M.Sc(IT) SEMESTER-3 **PAPER: MITM2101T**

Object Oriented Programming Using C++

UNIT No.1

Center for Distance and Online Education, PunjabiUniversity,Patiala

Lesson No:

1. **Evolution of Object Oriented Programming**
2. **C++ Identifiers, Keywords, Constants & Variables**
3. **C++ Operators**
4. **Special Operators, Operator Precedence and Associativity**
5. **Type Conversion**
6. **Input, Output Statements**
7. **File Inputs/Outputs**
8. **C++ I/O System**
9. **C++ File I/O**
10. **Conditional, Iterative and Control Statement**
11. **Arrays**
12. **Arrays as Character Strings**
13. **Structures and Unions**
14. **Arrays of Objects, Allocating Arrays and The Vector Container**
15. **Functions**
16. **Pointers**

(Syllabus)

MITM2101T: Object Oriented Programming Using C++

Maximum Marks: 70 **Maximum Time: 3 Hrs.**

Minimum Pass Marks: 35%

Course Objective: This course is designed to explore computing and to show students the art of computer programming. Students will be able to learn Understand object oriented programming and advanced C++ concepts for writing good programs. On completion of this course, the students will be able to

- Write, compile and debug programs in C++language.
- Use different data types, operators and console I/O function in a computer program.
- Design programs involving decision control statements, loop control statements and case control structures.
- Understand the implementation of arrays, pointers and functions and apply the dynamics of memory by the use of pointers.
- Comprehend the concepts of structures and classes: declaration, initialization and implementation.
- Apply basics of object oriented programming, polymorphism and inheritance.
- Use the file operations, character I/O, string I/O, file pointers, pre-processor directives and create/update basic data files.

Course Content

SECTION A

Evolution of OOP: Procedure Oriented Programming, OOP Paradigm, Advantages and disadvantages of OOP over its predecessor paradigms. Characteristics of Object Oriented Programming.

Introduction to C++: Identifier, Keywords, Constants. Operators: Arithmetic, relational, logical, conditional and assignment. Size of operator, Operator precedence and associativity. Type conversion, Variable declaration, expressions, statements, manipulators. Input and output statements, stream I/O, Conditional and Iterative statements, breaking control statements. Storage Classes, Arrays, Arrays as Character Strings, Structures, Unions, Bit fields, Enumerations and User defined types.

Pointers: Pointer Operations, Pointer Arithmetic, Pointers and Arrays, Multiple indirections, Pointer to functions. Functions: Prototyping, Definition and Call, Scope Rules. Parameter Passing by value, by address and by reference, Functions returning references, Const functions, recursion, function overloading, Default Arguments, Const arguments, Pre-processor, Type casting.

SECTION B

Classes and Objects: Class Declaration and Class Definition, Defining member functions, making functions inline, Nesting of member functions, Members access control. THIS pointer. Objects: Object as function arguments, array of objects, functions returning objects, Const member. Static data members and Static member functions, Friend functions and Friend classes.

Constructors: properties, types of constructors, Dynamic constructors, multiple constructors in classes. Destructors: Properties, Virtual destructors. Destroying objects, Rules for constructors and destructors. Array of objects. Dynamic memory allocation using new and delete operators, Nested and container classes, Scopes: Local, Global, Namespace and Class.

Inheritance: Defining derived classes, inheriting private members, single inheritance, types of derivation, function redefining, constructors in derived class, Types of inheritance, Types of base classes, Code Reusability. Polymorphism: Methods of achieving polymorphic behavior.

Operator overloading: overloading binary operator, overloading unary operators, rules for operator overloading, operator overloading using friend function. Function overloading: early binding, Polymorphism with pointers, virtual functions, late binding, pure virtual functions and abstract base class. Difference between function overloading, redefining, and overriding.

Templates: Generic Functions and Generic Classes, Overloading of template functions. Exception Handling catching class types, handling derived class exceptions, catching exceptions, restricting exception

Pedagogy:

The Instructor is expected to use leading pedagogical approaches in the class room situation, research-based methodology, innovative instructional methods, extensive use of technology in the class room, online modules of MOOCS, and comprehensive assessment practices to strengthen teaching efforts and improve student learning outcomes.

The Instructor of class will engage in a combination of academic reading, analyzing case studies, preparing the weekly assigned readings and exercises, encouraging in class discussions, and live project based learning.

Text and Readings: Students should focus on material presented in lectures. The text should be used to provide further explanation and examples of concepts and techniques discussed in the course:

- Herbert Schildt, "The Complete Reference C++", Tata McGraw-Hill.
- Deitel and Deitel, "C++ How to Program", Pearson Education.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications.
- BjarneStrautrup, "The C++ Programming Language", Addition- Wesley Publication Co.
- Stanley B. Lippman, JoseeLajoie, "C++ Primer", Pearson Education.
- E. Balagurusamy, "Object Oriented Programming with C++", Tata McGraw-Hill.

Scheme of Examination

- English will be the medium of instruction and examination.
- Written Examinations will be conducted at the end of each Semester as per the Academic Calendar notified in advance
- Each course will carry 100 marks of which 30 marks shall be reserved for internal assessment and the remaining 70 marks for written examination to be held at the end of each semester.
- The duration of written examination for each paper shall be three hours.
- The minimum marks for passing the examination for each semester shall be 35% in aggregate as well as a minimum of 35% marks in the semester-end examination in each paper.
- A minimum of 75% of classroom attendance is required in each subject.

Instructions to the External Paper Setter

The question paper will consist of three Sections: A, B and C. Sections A and B will have four questions each from the respective section of the syllabus and will carry 10.5 marks for each question. Section C will consist of 7-15 short answer type questions covering the entire syllabus uniformly and will carry a total of 28 marks.

Instructions for candidates

- Candidates are required to attempt five questions in all, selecting two questions each from section A and B and compulsory question of section C.
- Use of non-programmable scientific calculator is allowed.

Lesson No. 1 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

Evolution of Object Oriented Programming

Objectives:
After reading this lesson you should be able to:

Understand the general concept of Procedural Oriented Programming and Object Oriented Programming Understand how Object Oriented Programming was born and developed

- Understand the concept Programming Paradigm.
- Understand the main characteristics of Object Oriented Programming
- Understand the difference between Object Oriented Programming and Procedural Oriented Programming.
- Understand the main characteristics of Object Oriented Programming .
- Understand the main advantages and disadvantage of using Object Oriented Programming .

Introduction

Programmers eventually discovered that it made a program clearer and easier to understand if they were able to take a bunch of data and group it together with the operations that worked on that data. Such a grouping is called an *object* or *class.* Designing programs by designing classes is known as *object-oriented design (OOD).*In the recent decades programmes have became more complex. Hence it has become difficult and time consuming to create, manage and maintain programmes using the procedural programming techniques. Thus in order to solve the above problems encountered by the procedural approach for designing large and complex programs a new concept called 'Object Oriented Programming' emerged. Object oriented programming is a powerful programming concept that is used to develop software application and programmes today. Unlike the conventional programming approach, the object oriented approach lays more stress on data rather than on functions and does not allow data to be accessed freely in the system. The basic approach of the object oriented programming is to combine both data and function that operate on that data into a single unit. Such a unit is called an object. The only way to access the data of an object is by its functions. This secures the data from accidental modification from the outside function.

Over the years, it was realized that, while C is exceedingly powerful, it had some aspects that made it hard to learn and error prone. Further, while complex problems were successfully broken down into smaller functional units in C, the idea of treating the data and the functions that operate on that data as an object or entity led to the development of C++.

1.1 Programming Techniques

1.1.1 Procedural oriented programming (POP)

The repetition of sequence of statements which are needed at different locations in the program has lead to the creation of procedures. The sequences of statements are given a name and a technique is used to call and return from these procedures. A procedure call is used to invoke the procedure. After the sequence is processed program control shift back to the location just after the position from where the call was made. The concept is shown in figure 1 below.

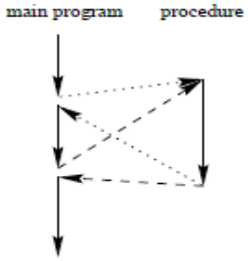


Figure 1

The programs are more structured and can be viewed as a sequence of procedure calls. The main program is responsible to pass data to the individual procedures which process the data and return it to the main program and once the program has finished the resulting data is presented as the output. It is easier to find errors in structured programs than unstructured especially if program size is large. Thus procedural programming approach is a top down approach in which a program is divided into procedures/functions that perform specific functions as shown in Figure 2.

