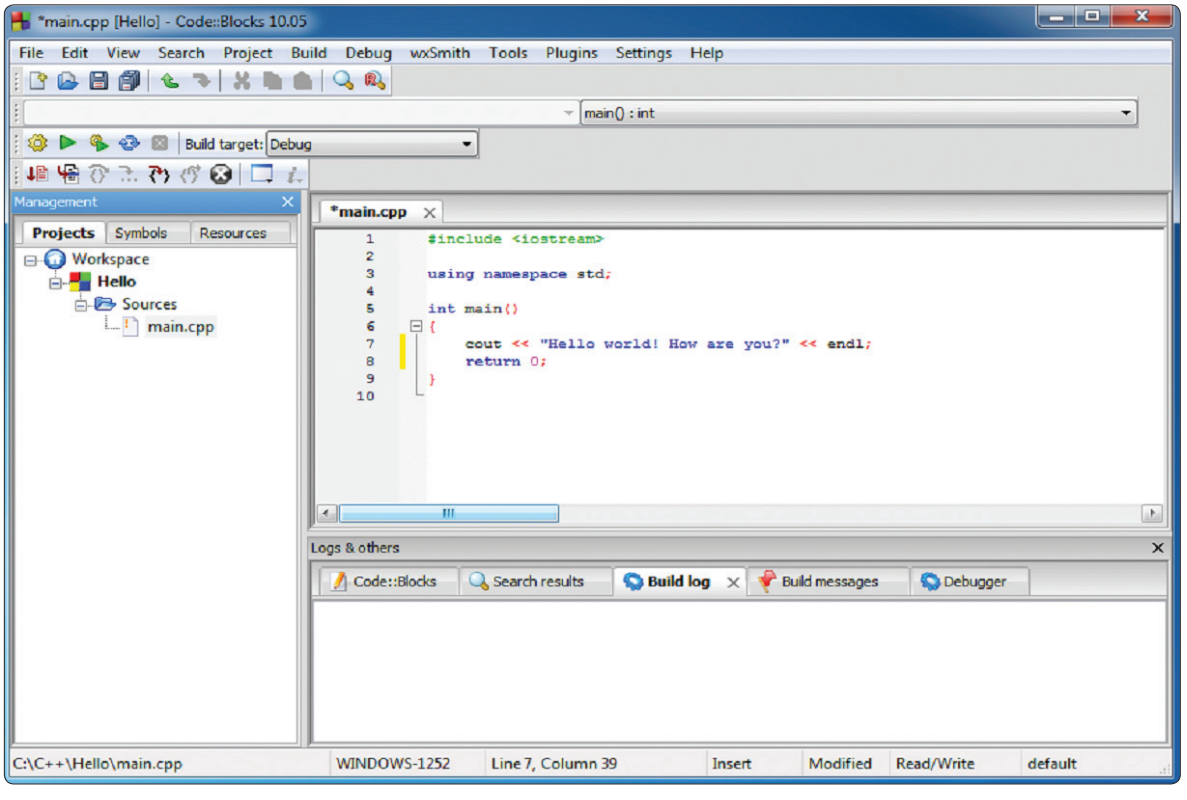
1. What Is Code::Blocks IDE?
2. Getting Started with Code::Blocks
3. Starting a C++ Program

uses the GNU General Public License 3. This means that you can use Code::Blocks to create commercial applications. The GNU License kicks in when you modify the actual source code of Code::Blocks and its related plug-ins and you will have to provide the source code for those modifications.

For our lessons we will be using Code::Blocks 10.05 with MinGW. You can download the latest version of Code::Blocks from their website at http://www.codeblocks.org/downloads/

Getting Started with Code::Blocks

After installing Code::Blocks, you can run the application by selecting the program. When the program launches, you will see the Code::Blocks IDE user interface window on your screen.



The Code::Blocks IDE

The user interface window has various panels. Let us familiarize ourselves with these panels.

1. **Management:** This panel contains the Projects tab which displays all open projects. The management panel also has a Symbols tab that shows symbols, variables, etc., being used. The Resources Tab can also be seen in this panel.

2. **Editor:** In the illustration above, you can find the editor panel with the tab that contains main.cpp. This is the place where you type your code. If you notice, the lines in the editor panel are marked for easy reference.

3. **Logs & others:** This window is used to output search results and messages from the compiler.

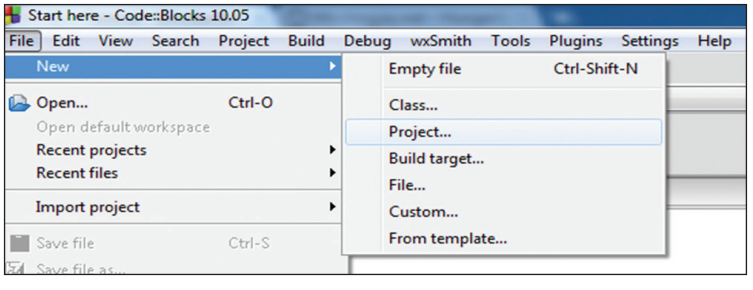
4. **Status Bar:** The status bar can be seen at the bottom of the Code::Blocks user interface. It contains the following information:

    - Absolute path of the file opened on the editor, which can be seen as C:\C++\Hello\main.cpp
    - Default character encoding of the OS. This can be seen as "default." The character encoding as shown in the image is WINDOWS-1252.
    - Row and column of the current cursor position in the editor panel
    - Keyboard mode for inserting text (either Insert or Overwrite)
    - Current state of a file. If the project has been modified, the word "Modified" will appear on the status bar; if not, that area will be empty.

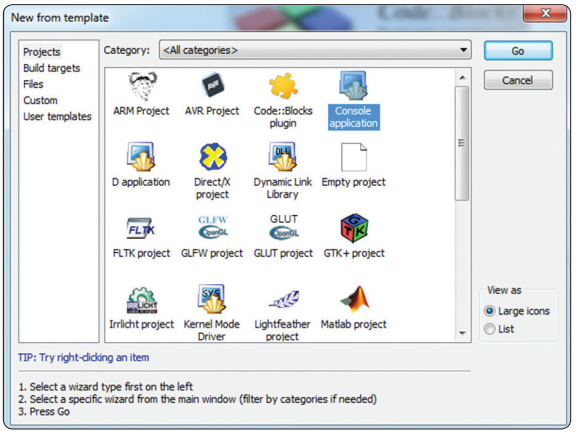
Code::Blocks has many more advanced features suited for professional C++ developers. However, for now, it's good to familiarize ourselves with the basic interface of Code::Blocks.

Starting a C++ Program

Let us begin creating our first C++ Program by following these simple instructions. First, let us open Code::Blocks. Once the IDE has loaded, look at the menu and select File > New > Project.



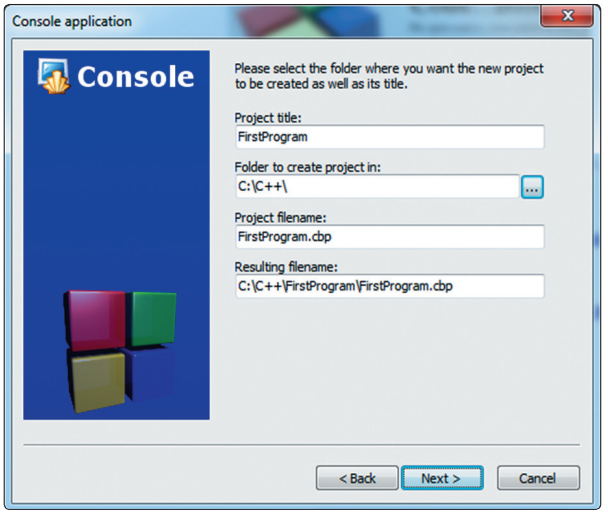
A pop-up window will be launched asking us what kind of project template we would want to use. Select **Console application** and then click on the Go button. Note that for our examples in this book, we will use Console application. This means that the code that we will create will run on the command prompt of Windows.



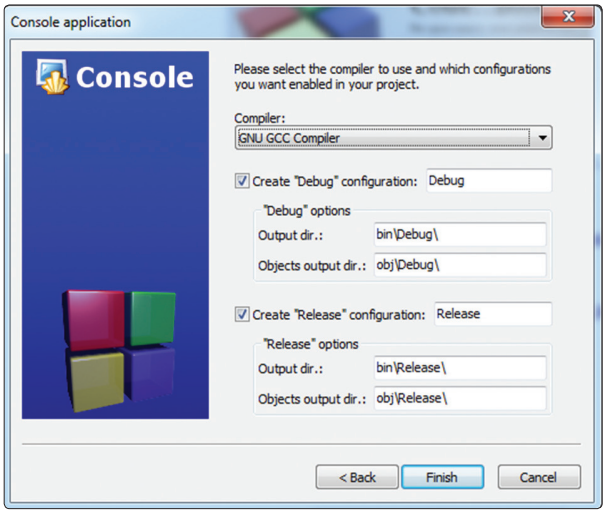
After choosing **Console application** as our project template, you will be asked to select what programming language you want to use. Remember, Code::Blocks can be used to program

for C or C++. Since our lessons are for C++, choose that in the options list and then click on the Next button.

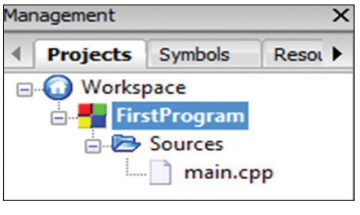
Code::Blocks will then ask you certain details regarding your Project such as the Project title, Project folder, Project filename. After you supply the information, Code::Blocks will show the full path of the resulting filename. For our first program, just copy the values seen on the image and then click on the Next button.

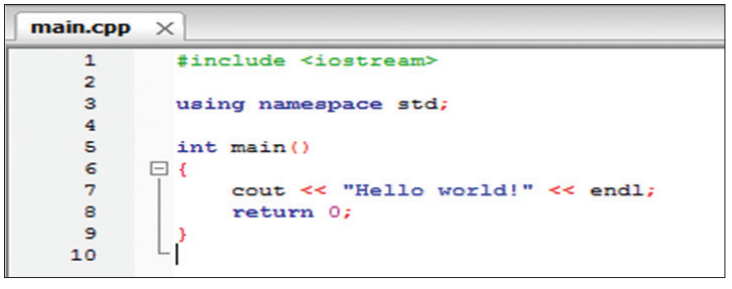


After you submit your project details, you will be asked what compiler to use. For our lessons, let us use the default compiler as shown on the image and then click on the Finish button.



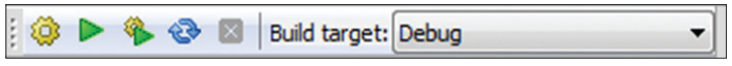
On the Management Panel of the IDE, you will see that Code::Blocks already supplied us with something to work with. Expand the Sources directory and we will see a file called *main.cpp* (**.cpp* is the C++ source code file extension name).

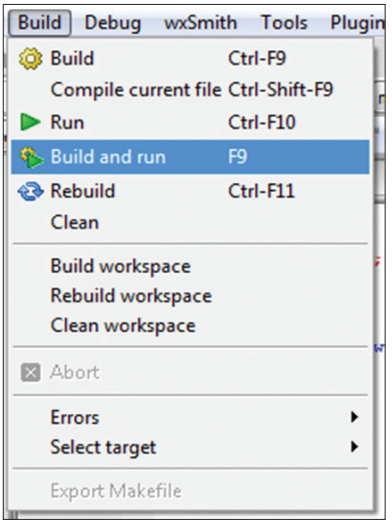




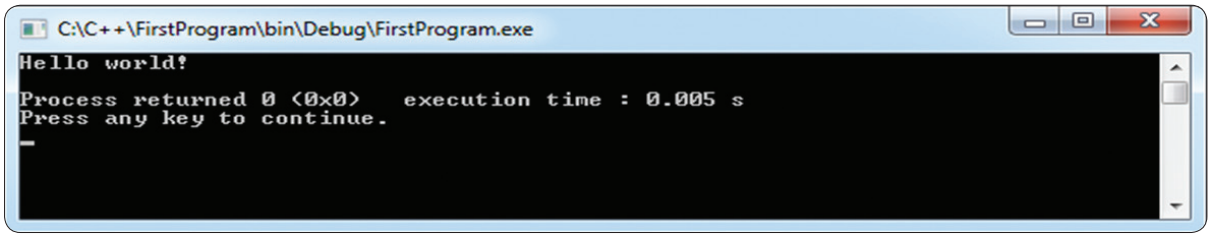
Double-click on *main.cpp* and you will see that the source code will be displayed on the Editor panel.

Code::Blocks has conveniently provided a simple "Hello World" program for us. We will explain the details behind the source code in the next lesson but for now, let us focus on how we can run this pre-written program. Look at the Build toolbar or the Build option of the main menu.





Before we can run our source codes, we have to build it. The key difference between Build and Compile is when you choose Compile, the source code is compiled as an object file. With Build, the compiler compiles the source code to an object file and links all the files created to create an executable application. For our purposes, we will just use the **Build and run** option by selecting it on the Build menu, clicking on the gear and play icons of the Build tab, or by pressing the F9 shortcut. Once we do that, we will see the results of the pre-written program as it launches the command prompt window.



And there we have it! We successfully ran the pre-built Code::Blocks "Hello World!" program! In our next lesson, we will deconstruct the source code and delve into more programming concepts to create more complex applications.

Keyboard Shortcuts

Keyboard shortcuts can help speed up your usage of Code::Blocks. While you may not use all the keyboard shortcuts available, it helps to know a few. Below are some useful keyboard shortcuts.

For Editing

Function	Shortcut Key
Undo last action	Ctrl-Z
Redo last action	Ctrl-Shift-Z

Swap header / source	F11
Comment highlighted code	Ctrl-Shift-C
Uncomment highlighted code	Ctrl-Shift-X
Auto-complete / Abbreviations	Ctrl-Space/Ctrl-J
Toggle bookmark	Ctrl-B
Goto previous bookmark	Alt-PgUp
Goto next bookmark	Alt-PgDown

For Files

Function	Shortcut Key
New file or project	Ctrl-N
Open existing file or project	Ctrl-O
Save current file	Ctrl-S
Save all files	Ctrl-Shift-S
Close current file	Ctrl-F4/Ctrl-W
Close all files	Ctrl-Shift-F4/Ctrl-Shift-W

For Building / Running

Function	Shortcut Key
Build	Ctrl-F9
Compile current file	Ctrl-Shift-F9
Run	Ctrl-F10
Build and Run	F9
Rebuilt	Ctrl-F11

SUMMARY

In this lesson, we have discussed the items and commands that you can see when programming using the Code::Blocks IDE, such as the management panel, editor pane, status bar, etc. As for the main menu, most of the menu items have self-explanatory names like Save, Build, and Run so it is quite easy for the user to know the basic functionalities of C++ like saving, building, and running a program.

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

Answer the following questions in the spaces provided.

1. What does "IDE" stand for?

   _____

2. What does "GUI" stand for?

   _____

3. Which section of the main menu of Code::Blocks contains commands used for creating and opening files?

   _____

4. Which section of the main menu of Code::Blocks contains commands to display the panels?

   _____

5. Which section of the main menu of Code::Blocks contains commands used for finding specific words in your source code?

   _____

6. Which section of the main menu of Code::Blocks contains commands used for executing your program?

   _____

7. Which section of the main menu of Code::Blocks contains commands used for compiling your code?

   _____

8. Which section of the main menu of Code::Blocks contains commands for debugging your program?

   _____

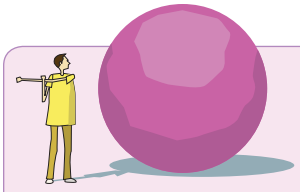
9. Which section of the main menu of Code::Blocks contains commands for managing your project file?

   _____

10. Which section of the main menu of Code::Blocks contains commands used for changing the configuration file directories, compilers, linkers, and coding environment?

    _____

CUT THIS PAGE



SKILLS WARM-UP

Write the letter of the menu in which the options in column A fall under. (Note that the answers in Column B could be used more than once.)

	A	B
_____	1. New	A. File
_____	2. Full Screen	B. Edit
_____	3. Add Files	C. View
_____	4. Go to Declaration	D. Search
_____	5. Save as	E. Project
_____	6. Comment	F. Build
_____	7. Close all	G. Debug
_____	8. Run	H. Tools
_____	9. Step Out	I. Plugins
_____	10. About	J. Help
_____	11. Notes	K. Settings
_____	12. Replace	
_____	13. To Do List	
_____	14. Remove Files	
_____	15. Errors	

WORD BANK

Integrated Development Environment (IDE) - makes programming easier because instead of having to manually code the commands so that MS-DOS will be able to execute your programs, it offers you an interface that combines the individual actions involved in writing, compiling, and running programs

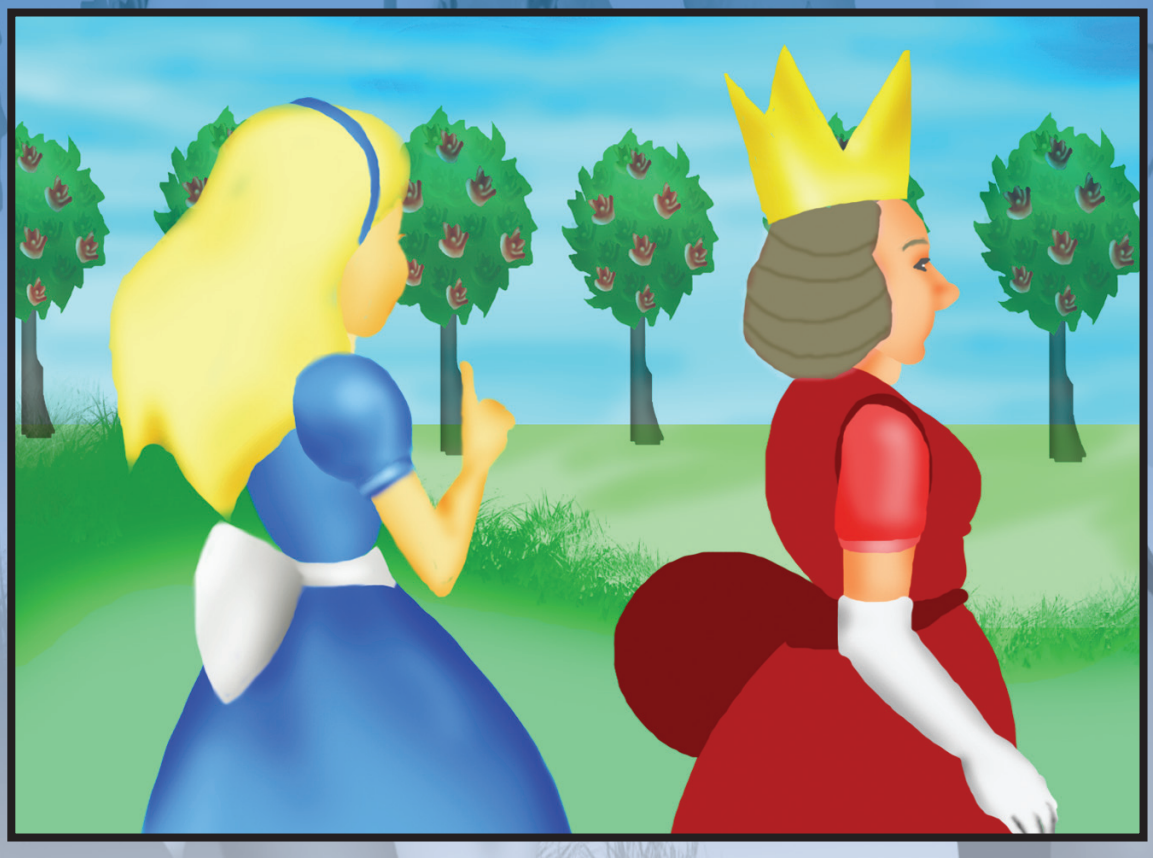
PERFORMANCE TASK

Create a group composed of 5 members. Each member should take turns getting a feel of the Code::Blocks IDE. One of the more interesting features of Code::Blocks is its debugger. Research on this and write down your thoughts on how this will help you program.

PERFORMANCE TASK

Create a group composed of 5 members to research other available IDEs that can be used in writing C++. Research the different specifications of these IDEs such as the IDE's developer, how much it costs, its system requirements, etc. Discuss the similarities and differences between these IDEs and the Code::Blocks. Identify each particular IDE's strengths and weaknesses. Use the space provided for your notes.

LESSON 4



"She's all right again now," said the Red Queen. "Do you know Languages? What's the French for fiddle-de-dee?"

"Fiddle-de-dee's not English," Alice replied gravely.

"Who ever said it was?" said the Red Queen.

Alice thought she saw a way out of the difficulty this time. "If you'll tell me what language fiddle-de-dee is, I'll tell you the French for it!" she exclaimed triumphantly.

But the Red Queen drew herself up rather stiffly, and said "Queens never make bargains."

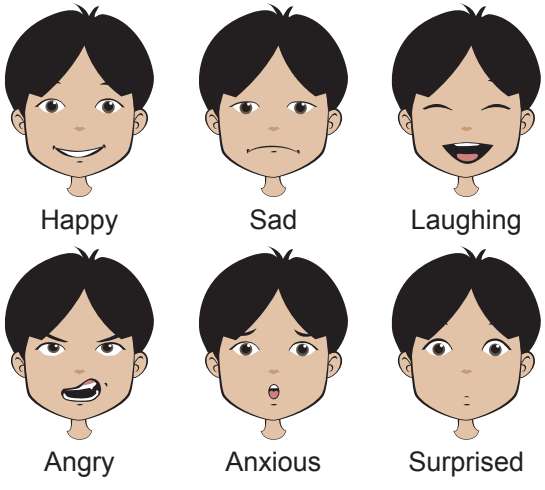
"I wish Queens never asked questions," Alice thought to herself.

—from *Alice Through the Looking-Glass*, Lewis Carroll

C++ Basics

Human language is our primary means of expressing our thoughts and feelings towards one another. Using carefully chosen words that have been strung together according to certain grammatical rules and subject to other linguistic nuances, we are able to express a wide range of emotions and ideas. We can attribute much of the progress we have achieved on our ability to communicate coherent ideas. Expressing ourselves is an important part of language and communication.

In programming, an **expression** refers to any valid combination of symbols that represent values and operations. Each programming language has its own rules regarding what constitutes valid expressions. But just as human expressions convey definite thoughts or feelings, expressions in programs are evaluated to get specific results.



Parts of a C++ Program

Even if different C++ programs have various types of codes and styles, they still contain the basic parts that make up any C++ program. In general, C++ programs are made up of objects, functions, variables, expressions, constants, and a host of other components.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Discover the different parts of a C++ program.
2. Identify the different directives used in a C++ program.
3. Name variables according to the rules of C++.
4. Use constants, operators, and expressions correctly.

LESSON OUTLINE

1. Parts of a C++ Program
2. Global Declarations
3. Data Types
4. Comments
5. Keywords
6. Variables
7. The cout and cin Statements
8. Constants
9. Operators
10. Expressions

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       cout << "Hello world!" << endl;
8       return 0;
9   }
10
```

Let's go back and study the "Hello World!" source code from Lesson 3 and derive the parts of a C++ program from there.

To start off, we have the preprocessor directives at the topmost part of a C++ program. This is where all the functions are imported or, in C++ terms, "included." Without it, you won't be able to execute any command at all, since commands are always included from other functions. Preprocessor directives always start with a hash symbol (#). In the example, the preprocessor directive is called `iostream` as seen in line 1.

In line 3, we are telling the compiler to use a group of functions included in the standard library or `std`. The function that we are using is the `cout` function in line 7. Note the semicolon at the end of the statement; this tells the compiler that the command has ended and this is part of the C++ syntax.

From lines 5 to 9, we have a function. **Functions** consist of lines of code grouped together that perform several tasks in a program. They are easily recognizable because function names are immediately followed by parentheses. The number of functions found in a C++ program can range from one to as many as the user desires. All C++ programs need at least the `main()` function to execute properly.

If a program is executed without a `main()` function anywhere on it, it will not run at all! The `main()` is required to be present for the program to compile and run, assuming that all other lines of code are in order. Line 5 shows the declaration of the function with an `int` data type wherein we expect that the result of executing the `main()` function will return an integer value.

The curly braces in lines 6 and 9 mark the beginning and end of the function. Between the two curly braces are the code blocks where we can write our programming code. The function `cout` (pronounced as "C out") is used to output something on the console. It uses the "<<" insertion operator to indicate what to output, and in this case, the string within the quotes will be outputted. `endl` means to display a new line.

Before the function ends, we have to return a value to show whether or not the function successfully executed. A return value of 0 means success. We can see this in line 8.

Right now this may seem very confusing but just like with anything, programming is a skill honed with practice and experience. The following sections will discuss other parts of a C++ program.

SYNTAX

General Format of a C++ Program

```
Preprocessor Directives
Global Declarations

Data type f1(parameter list);


Data type f2(parameter list);
.
.
.
Data type fn(parameter list);


main()
{
local variables
statements
}
Data type f1(parameter list)
{
local variables
statements
}

Data type f2(parameter list)
{
local variables
statements
}
.
.
.
Data type fn(parameter list)
{
local variables
statements
}
```

Global Declarations

Global declarations contain user-defined variables and constants. Constants will be discussed later but for now, it is important to remember that they should be given initial values which they retain throughout the whole program. With constants, the values they contain cannot be changed during the execution of the program. **Variables**, on the other hand, change values throughout the program.

CONNECT

When declaring a variable, make sure that it always starts with a letter or an underscore (_). Never declare a variable that starts with a number, has spaces in-between, or has the same name as a C++ keyword.

Data Types

In the previous lesson, we came across variable declaration. When we declare variables, we also declare the variable's data type. The **data type** of a variable determines what kind of values it can store. When a program containing the variables is executed, the OS reserves memory locations to hold these variables.

The most commonly used variables in C++ are `bool`, `char`, `int`, `float`, and `double`. A `bool` (boolean) is a data type that returns true and false, although C++ converts `bool` variables to the values `1` and `0`. A value of `1` represents a true value while a value of `0` represents false. Any other number can be assigned to a `bool` variable. The following are valid statements in C++:

```
bool bool1;
bool1 = false;
cout<<bool1;              //prints 0
bool1 = 100;
cout<<bool1;              //prints 1
bool1 = -100;             //prints 1
```

Since most programs perform calculations, variables should be able to store numerical information. The data types `unsigned int`, `short int`, `int`, `unsigned long`, and `long` store numerical values. They only differ on the size a variable of their type occupies in the memory and/or the range of values it can hold. If the value assigned to an integer data type contains

a decimal portion, then the decimal part is cut. `float`, `double`, and `long double` accept numerical values too but they store the decimal parts of the values assigned to them.

A `char` is a data type that can store a single character.

An `enum` data type lets its users create constants with specified values.

Data Type	Size	Value Range
bool	8 bits*	true or false
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	1.2e-38 to 3.4e38
double	64 bits	1.7e-308 to 1.7e308
long double	80 bits	3.4e-4932 to 1.1e4932

Local Variables

Local variables are similar to variables found under global declarations but they can only be accessed and manipulated within the code block they were initialized in.

Statements

A **statement** is an instructional code that commands the computer to do a certain action upon its execution. A semicolon (`;`) signifies the end of any statement. The declaration of a variable in a function is also considered as a statement since the entire line of code ends with a semicolon.

* Before, C++ compilers had a bool type represented internally as a long int—so its size was 32 bits. But now the ANSI (American National Standards Institute) compilers usually provide an 8-bit bool.

Comments

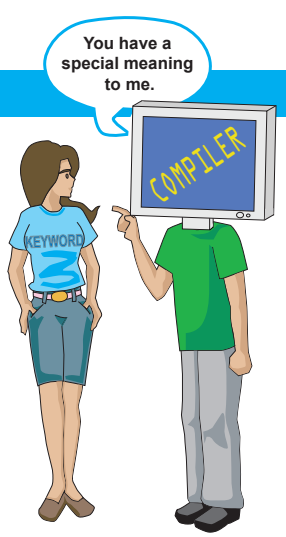
Comments are lines of code that are not executed during the program. They are used either to give a brief labeling and explanation to lines of codes or to temporarily disable lines of code during the development phase. There are two kinds of comments that C++ utilizes:

- **Single-line Comment** – a double slash (`//`) at the beginning of a line of code disables the entire statement and considers it merely as a comment
- **Multi-line Comment** – several lines of code can be disabled by placing the code block between a slash-asterisk (`/*`) and an asterisk-slash (`*/`)

Keywords

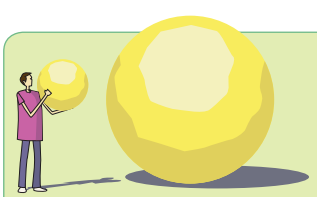
Keywords are reserved words in a programming language that have specific uses (e.g., executing a command). Examples are the statements `if`, `do-while`, and `switch`; and the functions `main()`, `getch()` which asks the user for an input before proceeding, and `clrscr()` which clears the screen and erases all the text outputs before the declaration of a statement.

Variables



As mentioned before, variables are defined by the user and are given initial values. A good rule to follow when naming variables is to make the names self-explanatory in the sense that the name given to a variable reflects what it does, to easily trace possible errors that may occur.

Variables are also called identifiers. Identifiers are user-defined word used to represent a value.



EXAMPLE

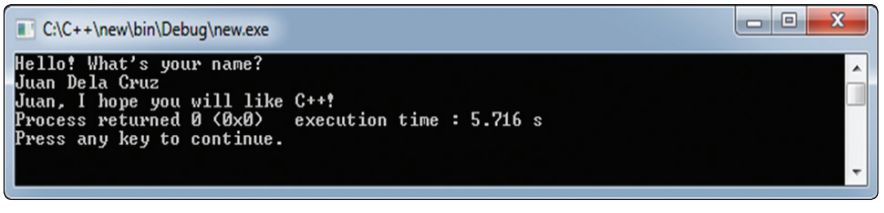
Most of the time, variable declarations consist of a single statement, like `int studentAge;`, where `int` is a data type and `studentAge` is the variable name.

The cout and cin Statements

Going back to our "Hello World!" source code from Lesson 2, let's say we modify that code to include another function and put together what we have learned so far. The code `cout << "Hello World" <<endl;` displays the sentence on your screen because the `cout` statement outputs what is within the statement. In contrast, the `cin` statement asks for an input from the user. Create a new C++ program with the following lines of code and observe what happens:

```
1   #include <iostream>
2   #include <string>
3   using namespace std;
4
5   int main()
6   {
7       string yourname;
8
9       cout << "Hello! What's your name?" << endl;
10      cin>> yourname;
11      cout << yourname << ", I hope you will like C++!";
12      return 0;
13  }
14
```

The statement in line 10 asks for user input then places whatever the input is on, the variable `yourname` which was declared earlier in line 7. Note that in C++, strings are not a built-in data type. Whenever we want to use and manipulate strings, we must add the necessary preprocessor directives seen in lines 2 and 3. To see if it works, the next line displays the value of `yourname` after the user input. Remember to separate the variable from the string literal (the statement within the quotation marks) using the insertion operator. We can see the results of the code below.



Did you notice that the program only displayed "Juan" instead of "Juan Dela Cruz"? That's because `cin` will only read the string until the next separator which is, in this case, a space character. We can solve that problem by using the `getline` function.

```
getline(cin, yourname);
```

The `getline` function needs the following parameters to work: the stream where the extraction is performed and the variable to hold the extracted data. Try editing line 10 and see what the output will be like.

Constants

Like variables, **constants** also act like containers which are user-defined and are given initial values. However, the values of constants cannot change during the program. There are two ways to declare constants:

1. First is by declaring a statement under the preprocessor directives and making use of the command `#define`. Doing so will yield a constant declaration like this:

```
#define standardAge 18
```

   where `standardAge` is the constant name and `18` is the declared value.

2. The other way is by declaring a statement and making use of the `const` command. Doing so will yield a constant declaration like this:

```
const int standardAge = 18;
```

   where `int` is the data type for the constant, `standardAge` is the constant name, and `18` is the declared value. Note that the whole statement ends with a semicolon.

Operators

Operators are symbols that execute a corresponding action during the program. In C++, there are several categories of operators, each having a unique feature. Below are the commonly known categories of operators.

Assignment Operator
Mathematical Operators
Increment and Decrement Operators
Relational Operators
Logical Operators

Assignment Operator (=)

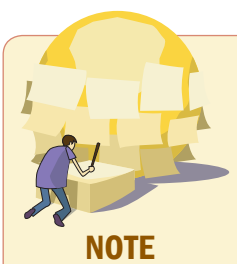
As the name implies, this operator assigns a value to an operand. The operand that will take the value is found on the left side of the assignment operator while the assigned values are found on the right side.

For example, `a=b+c;` signifies that you are assigning a with the sum of b and c.

Mathematical Operators (+, -, *, /, %)

Mathematical operators have the same function as to what they symbolize in arithmetic: addition (+), subtraction (-), multiplication (*), and division (/). The only operator that you might not be familiar with is the modulo operator (%). This is similar to the division operator but instead of giving the quotient, it will yield the remainder.

For example, if b = 30, c = 9, and d = 6, `a=b/d;` will yield the answer 5 for `a`. On the other hand, `a=b%d;` will yield the answer 0 for `a`, since the remainder, when 30 is divided by 6, is 0. Thus, `a=b%c;` will yield the answer of `3`.



NOTE

An `int` divided by an int will result to an `int` (omitting the decimal part). Consider the code below:

```
int j  = 10, k = 3;
cout<<j/k;                  //prints out 3
```

Increment (++) *and Decrement Operators* (--)

These operators increase and decrease the original values assigned to them by 1. Simply put, `a++;` will yield an equivalent result to `a=a+1;`. Similarly, `a--;` is the same as `a=a-1;`

Relational Operators (==, !=, >, <, >=, <=)

These operators perform a comparison test between variables and constants—whether they are equal (==, two equal signs), not equal (!=, an exclamation point and an equal sign), greater than (>), less than (<), greater than or equal (>=, a greater than sign with an equal sign), and less than or equal to the other (<=, a less than sign with an equal sign). The resulting value would either be true (signified by 1) or false (signified by 0).

For example, if a = 30, b = 30, c = 9, and d = 6, `a == b;` will yield 1, since `a` is indeed equal to `b`. On the other hand, `a!=b` will yield 0 because the operator wants to test whether `a` is not equal to `b`. Since it has been established earlier that `a` is indeed equal to `b`, this equation will yield 0 instead.

Logical Operators (&&, ||, !)

These operators further expand the relational operators since they add more control to it. For example, the equation `if (a==b)` will immediately test whether or not `a` is equal to `b`.

But what if you also want to test if `a` is also equal to a variable `c`? Your new equation would now be: `if (a==b && a==c)` so that the program tests if `a` is indeed equal to `b`, AND test if `a` is also equal to `c`. If it is, then the expression evaluates to true (returning a value of 1).

On the other hand, if the new equation is `if (a==b || a==c)`, the program will instead test if `a` is equal to `b`, ***or*** test if `a` is equal to `c`. If either of the two tests returns true then the whole expression evaluates to true.

The logical operator **`&&`** will test if all the indicated statements are true, while the logical operator **`||`** will test if either one of the indicated statements is true.

The remaining logical operator **`!`** reverses whatever output the other logical operators return. For example, the statement `!(a==b);` will have a reversed output, meaning if the value of `a` and `b` are equal, it will return false instead of true, and vice versa.

Expressions

Expressions refer to statements that return a value after being invoked or executed. The previous examples of mathematical equations (`a=b+c;`) and logical statements (`if (a==b)`) are all expressions since they return a certain value after execution. However, certain rules are followed by expressions when being evaluated. Below is a table indicating precedence in C++ of the mentioned operators:

C ++ Operator Precedence

()
! ++ --
* / %
+ -
< <= >= >
== !=
&&
||

When the computer encounters a statement like `a=b*(c+d/e);`, the expression inside the parentheses would be evaluated first. In `c+d/e`, the division equation would be evaluated before the addition equation. Afterwards, the entire evaluated expression would be multiplied to b. And in the end, the expression would be assigned to the variable a.

SUMMARY

This lesson featured the ins and outs of using C++, like the main components that make up a C++ program, expression, and functions that are inherent or user-defined, and other components that are important in order for any C++ program to run properly. These components are generally the same in any programming language since most applications follow the same rules on operator precedence, expression evaluation, and declaration of variables or functions.

WORD BANK

char – variable that holds a single character (one byte), which may be a number, a letter, or a symbol

Constant – fixed value used in certain mathematical or programming contexts

Expression – statement that is composed of at least one operator and one operand; it evaluates to a particular value based on the result of the operation

Floating-point Literal or **float** – variable that stores real numbers or numbers with fractional parts

Global Declarations – contains the user-defined variables and constants

int – variable that holds positive or negative whole numbers

Keyword – reserved words in a programming language that have specific uses

Operand – constant, variable, or expression which an operator will act upon

Operator – action that is applied on the operand/s, resulting in some value; symbol that represents actions or computations to be performed

Operator Precedence Rule – hierarchy that determines the sequence at which the operators in a given compound expression (two or more operators) are to be performed

Preprocessor Directives – allow the programmer to use different functions effectively

Variable – symbol used in a program that stands for a value or data stored in a computer's memory

PERFORMANCE TASK

Create a group composed of five members to do group research on C++ coding conventions and tips on how to write clean and readable code. You may do an Internet search for "C++ coding conventions" and list down several tips regarding code indention, variable naming conventions, the use of braces, and helpful uses of comments. Use the space below for your inputs. Discuss your results in class for additional insight.

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

A. On the blanks provided, identify the C++ related concepts described below.

_____ 1. refers to any valid combination of symbols that represent values and operations
_____ 2. topmost part of a C++ program where all the functions are imported
_____ 3. the part of a C++ program that contains user-defined variables and constants
_____ 4. values that remain fixed throughout the program
_____ 5. symbols used in a program that stand for values or data stored in a computer's memory
_____ 6. lines of code grouped together that perform several tasks in a program
_____ 7. determines the kind of values a variable can store
_____ 8. variables that can only be accessed and manipulated within the code block they were initialized in
_____ 9. an instructional code that commands the computer to do a certain action upon its execution
_____ 10. lines of code which are not executed
_____ 11. reserved words that have specific uses
_____ 12. symbols that perform a corresponding action during a program
_____ 13. a type of comment that only spans more than one line
_____ 14. hierarchy that determines the sequence at which operators in a compound expression are to be performed
_____ 15. keyword used to display values on the screen

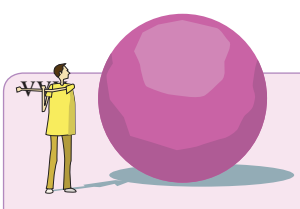
B. Identify the different C++ data types and indicate the number of bits they take up in the computer's memory and range of values they can hold.

Data type	Size	Value Range

CUT THIS PAGE

C. Enumerate the different operations and operator symbols that fall under each of the operator categories.

Category	Operation	Symbol
Assignment		
Mathematical		
Increment and Decrement		
Relational		
Logical		



SKILLS WARM-UP

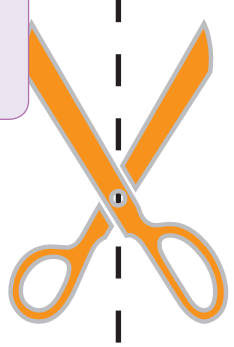
Put a check mark beside the valid variable names and a cross mark beside the invalid ones.

_____ 1. MyAge
_____ 2. Sdf12
_____ 3. 100Total
_____ 4. Number++
_____ 5. _NumberOfErrors
_____ 6. total1
_____ 7. first name
_____ 8. main
_____ 9. _CocoKing
_____ 10. 246810

SKILLS WORKOUT

Evaluate the following expressions:

_____ 1. 100 * 20 – 50 / 5
_____ 2. (a > b) && (100 = 10*10)
_____ 3. !((1000 + 10) >= 1000)
_____ 4. 25 % 2 == 1
_____ 5. 25 / 2 == 12.5



PERFORMANCE TASK

Divide the class into groups of at most five members each. Each group is to come up with ten mathematical or scientific formulas. You may consult your notes from other math and science subjects and recall your favorite formulas. Use what you've learned in this chapter to write down below the C++ expressions equivalent to these formulas.

Formula	C++ Expression
Ex. Area of a triangle	(float)(base * height)/2;

LESSON 5

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.

—from *The Road Not Taken*, Robert Frost

Program Flow of Control

We are always given choices. Sometimes, the choice may be as simple as what to eat for lunch, or whether to ride the bus or the jeep to go to school. Sometimes it may be as enormous as whether or not to cheat during an exam, or to spill a secret you promised to keep. These choices are the result of a mixture of careful consideration and gut reaction. We are given the freedom to choose our own paths in life. But once we have decided on where we're going, we can't look back.

Similarly, with programming, we are asked to make certain decisions as to what action the program will perform. Previously, we were given an understanding of the different types of expressions that computer programs usually handle. Indeed, a lot of the actions that most programs perform have to do with the evaluation of such expressions. But what is even more valuable would be the capability to choose which among several alternative steps to take based on the result of an expression.

In C++, decisions are made using statements like if, if-else, nested if, and switch. In this lesson, we will examine how these conditional statements are applied to simple programming problems. We will use the boolean expressions you learned from the previous lesson. Boolean expressions specify the conditions that handle a program's flow of control.

In programming, **flow of control** refers to the order or sequence in which computational steps occur during program execution. When we speak of **statement-level flow of control**, we mean the sequence in which program statements are executed. A similar order is also observable at a higher level—the flow of control that occurs between program modules or subprograms. Going down to a lower level, we have already seen in the previous lesson how operator precedence and associativity rules dictate the order in which multiple operations are evaluated in an expression.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Analyze control structures that affect program flow of control.
2. Recognize problems that can be solved using conditional statements.
3. Identify common selection control structures that can be used to control program flow.

LESSON OUTLINE

1. One-Way Selection Statement
2. Compound Statements and Blocks
3. Two-Way Selection Statement
4. Multi-Way Selection Statement

A programming language requires a minimum of two mechanisms in order to ensure that it has sufficient flexibility and computing power when dealing with programming problems:

1. Some means of choosing among several alternative paths of execution; and
2. Some means of specifying the repeated execution of a certain group of statements or section of a program.

Statements that provide these types of capabilities are called control statements. A **control statement** is a program statement whose purpose is to control or determine how subsequent statements are to be executed. A **control structure**, on the other hand, refers to the control statement along with the collection of program statements that it controls. This group of statements could be among the alternative control flow paths the program may go into, or it could be the portion of the program that is to be executed repeatedly for a definite number of times.

One-Way Selection Statement

SYNTAX

```
if (condition)
{
statement_1;
statement_2;
  :
statement_n;
}
```

Control Structure

In the above, the control statement is the `if` statement, while the control structure is the entire code. This also shows an example of a control structure that handles conditions, which will be taken up in detail later.

The simplest among all the conditional statements is the **`if`** statement. More often than not, we face the consequences of our decisions. For example, if we don't eat then we get hungry. The condition here is "not eating." When this condition is true, the consequence is "we get hungry." It is the same with the `if` statement. If the condition in the `if` statement is `true`, the succeeding statement will be executed. With the `if` statement, you can also execute several statements, just enclose them within open and close braces (**`{}`**).

NOTE

It is a good programming habit to enclose the statement/s that the `if` statement will execute between two braces (open and close) even if there is only one statement. Also, it is good to indent the statements within `if` statements to facilitate readability. A code that is readable is easier to analyze and debug.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6       int x;
7
8       cout << "Enter an Integer." << endl;
9       cin >> x;
10
11      if(x % 2 == 0)
12      {
13          cout << x << " is an EVEN number.";
14      }
15
16      if(x % 2 == 1)
17      {
18          cout << x << " is an ODD number.";
19      }
20
21      return 0;
22  }
23
```

The code above shows us how the `if` statement works. It works by asking the user to enter an integer. The program will then store that integer in a variable. To determine if the input is an odd or even number, we need to divide the number by 2 and evaluate the remainder. If the expression has no remainder, the number being evaluated is even. If there is a remainder, then the number being evaluated is odd. To do this, we will use the modulo operator. The code above shows two `if` statements for two conditions. The `if` statement from lines 11 to 14 tests whether or not the number is even while the other statement tests if the number is odd.

In this case, the test for the even number goes first, and if the condition is met, the output will say that it is an even number. However, given the structure of our program, the other `if` statement from lines 16 to 19 will also be evaluated even if the number was determined to be an even number. While the program gives us the correct result, it is not efficient since both statements have to be evaluated even if one has already been met.

In programming, there are several ways to come up with the same result. Read on to learn another way of programming that will get us the same results as the program above.

Compound Statements and Blocks

A **compound statement** is a collection of individual program statements that have been grouped together and enclosed with braces. This mechanism is used to group a sequence of program statements and get them to be treated syntactically as only one statement, for use when the structure of the programming language requires it. As a result, it becomes possible to use a compound statement where a single program statement can be used.

SYNTAX

```
if (i == 10)
{
      statement_1;
      statement_2;
      :
      statement_n;
};
```

Compound Statement

In some programming languages, compound statements are also called blocks. More specifically, a **block** is a compound statement that contains data declarations in the beginning. Variables that are declared within a block can only be used and referenced within that block, and defines the local scope that the variables can be used.

Two-Way Selection Statement

Two factors have posed considerable influence in determining how well accepted and widely used a programming language can become:

1. The ease with which programs in it can be written, or its **writeability**; and
2. The ease with which programs in it can be understood, modified, and maintained, or its **readability**.

In general, having a greater number of control statements enhances the writeability of a programming language. In this regard, all programming languages that have become widely used offer more than what is minimally required.

At times, we are given conditions that may not be true. When you need to execute specific statements depending on whether the condition is true or false, use the **`if-else`** statement. The following program is an implementation of a two-way selection statement for determining whether the user input is an even or odd number.

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

int main()
{
    int x;

    cout << "Enter an Integer." << endl;
    cin >> x;

    if(x % 2 == 0)
    {
        cout << x << " is an EVEN number.";
    }

    if(x % 2 == 1)
    {
        cout << x << " is an ODD number.";
    }

    return 0;
}
```

The code will have the same results as the one in page 53. With the previous code, there were two `if` statement blocks. With this code, there is only one `if-else` statement block. That means if the expression evaluates to false, the statements within the scope of the `else` part will be executed. If the expression is true, the code no longer needs to go through the `else` statement but on to the command after the `if-else` block.

The `if-else` statement is good if there are only two possible conditions for the statement. This works for queries such as True or False. But what if we expect other possible **results?**

Multi-Way Selection Statement

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

int main()
{
    int grade;

    cout << "Enter grade:";
    cin >> grade;
    cout << "Converted grade is: ";

    if(grade < 83)
    {
        cout << "C";
    }
    else
    {
        if(grade < 92)
        {
            cout << "B";
        }
        else
        {
            cout << "A";
        }
    }

    return 0;s
}
```

Many times, we have several paths to choose from when we decide. When you want to evaluate several conditions, use the **nested `if`** structure.

Explaining the Nested if *Application*

In this program, we declared a variable that stores user input, then we displayed the equivalent letter score of the numeric grade. Line 12 tests if `grade` is less than 83. If it is, the converted grade that will be printed is "`C`" because a grade less than 83 will make the statement above true. In addition to this, C++ will not execute the code from the first `else` clause up to the close brace before the `return` command.

If the value of `grade` is 83 and above, the statement within line 12 will be ignored. Within the scope of the first `else` statement is another `if-else` statement. The `if` statement tests if the value of `grade` is less than 92. If it is, "`B`" is printed. If the value of `grade` is at least 92, then C++ ignores the statements within the second `if` statement and proceeds to printing "`A`" as the converted grade.

Try changing the values for the variable `grade` to test the program.

We can keep on nesting `if-else` statements within the `else` statement block but that would make our code too confusing. The **`switch`** statement can be an alternative to the nested `if` statement because in the `switch` statement, you also have several options. However, the values to be tested in the `switch` statement are limited to those of type `int` and `char`. The expression in each case is evaluated until a match is found. If no match is found, an optional `default` portion is executed. Take note that at the end of each `case` portion is a **`break`** command. Without it, C++ goes on executing the statements in the next `case` clauses.

SYNTAX

```
switch(<expression>)
{
    case <constant1>:
          <statements>
          break;
    case <constant2>:
          <statements>
          break;
          :
          :
    default:
          <statements>
          break;
}
```

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

int main()
{
    int month;

    cout << "Enter the value of the month:";
    cin >> month;

    switch(month)
    {
        case 1:
            cout<<"January has 31 days.";
            break;
        case 2:
            cout<<"February has 28 days.";
            break;
        case 3:
            cout<<"March has 31 days.";
            break;
        case 4:
            cout<<"April has 30 days.";
            break;
        case 5:
            cout<<"May has 31 days.";
            break;
        case 6:
            cout<<"June has 30 days.";
            break;
        case 7:
            cout<<"July has 31 days.";
            break;
        case 8:
            cout<<"August has 31 days.";
            break;
        case 9:
            cout<<"September has 30 days.";
            break;
        case 10:
            cout<<"October has 31 days.";
            break;
        case 11:
            cout<<"November has 30 days.";
            break;
        case 12:
            cout<<"December has 31 days.";
            break;
        default:
            cout<<"Sorry that is not a valid month";
            break;
    }
  return 0;
}
```

Explaining the switch *Application*

The `switch` statement starts where the value of the variable `month` is being tested. This means that the value of `month` is checked against the `case` constants defined. In this case, the constants are the numerical equivalents of the months of a year.

There are 12 case constants (one for each month of the year) and one default case when the value of `month` is not within the specified range. Each `case` ends with a `break` command so that if the `switch` statement finds a matching value, all succeeding lines within the `switch` statement are ignored, making the program more efficient.

SUMMARY

Control statements are used to dictate the sequence of execution of instructions or expressions that are written in programs. This allows the computer to understand which instruction should be performed first. A control structure refers to the control statement and the group of statements that it encloses and controls. Individual program statements that are put together and considered or treated as a single statement are called compound statements.

To recap, here is a table of useful syntaxes and examples:

SYNTAX	EXAMPLE

`if (<boolean condition>)`
`{`
`  <statement>`
`}`
	`if(x!=0)`
	`{`
	`  x=(int)x/2;`
	`}`

`if (<boolean condition>)`
`{`
`  <statement/s>`
`}`
`else`
`{`
`  <statement/s>`
`}`
	`if (A%2==0)`
	`{`
	`  cout<<A<<" is EVEN";`
	`}`
	`else`
	`{`
	`  cout<<A+" is ODD";`
	`}`

```
if (<boolean condition>)
{
   <statement/s>
}
else
{
   if (<boolean condition>)
   {
      <statement/s>
   }
   else
   {
      <statement/s>
   }
}
```

```
if (number1>10)
{
   cout<<"greater than 10";
}

else
{
   if (number1<10)
   {
   cout<<"less than 10";
   }
   else
   {
   cout<<"equals 10";
   }
}
```

```
switch(<expression>)
{
      case <constant1>:
           <statements>

      break;
      case <constant2>:

           <statements>

           break;

      :

      :

      :
      default:

           <statements>

           break;
}
```

```
switch(Number)
{
   case 1:
          cout<<"One";
          break;
   case 2:
          cout<<"Two";
          break;
   case 3:
          cout<<"Three";
          break;
   default:
          cout<<"Sorry!";
          break;
}
```

WORD BANK

Block – a compound statement that includes variable declarations at the beginning

Compound Statement – a collection of individual program statements that have been grouped together and enclosed with delimiting symbols

Control Statement – a program statement whose purpose is to control or determine how subsequent statements are to be executed

Control Structure – the control statement along with the collection of program statements that it controls

Flow of Control – the order of execution of program statements

PERFORMANCE TASK

Every day we make a lot of decisions, simple decisions—from opting to wake up when the alarm clock rings to choosing which clothes to wear—or complex decisions such as picking the course to take up in college.

Divide your class into groups of five members and then discuss and compare your morning routines from waking up until the moment you arrive at your seat in the classroom. Identify the different decisions you make and create a simple flow chart that reflects your actions. Use the space below for your inputs.

C++ PROGRAMMING FUNDAMENTALS
Program Flow of Control

L5

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

1. What is the statement-level flow of control?

   _____

   _____

2. What are the differences between if, if-else, nested if, and switch statements?

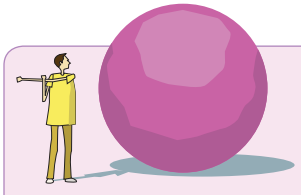
   _____

   _____

3. What does the break command do?

   _____

   _____



SKILLS WARM-UP

On the blanks provided, identify the C++ related concepts described below.

_____ 1. refers to the sequence in which computational steps occur during program execution

_____ 2. refers to the sequence in which program statements are executed

_____ 3. a statement that determines how subsequent statements are to be executed

_____ 4. a collection of program statements that have been grouped together and enclosed with braces

_____ 5. a collection of statements that include data declarations at the beginning

_____ 6. refers to the control statement and the program statements it controls

_____ 7. the ease with which programs can be written

_____ 8. the ease with which programs can be understood, modified, and maintained

CUT THIS PAGE

Write the screen output produced by the following code segments.

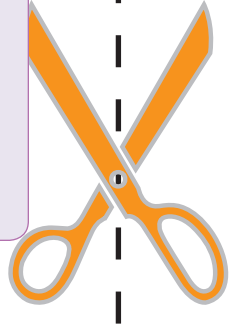
SKILLS WORKOUT

1.	`int x = 3;`
	`if (x > 2)`
	`{`
	`  cout << "A" << endl;`
	`}`
	`if (x == 3)`
	`{`
	`  cout << "B" << endl;`
	`}`
	`cout << "C";`
		Output:

2.	`int x = 100;`
	`if (x > 75)`
	`{`
	`  cout << "Pass";`
	`}`
	`else`
	`{`
	`  cout << "Fail";`
	`}`
		Output:

3.	`int x = 3;`
	`if (x < 3)`
	`{`
	`  if (x>1)`
	`  {`
	`  cout << "FLYR";`
	`  }`
	`  else`
	`  {`
	`  cout << "PERT";`
	`  }`
	`}`
	`else`
	`{`
	`  if (x<2)`
	`  {`
	`  cout << "GEW";`
	`  }`
	`  else`
	`  {`
	`  cout << "TWOR";`
	`  }`
	`  cout << "TWAX"`
	`}`
		Output:

4.	`int direction = 3;`
	
	`switch (direction)`
	`{`
	`  case 0: cout << "Left"; break;`
	`  case 1: cout << "Up"; break;`
	`  case 2: cout << "Right"; break;`
	`  default: cout << "Down"; break;`
	`}`
		Output:

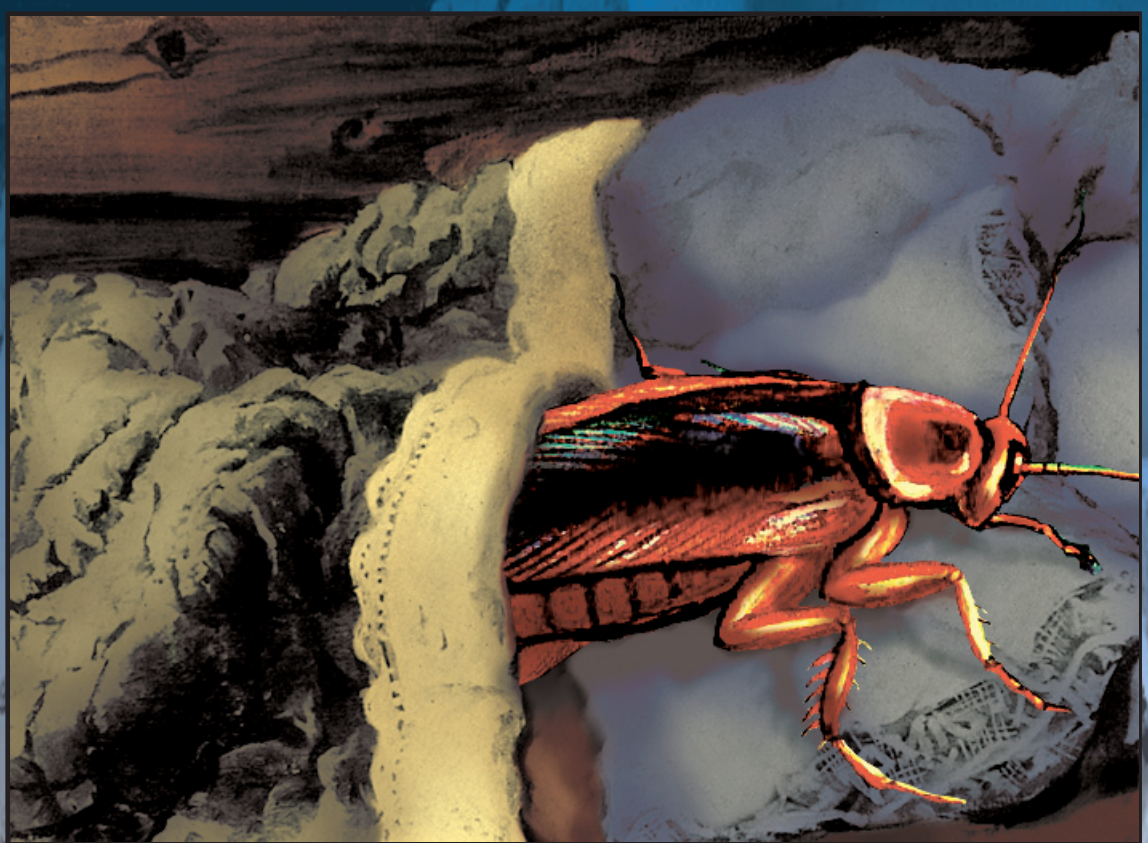


PERFORMANCE TASK

For this exercise, each group, composed of five students, will use C++ expressions to model the formulas used to compute the monthly water and electric bills.

The monthly electric and water bills are computed by using complex formulas. For example, the actual price per cubic meter of water is determined by the total consumption. For example, if your total water consumption is below 100 cubic meters, the price is 10 pesos/cubic meter, but if your consumption is within 100-200 cubic meters, it costs 15 pesos. The price varies in each bracket. Find out how your monthly water and electric bills are computed and model the computations using C++ expressions.

LESSON 6



"How about if I sleep a little bit longer and forget all this nonsense," he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before.

—from *Metamorphosis*, Franz Kafka

Handling Repetitions

In Kafka's *Metamorphosis*, salesman Gregor Samsa led a boring, repetitive life—until he woke up one rainy morning to find himself transformed into a giant cockroach. It is human nature to get bored and to lose interest in certain activities if these have been done repeatedly for a very long time. Of course, just because we're bored doesn't mean that we will immediately transform into vermin. However, doing anything repetitively over a long period of time does lead to fatigue and inattention on the part of the person.

Monotony and boredom over highly repetitive tasks usually cause either or both of the following: (a) the person loses concentration and begins to commit careless mistakes, or (b) he quits and walks away from the job. One of the greatest strengths of a computer is its ability to perform repetitive tasks continuously without getting bored or tired. With computers, we only have to worry about our own fatigue.

But just exactly how are such repetitive tasks handled in computer programs? Using only sequential and selection control structures, one way of accomplishing the repeated execution of a particular operation is by typing the same code in the program over and over again. This is an extremely inefficient or even unworkable way of doing it.

Fortunately, there are **repetition control structures** specifically meant to handle the repeated execution of portions of a program. These control structures are called **loops**. With loops, a test condition for repetition is already built-in, and a separate `goto` statement is no longer required. Each pass through the loop is called an **iteration**.

Counter-Controlled Loops

A **counter-controlled loop** is a control structure that allows us to specify the repeated execution of a group of

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Analyze problems that can be solved using loops.
2. Construct programs with loops.
3. Test and run programs involving loops.
4. Recognize the differences between various repetition control structures and how they work.
5. Recognize the conditions or situations where each particular repetition control structure is appropriate to use.

LESSON OUTLINE

1. Counter-Controlled Loops
2. Condition-Controlled Loops

program statements a definite number of times. A variable that serves as an **index** or **counter** is used to indicate how many times an enclosed block of statements is to be executed repeatedly.

The for *Loop*

The most common type of counter-controlled loop is the **`for`** loop. The `for` loop is used when the number of repetitions or iterations is already known before entering the loop.

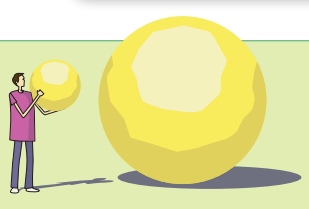
SYNTAX

The `for` loop follows this format:

```
for (<initialization>; <condition>; <increment/operations>)
{
   <statement/s>
}
```

From the syntax, you can see that the `for` loop has three main parts separated by a semicolon (`;`):

1. `initialization` – this is where you assign values to variables that will be used in your loop.
2. `condition` – made up of conditional statements that (if true) determine whether or not the statements within the `for` loop will be executed.
3. `increment/operations` – can be a place where operations are done every time the loop iterates.



EXAMPLE

`for` loop that prints the numbers 1–10:

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int i;
8       for (i=1; i<=10; i++)
9       {
10          cout<<i<<endl;
11      }
12      return 0;
13  }
```

Let us deconstruct the example given in the previous page:

In line 7, we declare the variable `i` as an integer but we do not assign a value. However, once we begin the `for` loop statement in line 8, we assign `i` as the value of `1`.

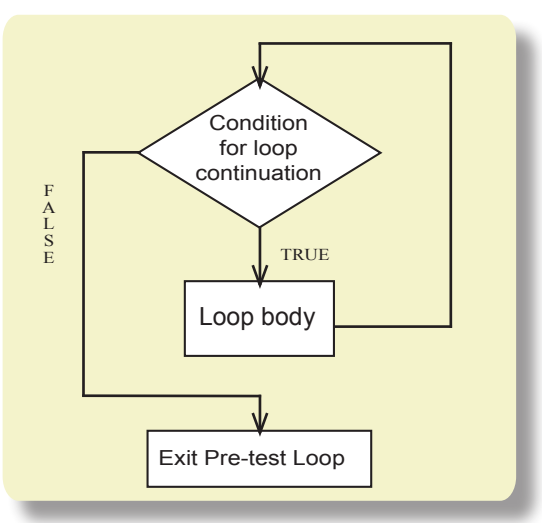
After assigning a value to `i`, we check if the value of `i` is within the condition. In this case, we check if `i` is less than or equal to 10 and if the program processes the code within the brackets, which is to print the current value of `i` on the screen. At the end of an iteration, the value of `i` is incremented by 1 which is written as `i++;`.

The expected output of the program is to be able to print the number 1 to 10 on your screen.

Condition-Controlled Loops

It is not always possible to know in advance how many times a particular set of operations should be repeated. In such cases, we need a loop that reacts to changing conditions. A **condition-controlled loop** is a loop that tests a condition to determine whether a block of enclosed statements should be executed repeatedly.

One type of condition-controlled loop evaluates an expression prior to entering the loop. This is called the **pre-test loop**. This condition determines whether program flow can proceed into the loop body. If the condition is satisfied, the statement block inside the loop structure is executed. Once the last statement is executed, control is transferred back to the beginning of the loop where the pre-test is evaluated again. As long as the expression remains true, the execution of the statement block inside the loop will be repeated.



Flowchart of a pre-test loop

If the condition is not satisfied, then the loop execution ends. The loop body is disregarded and control is transferred to the next executable statement after the loop structure. If at the very start, the result of the pre-test condition is already false, then the loop body won't even be executed.

The while *Loop*

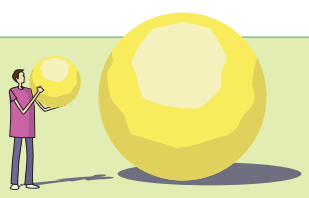
The **while** loop is the most common example of a loop structure that uses a pre-test.

SYNTAX

The while loop follows this format:

```
while (<expression>)      /* pre-test loop */
{
      statement_list;
}
statement_list2;
```

The while loop is quite similar to a for loop; however, at the start of the loop, it merely checks the condition. Just like our earlier example, let's study how we can print the numbers 1 to 10 on your screen, this time, using the while loop.



EXAMPLE

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int i=1;
8       while(i<=10)
9       {
10          cout<<i<<endl;
11          i++;
12      }
13      return 0;
14  }
```

If we compare the code using for loop and while loop, we can see certain similarities. Line 7 declares the variable and also assigns a value of 1 to i. Line 8 shows us that there is just one condition. And finally, in line 11, the value i is incremented by 1 right after the value of the variable is displayed on the screen.

What was done in just one line using the for loop was done with 3 lines in the while loop. Perhaps another example can better illustrate the while loop.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int vTotal=0; /*Accumulator for students' grades*/
8       int vGrade_Counter = 1; /*Counter for no. of students processed so far*/
9       int vNum_Students;
10      float vGrade, vAverage;
11
12      cout << "Enter the number of students in the class: ";
13      cin >> vNum_Students;
14
15      while(vGrade_Counter <= vNum_Students)
16      {
17          cout << "Enter a grade: ";
18          cin >> vGrade;
19          vTotal = vTotal + vGrade;
20          vGrade_Counter = vGrade_Counter + 1;
21      }
22      /* compute the class average */
23      vAverage = (float)vTotal / vNum_Students;
24      cout << "Class average is: " << vAverage;
25      return 0;
26  }
```

The example above computes the average grade of a class. It asks the user how many students there are in the class, and later, what their individual grades are. A **counter** keeps track of how many students have been processed so far. The `while` loop uses the counter as part of the pre-test condition that determines whether the loop body should be executed again or not.

Line 10 shows that the variables `vGrade` and `vAverage` are declared as `float` so that they will be able to store the decimal portions of the entered grade and average of the entered grades, respectively.

Line 23 shows that the code `(float)` was placed beside the variable `vTotal`. Since vTotal and `vNum_Students` were declared both as integers, when you divide one with the other, only the integer part of the quotient will be assigned to the float `vAverage`. So if the value of `vTotal` is 1001 and `vNum_Students` is 10, 100 will be assigned to `vAverage`, not 100.1. What we did in this line is casting. **Casting** is the process of converting a value from a specific data type into another.

When performing arithmetic operations, the result will be a type that occupies at least the same amount of memory that is occupied by the biggest operand in the statement. The variables `vAverage` and `vTotal` are both float datatypes while `vNum_Students` is an integer datatype. Since a float datatype is bigger than an integer datatype, the resulting operation between `vTotal` and `vNum_Students` will yield a float value.

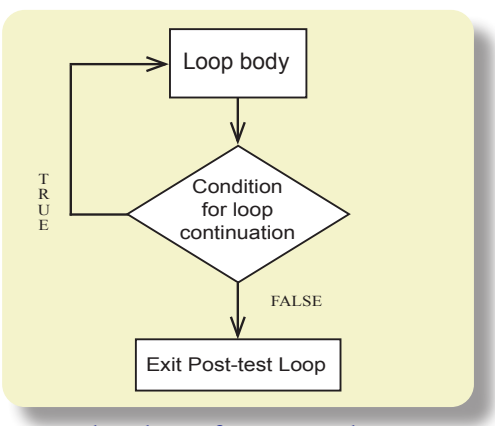
The do-while *Loop*

Another type of repetition control structure tests the condition at the end of the loop body. This is called the **post-test loop**. Since the expression is evaluated at the end of the loop structure, this assures us that the enclosed statement block inside the loop structure will be executed at least once. The **`do-while`** loop is the most common example of a loop structure that uses a post-test.

SYNTAX

The `do-while` loop follows this format:

```
do {
      statement_list1;
} while (<condition>);     /* post-test loop */
statement_list2;
```



Flowchart of a post-test loop

Just like the other examples before it, the code will output the numbers 1 to 10 on your screen. The key difference between the `for` loop and the `while` loop examples is that the condition is checked only at the end of the loop, as seen in line 13. Let's look at another example that can better illustrate the `do-while` loop.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int i = 1;
8       do
9       {
10          cout << i << endl;
11          i++;
12      }while (i <= 10);
13      return 0;
14  }
15
```

The code checks if the inputted password is the same as a the predefined password. The value of the correct password is defined at the start of the code in line 1 as a constant. In line 8 we create a variable to hold our input for the password. Once we enter the `do-while` loop, the program will prompt the user for a password. In fact, the program will repeatedly ask the user to input the value of the predefined password until the user gets it right.