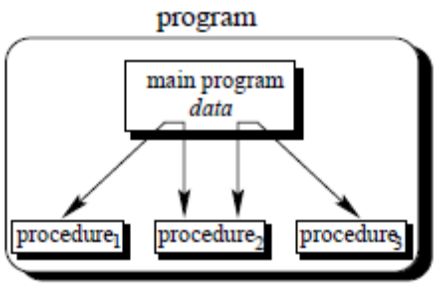


Figure 2

1.1.4 Object oriented programming (OOP)

The term object oriented means that we organize a problem as a collection of discrete objects that contains data and functions that operate on that data. The basic approach of the object oriented approach is to combine both data and function that operate on that data into a single unit. Such a unit is called an object. The only way to access the data of an object is by its functions. This secures the data from accidental modification from the outside function. Data of an object can be accessed only by the function of that object. Also different objects in the system communicate with each other by calling the functions of the other objects as shown in the figure 3.

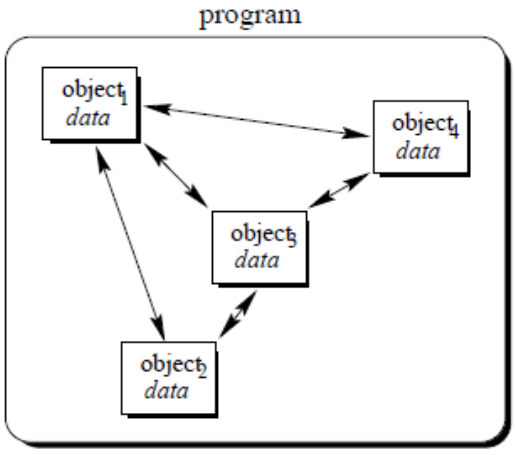


Figure 3

Object oriented problem solving approach works similar to the ways how humans solve problems from the user's perspective. Objects are often used to model real world entities that we normally find in everyday life. An object can be a person, place, thing, concept or anything you see in the real world. In the program, data describing the object are referred to as attributes of an object. The behavior of an object is implemented using functions. Each object has identity that distinguishes it from all other objects. An object student is shown in figure 4 below:

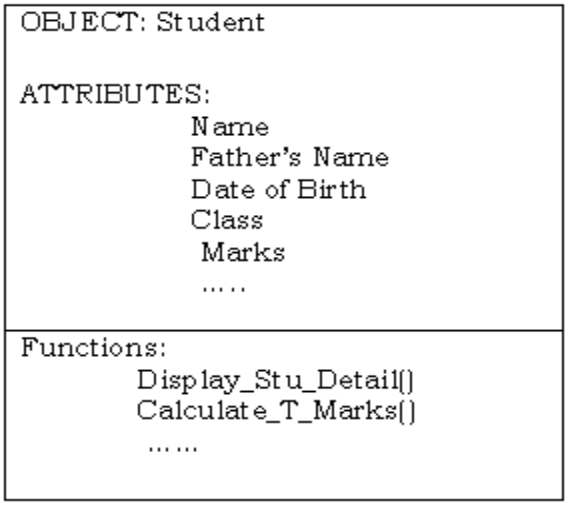


Figure 4

1.2 Basics Concepts related to Object Oriented Programming

1.2.1 Classes and Objects

An object is just an instance of a data type. For example, when you declare a variable of type int, you are creating an instance of the int data type. A class is like a data type in that it is the blueprint upon which an object is based. When you need a new object in a program, you create a class, which is a kind of template for the object. Then, in your program, you create an instance of the class. This instance is called an object.

Classes are really nothing more than user-defined data types. As with any data type, you can have as many instances of the class as you want. For example, you can have more than one window in a Windows application, each with its own contents or think again about the integer data type (int). You can declare many integers, as many as you like. The same is true of classes. After you define a new class, you can create many instances of the class. Each instance (called an object) normally has full access to the class's methods and gets its own copy of the data members.

Declaring the class does not allocate memory for that class. It is just a way of representing what data that the class has and what the class can do. A class needs to be accessed through objects. An object can also be defined as a particular instance of a class. Consider the example of the class car it does not represent a single car, rather it represent all possible cars by listing the characteristics and behaviors they all can have. In

creating objects a specific car/object needs to be declared which has a specific color, type of care and year of make. There is a general format for declaring an object from a class. That is the class name followed by the object name followed by a semicolon.

class_name object_name;

Object name can be any name, which is suitable for the identification of that object. Here an object named Zen has been created assuming that a type of a car is Zen. This object called 'Zen' is an instance of the class car. A class can have any number of objects.

1.2.2 Data Abstraction Types (ADT)

Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. Consider a class Car, which is made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other. You try to obtain your own abstract view or model of the problem. This process of modeling is called abstraction and is illustrated in figure 5 below

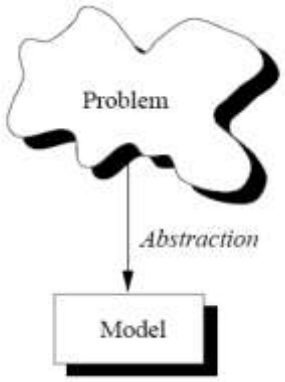


Figure 5

A procedure abstraction lets you ignore the implementation of a procedure, and focus only on the arguments and return values of the procedure. It lets you ignore the way data is represented in memory, and think in terms of what operations can be performed on the data

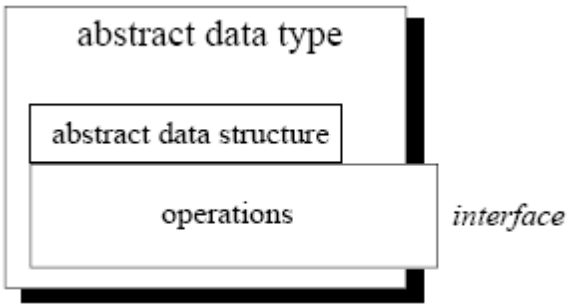


Figure 6

ADTs as shown in the figure 6 above allow the creation of instances with well defined properties and behaviors. In object orientation ADTs are referred to as classes. Therefore a class defines properties of objects which are the instances in an object oriented environment.

Object Oriented Programming and Procedural Oriented Programming differ in terms of the kind of abstractions we create. Procedural Oriented Programming is concerned with creating and using procedural abstractions which represent actions taken by the program (such as computing the sine of a number) Object Oriented Programming is concerned with data abstractions which represents the objects on which the procedures are acting (such as integers, strings, or windows)

1.2.3 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation. It is also commonly referred to as “Information Hiding”. Encapsulation helps to hide, inside the object, both the data fields and the methods that act on that data. Access to the data is controlled by this feature, forcing programs to retrieve or modify data only through the object's interface.

If a class allows another to modify its attributes or invoke its methods, the attributes and methods are said to be part of the class’ public interface. If a class doesn’t allow another to modify its attributes or invoke its methods, those are part of the class’ private implementation. For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

1.2.4 Inheritance

Inheritance describes the ability to create new classes based on an existing class. The new class inherits all the properties and methods and events of the base class, and can be customized with additional properties and methods. For example, you can create a new class named "Truck" based on the "Car" class. The "Truck" class inherits the "Color" property from the "Car" class and can have additional properties such as "FourWheelDrive." It is the process by which one object can acquire the properties of another object. This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Inheritance is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way. Just as a child inherits genes from its parent, a class can inherit attributes and behaviors from its parent. The parent class is common referred to as the super-class and the child class as the sub-class.

Inheritance enables us to create a class that is similar to a previously defined class, but one that still has some of its own properties. Create a new class by inheritance. This new class inherits all the data and methods from the tested base class. The level of inheritance can be controlled with the public, private, and protected keywords.

1.2.5 Polymorphism

The last major feature of object-oriented programming is polymorphism. By using polymorphism, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way. In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat. This same principle can also apply to programming. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, push() and pop() , that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack—the functions push() and pop() —are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data. Thus Polymorphism means that you can have multiple classes that

can be used interchangeably, even though each class implements the same properties or methods in different ways.

Polymorphism is essential to object-oriented programming because it allows you to use items with the same names, no matter what type of object is in use at the moment. For example, given a base class of "Car," polymorphism enables the programmer to define different "StartEngine" methods for any number of derived classes. The "StartEngine" method of a derived class named "DieselCar" may be completely different from the method with the same name in the base class. Other procedures or methods can use the "StartEngine" method of the derived classes in exactly the same way, no matter what type of "Car" object is being used at the time.

It helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the specific action The programmer, don't need to do this selection manually. You need only remember and utilize the general interface. The first object-oriented programming languages were interpreters, so polymorphism was, of course, supported at run time. However, C++ is a compiled language. Therefore, in C++, both run-time and compile-time polymorphism are supported.

1.3 A brief history of Object Oriented Programming

Although the concept that everything is an object has been around since the beginning of man, the concept of software objects didn't start to take shape until the early 1950s. There are, in fact, many definitions of an object, such as the one by Grady Booch [Booch, p77]: "An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable". OOs early growth was inhibited by a number of factors, which, among other reasons included: lack of technology, high cost of computer memory and hardware, and a great reluctance to change technologies, by management. "The term object was first formally applied in the SIMULA language, and objects typically existed in SIMULA programs to simulate some aspect of reality" [Booch, p77]. Kristen Nygaard is credited with the development of SIMULA, the first software application to use OO, while doing work in Operational Research in the 1950s and early 1960s at the Norwegian Computing Center, Oslo, Norway. The research that he was doing there created the need for precise tools so he could describe and simulate complex man-machine systems. SIMULA was originally designed and implemented as a language to simulate discrete events, but was later upgraded and re-released as a full scale programming language. Although SIMULA never became widely used, the features used by it were highly influential on modern programming methodology. Among other things, SIMULA introduced the object-oriented programming concepts like classes, objects, inheritance, encapsulation and polymorphism, as were described above.

In 1970's the term Object-oriented programming were introduced in the language called 'smalltalk' in order to represent the concepts of objects and messages. Smalltalk was created by Alan Kay at Xerox PARC. It is the first programming language to be called

'object-oriented'. C++ was created as a bridge between object-oriented programming and C, the world's most popular programming language for commercial software development. The goal was to provide object-oriented design to a fast, commercial software development platform. The concept of object oriented programming became popular in 1990's with the popularity of graphical user interfaces, event-driven programming and with the development of computer games. After that, a number of object oriented languages have emerged including C++, Java, as well as Visual Basic .NET and C# which were designed for Microsoft's .NET platform. The top four OO languages in use today include Smalltalk, Eiffel, C++ and Java.

1.4 Programming Paradigms

A programming paradigm is a fundamental style of computer programming. (Compare with a methodology, which is a style of solving specific software engineering problems). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints...) and the steps that compose a computation (assignation, evaluation, continuations, data flows...). A programming language can support multiple paradigms. For example programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements.

In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures. Just as different groups in software engineering advocate different methodologies, different programming languages advocate different programming paradigms. Some languages are designed to support one particular paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby and Oz).

With the advent of high-level languages, programming became accessible to more people; writing program code was no longer exclusively the domain of specially trained scientists. As a result, computing was used in increasingly complex roles. It was soon clear, however, that a more efficient way of programming was needed, one that would eliminate the obscure and complex "spaghetti code" that the early languages produced.

Programmers needed a new way of using high-level languages, one that enabled them to partition their programs into logical sections that represented the general tasks to be completed. Thus, the structured-programming paradigm was born. Structured programming encourages a top-down approach to programming, in which the programmer focuses on the general functions that a program must accomplish rather than the details

of how those functions are implemented. When programmers think and program in top-down fashion, they can more easily handle large projects without producing tangled code.

Today, the need for efficient programming methods is more important than ever. The size of the average computer program has grown dramatically and now consists of hundreds of thousands of code lines. With huge programs, reusability is critical. Again, a better way of programming is needed-and that better way is object-oriented programming.

Object oriented programming is a technique for programming – a paradigm for writing "good" programs for a set of problems. If the term "object oriented programming language" means anything, it must mean a programming language that provides mechanisms that support the object oriented style of programming well. There is an important distinction here. A language is said to *support* a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; it merely *enables* the technique to be used. For example, you can write structured programs in Fortran77 and object oriented programs in C, but it is unnecessarily hard to do so because these languages do not directly support those techniques.

Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the more subtle form of compile time and/or runtime checks against unintentional deviation from the paradigm. Type checking is the most obvious example of this; ambiguity detection and runtime checks are also used to extend linguistic support for paradigms. Extra linguistic facilities such as libraries and programming environments can provide further support for paradigms.

One language is not necessarily better than another because it possesses a feature the other does not. There are many examples to the contrary. The important issue is not so much what features a language possesses, but that the features it does possess are sufficient to support the desired programming styles in the desired application areas:

- All features must be cleanly and elegantly integrated into the language.
- It must be possible to use features in combination to achieve solutions that would otherwise require extra, separate features.
- There should be as few spurious and "special purpose" features as possible.
- A feature's implementation should not impose significant overheads on programs that do not require it.
- A user should need to know only about the subset of the language explicitly used to write a program.

The first principle is an appeal to aesthetics and logic. The next two are expressions of the ideal of minimalism. The last two can be summarized as "what you don't know won't hurt you." C++ was designed to support data abstraction, object oriented programming, and generic programming in addition to traditional C programming techniques under

these constraints. It was *not* meant to force one particular programming style upon all users. Each paradigm builds on its predecessors, each adds something new to the C++ programmer's toolbox, and each reflects a proven design approach. The presentation of language features is not exhaustive. The emphasis is on design approaches and ways of organizing programs rather than on language details. Object-oriented programming by itself is not adequate to easily solve all programming problems, and thus combination of various approaches (multiparadigm) programming languages is required.

1.5 Main Characteristics of Object Oriented Programming

- It lays more emphasis on data than on functions.
- Data and functions that operate on that data are unified together into a single unit called an object
- Programs are divided into classes and not functions.
- It lays more emphasis on what to do and not how to solve a given problem.
- The object oriented approach represents real world modeling.
- It follows the bottom-up approach in which we divide the problem to be solved into small and elementary building blocks known as object.
- Using this approach, complex user defined data types can be created.
- Modification and maintenance of a program is very easy as if you want to modify the data in an object then only functions of that object needs to be rewritten.
- It is easy to understand the structure and the working of the complex system.

1.6 Advantages and Disadvantages of Object Oriented Programming over Conventional programming

Advantages of Object Oriented Programming

- Procedural Oriented Programming is non real world modeling whereas Object Oriented Programming models the real world better because everything in the world is an object.
- Scope of reusability is higher in case of Object Oriented Programming. Object Oriented Programming makes programming faster and easier because of the reuse of existing, previously tested classes from a library. Reuse can also be accomplished by inheritance.
- Development Life Cycle - Object Oriented Programming makes faster development of programs because rapid prototyping of models is achieved due to the reuse of existing models of corporation's processes.
- Maintenance – In Procedural Oriented Programming as the code size grows, maintaining code becomes difficult as any major changes required for later functionality may result in a lot of changes in code. Object oriented programming makes maintenance easier because code only needs to be changed in one place. This is made possible by encapsulation.

- Quality - Object Oriented Programming makes testing easier and more reliable because the existing components are used in application development and these components are previously tested.
- Object Oriented Programming does automatic garbage-collection better.
- Unauthorized access to data - In case of Procedural Oriented Programming as data is global so unauthorized functions can manipulate this data accidentally and thus make it inconsistent where as in case of Object Oriented Programming's access to data is controlled by the use of encapsulation.
- Decoupling - It allows for the separation of object interactions from classes and inheritance into distinct layers of abstraction.

Disadvantages of Object Oriented Programming

- The run time overhead is more in case of applications built in Object Oriented Programming than using Procedural Oriented Programming.
- Object oriented Programming is domain centered and knowledge intensive

Summary

- The main programming techniques are procedural oriented programming and object oriented programming.
- In procedural oriented programming the main program is passes data to the individual procedures which process the data and return it to the main program and once the program has finished the resulting data is presented as the output.
- The concept of object-oriented programming (OOP) is a programming technique that uses the concept of 'objects' and their interactions between them.
- Classes are really nothing more than user-defined data types and objects instances of these data types.
- A Class is a collection of different types of variables, which can hold data and a set of related functions. The variables in a class are referred as member variables or data members. The functions of a class are referred to as member functions or methods of a class.
- Object is a particular instance of a class and classes are accessed using objects.
- The main concepts related to object oriented programming is data abstraction, encapsulation, polymorphism and inheritance.
- SIMULA was the first programming language to introduce the object-oriented programming concepts like classes, objects, inheritance, encapsulation and polymorphism. But the term Object-oriented programming was introduced in the language called 'smalltalk' in order to represent the concepts of objects and messages.
- C++ was created as a bridge between object-oriented programming and C.
- Programming paradigm is a fundamental style of computer programming. Object oriented programming language provides a mechanism that support the object oriented style of programming well.
- Object oriented programming lays more emphasis on data than on functions and represents real world modeling.