```
1  #define cPasswd 25
2  #include <iostream>
3
4  using namespace std;
5
6  int main()
7  {
8      int vPasswd;
9
10     do
11     {
12         cout<<"Please enter password: ";
13         cin>>vPasswd;
14     }
15     while (vPasswd != cPasswd);
16
17     cout<<"Password is correct. You may proceed.";
18     return 0;
19 }
```

The loop will terminate and proceed to the next executable statement outside the `do-while` loop only after the user has provided the correct password.

SUMMARY

In order to take full advantage of the ability of computers to perform repetitive tasks without tiring out or wavering in accuracy and consistency, programming languages have specific control structures that allow the repeated execution of portions of a code. This eliminates the need to repeatedly type the same code just to achieve the repetition. A counter-controlled loop is used if the repetition is to be executed a definite number of times. A condition-controlled loop is used if the number of repetitions is not known, and a logical condition is needed to test whether the loop body should be executed again.

WORD BANK

Condition-Controlled Loop – repetition control structure; specifies repeated execution of a group of statements based on an event or condition

Counter-Controlled Loop – repetition control structure; specifies repeated execution of a group of statements a number of times

Index Variable – counter used to keep track of iterations in a counter-controlled loop

Iteration – refers to each pass within a repetition control structure

Pre-Test – test performed at the start of a condition-controlled loop

Post-Test – a test performed at the end of a condition-controlled loop

Repetition Control Structure – a **loop**; control structure used to enable the repeated execution of a group of statements in a program

PERFORMANCE TASK

In Lesson 5, the **break** keyword and its functions in switch statements was introduced. The **break** keyword also has a specific function when used in loops.

Create a group composed of 3 to 5 members and research on the functions of the **break** keyword when used in loops, as well as on the **continue** keyword. Take note of the syntax and specific purposes of these keywords. Discuss among your groupmates some possible scenarios where these keywords can be used. Present the results of your discussion in class for additional insights.

PERFORMANCE TASK

Computers are great at performing repetitive tasks. Think of some real world processes that involve repetition. Think of how these tasks can be performed better with the aid of computers. Create an equivalent flowchart or a pseudocode for these processes and, if possible, implement these processes using C++ code.

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

1. Why is it impractical to just retype the portion of the program that needs to be executed repeatedly, rather than use repetition control structures?

   _____

   _____

2. What is the difference between a counter-controlled loop and a condition-controlled loop? When is it appropriate to use one instead of the other?

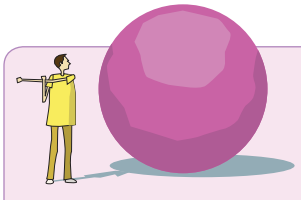
   _____

   _____

3. What is the difference between a `while` loop and a `do-while` loop? When is it appropriate to use one instead of the other?

   _____

   _____



SKILLS WARM-UP

On the blanks below, identify the concepts related to C++ described by each item.

_____ 1. control structures used to facilitate the repeated execution of a group of statements within a program; also known as repetition control structures

_____ 2. refers to each pass within a repetition control structure

_____ 3. a type of loop that tests a condition to determine whether the block of enclosed statements should be executed repeatedly

_____ 4. a type of loop that specifies the repeated execution of a group of statements a definite number of times.

_____ 5. a control variable or counter used to keep track of the number of iterations

CUT THIS PAGE

_____ 6. a type of condition-controlled loop that evaluates the condition expression prior to the execution of the loop body

_____ 7. a type of condition-controlled loop that evaluates the condition expression after the execution of the loop body

_____ 8. the part of a **for** loop where values are assigned to the variables that will be used in the loop

_____ 9. the part of a **for** loop that is made up of conditional statements that determine whether or not the statements within the **for** loop will be executed

_____ 10. the part of a **for** loop where values of loop control variables are updated

Answer the following questions briefly.

SKILLS WORKOUT

1. What is the output produced by the following code?

```
int counter = 7;
while (counter >= 0) {
   cout << counter << endl;
   counter = counter - 2;
}
```

2. What would be the output of the previous code if ">" was replaced by "<" ?

3. How would you rewrite the code in #1 using a `for` loop ?

4. What would probably be the most important difference between the `do-while` loop and the `while` loop ?


techFactorsInc

LESSON 7

What was to be done? To turn and fly was now too late; and besides, what chance was there of escaping ghost or goblin, if such it was, which could ride upon the wings of the wind? Summoning up, therefore, a show of courage, he demanded in stammering accents, "Who are you?" He received no reply. He repeated his demand in a still more agitated voice. Still there was no answer. Once more he cudgeled the sides of the inflexible Gunpowder, and, shutting his eyes, broke forth with involuntary fervor into a psalm tune. Just then the shadowy object of alarm put itself in motion, and with a scramble and a bound stood at once in the middle of the road. Though the night was dark and dismal, yet the form of the unknown might now in some degree be ascertained. He appeared to be a horseman of large dimensions, and mounted on a black horse of powerful frame. He made no offer of molestation or sociability, but kept aloof on one side of the road, jogging along on the blind side of old Gunpowder, who had now got over his fright and waywardness.

—from *The Legend of Sleepy Hollow*, Washington Irving

Arrays

When you are in a situation where you're not in control of things, like Ichabod Crane when he meets the Headless Horseman for the first time, it usually leads to panic and confusion. Even his horse Gunpowder is skittish and refuses to follow his directions. When things are out of control, they usually tend to go in all directions.

This is the same with computer programming. If there is no control over the data or the program elements, then there will be a tendency for things to get messed up. There are different kinds of data that computers can deal with, ranging from numerical to character, to date/time, and logical values. Thus, it is a useful programming practice to group several related, though distinct data items into one unit. The most common **structured data type** is the array.

What is an Array?

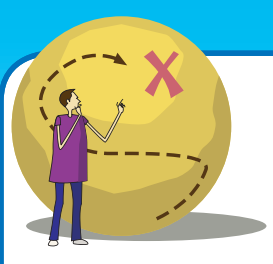
An **array** is a collection of variables of the same data type. Each value is referred to as an **array element**. Suppose we need to compute for the average of 5 quiz scores. These can be stored in variables quiz1, quiz2, …quiz5. Take a look at the program below, which shows a solution with only the use of variables:

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       float quiz1, quiz2, quiz3, quiz4, quiz5;
8       float average;
9
10      cout<<"Enter Quiz 1: ";
11      cin>>quiz1;
12
13      cout<<"Enter Quiz 2: ";
14      cin>>quiz2;
15
16      cout<<"Enter Quiz 3: ";
17      cin>>quiz3;
18
19      cout<<"Enter Quiz 4: ";
20      cin>>quiz4;
```

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Discover what arrays are and how to declare them.
2. Analyze strings and how to use character arrays to make them.
3. Apply arrays in programs.



LESSON OUTLINE

1. What is an Array?
2. Multidimensional Arrays
3. Char Arrays

```
21
22          cout<<"Enter Quiz 5: ";
23          cin>>quiz5;
24
25          average = (quiz1 + quiz2 + quiz3 + quiz4 + quiz5)/5;
26          cout<<"The average quiz score is " << average;
27          return 0;
28      }
```

```
Enter Quiz 1: 10
Enter Quiz 2: 20
Enter Quiz 3: 30
Enter Quiz 4: 40
Enter Quiz 5: 50
The average quiz score is 30
Process returned 0 (0x0)   execution time : 9.907 s
Press any key to continue.
```

The program above seems simple enough and will run just fine. But that's just for five quiz scores. What if we have 10 quiz scores? What about 100 quiz scores? Can you imagine creating 100 variables and going through the entire process of getting the quiz score values 100 times? Tedious, isn't it?

The problem with the code above is that it gets caught up with a lot of variable declaration and initialization. For such a simple problem of getting the average quiz scores, the use of looping or iteration constructs is difficult to apply when variables are used, making the program hard to write. What the program needs is to store quizzes in an array that would allow the use of iteration in computation and lessen the lines of code to write.

Defining an Array

Generally, an array is defined using this format:

```
datatype arrayName[size];
```

The elements of an array definition are as follows:

- `datatype` is a valid data type in C++
- `arrayName` is a user-defined name
- `size` is an integer which indicates the size of the array

Following our earlier example, if we are to declare an array to hold 5 quiz scores, it will be written as:

```
float quiz[5];
```

Array Elements

One can access the content of the array by referring to an offset from the array name. Array elements are counted from zero. Therefore, the first element of an array is `arrayName[0]`. In the previous example, `quiz[0]` is the first score, `quiz[1]` is the second, and so on.

Initializing Arrays

To initialize an array, values are assigned right after declaration with the use of the assignment operator separated by commas and enclosed by braces. For example,

```
float quiz[5] = {95, 75, 69, 90, 78};
```

declares `quiz` to be an array of float values. It assigns `quiz[0]` to be `95`, quiz`[1]` to be `75`, and so on.

When we are working with arrays, we may not know the size of the array or the values of the array elements. Just like in our quiz program, we do not know the values of the individual quiz scores.

If you did not specify the size of an array, an array that is big enough to hold the values you initialized would be created. Therefore, `int quiz[] = {95, 75, 69};` would only create three memory locations for the array `quiz`.

To assign values to an array, we can do it this way:

```
quiz[0] = 95;
quiz[1] = 75;
quiz[2] = 69;
```

Using Arrays

Now we are ready to use an array in a program. Take note of all the important things learned as far as declaring, initializing, and accessing elements of an array are concerned. Let's solve the problem mentioned at the start of this lesson. We have seen how long our program would look like if we did not use an array. What follows is the program when arrays are used instead:

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       float quiz[5];
8       float sum= 0, average;
9       int i;
10
11      for (i=1; i<=5; i++)
12      {
13          cout << "Enter quiz " << i << ": ";
14          cin >> quiz[i-1];
15          sum = sum + quiz[i-1];
16      }
17
18      average = sum/5;
19      cout << "Average quiz = " << average;
20
21      return 0;
22  }
```

Let us deconstruct the given code. Line 7 shows the declaration of the array quiz. We then use a `for` loop to insert elements to the array. Line 14 shows how we can add elements to an array; note that we specify what position the element is placed in. We add up the array elements with the code in line 15.

This clearly illustrates that writing the program is easier with the use of the array. Imagine how long the program will be if we need to process 100 test scores without using an array!

Multidimensional Arrays

So far, we have seen an array as a linear or sequential set of information, that is, a one-dimensional array. However, arrays can be represented using more than one dimension or what we call a **multidimensional array**. We can use multidimensional arrays when we are plotting points in a graph. A line graph has two dimensions: x- and y- axis.

Initializing Multidimensional Arrays

We can initialize multidimensional arrays by assigning the list of values to array elements in order, with the last array subscript changing, while each of the former holds steady. Therefore, if we have the array

```
int myArray[5][3];
```

the first three elements go into `myArray[0]`, the next three into `myArray[1]`, and so on. We initialize this array by writing

```
int myArray[5][3] = {5,10,15,3,6,9,2,4,6,10,20,30,4,8,12};
```

Using Multidimensional Arrays

Suppose we have the following data:

	Quiz 1 Scores	Quiz 2 Scores	Quiz 3 Scores
Student 1	95	88	95
Student 2	95	74	80
Student 3	70	70	90
Student 4	71	80	85
Student 5	90	79	95

They represent the three quizzes obtained by five students. To use a one-dimensional array would be inappropriate since we are now dealing with two dimensions: student number and quiz number. Each datum represents a certain quiz obtained by each student.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       float quizscores[5][3] = { {95,88,95}, {95,74,80}, {70,70,90}, {71,80,85}, {90,79,95}};
8
9       for (int i = 0; i<5; i++)
10          for (int j=0; j<3; j++)
11          {
12              cout << "quizscores[" << i << "][" << j << "]: ";
13              cout << quizscores[i][j]<< endl;
14          }
15      return 0;
16  }
17
```

By now most of the code must seem familiar. In this program, we use one multidimensional array to store the quiz scores per student and we use a single dimension array to hold the average test score result for each student.

Let's focus on the multidimensional array named `quizscores` in line 7. How do we know which value is which? If we run the program above, it should display the value of the quiz scores stored in the multidimensional array.

```
quizscores[0][0]: 95
quizscores[0][1]: 88
quizscores[0][2]: 95
quizscores[1][0]: 95
quizscores[1][1]: 74
quizscores[1][2]: 80
quizscores[2][0]: 70
quizscores[2][1]: 70
quizscores[2][2]: 90
quizscores[3][0]: 71
quizscores[3][1]: 80
quizscores[3][2]: 85
quizscores[4][0]: 90
quizscores[4][1]: 79
quizscores[4][2]: 95

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Let us modify the prior program to get the average quiz scores for each student by using a multidimensional array.

```
1   #include <iostream>
2
3   using namespace std;
4
5    int main()
6   {
7       float quizscores[5][3] = { {95,88,95}, {95,74,80}, {70,70,90}, {71,80,85}, {90,79,95}};
8       float average[5];
9       float sum = 0;
10
11      for (int i = 0; i<5; i++)
12      {
13           for (int j=0; j<3; j++)
14          {
15              sum = sum + quizscores[i][j];
16          }
17          average[i]=sum/3;
18          sum=0;
19      }
20
21     for (int i=0;i<5;i++)
22      {
23          cout << "The average quiz score of student " << i << " is " <<
24          average[i];
25          cout << endl;
26      }
27
28      return 0;
29  }
30
```

```
The average quiz score of student 0 is 92.6667
The average quiz score of student 1 is 83
The average quiz score of student 2 is 76.6667
The average quiz score of student 3 is 78.6667
The average quiz score of student 4 is 88

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

To get the average quiz score of each student, we had to add all the quiz scores of that student and store them in an array as seen in lines 11-19. To display the results on the console, we had to loop through the `average` array as seen in lines 21-26.

Char Arrays

In C++, a **string** is an array of characters ending with a null character. You can declare and initialize a string in the same way as you would any other array. For example:

```
char Welcome[] = { 'h', 'e', 'l', 'l', 'o', '\0'};
```

The last character `'\0'` is the null character. Although this character-by-character approach works, it is difficult to type and is prone to errors. C++ enables the programmer to use a shorthand form of the previous line of code:

```
char Welcome[] = "hello";
```

Based on this line, we can take note of the following things:

- Instead of single-quoted characters separated by commas and surrounded by braces, we can have a double-quoted string without commas and braces.
- We don't need to add a null character because the compiler automatically does it.

Here is a program that uses a string:

```
#include <iostream>

using namespace std;

int main()
{
    char StudName[20];
    int i=0;

    cout << "Enter your name: ";
    cin >> StudName;
    cout << "Your name is " << StudName << endl;

    while (StudName[i])
    {
        cout << "Array Position " << i << ": " << StudName[i] <<endl;
        i++;
    }
    return 0;
}
```

```
Enter your name: Joseph
Your name is Joseph
Array Position 0: J
Array Position 1: o
Array Position 2: s
Array Position 3: e
Array Position 4: p
Array Position 5: h

Process returned 0 (0x0)   execution time : 6.895 s
Press any key to continue.
```

SUMMARY

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an array element. You can declare an array by writing the type followed by the array name and the subscript. It is possible to have arrays of more than one dimension. Each dimension is represented by a subscript in the array. Therefore, a two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. Arrays can have any number of dimensions.

WORD BANK

Arrays – groups of data of the same type referenced under one name

Array Element – individual content of the array

String – an array of characters ending with a null character

Structured Data Type – a number of distinct data items, of different types, grouped together as one unit

Subscript – the number of elements in the array, surrounded by brackets

PERFORMANCE TASK

The examples used in this lesson use valid array subscripts to address individual array elements. This means that the values for the array subscripts are within 0 and the maximum number of elements is –1.

Form a group of five members and discuss what would happen if your subscripts with invalid values are used to address array elements. Specifically, discuss what would happen if an array subscript has a value that is:

a. not a whole number
b. negative
c. not a number
d. equal to or greater than the number of array elements.

Write a C++ code to experiment with these scenarios and observe the results when you compile and run the program. Create tips and reminders to help you avoid problems when working with arrays. Present the results of your discussions in class for additional insight.

SCORE

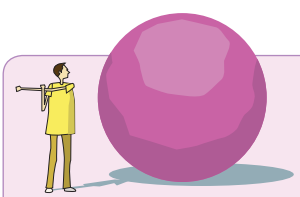
NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

1. Write a program that initializes all 10 elements of an integer array named Numbers to 0. Use the `for` loop.
2. Write a program that accepts a series of integers and displays if the integer is odd or even. The program stops accepting integers if the user inputs a negative number.
3. What are the first and the last elements of SomeArray[100]?
4. How many elements are in the array SomeArray[10][5][20]?



SKILLS WARM-UP

Write **True** on the blank provided if the statement is true, otherwise write **False**.

_____ 1. Arrays can be used to group similar data together.

_____ 2. An Array can be declared by writing its data type, followed by the array name, and the number of elements enclosed in square brackets.

_____ 3. Array elements can be given initial values when an array is declared.

_____ 4. Individual elements of an array can be addressed by appending the subscript enclosed in curly brackets at the end of the array name.