- Scope of reusability is higher in case of Object Oriented Programming.
- The run time overhead is more in case of applications built in Object Oriented Programming

Self Check Exercise

1. Define Procedural programming?
2. Describe how object oriented programming has evolved?
3. How is Object Oriented Programming better than procedural programming?
4. How does a class relate to an object?
5. What are the three major concepts used in Object Oriented Programming? Define each of these three concepts.
6. What is meant by Data Abstraction?

Suggested Readings

- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, "The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, "The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

Lesson No. 2 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

C++ Identifiers, Keywords, Constants & Variables

Objectives

After reading this lesson you should be able to:

- Understand the concept of Identifiers, Keywords, Constants and Variables.
- Understand the different data types and how to use them.
- Understand the variable scope, declaration and initialization.

Introduction

Tokens are the various c++ program elements which are identified by the compiler. The tokens supported by c++ include keywords, variables, constant, special characters, operators etc. The detail of each one is given below:

2.1 Identifiers

Identifiers are names given to various programming elements such as variables, functions, symbolic constants, arrays, classes etc. Identifiers are a means of referral both for program designers and for the compiler.

Example:

```
int count
char name[10][20]
```

In the above example count and name are the identifiers

2.1.1 Naming Rules for Identifiers

- The identifiers must indicate to some degree the purpose of the item being labeled by the identifier.
- A name (identifier) consists of a sequence of letters and digits.
- The first character must be a letter.
- The underscore character '_' is considered a letter.
- C++ imposes no limit on the number of characters in a name. Although for some compilers only the 32 first characters of an identifier are significant
- Names starting with an underscore are reserved for special facilities in the implementation and the runtime environment, so such names should not be used in application programs.
- The C++ language is "case sensitive", that means that a same identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Uppercase and lowercase letters are distinct, so *C o u n t* and *c o u n t* are different names. But in general, it is best to avoid names that differ only in slightly.
- C++ and compiler specific keywords cannot be used as I*dentifiers and also* should not have the same name as functions that are in the C or C++ library.

CORRECT	INCORRECT
Result1 A_size _fun_average	1result A-size !fun_average

2.2 Keywords

Keywords are reserved words that have standard predefined meaning in c++. The keywords cannot be used as variable names. The keywords are always written in lowercase. The keywords are shown in the table below:

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

Additionally, some reserve words are there to represent some of the operators alternatively under certain circumstances. They are

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Also there are some keywords specific to the compilers

There are some keywords in C++ but not in C, such keywords are shown in the tables below:

C++ Keywords That Are Not C Keywords					
and	*and_eq*	*asm*	*bitand*	*bitor*	*bool*
catch	*class*	*compl*	*const_cast*	*delete*	*dynamic_cast*
explicit	*export*	*false*	*friend*	*inline*	*mutable*
namespace	*new*	*not*	*not_eq*	*operator*	*or*
or_eq	*private*	*protected*	*public*	*reinterpret_cast*	*static_cast*
template	*this*	*throw*	*true*	*try*	*typeid*
typename	*using*	*virtual*	*wchar_t*	*xor*	*xor_eq*

C++ Keywords That Are C Macros							
and	*and_eq*	*bitand*	*bitor*	*bool*	*compl*	*false*	
not	*not_eq*	*or*	*or_eq*	*true*	*wchar_t*	*xor*	*xor_eq*

2.3 Constants (Literals)

A constant is a value that does not change during the program execution. They are also called literals. Constants can be of any of the basic data types and are classified as integer constants, floating point constants, character constants and string constants. Constants are used to give concrete values to variables or to express messages which are printed by our programs, for example, when we wrote:

```
int Count;
Count=5;
```

The 5 in this piece of code is a integer constant.

2.3.1 Integer Constants

Integer constants are specified as numbers without fractional components. For example, 1776, 707, -273 are integer constants. There are also several variations of the basic types like int, long, unsigned long unsigned int etc that you can generate using the type modifiers.

2.3.2 Floating-point Constants

Floating-point constants require the decimal point followed by the number's fractional component. For example, 11.123 is a floating-point constant. C/C++ also allows you to use scientific notation for floating-point numbers as shown in the following example

```
6.02e23   // 6.02 x 10^23
1.6e-19   // 1.6 x 10^-19
```

There are two floating-point types: float and double. The default type for floating point literals is double.

2.3.3 Octal and Hexadecimal Constants

It is sometimes easier to use a number system based on 8 or 16 rather than 10 (standard decimal system). The number system based on 8 is called octal and uses the digits 0 through 7. In octal, the number 10 is the same as 8 in decimal. The base 16 number system is called hexadecimal and uses the digits 0 through 9 plus the letters A through F, which stand for 10, 11, 12, 13, 14, and 15, respectively. C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75      // decimal
0113    // octal
0x4b    // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

2.3.4 Character and string Constants

There also exist non-numerical constants, like chars and strings. Character constants are enclosed between single quotes and a string which is a set of characters enclosed in double quotes. For example

'z'
'p'
"Hello world"

'z' and 'p' are both character constants and "Hello world" and "how do you do?" are both string constants. Both C and C++ define wide characters which are 16 bits long. To specify a wide character constant c++ uses a built-in type called wchar_t. When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Consider the difference between these two expressions: x, 'x', x alone would refer to a variable whose identifier is x, whereas 'x' (enclosed within single quotation marks) would refer to the character constant 'x'.

C/C++ include the special backslash character constants shown in Table below. These are also referred to as escape sequences. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

\n	Newline: Go to the next line
\r	carriage return
\t	Tab: Advance to the next tab stop (eight-column boundary)
\v	vertical tab
\b	Backspace: Move the cursor to the left one character
\f	form feed (page feed) Go to top of a new page
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	Question mark (?)
\\	backslash (\)

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A). String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line. You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"this forms" "a single" "string" "of characters"
```

2.3.5 Boolean Constants

There are only two valid Boolean values: true and false. These can be expressed in C++ as values of data type bool by using the Boolean literals true and false.

2.3.6 Defined constants (#define)

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

```
For example:
#define PI 3.14159265
#define TAB '\t'
```

This defines two new constants: PI and TAB. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, for example:

```
r=7;
circle = 2 * PI * r;
```

In fact the only thing that the compiler preprocessor does when it encounters #define directives is to literally replace any occurrence of their identifier (in the previous example, these were PI and TAB) by the code to which they have been defined (3.14159265 and '\t' respectively). The #define directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (;) at its end. If you append a semicolon character (;) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

2.3.7 Declared constants (const)

With the const prefix you can declare constants with a specific type in the same way as you would do with a variable:

2.4. Variables

A variable is a symbolic named location in memory that is used to hold a value that may be used by the program. The value of the variable may change during the program execution but at a given instance only one value can be stored in it. All variables must be declared before they can be used. Every variable must have a name and data type associated with it. The general form of a declaration is

```
data type list of  variable;
```

2.4.1 Scope of Variables

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block. Another type of variables is declared in the definition of function parameters and is called formal parameters. These variables are local in scope. The local and global variables are shown in the following figure.

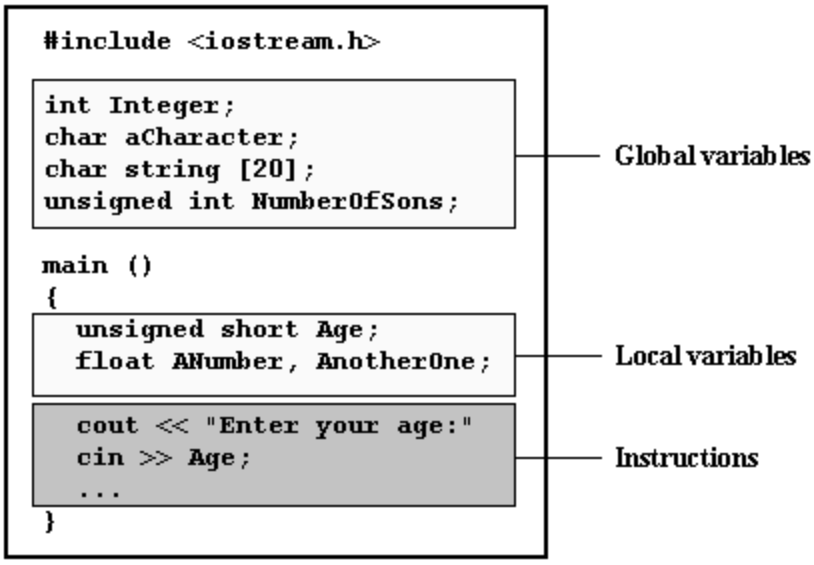


Figure 1

The variable can be put into three different categories as given below:

- **Local Variables**

  Variables that are declared inside a function are called local variables. Local variables may be referenced only by statements that are inside the block in which the variables are declared. A block of code begins with an opening curly brace and terminates with a closing curly brace. Local variables exist only while the block of code in which they are declared is executing. That is, a local variable is created upon entry into its block and destroyed upon exit.

  For example, consider the following two functions:

```
void func1(void)
{
int x;
x = 10;
}
void func2(void)
{
int x;
x = -199;
}
```

  The integer variable x is declared twice, once in func1 () and once in func2 (). The x in func1 () has no bearing on or relationship to the x in func2 (). This is because each x is only known to the code within the same block as the variable declaration.

- **Formal Parameters**
  If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the formal parameters of the function. They behave like any other local variables inside the function. As shown in the following program fragment, their declarations occur after the function name and inside parentheses:

```
/* Return 1 if c is part of string s; 0 otherwise */
int is_in(char *s, char c)
{
while(*s)
    if(*s==c) return 1;
    else s++;
return 0;
}
```

  The function is_in() has two parameters: s and c. This function returns 1 if the character specified in c is contained within the string s; 0 if it is not. Then you may use them inside the function as normal local variables. As local variables, they are also dynamic and are destroyed upon exit from the function.

- **Global Variables**
  Unlike local variables, global variables are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution. You create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in. In the following program, the variable count has been declared outside of all functions. Although its declaration occurs before the main() function, you could have placed it anywhere before its first use as long as it was not in a function. However, it is usually best to declare global variables at the top of the program.

```
#include <stdio.h>
int count; /* count is global */
void func1(void);
```

2.4.2 Fundamental Data Types

When programming, we store the variables in our computer's memory, but the computer has to know what we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In

addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.
Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one in the following table

Name	Description	Size*	Range*
Char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
Float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
Double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2bytes	1 wide character

Table 1

The values of the columns Size and Range depend on the architecture of the system where the program is compiled and executed. The values shown above are those found on most 32bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one *word*) and the four integer types char, short, int and long must each one be at least as large as the one preceding it. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

2.4.3 Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float) followed by a valid variable identifier. For example:

```
int value;
float f_number;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier value. The second one declares a variable of type float with the identifier f_number. Once declared, the variables value and f_number can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int x, y, z;
```

This declares three variables (x, y and z), all of them of type int.

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned short int num_count;
signed int MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword signed)

An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable. short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

The following program shows the use of variables:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

2.4.4 Initialization of variables

When declaring a variable, its value is by default undetermined. But you may want a variable to store a concrete value at the same moment that it is declared. In order to do that, you can initialize the variable. There are two ways to do this in C++. The first one, known as c-like, is done by appending an equal sign followed by the value to which the variable will be initialized:

```
type identifier = initial_value ;
```

For example, if we want to declare an int variable called a initialized with a value of 0 at the moment in which it is declared, we could write: int a = 0; The other way to initialize variables, known as constructor initialization, is done by enclosing the initial value between parentheses (()):

```
type identifier (initial_value) ;
```

Both ways of initializing variables are valid and equivalent in C++. The example program is given below:

```
// initialization of variables

#include <iostream>
using namespace std;

int main ()
{
  int a=5;               // initial value = 5
  int b[2];              // initial value = 2
  int result;            // initial value undetermined

  a = a + 3;
  result = a - b;
  cout << result;

  return 0;
}
```

Summary

- The tokens supported by c++ include keywords, variables, constant, special characters, operators.
- Identifiers are names given to various programming elements such as variables, functions, symbolic constants, arrays, classes etc.
- Keywords are reserved words that have standard predefined meaning in c++.
- A constant is a value that does not change during the program execution. They are also called literals.
- A variable is a symbolic named location in memory that is used to hold a value that may be used by the program.
- When programming, we store the variables in our computer's memory, but the computer has to know the data type of the variable which we want to store to allocate the required space.
- In order to use a variable in C++, we must first declare it specifying which data type we want it to be.
- When a variable is declared its value is by default undetermined.
- We need to initialize the variable before using it.

Self Check Exercise

1. What are the rules to be followed for naming the various programming elements in C++?
2. What is meant by literals?
3. Which are the broad categories of variable? How are they different from each other?
4. What is the difference between declaring and inializing a variable? What will happen if we don't inialize a variable?
5. How many different data types are there in C++ ? Explain in brief.
6. Which are the data types which are there in C++ but not in C?

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co., 2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

Lesson No. 3 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

C++ Operators

Objectives

After reading this lesson you should be able to:

- Understand the concept of assignment operator, compound assignment operator and type conversion in assignment operators.
- Understand the concept of arithmetic operators and their rules of precedence
- Understand the Relational, Logical and Bitwise Operators and also their rules of precedence.

Introduction

Operators are special type of function (some of which can be overloaded in C++. An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary function calls, but the effect is the same.

C/C++ is very rich in built-in operators. There are four main classes of operators: arithmetic, relational, logical, and bitwise. In addition, there are some special operators for

particular tasks. Once we know of the existence of variables and constants, we can begin to operate with them. For that purpose, C++ integrates operators. Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs that are not part of the alphabet but are available in all keyboards. This makes C++ code shorter and more international, since it relies less on English words, but requires a little of learning effort in the beginning.

This section covers the basic operators in C and C++. All operators produce a value from their operands. This value is produced without modifying the operands, except with the assignment, increment, and decrement operators.

3.1 The Assignment Operator (=)

You can use the assignment operator within any valid expression. Where an expression may be as simple as a single constant or as complex as you require. C/C++ uses a single equal sign to indicate assignment. The assignation operator assigns a value to a variable.

```
a = 5;
```

This statement assigns the integer value 5 to the variable a. The part at the left of the assignation operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these.

```
a = b;
```

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost. Consider also that we are only assigning the value of b to a at the moment of the assignation. Therefore a later change of b will not affect the new value of a. The most important rule of assignation is the *right-to-left* rule: The assignation operation always takes place from right to left, and never the other way
For example, let us have a look at the following code:

```
// assignation operator

#include <iostream>
using namespace std;

int main ()
{
  int a, b;       // a:?, b:?
  a = 10;         // a:10, b:?
  b = 4;          // a:10, b:4
  a = b;          // a:4, b:4
  b = 7;          // a:4, b:7

  cout << "a:";
  cout << a;
  cout << " b:";
  cout << b;

  return 0;
}
```

This code will give us as result that the value contained in a is 4 and the one contained in b is 7. Notice how a was not affected by the final modification of b, even though we declared a = b earlier (that is because of the *right-to-left rule*). A property that C++ has over other programming languages is that the assignation operation can be used as the rvalue (or part of an rvalue) for another assignation. For example:

```
a = 2 + (b = 5);
```

is equivalent to:

```
b = 5;
a = 2 + b;
```

that means: first assign 5 to variable b and then assign to a the value 2 plus the result of the previous assignation of b (i.e. 5), leaving a with a final value of 7. The following expression is also valid in C++:

```
a = b = c = 5;
```

It assigns 5 to the all the three variables: a, b and c.

3.2 Arithmetic operators

3.2.1 Arithmetic operators (+, -, *, /, %)

The basic mathematical operators are the same as the ones available in most programming languages: addition (**+**), subtraction (**-**), division (**/**), multiplication (*****), and modulus (**%**) this produces the remainder from integer division. Integer division truncates the result (it doesn't round). The modulus operator cannot be used with floating-point numbers.

C and C++ also use a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign,

and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable x and assign x to the result, you say: x += 4;.

Thus there are five arithmetical operations supported by the C++ language are:

+	addition
-	subtraction
*	multiplication
/	division
%	modulo

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to seeing may be *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values.

When you apply / to an integer or character, any remainder will be truncated. For example, 5/2 will equal 2 in integer division. The modulus operator **%** also works in C/C++ as it does in other languages, yielding the remainder of an integer division. However, you cannot use it on floating-point types. The following code fragment illustrates the working of modulus (**%**):

```
int x, y;
x = 5;
y = 2;
printf("%d ", x/y); /* will display 2 */
printf("%d ", x%y); /* will display 1, the remainder of the integer division */
x = 1;
y = 2;
printf("%d %d", x/y, x%y); /* will display 0 1 */
```

The last line prints a 0 and a 1 because 1/2 in integer division is 0 with a remainder of 1. The unary minus multiplies its operand by –1. That is, any number preceded by a minus sign switches its sign.

3.2.2 Increment and Decrement

C/C++ includes two useful operators not generally found in other computer languages. These are the increment and decrement operators, **++** and −−. The operator **++** adds 1 to its operand, and −−subtracts one. In other words:

x = x+1; is the same as ++x; and x = x-1; is the same as - -x;. Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand. For example, x = x+1; can be written ++x; or x++; . There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement

operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, instance,

```
x = 10;
y = ++x;
```

sets y to 11. However, if you write the code as

```
x = 10;
y = x++;
```

y is set to 10. Either way, x is set to 11; the difference is in when it happens. Most C/C++ compilers produce very fast, efficient object code for increment and decrement operations. For this reason, you should use the increment and decrement operators when you can instead of assignments like x=x+1.

3.2.3 Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence, which are generally the same as those followed in algebra:

- Operators in expressions contained within pairs of parentheses are evaluated first. Thus, parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer. Parentheses are said to be at the "highest level of precedence." In cases of nested, parentheses such as ((a+b)+c) the operators in the inner most pair of parentheses are applied first.
- Multiplication, division and modulus operations are applied next. If an expression contains several multiplications, divisions and modulus operations, operators are applied from left to right. Multiplication, division and modulus are said to be on the same level of precedence.
- Addition and subtraction operations are applied last. If an expression contains several additions and subtraction operations, operators are applied from left to right. Addition and subtraction also have the same level of precedence.

The set of rules of operator precedence defines the order in which C++ applies operators. When we say that certain operators are applied from left to right, we are referring to the associativity of the operators. For example, in the expression

```
a+b+c
```

the addition operators (+) associate from left to right, so a+b is calculated first, then c is added to the sum to determine the value of the whole expression. The precedence of the arithmetic operators is shown in the table below

highest ↓ lowest	++ – – – (unary minus) * / % + –

Operators on the same level of precedence are evaluated by the compiler from left to right. Of course, you can use parentheses to alter the order of evaluation. C/C++ treats parentheses in the same way as virtually all other computer languages. Parentheses force an operation, or set of operations, to have a higher level of precedence.

3.3 Relational and Logical Operators

In the term relational operator, relational refers to the relationships that values can have with one another. In the term logical operator, logical refers to the ways these relationships can be connected. Because the relational and logical operators often work together, they are discussed together here. The idea of true and false underlies the concepts of relational and logical operators. In C, true is any value other than zero. False is zero. Expressions that use relational or logical operators return 0 for false and 1 for true. C++ fully supports the zero/non-zero concept of true and false. However, it also defines the bool data type and the Boolean constants true and false. In C++, a 0 value is automatically converted into false, and a non-zero value is automatically converted into true. The reverse also applies: true converts to 1 and false converts to 0. The outcome of a relational or logical operation is true or false. But since this automatically converts into 1 or 0, the distinction between C and C++ on this issue is mostly academic.

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

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Example of Relational operators

```
(7 == 5)   // evaluates to false.
(5 > 4)    // evaluates to true.
(3 != 2)   // evaluates to true.
(6 >= 6)   // evaluates to true.
(5 < 5)    // evaluates to false.
```

The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignation operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

3.3.2 Logical operators (!, &&, ||)

The Operator ! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand. For example:

```
!(5 == 5)   // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4)   // evaluates to true because (6 <= 4) would be false.
!true       // evaluates to false
!false      // evaluates to true.
```

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

|| OPERATOR

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

For example:

```
( (5 == 5) && (3 > 6) )  // evaluates to false ( true && false ).
( (5 == 5) || (3 > 6) )  // evaluates to true ( true || false ).
```

- Both the relational and logical operators are lower in precedence than the arithmetic operators. That is, an expression like 10 > 1+12 is evaluated as if it were written 10 > (1+12). Of course, the result is false.
- You can combine several operations together into one expression, as shown here: 10>5 && !(10<9) || 3<=4
- Although neither C nor C++ contain an exclusive OR (XOR) logical operator, you can easily create a function that performs this task using the other logical operators. The outcome of an XOR operation is true if and only if one operand (but not both) is true. The following program contains the function xor() , which returns the outcome of an exclusive OR operation performed on its two arguments:

```
#include <stdio.h>
int xor(int a, int b);
int main(void)
{
printf("%d", xor(1, 0));
printf("%d", xor(1, 1));
printf("%d", xor(0, 1));
printf("%d", xor(0, 0));
return 0;
}
/* Perform a logical XOR operation using the two arguments. */
int xor(int a, int b)
{
return (a || b) && !(a && b);
}
```

The following table shows the relative precedence of the relational and logical operators:

Highest ⬇ **Lowest**	!
	> >= < <=
	== !=
	&&
	||

As with arithmetic expressions, you can use parentheses to alter the natural order of evaluation in a relational and/or logical expression.

- Remember, all relational and logical expressions produce either a true or false result. Therefore, the following program fragment is not only correct, but will print the number 1.

```
int x;
x = 100;
printf("%d", x>10);
```

3.4 Bitwise Operators

3.4.1 Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store. Unlike many other languages, C/C++ supports a full complement of bitwise operators. Since C was designed to take the place of assembly language for most programming tasks, it needed to be able to support many operations that can be done in assembler, including operations on bits. Bitwise operation refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the char and int data types and variants. You cannot use bitwise operations on float, double, long double, void, bool, or other, more complex types. Table below lists the operators that apply to bitwise operations. These operations are applied to the individual bits of the operands.

operator	description
&	Bitwise AND
|	Bitwise Inclusive OR
^ ~	Bitwise Exclusive OR Unary complement (bit inversion)
<<	Shift Left
>>	Shift Right

The bitwise AND, OR, and NOT (one's complement) are governed by the same truth table as their logical equivalents, except that they work bit by bit. The exclusive OR has the truth table shown here:

p	q	p ^q
0	0	0
1	0	1
1	1	0
0	1	1

As the table indicates, the outcome of an XOR is true only if exactly one of the operands is true; otherwise, it is false.

Bitwise operations most often find application in device drivers—such as modem programs, disk file routines, and printer routines —because the bitwise operations can be used to mask off certain bits, such as parity. (The parity bit confirms that the rest of the bits in the byte are unchanged. It is usually the high-order bit in each byte.)

Think of the bitwise AND as a way to clear a bit. That is, any bit that is 0 in either operand causes the corresponding bit in the outcome to be set to 0. For example, the following function reads a character from the modem port and resets the parity bit to 0:

```
char get_char_from_modem(void)
{
char ch;
ch = read_modem(); /* get a character from the
modem port */
return(ch & 127);
}
```

- Remember, relational and logical operators always produce a result that is either true or false, whereas the similar bitwise operations may produce any arbitrary value in accordance with the specific operation. In other words, bitwise operations may produce values other than 0 or 1, while logical operators will always evaluate to 0 or 1.

3.4.2 Bit-Shift Operators(>>, <<)

The bit-shift operators, >> and <<, move all bits in a variable to the right or left as specified.

- The general form of the shift-right statement is *variable* >> *number of bit positions*
- The general form of the shift-left statement is *variable* << *number of bit positions*

As bits are shifted off one end, 0's are brought in the other end. (In the case of a signed, negative integer, a right shift will cause a 1 to be brought in so that the sign bit is preserved.) Remember, a shift is not a rotate. That is, the bits shifted off one end do not come back around to the other. The bits shifted off are lost.

Bit-shift operations can be very useful when you are decoding input from an external device, like a D/A converter, and reading status information. The bitwise shift operators can also quickly multiply and divide integers. A shift right effectively divides a number by 2 and a shift left multiplies it by 2. The following program illustrates the shift operators:

```
/* A bit shift example. */
#include <stdio.h>
int main(void)
{
unsigned int i;
int j;
i = 1;

/* left shifts */

for(j=0; j<4; j++) {
i = i << 1;       /* left shift i by 1, which is same as a multiply by 2 */
printf("Left shift %d: %d\n", j, i);
}

/* right shifts */

for(j=0; j<4; j++) {
i = i >> 1; /* right shift i by 1, which is same as a division by 2 */
printf("Right shift %d: %d\n", j, i);
}
return 0;
}
```

3.5 Compound assignation (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignation operators:

expression	is equivalent to
value += increase;	value = value + increase;
a -= 5;	a = a - 5;
a /= b;	a = a / b;
Price *= units + 1;	price = price * (units + 1);

and the same for all other operators. The example is shown below