_____ 5. A variable with an **`int`** data type can be used as a subscript to address individual array elements.

_____ 6. The elements of arrays can only have **`int`** data types.

_____ 7. The individual elements of one array can have different data types.

_____ 8. Strings can be expressed as a multi-dimensional array of characters.

_____ 9. We can use nested loops to simplify the sequential access of multi-dimensional array elements.

_____ 10. We can initialize arrays even though we did not specify the array size.

CUT THIS PAGE

SKILLS WORKOUT

Answer the following questions.

1. Given the following array declaration:

```
int hours[20];
```

   a. What is the array name? _____
   b. What is the data type of each element of the array? _____
   c. What is the maximum number of elements the array can contain? _____
   d. How can you access the first element of the array? _____
   e. How can you access the last element of the array?_____

2. What is the output produced by the following code?

```
int n[] = {1, 2, 3, 4, 5};
       for (int i=4; i>0; i--)
             {
             cout << n[i] << " ";
             }
```

3. How would you declare a variable named n which is a 3 by 3 array of integers?

PERFORMANCE TASK

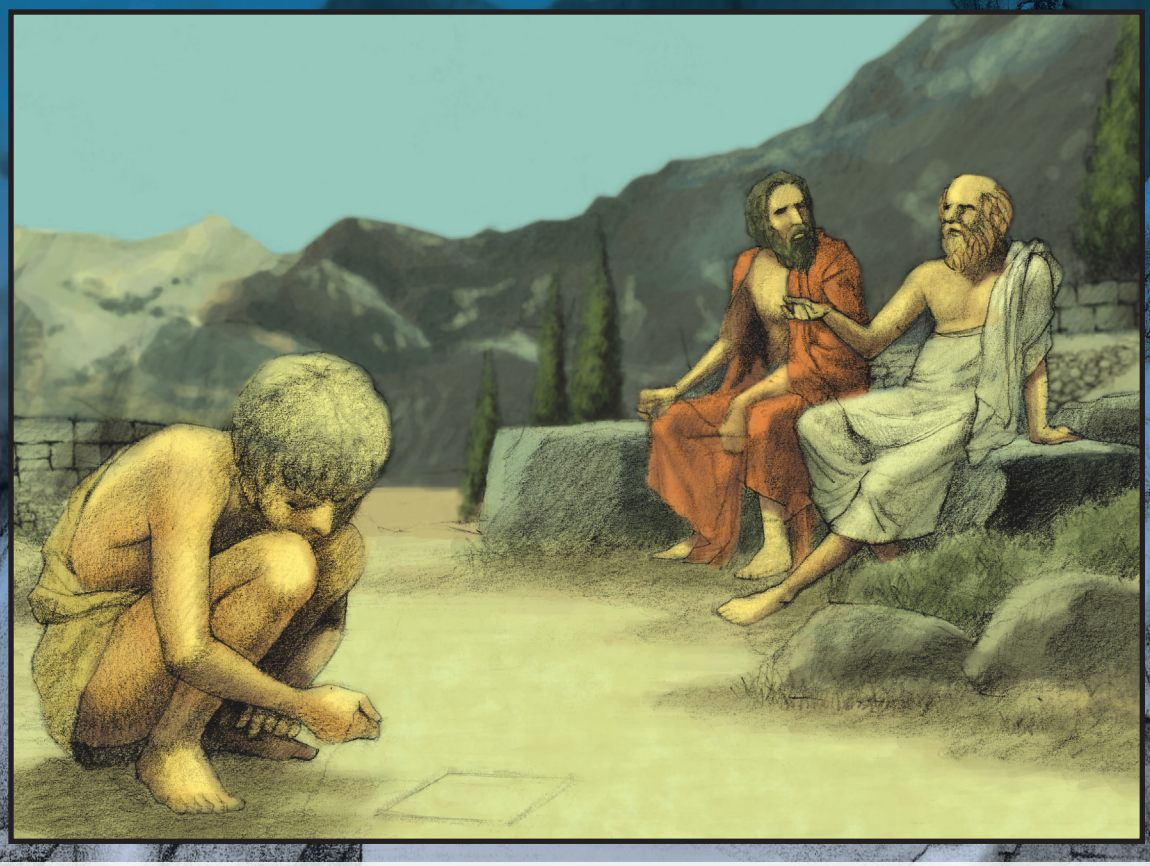
Several operations can be done on arrays, such as sorting the contents of one array or copying the contents of one array to another. Form a group of five members and discuss with your groupmates how you could write your own C++ functions which use loops and other previously-discussed C++ concepts to perform these two array operations. Create the two function methods below after talking these over with your groupmates:

a. A function named "sortIntArray" that takes only one parameter, an array of integers. This method takes an array of integers whose elements are initially unsorted. When the function is called, it sorts the elements in increasing order.

b. A function named "copyIntArray" that takes two parameters, the source array of integers, and the destination array of integers whose number of elements is the same as the source. When this function is called, it should copy every element of the source array into the destination array.

Present your results in class.

LESSON 8



Socrates: *Do you see, Meno, what advances he has made in his power of recollection? He did not know at first, and he does not know now, what is the side of a figure of eight feet: but then he thought that he knew, and answered confidently as if he knew, and had no difficulty; now he has a difficulty, and neither knows nor fancies that he knows.*

Meno: *True.*

Socrates: *Is he not better off in knowing his ignorance?*

Meno: *I think that he is.*

Socrates: *If we have made him doubt, and given him the "torpedo's shock," have we done him any harm?*

Meno: *I think not.*

Socrates: *We have certainly, as would seem, assisted him in some degree to the discovery of the truth; and now he will wish to remedy his ignorance, but then he would have been ready to tell all the world again and again that the double space should have a double side.*

—from *Meno*, Plato

Memory Management

How do you acquire knowledge, i.e., real knowledge that some call "wisdom"? Is it by observing your surroundings? Is it by listening to the teachings of your parents and professors? Or is it by searching within yourself? Maybe, just maybe, there are questions to which neither experience nor education can provide the answers. In the dialogue, Socrates was trying to get the boy give the solution to a mathematical problem, even if the boy did not have a formal education on the matter. Socrates was pointing out to Meno that there is such a thing as "knowledge by recollection."

10100100010010101011000001010100001000000101011010100010101110101000101010001010101000101010000010101000010101000010001010100010001001000001010000101000010000101001000101001000100010100010000011010100010010100100100010100100101010010000010001010000001010000010000000100000100000100001001000001000010100100010010001001000100001010010101010000101000010101010000011100001001000010000010001000010000001001000100010100

In the computer world, though, memory is simply a question of bytes. A **byte** is a series of 1s and 0s usually grouped as a sequence of eight bits. In Lesson 4, you have encountered a list of data type sizes measured in bits. It is good to know that one of the most powerful features of C++ is the ability to manipulate the memory locations occupied by the variables used within a program. In programming, you don't just declare additional variables to solve a problem—you take into consideration the hardware resources the program requires for it to run properly. This is especially true when creating large programs wherein hardware resources are limited.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Use pointers.

LESSON OUTLINE

1. Pointers

Pointers

C++ lets its users manipulate memory locations through the use of pointers. **Pointers** are variables used to store memory locations (other variables only store basic data types). Consider the following program.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7       int i=8;
8       int *ptr = NULL;
9       ptr = &i;
10
11      cout<<"Value of i:\t\t  "<<i;
12      cout<<endl<<"Memory Address of i:\t"<<&i;
13      cout<<endl<<"Value of ptr:\t\t"<<ptr;
14      cout<<endl<<"Value of *ptr:\t\t"<<*ptr;
15
16      *ptr = 12;
17
18      cout<<endl<<endl<<"*ptr was assigned the value of 12.";
19      cout<<"\n\nValue of i:\t\t  "<<i;
20      cout<<endl<<"Memory Address of i:\t"<<&i;
21      cout<<endl<<"Value of ptr:\t\t"<<ptr;
22      cout<<endl<<"Value of *ptr:\t\t"<<*ptr;
23
24      return 0;
25  }
```

Let us begin to understand the code above by studying the variable declarations.

Line 8 shows how a pointer is declared. The asterisk before the variable `ptr` indicates that an identifier will be a pointer variable—in this example, an `int` (the asterisk was used as a multiplication symbol during the past lessons, but this time, it is used to tell the compiler to reserve a memory space that can hold a memory address). We assigned ptr with a NULL value. Assigning a NULL value to a pointer is typically done to wipe out any garbage values—values that have no use at all—that the pointer may initially have and establish that it has not yet been assigned a valid memory address.

Line 9 shows the memory address in the computer where the variable is stored. The ampersand symbol `(&)` refers to the address of a variable. Take note that when you run this program, it is very likely that you will have a different memory address printed than what is shown here because different computers store variables in different addresses.

Finally, lines 11-14 print out the values of our variables and its memory addresses.

Now that we have gone through the first half of the code, let us see what happens when we change the value of the variable that a pointer points to. As discussed, `*ptr` is the content of the address stored in `ptr`. Thus, the executed code `*ptr = 12;` translates to "assign the value of `12` to the contents of the address stored in `ptr`." Since `ptr` points to `i`, it is the same as assigning `12` to the variable `i`. Lines 18-22 will print out the result after *ptr was assigned a value of two.

To understand this better, refer to the following image which shows the output of the program, and follow the sequence of our statements.

```
Value of i:        8
Memory Address of i:    0x28ff08
Value of ptr:           0x28ff08
Value of *ptr:          8

*ptr was assigned the value of 12.

Value of i:             12
Memory Address of i:    0x28ff08
Value of ptr:           0x28ff08
Value of *ptr:          12
Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6
7       int i=8,j=2;
8       int *ptr = NULL;
9       ptr = &i;
10
11      cout<<"Value of i:\t\t  "<<i;
12      cout<<endl<<"Memory Address of i:\t"<<&i;
13      cout<<"\nValue of j:\t\t"<<j;
14      cout<<endl<<"Memory Address of j:\t"<<&j;
15      cout<<endl<<"Value of ptr:\t\t"<<ptr;
16      cout<<endl<<"Value of *ptr:\t\t"<<*ptr;
17
18      ptr = &j;
19      *ptr = 12;
20
21      cout<<endl<<endl<<"&j was assigned to ptr.";
22      cout<<endl<<"*ptr was assigned the value of 12.";
23      cout<<"\n\nValue of i:\t\t"<<i;
24      cout<<endl<<"Memory Address of i:\t"<<&i;
25      cout<<"\nValue of j:\t\t"<<j;
26      cout<<endl<<"Memory Address of j:\t"<<&j;
27      cout<<endl<<"Value of ptr:\t\t"<<ptr;
28      cout<<endl<<"Value of *ptr:\t\t"<<*ptr;
29
30      //cout<<sizeof(ptr)<<endl;
31      //cout<<sizeof(i)<<endl;
32
33      return 0;
34  }
```

Another program that extends point.cpp

The program added another variable to point.cpp. A variable `j` was declared and initialized to `2`. The value and the location of the variable `j` are then printed also. Lines 18-19 show that the address stored in `ptr` was changed from `&i` to `&j`. This means that the variable `ptr` now points to the location of the variable `j`. `12` was assigned to `*ptr`. So instead of the changes in the pointer variable affecting the variable `i`, the variable `j` is affected. Look closely at the output.

After close examination, notice that the value of `j` becomes `12` and the value of `ptr` changes from `0x8fa5fff4` to `0x8fa5fff2`.

```
Value of i:               8
Memory Address of i:      0x28ff08
Value of j:               2
Memory Address of j:      0x28ff04
Value of ptr:             0x28ff08
Value of *ptr:            8

&j was assigned to ptr.
*ptr was assigned the value of 12.

Value of i:               8
Memory Address of i:      0x28ff08
Value of j:               12
Memory Address of j:      0x28ff04
Value of ptr:             0x28ff04
Value of *ptr:            12
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

SUMMARY

This lesson introduced pointers and how to use them. In the next lesson, more applications of pointers will be discussed. One of the most powerful features of C++ is the ability to manipulate the memory locations being occupied by the variables used within a program. This gives the program more flexibility while also granting the programmer more control.

SYNTAX	EXAMPLE
`<Data type> *<variable name>;`	`int *ptr;`

WORD BANK

Byte – a series of 1s and 0s usually grouped as a sequence of eight bits

***ptr** – if ptr is declared as a pointer, this translates to "the contents of the address stored in the variable ptr"

&x – translates to "the address of the variable x"

Pointer – a variable that is used to store memory locations

Garbage Values – values that are of no use; normally, these are the values assigned by the compiler to declared variables when they are not initialized

SCORE

NAME: _____

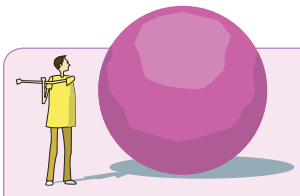
SECTION: _____

DATE: _____

SELF-CHECK

In your own words, explain how pointers are used. You may use point.cpp as your sample program.

CUT THIS PAGE



SKILLS WARM-UP

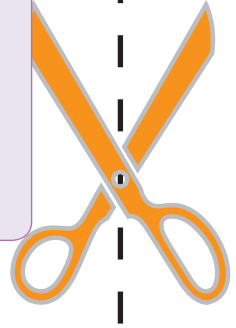
In the blanks below, identify the concepts related to C++ described by each item.

_____1. a sequence of 1s and 0s usually grouped into sequences of eight bits
_____2. variables used to store memory locations
_____3. the value assigned to pointers to establish that that particular pointer has not been assigned a valid memory address
_____4. values assigned to uninitialized variables
_____5. operator used when declaring a variable as a pointer
_____6. operator used on a variable to refer to the address of a variable instead of its value
_____7. operator used with a pointer variable to refer to the value stored in the memory address contained by the pointer variable

SKILLS WORKOUT

Write down the C++ code equivalent to the following statements.

1. Declare a variable named **`intptr`** which is a pointer to an **`int`**.
2. Declare an **`int`** variable named **`intval`**.
3. Declare a variable named **`intptrptr`** which is a pointer to a pointer to an **`int`**.
4. Set the value of **`intptr`** to the address of **`intval`**.
5. Set the value of **`intptrptr`** to the address of **`intptr`**.
6. Set the value of **`intval`** to zero using **`intptr`**.
7. Set the value of **`intval`** to 0 using **`intptrptr`**.
8. Display the address of **`intval`** using **`cout`** and the **`&`** operator.
9. Display the address of **`intval`** using **`cout`** and **`intptr`**.
10. Display the value of **`intval`** using **`cout`** and **`intptrptr`**.



PERFORMANCE TASK

Create a group composed of five members to research on pointers. Research deeper into pointers and assess why knowledge about pointers is essential for every C++ programmer. Research the typical programming scenarios where pointers are used. Research common errors encountered when using pointers, then discuss how to avoid or correct these errors. Use the space below for your notes. Present the results of your research in class for additional insight.

PERFORMANCE TASK

Create a group composed of five members to do research work on two C++ keywords: new and delete. Research the uses of these keywords and how they are related to pointers. Use the space that follows for your notes. Present the results of your research in class for additional insight.

LESSON 9

The door of Scrooge's counting-house was open that he might keep his eye upon his clerk, who, in a dismal little cell beyond, a sort of tank, was copying letters. Scrooge had a very small fire, but the clerk's fire was so very much smaller that it looked like one coal. But he couldn't replenish it, for Scrooge kept the coal-box in his own room; and so surely as the clerk came in with the shovel, the master predicted that it would be necessary for them to part. Wherefore the clerk put on his white comforter, and tried to warm himself at the candle; in which effort, not being a man of a strong imagination, he failed.

—from *A Christmas Carol*, Charles Dickens

Functions

Like most clerks during the 19th century, Dickens's Bob Cratchitt was forced to do his data organization by hand. Everything was written down in the record books and organized in such a manner that everything was labeled neatly and properly. Nowadays, data manipulation has gone a long way from being etched onto paper by quill and inkpot. Computer programs have made our lives easier in terms of organizing and accessing large amounts of different types of data.

What Is a Function?

Functions are generally used to make programs modular. This means that programs are subdivided into modules or "functions" which allow programs to be read or modified easily. Functions contain statements that perform a task or may return values. These values may be used by the program or parts of the program later on.

The use of functions introduces the implementation of encapsulation, which is a characteristic of OOP. A function encapsulates a set of operations and returns values to the program or other calling functions. When we say encapsulate, it means statements are grouped as one and placed in a function. It allows data to be hidden from other units of the program. Consider a capsule—when we take the medicine we don't have to get the powdered medicine out of the capsule. We take it because we know what it is for and we don't have to see what it looks like inside.

You may program functions in C++ even without function declaration. This is for lazy programmers. Although, if there is no function declaration on top of the source code, the function declaration should always be on top of the main method. The compiler reads the code starting from the top. Therefore, since the compiler is more concerned with the main method, your function declaration on top will be registered ahead before the main program uses it.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Recognize the different parts of a function.
2. Declare and define functions.
3. State the differences between global and local variables.

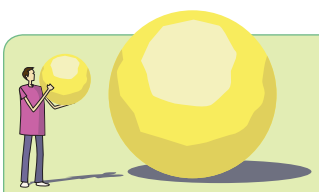
LESSON OUTLINE

1. What Is a Function?
2. Writing a Function
3. Using Functions
4. Global and Local Variables
5. Overloading Functions

NOTE

You have learned that every C++ program has at least one function, which is `main()`. When a C++ program is invoked, this is executed first. However, the `main()` function may call other functions, too.

When a function only performs a task, the keyword `void` is used followed by the function name and a pair of parentheses. This means that the function will not return a value. However, when the function returns a value, instead of using `void`, the function header will now contain the data type of the value the function is supposed to return.



EXAMPLE

```
void printName();       //this function returns nothing
int ComputeAgeInDays(int age);     //returns an int
char Answer();          //returns a char
```

Every function has its own name, and when that name is encountered by the compiler, the execution of the program is transferred to the body of the function. This is known as "calling the function." When the function returns, execution resumes on the next line.

Functions make the main program look organized and they contain statements on how to execute the program. In a C++ program, when there is a function call, control is transferred from the function to the last statement, where the function returns a value or contains an exit statement. After the function has completed its execution, the program resumes on the next statement right after the function call.

CONNECT

Using functions has a number of benefits:

1. It makes programs significantly easier to understand and maintain. The program can consist of a series of function calls rather than countless lines of code.

2. Well-written functions may be reused in multiple programs. If the functions are already written, they may be imported into the program that needs it. The C standard library is an example of the reuse of functions.

3. Different programmers working on a large project can divide the workload by writing different functions which allow the programs to be developed faster.

Writing a Function

SYNTAX

A general format for writing a function prototype or function declaration is:

```
datatype functionName(parameter list);
```

A general format of writing a function definition is:

```
datatype functionName(parameter list)
{
      statement1;
      statement2;
      :
      :
      statementN;
      return VariableOrExpression;
}
```

Let us create a program that computes for the value of a number when raised to another number.

```
1   #include <iostream>
2
3   using namespace std;
4
5   int raiseTo(int base, int power);
6   int x, b, p;
7
8   int main()
9   {
10      cout << "Enter base: ";
11      cin >> b;
12
13      cout << "Enter power: ";
14      cin >> p;
15      cout << endl;
```

```
16
17      x = raiseTo(b,p);
18      cout << b << " when raised to " << p << " equals " << x;
19      return 0;
20  }
21
22  int raiseTo(int base, int power)
23  {
24      int ctr, prod = 1;
25      for (ctr=1; ctr<=power; ctr++)
26      {
27          prod = prod * base;
28      }
29
30      return prod;
31  }
```

```
Enter base: 2
Enter power: 3

2 when raised to 3 equals 8
Process returned 0 (0x0)   execution time : 4.150 s
Press any key to continue.
```

The general rule in using functions is to declare it first then define it later. The function prototype, also called function declaration, consists of the return type, a function name, and a parameter list. It is a way of telling the compiler the data types of any return value and parameter so it can perform error-checking. You can see the function declaration in line 5 of the code.

```
int raiseTo(int base, int power);
```

Let us deconstruct the function declaration and identify the 3 elements involved. The function name is `raiseTo` and the function is expected to return an integer value after it is called. The variables `base` and `power` are the parameters of the function. Parameters are like local variables of a function. Usually found and declared in the function header, they accept values from the program or another function.

Lines 22 – 31 contain the actual function definition of `raiseTo`. The function definition consists of the prototype and a function body, which is a block of code enclosed in parentheses. The definition creates the actual function in memory. Functions may not need to be declared if the prototype is located just before `main()` or another function since the definition serves as the declaration, too. However, if it is placed after `main()` and no function declaration was made, it would result in an error since the prototype is required before it can be called in the program.

In the sample code provided, the actual function `raiseTo` is called within the main function in line 17 which is written as:

```
x = raiseTo(b,p)
```

It's quite easy to call a function. First, we assign a variable that will hold the result of the function. Then we type in the function name and provide the arguments to that function. Arguments are the values which may be variables or expressions passed on to the function and are normally found in the function call. The data type of the arguments should match the data types of the parameters. Also, the number of arguments should be equal to the number of parameters.

Using Functions

When using functions, there are important things to remember:

1. A function must be declared prior to its use except when it is already defined.
2. The parameter list of a function may have parameters of any data type.
3. Parameters are optional.
4. The number and data types of parameters should match the number and data types of the arguments passed on to it.
5. The `return` statement is used to return a single value from a function. It usually appears at the last line of a function but it may also appear anywhere within a function.
6. The `return` statement is optional.
7. If the data type of the value returned by a function is not declared, it is converted to `int`.
8. If a function does not return a value, its return type will be `void`.

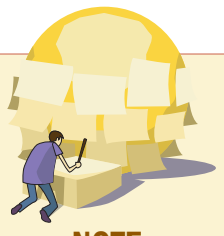
Global and Local Variables

We declare variables in two areas. The variable declaration in line 6 happens before any other code while the variable declaration in line 24 is within the `raiseTo` function. What's the difference between those two?

Observe that variables can be declared within a function like the one in line 24. These variables are local variables and are named as such since these variables can only be used within the function. They have a local scope. **Scope** refers to the section of code where a variable is allowed to be accessed or referenced—that is, variables declared within a function can only be used within that function and the other functions it calls. The life of local variables ends when a function is terminated or completed.

In the program, computing the power of an input number, where the `raiseTo()` function is used, the declaration `int ctr, prod = 1;` contains the `ctr` and `prod` variables which are local variables that can only be referenced inside the `raiseTo()` function. Should `main()` reference these variables, the compiler would return an error.

Another type of variable used in a C++ program is the global variable. Global variables, such as the ones in line 6, are declared outside any function. They have a global scope and thus are available for any function in the program, including `main()`. The life of global variables ends when `main()` is terminated or completed. In the same program mentioned earlier, the variables `x`, `b`, and `p` are global.



NOTE

Local variables and global variables may have the same names. When inside the `main()` program, global variables are referenced. However, when local variables with the same global variable names are referenced within the function they are declared in, this reference is made to local variables.

Pass-by-Reference Method

As mentioned, functions in C++ generally implement the pass-by-value method, which means that parameters only accept values from other functions of the program—the function will not be able to change the values of the arguments of the calling function.

Consider a program which passes two values on to a function and the function exchanges these two numbers, after which the program will display the exchanged values.

```
1   #include <iostream>
2   using namespace std;
3
4   void swap(int num1, int num2);
5
6   int main()
7   {
8     int n1,n2;
9
10    cout << "This program swaps two numbers." << endl;
11    cout << "Enter first number: ";
12    cin >> n1;
13
14    cout << "Enter second number: ";
15    cin >> n2;
16
17    swap(n1,n2);
18
19    cout << endl;
20    cout << "The swapped numbers in main() are " << n1<< " and " << n2;
21
22    return 0;
23  }
24
25  void swap(int num1, int num2)
26  {
```

```
27      int temp;
28      temp = num1;
29      num1 = num2;
30      num2 = temp;
31      cout << "The swapped numbers in swap() are " << num1 << " and " << num2;
32  }
```

```
This program swaps two numbers.
Enter first number: 10
Enter second number: 20
The swapped numbers in swap() are 20 and 10
The swapped numbers in main() are 10 and 20
Process returned 0 (0x0)   execution time : 5.954 s
Press any key to continue.
```

The function swaps the numbers as reflected in the output display inside the body of the function. But outside, the values 10 and 20 are not exchanged. This happens because when parameters use the pass-by-value method, parameters act as local variables, meaning another memory location within the computer was allocated for these parameters. So changes made on the parameters would not be reflected outside the function since the scope of local variables is only within the function's environment.

In the preceding example, what is needed is that the changes made to parameters be passed on back to the calling function. This will be made possible by the *pass-by-reference* method. Instead of values being passed on to the function's parameters, the addresses of the arguments are passed and therefore no new memory allocation is made. The parameters, as well as the arguments, only refer to the same memory location. In C++, pointers are used to pass by reference. There are two operators used with pointers: one is the **indirection** operator (`*`) which returns the value of the address stored by the pointer and the **address-of** operator (`&`) which returns the address of the pointer.

Consider the modified program for exchanging two numbers. This time, pointers are used.

```
1   #include <iostream>
2   using namespace std;
3
4   void swap(int *num1, int *num2);
5
6   int main()
7   {
8       int *n1, *n2;
9       int input1, input2;
10
11      cout << "This program swaps two numbers." << endl;
12      cout << "Enter first number: ";
13      cin >> input1;
14      cout << "Enter second number: ";
15      cin >> input2;
16
17      swap(&input1, &input2);
18
19      cout << endl;
20      cout << "The swapped numbers in main() are " << input1 << " and " << input2;
```

```
22      return 0;
23  }
24
25  void swap(int *num1, int *num2)
26  {
27      int temp;
28      temp = *num1;
29      *num1 = *num2;
30      *num2 = temp;
31      cout << endl;
32      cout << "The swapped numbers in swap() are " << *num1 << " and " << *num2;
```

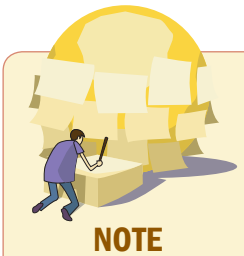
```
This program swaps two numbers.
Enter first number: 10
Enter second number: 20

The swapped numbers in swap() are 20 and 10
The swapped numbers in main() are 20 and 10
Process returned 0 (0x0)   execution time : 1.825 s
Press any key to continue.
```

Overloading Functions

In C++, it is valid to give several functions the same name. The only requirement is that it should have a different set of parameters, whether data types or a number of parameters. This is known as **function overloading**. For example:

```
float area(int length, int width);
 float area(float side);
```

The two functions have the same name `area()` but contain different parameters. Take note that they may return values of different data types.



NOTE

Function overloading allows for functions to have many forms. The method described earlier is an example of the C++ implementation of polymorphism.

Function polymorphism allows the function to be overloaded. There is only one function name but a number of tasks are associated with the function. This is significant especially in programs, which require the same function to be performed on different types of values. Consider a program which computes for the area of a circle and a rectangle. The equation to compute the area for each shape is different.

- The area of a circle is computed using the equation $3.14 * r^2$ (r = radius)
- The area of a square is computed using the equation s^2 (s = length of one side)
- The area of a rectangle is computed using the equation l*w (length x width)

With function overloading, the program may be constructed as:

```
#include <iostream>

using namespace std;

float area(float radius);
int area (int length, int width);
int area (int side);
float r;
int l,w,s;

int main()
{
    int choice;
    float result;

    cout<<"This program computes for the area based on the option chosen."<<endl;
    cout<<"Enter 1 to select a circle."<<endl;
    cout<<"Enter 2 to select a rectangle."<<endl;
    cout<<"Enter 3 to select a square."<<endl;
    cout<<"Enter your choice:";
    cin>>choice;

    switch(choice)
    {
        case 1:
        {
            cout<<"Enter the radius of the circle:";
            cin>>r;
            result=area(r);
            break;
        }
        case 2:
        {
            cout<<"Enter length of the rectangle:";
            cin>>l;
            cout<<"Enter width of the rectangle:";
            cin>>w;
            result=area(l,w);
            break;
        }
        case 3:
        {
            cout<<"Enter sides of a square:";
            cin>>s;
            result = area(s);
            break;
        }
        default:
            cout<<"Incorrect choice."<<endl;
    }
    cout<<endl<<"The area of your preferred figure is "<<result;
    return 0;
}

float area(float r)
{
    return(3.14 * r * r);
}

int area(int l, int w)
```

```
61  {
62      return(l*w);
63  }
64  int area (int s)
65  {
66      return(s*s);
67  }
68
```

Result for computing the area of a circle:

```
This program computes for the area based on the option chosen
Enter 1 to select circle.
Enter 2 to select rectangle.
Enter 3 to select square.
Enter your choice: 1
Enter radius of a circle: 5

The area of your preferred figure is 78.5
Process returned 0 (0x0)   execution time : 7.527 s
Press any key to continue.
```

Result for computing the area of a rectangle:

```
This program computes for the area based on the option chosen
Enter 1 to select circle.
Enter 2 to select rectangle.
Enter 3 to select square.
Enter your choice: 2
Enter length of a rectangle: 6
Enter width of a rectangle: 4

The area of your preferred figure is 24
Process returned 0 (0x0)   execution time : 9.348 s
Press any key to continue.
```

Result for computing the area of a square:

```
This program computes for the area based on the option chosen
Enter 1 to select circle.
Enter 2 to select rectangle.
Enter 3 to select square.
Enter your choice: 3
Enter sides of a square: 9

The area of your preferred figure is 81
Process returned 0 (0x0)   execution time : 9.055 s
Press any key to continue.
```

SUMMARY

Functions are used to modularize the programs created. They can either return values to another function calling them or returning nothing at all except for performing certain tasks. The variables or values passed on to functions are called arguments, while the variables declared on function headers are called parameters. The number and type of arguments passed by a function should correspond to the number and type of the parameters of the accepting functions.

WORD BANK

Address-of Operator (`&`) – returns the address of the pointer

Arguments – values which may be variables or expressions passed on to the function and are normally found in the function call

Indirection Operator (`*`) – returns the value of the address stored by the pointer

Function Overloading – creating more than one function with the same name

Function Prototype or **Function Declaration** – consists of the return type, a function name, and a parameter list; a way of identifying the data types of any return value and parameter for the compiler so it can perform error-checking

Global Variables – have a global scope, and thus, are available from any function in the program, including `main()`

Local Variables – can only be used within the function; have a local scope

Parameters – found and declared in the function header; accept values from the program or another function

Pass-by-Reference Method – the addresses of the arguments are passed; the parameters, as well as the arguments, only refer to the same memory location

Pass-by-Value Method – parameters only accept the values passed on to them

Return Statement – used to return a single value from a function

Return Values – the results of the function which may be passed to another function

Scope – the section of code where a variable is allowed to be accessed or referenced

PERFORMANCE TASK

Did you know that functions can invoke themselves? These functions are called recursive functions. Your group task for this exercise is to research on recursive functions. Determine scenarios in which these types of functions can be useful. Also determine any important things to remember when writing/using recursive functions. Discuss your results in class for further insight and formulate a code of your own recursive function. Use the space provided for your notes.

PERFORMANCE TASK

Together with your group, write a program that accepts a number and determines if it is odd or even. The program should use a function to test if the number is odd or even.

Together with your group, write a function that accepts two integers and returns the result of dividing the first by the second. Note that the process of division should not be carried out if the second number is zero. In such a case, the function should return -1 and should display "Division by zero is not allowed."

SCORE

NAME: _____

SECTION: _____

DATE: _____

SELF-CHECK

On the blanks provided, identify the concepts related to C++ described by the items below.

_____1. the keyword used to specify that a function will not return a value

_____2. value that results from a function

_____3. variables or expression that are passed on to functions

_____4. these are found and declared in the function header, and they accept values from the program or another function

_____5. a way of telling the compiler of the data types of any returning value and the parameters of a function

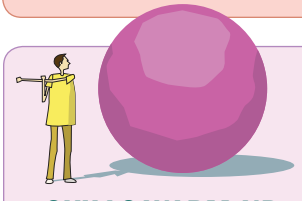
_____6. the process of creating more than one function with the same name

_____7. the section of code where a variable is allowed to be accessed or referenced

_____8. variables that can only be accessed within a function

_____9. a method of passing values to a function indirectly by providing the memory addresses of the values to be passed as parameters

_____10. a method of passing values to a function directly by providing the values as parameters to the function call



SKILLS WARM-UP

Write **True** on the blank provided if the statement is true, otherwise, write **False**.

_____ 1. A function always returns a value.

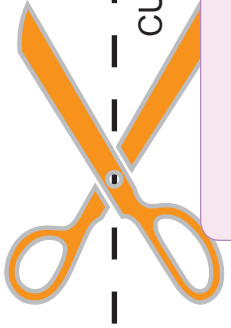
_____ 2. A function can return multiple values using the return keyword.

_____ 3. A function must be declared prior to its use except when it is already defined.

_____ 4. Function parameters are optional.

_____ 5. The return statement may appear anywhere within a function.

CUT THIS PAGE



_____ 6. When the data type of the value returned by a function is not declared, it is converted to void.

_____ 7. The return statement is only used in functions that return values.

_____ 8. Local variables should not have names that are already used by global variables.

_____ 9. Changes made on the parameters would not be reflected outside of the function if these values were passed by reference.

_____ 10. Functions that have the same set of parameters can have the same name.

SKILLS WORKOUT

Error Identification

Given the following programs, identify the line where an error occurs and state what should be done to correct it.

1.

```
#include <iostream.h>

int main()
{
      int x, y;
      cin >> x;
      y = funct(x);
      cout << y;
      return 0;
}

int funct(int x)
{
      return (x*x);
}
```

2.