```
// compund assignation

#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;          // equivalent to a=a+2
  cout << a;
  return 0;
}
```

Summary

- An operator takes one or more arguments and produces a new value.
- C++ has four main classes of operators: arithmetic, relational, logical, bitwise and additionally there are some special operators..
- Unlike other languages whose operators are mainly keywords, operators in C++ are mostly made of signs.
- The assignment operator within any valid expression. C/C++ uses a single equal sign to indicate assignment.
- There are five arithmetical operations supported by the C++ language (+, -, *, /, %)
- C/C++ includes two useful operators not generally found in other computer languages. These are the increment and decrement operators, ++ and −−.
- C++ applies the operators in arithmetic expressions in a precise sequence determined by the rules of operator precedence, which are generally the same as those followed in algebra. Operators on the same level of precedence are evaluated by the compiler from left to right. Also parentheses can be used to alter the order of evaluation.
- In order to evaluate a comparison between two expressions we can use the relational and equality operators. (==, !, >, <, >=, <=)
- C++ supports three logical operations & (AND), ||(OR) and ! (NOT).
- Both the relational and logical operators are lower in precedence than the arithmetic operators.
- relational and logical operators always produce a result that is either true or false.
- There are six bitwise operators. (&, |, ^, ~, <<, >>). These operators modify variables considering the bit patterns that represent the values they store.

Self Check Exercise

1. What are the precedence rules for arithmetic operators?
2. How many relational operators are there, explain each with an example?
3. What is the difference between Logical and bitwise operators?
4. How type conversion takes place in assignments?
5. What is the function of % operator?

Suggested Readings

- Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications,1994.
- E. Balaguruswamy, “Object Oriented Programming with C++”, Tata McGraw-Hill,2001. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Deitel and Deitel, “ C++ How to program” ,Pearson Education, 2001.

- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

Lesson No. 4 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

Special Operators, Operator Precedence and Associativity

Objectives

After reading this lesson you should be able to:

- Understand special operators and their use
- Understand the operator precedence for expressions with several different operators.
- Understand the associatively of Operators

Introduction

In addition to the four basic categories of operators (Arithmatic, Relational, Logical and Bitwise operators) there are a number of special operators for performing specific task in C++, some of them are common with C and some specific to C++ like point-to-member, new & free, scope resolution operator etc. Some of these special operators are discussed in next sections (4.1, 4.2 & 4.3). This lesson also discusses precedence and associativity of operators. Some of the special operators work independently and do not have any priority in relation to other operators. Such operators are not shown in the priority table in the operator precedence section (4.4) are not shown in the

4.1 Explicit type casting operator

Type casting operators allow you to convert a data of a given type to another type. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):

```
int i;
float f = 3.14;
i = (int) f;
```

The previous code converts the float number 3.14 to an integer value 3, the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int ( f );
```

Both ways of type casting are valid in C++.

4.2 Size of Operator (sizeof())

Size of is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes. For example, assuming that integers are 4 bytes and doubles are 8 bytes,

```
double f;
printf("%d ", sizeof f);
printf("%d", sizeof(int));
```

The code above display 8 4.

Remember, to compute the size of a type, you must enclose the type name in parentheses, this is not necessary for variable names. C/C++ defines (using typedef) a special type called size_t, which corresponds loosely to an unsigned integer. The value returned by sizeof is of type size_t. For all practical purposes, however, you can think of it (and use it) as if it were an unsigned integer value.

size of primarily helps to generate portable code that depends upon the size of the built-in data types. For example, imagine a database program that needs to store six integer values per record. If you want to port the database program to a variety of computers, you must not assume the size of an integer, but must determine its actual length using sizeof. This being the case, you could use the following routine to write a record to a disk file:

```
/* Write 6 integers to a disk file. */
void put_rec(int rec[6], FILE *fp)
{
int len;
len = fwrite(rec, sizeof(int)*6, 1, fp);
if(len != 1) printf("Write Error");
}
```

Coded as shown, put_rec() compiles and runs correctly in any environment, including those that use 16- and 32-bit integers. One final point: sizeof is evaluated at compile time, and the value it produces is treated as a constant within your program. This

operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:

```
a = sizeof (char);
```

This will assign the value 1 to a because char is a one-byte long type. The value returned by sizeof is a constant, so it is always determined before program execution.

- **Need for sizeof :** In many programs, there are situations where it is useful to know the size of a particular datatype (one of the most common examples is dynamic memory allocation using the library function malloc). Though for any given implementation of C or C++ the size of a particular datatype is constant, the sizes of even primitive types in C and C++ are implementation defined (that is, not precisely defined by the standard). This can cause problems when trying to allocate a block of memory of the appropriate size. For example, say a programmer wants to allocate a block of memory big enough to hold ten variables of type int. Because our hypothetical programmer doesn't know the exact size of type int, (s)he doesn't know how many bytes to ask malloc for. Therefore, it is necessary to use the operator sizeof

4.3 The [ ] and ( ) Operators

Parentheses are operators that increase the precedence of the operations inside them. For example consider the following two expressions

```
a+b*c
```

```
(a+b)*c
```

In the first expression first b is multiplied by c and the added to a as * has higher priority than +. But the precedence of operators is changed in the second expression by using parentheses in this first ais added to b and than multiplied by c.

Square brackets perform array indexing. Given an array, the expression within square brackets provides an index into that array. For example,

```
#include <stdio.h>
char s[80];
int main(void)
{
    s[3] = 'X';
    printf("%c", s[3]);
    return 0;
}
```

This code first assigns the value 'X' to the fourth value of an array s, and then prints that element.

4.4 Operator precedence

Operator precedence defines the order in which an expression evaluates when several different operators are present. Except few mostly all operators in C++ are ranked according to their precedence. And there are around 50 operators in C++, and these operators can affect the evaluation of an expression in an unexpected ways if we are not careful in taking into consideration their precedence. C and C++ have specific rules to determine the order of evaluation. When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

```
a = 5 + 7 % 2
```

we may doubt if it really means:

```
a = 5 + (7 % 2)    // with a result of 6, or
a = (5 + 7) % 2    // with a result of 0
```

The correct answer is the first of the two expressions, with a result of 6. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference come from mathematics) but for mostly all the operators which can appear in C++. From greatest to lowest priority, the priority order is as given in the table below:

Table 1

S.No	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	.* ->*	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right

S.No	Operator	Description	Grouping
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13	|	bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15	||	logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^= !=	assignment	Right-to-left
18	,	comma	Left-to-right

All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses signs (and). If you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also make the code easier to read.

4.5 Associativity of Operators

Associativity defines the precedence order in which operators are evaluated in the case that there are several operators at the same level in an expression. Thus when an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators. Associativity can be of two types

- Left to Right
- Right to Left

4.5.1 Left to Right associativity

Left to Right associativity means that the left operand must be unambiguous that means that it must not be involved in evaluation of any other sub-expression. Let us understand this with an example.

Consider the expression

a = 3 / 2 * 5 ;

Here there is a tie between operators of same priority, that is between / and *. This tie is settled using the associativity of / and *. But both enjoy Left to Right associativity. Figure below shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
/	3	2 or 2 * 5	Left operand is unambiguous, Right is not
*	3 / 2 or 2	5	Right operand is unambiguous, Left is not

Since both / and * have L to R associativity and only / has unambiguous left operand (necessary condition for L to R associativity) it is performed earlier.
Consider yet another expression

z = a * b + c / d ;

Here * and / enjoys same priority and same associativity (Left to Right). Figure given below shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
*	a	b	Both operands are unambiguous
/	c	d	Both operands are unambiguous

Here since left operands for both operators are unambiguous Compiler is free to perform ***** or **/** operation as per its convenience since no matter which is performed earlier the result would be same.

4.5.2 Right to Left associativity

Right to Left associativity means that the right operand must be unambiguous that means that it must not be involved in evaluation of any other sub-expression. Consider the following expression

a = b = 3;

Here both assignment operators have the same priority and same associativity (Right to Left). Figure given below shows for each operator which operand is unambiguous and which is not.

Operator	Left	Right	Remark
=	a	b or b = 3	Left operand is unambiguous, Right is not
=	b or a = b	3	Right operand is unambiguous, Left is not

Since both = have R to L associativity and only the second = has unambiguous right operand (necessary condition for R to L associativity) the second = is performed earlier.