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

float area(float radius);
int main()
{
      float x, y;
      clrscr();
      cout << "Enter radius: ";
      cin >> x;
      y = area();
      cout << y;
      getch();
      return 0;
}

float area(float radius)
{
      return (3.14*radius*radius);
}
```

PERFORMANCE TASK

Function Prototype

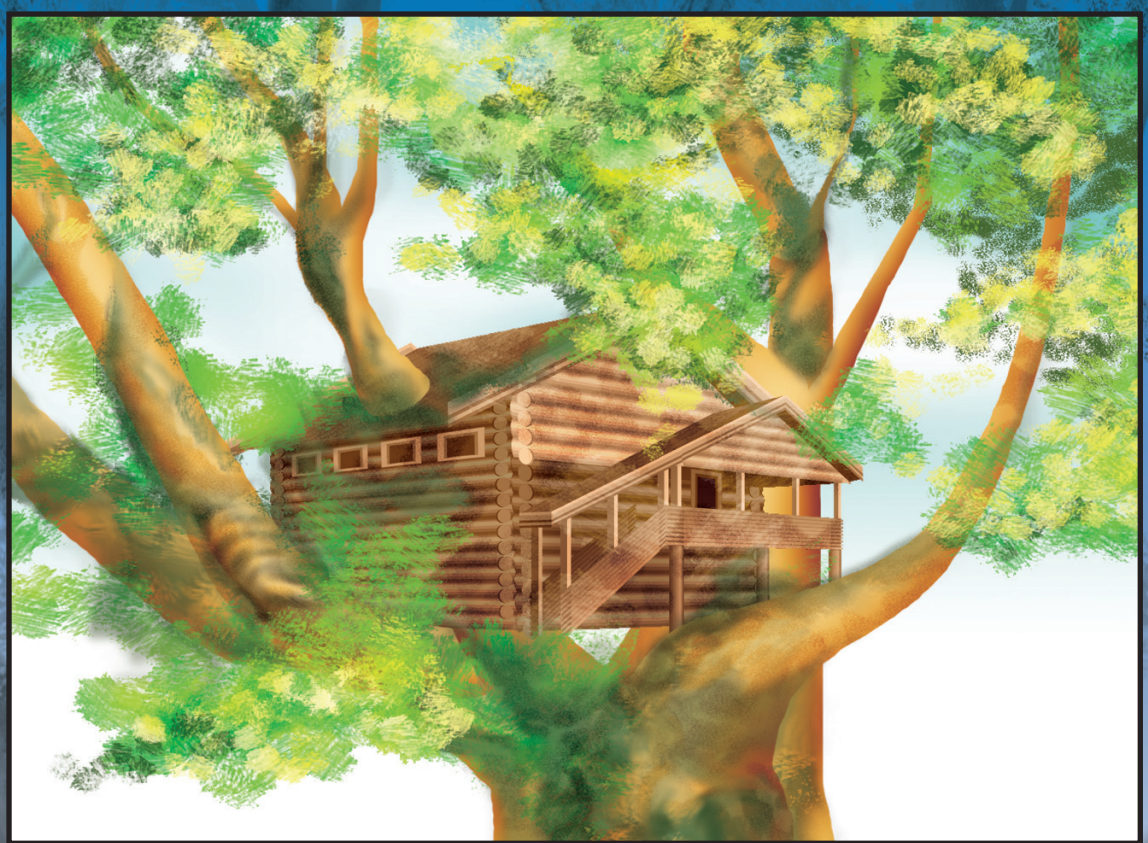
With the help of your group (form one), write the function prototypes given the following specifications.

1. Function Name: AreaTriangle
   Return Value: Float
   Parameter/s: Float Base and Height

2. Function Name: Cube
   Return Value: Integer
   Parameter/s: Integer Number

3. Function Name: Odd
   Return Value: Integer
   Parameter/s: Integer Number

LESSON 10



Fritz secured the ladder so firmly to the branch, that I had no hesitation in ascending myself. I carried with me a large pulley fixed to the end of a rope, which I attached to a branch above us, to enable us to raise the planks necessary to form the groundwork of our habitation. I smoothed the branches a little by aid of my axe, sending the boys down to be out of my way. After completing my day's work, I descended by the light of the moon, and was alarmed to find that Fritz and Jack were not below; and still more so, when I heard their clear, sweet voices, at the summit of the tree, singing the evening hymn, as if to sanctify our future abode. They had climbed the tree, instead of descending, and, filled with wonder and reverence at the sublime view below them, had burst out into the hymn of thanksgiving to God.

—from *Swiss Family Robinson*, Johann David Wyss

Basic Classes

In *Swiss Family Robinson,* a family stranded on a deserted island after a shipwreck survived on their own while they waited for someone to rescue them. Part of their survival depended on how well they managed to create a shelter for themselves using the materials available to them. As a result, they managed to build their own house using their ingenuity and resourcefulness.

Creating New Data Types

In C++, programmers can create new data types wherein each of these can have all the functionalities of the built-in data types. There is a need for such new types to be created because sometimes, the existing types do not really describe the objects or information. For example, even if we have integers and double data types in C++, using them does not fully describe the object or data it represents. Say we have to describe the object car, its model, its rate of acceleration or deceleration, even its mileage. Using a string of characters to define the model and integer or float types to describe the speed does not clearly describe the data. Thus, a need for new data types is evident when programmers cannot use the existing ones. This is made possible through the use of classes.

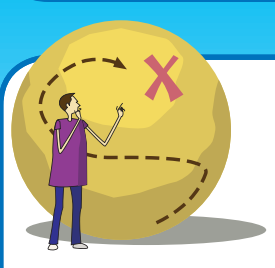
Classes and Members

As defined in Lesson 2, a class is an actual representation of an abstract data type. It therefore provides implementation details for the data structure and operations used. For example, if we have a Person class, which will be used to represent persons, the following attributes and methods may be defined: the person's gender, age, and message.

LESSON OUTCOMES

At the end of this lesson, the student will be able to:

1. Discover more about classes and objects.
2. Define new classes and create objects of that class.
3. Recognize member functions and member data.
4. Distinguish constructors from destructors.



LESSON OUTLINE

1. Creating New Data Types
2. Classes and Members
3. Private vs. Public
4. Constructors vs. Destructors

NOTE

A new type is created when a class is declared. A class is a programmer's data type, which encapsulates data, members, abilities, and methods. This allows programmers to model real-world objects and information.

With the given definition, data types in C++ would not be enough to describe the real-world object so classes are created. In `int x;`, we are declaring a variable x of data type integer. But in `Person Student;`, we are declaring Student of class Person.

A class contains members in the form of variables and functions, which are the data members and methods, respectively. The code snippet shows how members can be defined in a class.

Data Members
- Gender
- Age

Method
- Welcome

Declaring a Class

To declare a class, we use the keyword class followed by a user-defined name by the class definition in braces. The class definition contains the class members, its data, the class methods, and its functions.

The code snippet shows how to construct a Person class which is modelled after of a real-world human being.

```
class Person
{
 public:
    void Welcome();
    char Gender;
    int Age;
};
```

The keyword `public` indicates that the method `Welcome()` and the variables Gender and Age can be called from a code outside of the class. That is, they may be called by other parts of a program using objects of this class.

NOTE

The opposite of public is private. Members of the class declared as private cannot be directly accessed outside the class. Public and private keywords will be further discussed in the next section.

In the previous code snippet, the methods are declared but not defined. An implementation for each method must be written.

```
class Person
{
 public:
    void Welcome();
    char Gender;
    int Age;
};
void Person::Welcome()
{
   cout << "Good Day!" << endl;
}
```

Since the methods are implemented outside of the class definition, they must be identified as belonging to that class. This is done with the scope resolution operator (::). It identifies each method, like `Welcome()`, as belonging to the class Person.

Accessing Class Members

To access a member of a class, the dot operator (.) is used. Consider the class Person defined previously; if we have the declaration `Person Student` and we want to assign constant values to the data members of Student, this will be done using the dot operator:

```
Student.Gender = 'F';
Student.Age = 17;
```

When we call the method `Welcome()`, the dot operator is also used:

```
Student.Welcome().
```

Private vs. Public

All data and method members of the class are private by default—they can only be accessed within methods of the class itself. Meanwhile, public members can be accessed through any object of the class. To ensure that data abstraction, hiding, and isolation are implemented, the use of the keyword `private` is encouraged especially when dealing with data members or variables. It enables you to separate the details of how the data is stored from how it is used.

So the previous class definition may be modified as follows:

```
class Person
{
 public:
    void Welcome();
 private:
    char Gender;
    int Age;
};
void Person::Welcome()
{
   cout << "Good Day!" << endl;
}
```

All data members of the class are declared as private. Thus, if we have `Student.Age = 17;`, this would result in an error since it cannot access private data. The reference is obviously done outside the class definition and somewhere within the program or other function. In such cases, a need for accessor methods arises.

Accessor Methods

Even if data members are declared to be private and therefore inaccessible to other parts of the program, the use of **accessor methods** to *set* and *get* member variables allow data members to be accessed from any part of the program. These are the member functions that other parts of your program call to set and get values from the private member variables.

A public accessor method is a class member function used either to read or to set its private class member's value. See the modifications made in the Person class example on the next page.

Take note of the presence of the *get* and *set* methods, specifically `getAge()`, `setAge()`, `getGender()`, and `setGender()`. A different variable name is used as members of the class to distinguish it from the parameters of the member functions.

```
class Person
{
public:
    void Welcome();
    void setGender(char vGender);
    char getGender();
    void setAge(char vAge);
    int getAge();

private:
    char Gender;
    int Age;
};

void Person::setGender(char vGender)
{   Gender = vGender;  }

char Person::getGender()
{   return Gender;  }

void Person::setAge(char vAge)
{   Age = vAge; }

int Person::getAge()
{   return Age;  }


void Person::Welcome()
{
   cout << "Good Day!" << endl;
}
```

Using Classes

Now, let us write a program that would use the previously defined Person class. The program will ask for user's input for age and gender and will welcome the person addressed as Miss or Sir, and inform the user that he/she is ready to vote based on the input age. The program may look something like what's on the next pages:

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

class Person
{
public:
    void Welcome();
    void setGender(char vGender);
    char getGender();
    void setAge(char vAge);
    int getAge();

private:
    char Gender;
    int Age;
};

void Person::setGender(char vGender)
{
    Gender = vGender;
}

char Person::getGender()
{
    return Gender;
}

void Person::setAge(char vAge)
{
    Age = vAge;
}

int Person::getAge()
{
    return Age;
}

void Person::Welcome()
{
   cout << "Good Day!" << endl;
}

int main()
{
  int vAge;
  char vGender;
  Person Student;

  cout << "This program greets the person based on gender and" << endl;
  cout << "determines if he can vote or not" << endl << endl;

  cout << "Enter your age: ";
  cin >> vAge;

  cout << "Enter your gender (M/F): ";
  cin >> vGender;

  Student.setAge(vAge);
  Student.setGender(vGender);

  Student.Welcome();

  if (Student.getGender()=='F')
  {
      cout << "Miss, ";
  }
  else
  {
      cout << "Sir, ";
  }
```

```
71
72      if (Student.getAge()>=18)
73      {
74          cout << "good news! You are ready to vote.";
75      }
76      else
77      {
78         cout << "I'm sorry to inform you ";
79         cout << "that you cannot vote this year!";
80       }
81
82       return 0;
83
84  }
```

```
This program greets the person based on gender and
determines if he can vote or not

Enter your age: 11
Enter your gender (M/F): m
Good Day!
Sir, I'm sorry to inform you that you cannot vote this year!
Process returned 0 (0x0)   execution time : 4.331 s
Press any key to continue.
```

```
This program greets the person based on gender and
determines if he can vote or not

Enter your age: 18
Enter your gender (M/F): f
Good Day!
Sir, good news! You are ready to vote.
Process returned 0 (0x0)   execution time : 3.324 s
Press any key to continue.
```

Constructors vs. Destructors

Just as variables are declared and initialized, classes are also declared and initialized. **Initialization** is the process of declaring and creating an object with some value. This value is subject to change, of course. The process only makes sure that the objects (such as variables or classes) do not merely contain garbage or meaningless information.

Classes should also be initialized through the special method named ***constructor***. It is called when an object of the class is created. The constructor can be used to initialize variables, dynamically allocate memory, or set up any needed resources. It can take parameters as needed, but it cannot have a return value—not even void. The constructor is a class method with the same name as the class itself.

Another special method, the **destructor**, is called when an object is destroyed. When we declare a constructor, we would also want to declare a destructor. The destructor is used to free any memory that was allocated and to possibly release other resources. It always has the same name of the class, preceded by a tilde (~). It takes no argument and has no return value. When

we initialize a class, to make sure that a memory is allocated for it, we would also deallocate the memory allotted to the class by means of this method.

Below is the Person class with a constructor and a destructor added. The program is similar to the preceding sample because these methods allocate and deallocate memory spaces and are, of course, invisible.

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

class Person
{
    public:
        Person();         //Constructor
        ~Person();        //Destructor
        void Welcome();
        void setGender(char vGender);
        char getGender();
        void setAge(char vAge);
        int getAge();

    private:
        char Gender;
        int Age;
};

Person::Person()
{
    Age=0;               // initialize values to data members
    Gender='M';
}

Person::~Person()
{      // does nothing except deallocate memory
}

void Person::setGender(char vGender)
{
    Gender = vGender;
}

char Person::getGender()
{
    return Gender;
}

void Person::setAge(char vAge)
{
    Age = vAge;
}

int Person::getAge()
{
    return Age;
}

void Person::Welcome()
{
   cout << "Good Day!" << endl;
}

int main()
{
  int vAge;
```

```
58      char vGender;
59      Person Student;
60
61      cout << "This program greets the person based on gender and" << endl;
62      cout << "determines if he/she can vote or not" << endl << endl;
63
64      cout << "Enter your age: ";
65      cin >> vAge;
66
67      cout << "Enter your gender (M/F): ";
68      cin >> vGender;
69
70      Student.setAge(vAge);
71      Student.setGender(vGender);
72
73      Student.Welcome();
74      if (Student.getGender()=='F')
75      {
76          cout << "Miss, ";
77      }
78      else
79      {
80          cout << "Sir, ";
81      }
82
83      if (Student.getAge()>=18)
84      {
85          cout << "Good news! You are ready to vote.";
86      }
87      else
88      {
89         cout << "I'm sorry to inform you ";
90         cout << "that you cannot vote this year.";
91      }
92
93      return 0;
94  }
```

In the preceding example, the constructor was used to initialize member variables. The destructor, as defined above, performs no real actions. In other classes, the destructor might free memory that was allocated, release some resources, or perform some other clean up activity.

SUMMARY

In C++, programmers can create new data types wherein each can have all the functionalities of the built-in data types. All members of a class are private by default. This means that they can only be accessed within methods of the class itself. Public members can be accessed through any object of the class.

WORD BANK

Accessor Methods – public functions that set and get private member variables

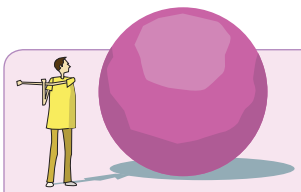
Constructor – called when an object of the class is instantiated or created; used to initialize variables, dynamically allocate memory, or set up any needed resources

SCORE

NAME: _____

SECTION: _____

DATE: _____



SKILLS WARM-UP

A. On the spaces provided, identify the concepts related to C++ described by each item below.

_____1. a representation of an abstract data-type
_____2. the term used to refer to the variables and functions contained in a class
_____3. the keyword used to indicate that a method or variable can be called from a code outside of the class
_____4. the operator used to access a member of a class
_____5. the operator used when declaring a method to specify that the method belongs to a specific class
_____6. an instance of a class
_____7. public methods used to allow private data members to be accessed from outside of the class
_____8. the process of declaring and creating an instance of a class with some value
_____9. a special method that is called when an object is destroyed
_____10. a special method when an object is created

B. Write **True** on the blank provided if the statement is true, otherwise write **False**.

_____ 1. A method declared as `private` can be called from another class provided that the class is declared as `public`.
_____ 2. A return type of the class constructor is `void`.
_____ 3. The class constructor must have the same name as the class.
_____ 4. Once an object is initialized, the values of its members cannot be changed.
_____ 5. The class destructor takes no arguments and has no return value.
_____ 6. To ensure that data abstraction, hiding, and isolation are implemented, the use of the keyword `private` is encouraged.
_____ 7. Accessor methods are required to start with *set* and *get*.
_____ 8. All data and method members of the class are private by default.
_____ 9. Declaring a destructor is required when you declare a constructor.
_____ 10. One class can have multiple destructors.

CUT THIS PAGE

SKILLS WORKOUT

1. Write the code that declares a class called Student with these data members: allowance, expenses, yearsStudying, and age.

2. Rewrite the Student class to make data members private, and provide public accessor methods to get and set each of the data members.

3. Write a program with the use of Student class. The program accepts user's input for allowance, expenses, years of studying, and age, and computes for the student's savings as well ratio of student life to his lifetime.

Destructor – called when an object is destroyed; used to free memory that was allocated, release some resources, or perform some other cleanup activity

Initialization – combines the definition of the variable with its initial assignment

PERFORMANCE TASK

In this exercise, each group, composed of five students, will explore the primary steps in Object-Oriented Programming. Each group should think of a real world object and then construct a class to represent this object. Discuss the essential variables and methods your object should possess. Identify as many variables and methods as you can and write these down on the table below.

// Class Name
// Variables
// Methods

After that, name the C++ code for declaring the class representing your real-world object.

PERFORMANCE TASK

One of the object-oriented programming concepts discussed in Lesson 2 is inheritance. C++, is an OOP language that supports inheritance. Many C++ algorithms depend on this feature. Your task is to create a group composed of 5 members to research on how inheritance is implemented in C++. Give code examples that illustrate this feature and give programming scenarios where inheritance can be useful.