Most of the operators have left – to – right priority as is clear from table 1 given in section 4.4.

Summary

- In C++ there are a number of special operators for doing a particular task.
- Type casting operators allows converting a data of a given type to another type.
- sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes.
- Parentheses are operators that increase the precedence of the operations inside them. Square brackets perform array indexing.
- Operator precedence defines the order in which an expression evaluates when several different operators are present.
- All these precedence levels for operators can be manipulated or become more legible by removing possible ambiguities using parentheses.
- Associativity defines the precedence order in which operators are evaluated in the case that there are several operators at the same level in an expression.

Self Check Exercise

1. How is sizeof operator useful?
2. Determine the hierarchy of operations and evaluate the following expression:

```
i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8
```

3. Is Parentheses an operator in C++, if yes what is its significance.
4. Explain the associativity of operators.
5. What is meant by left – to right associativity? Explain with an example.
6. List the operators which have right – to – left associativity.
7. What is meant by explicit type conversion?

Suggested Readings

- Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications,1994.
- E. Balaguruswamy, “Object Oriented Programming with C++”, Tata McGraw-Hill,2001. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Deitel and Deitel, “ C++ How to program” ,Pearson Education, 2001.
- Bjarne Strautrup, “ The C++ Programming Language”, Addison-Wesley Publication Co.,2001.
- Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

Lesson No. 5 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

Type Conversion

Objectives

After reading this lesson you should be able to:

- Understand the concept of type conversion/casting in a better way.
- Understand the two categories of type conversion (implicit and explicit type casting) clearly.
- Understand what is meant by expressions in C++.
- Understand how type casting takes place in expressions.

Introduction

5.1 Type Conversion

Type conversion is conversion of data from one type to another type. While developing applications we have to be very careful about type conversion else there will be a lot of logical errors which will be hard to detect and the application will not work as expected. There are two main types of type conversions

- Implicit conversion
- Explicit conversion

5.1.1 Implicit conversion

Implicit type conversion is done automatically by the compiler whenever data from different types is intermixed. When a value from one type is assigned to another type, the compiler implicitly converts the value into a value of the new type. Implicit conversions do not require any operator.

```
double a=3;

int b=3.14;
```

Here, the value 3 is promoted to a double value and then assigned to a and we have not had to specify any type-casting operator. In the second example, the fractional part of the double value is dropped because integers cannot support fractional values and only 3 is saved in b. Because converting a double to an int usually causes data loss (making it unsafe), compilers such as Visual Studio Express 2005 will typically issue a warning. These kind of conversions are known as a standard conversions. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};

class B { public: B (A a) {} };

A a;

B b=a;
```

Here, a implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

5.1.2 Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. Explicit conversion can be done using type cast operator and the general syntax for doing this is

```
datatype (expression);
```

Here in the above datatype is the type which the programmer wants the expression to gets changed to consider the following example

```
float L_value, R_value;
```

int (L_value+R_value);

In the above example the output of addition is converted to integer from float. In C++ there are two notations used for explicit type conversion:

- C-style casting
- C++-style casting

The C-style casting takes the synatx as

(type) expression

This can also be used in C++. But apart from the above the other form of type casting that can be used specifically in C++ programming language namely C++-style casting is as below namely:

type (expression)

This approach was adopted since it provided more clarity to the C++ programmers rather than the C-style casting. Consider the following expression as per C-style casting

(type) firstVariable * secondVariable

In the above expression it is not clear whether we want to type cast only the first variable or the total expression but when a programmer uses the C++ style casting it is much more clearer as below

type (firstVariable) * secondVariable

The following expression gives one example each of C-style casting and C++ style casting respectively:

```
short a=2000;

int b;

b = (int) a;    // c-like cast notation

b = int (a);    // functional notation
```

The same will be clearer with the help of following example:

```
#include <iostream.h>
void main()
{
int a;
float b,c;
cout<< "Enter the value of a:";
cin>>a;
cout<< "\n Enter the value of b:";
cin>>b;
c = float(a)+b;
cout<<"\n The value of c is:"<<c;
}
```

The output of the above program is

Enter the value of a: 10

Enter the value of b: 12.5

The value of c is: 22.5

Another way to classify type conversion is

5.2 Type Conversion in Assignments

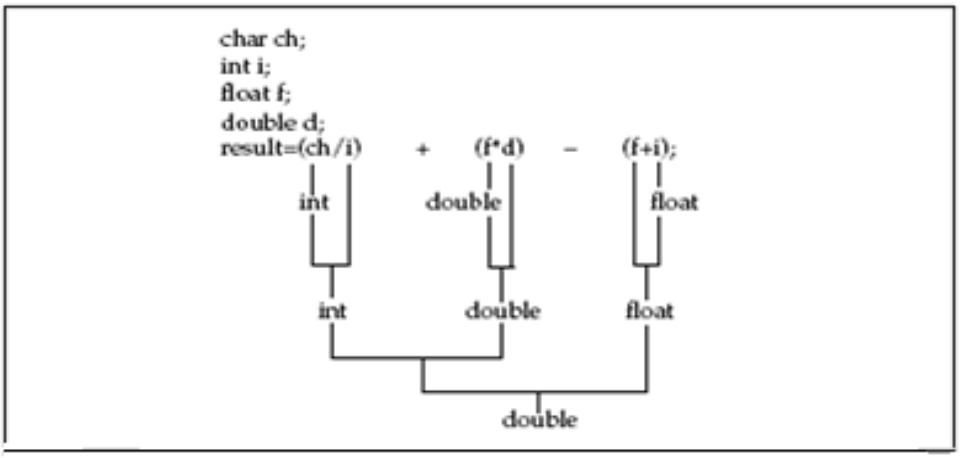
When variables of one type are mixed with variables of another type, a type conversion will occur. In an assignment statement, the type conversion rule is easy: The value of the right side (expression side) of the assignment is converted to the type of the left side (target variable), as illustrated here:

```
int x;
char ch;
float f;
void func(void)
  {
    ch = x; /* line 1 */
    x = f; /* line 2 */
    f = ch; /* line 3 */
    f = x; /* line 4 */
  }
```

In the above code if x were between 255 and 0, ch and x would have identical values. Otherwise, the value of ch would reflect only the lower-order bits of x. In line 2, x will receive the nonfractional part of f. In line 3, f will convert the 8-bit integer value stored in ch to the same value in the floating-point format. This also happens in line 4, except that f will convert an integer value into floating-point format. The Table below summarizes the assignment type conversions.

Target Type	Expression Type	Possible Info Loss
signed char	char	If value > 127, target is negative
char	short int	High-order 8 bits
char	int (16 bits)	High-order 8 bits
char	int (32 bits)	High-order 24 bits
char	long int	High-order 24 bits
short int	int (16 bits)	None
short int	int (32 bits)	High-order 16 bits
int (16 bits)	long int	High-order 16 bits
int (32 bits)	long int	None
int	float	Fractional part and possibly more
float	double	Precision, result rounded
double	long double	Precision, result rounded

The concept of implicit type casting will be clearer from the example given in the following figure



5.3 Type Conversion in Expressions

Expressions are formed from data and operators. Data may be represented either by variables or by constants. Operators, constants, and variables are the constituents of expressions. An *expression* in C/C++ is any valid combination of these elements. Because most expressions tend to follow the general rules of algebra, they are often taken for granted. However, a few aspects of expressions relate specifically to C and C++.

Neither C nor C++ specifies the order in which the subexpressions of an expression are evaluated. This leaves the compiler free to rearrange an expression to produce more optimal code. However, it also means that your code should never rely upon the order in which subexpressions are evaluated. For example, the expression

```
x = f1() + f2();
```

does not ensure that f1() will be called before f2() .

When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called type promotion. First, all char and short int values are automatically elevated to int. (This process is called integral promotion.) Once this step has been completed, all other conversions are done operation by operation, as described in the following type conversion algorithm:

Algorithm

IF an operand is long double

THEN the second operand is converted into long double

ELSE IF an operand is double

THEN the second operand is converted into double

ELSE IF an operand is float

THEN the second operand is converted into float

ELSE IF an operand is unsigned long

THEN the second operand is converted into unsigned long

ELSE IF an operand is long

THEN the second operand is converted into long

ELSE IF an operand is unsigned int

THEN the second operand is converted into unsigned long

5.4 Type Conversions between Classes

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators cannot be applied indiscriminately on classes and pointers to classes as it may lead to code that will be syntactically correct but may cause runtime errors.

To control the type conversions between classes, there are four specific casting operators:

- dynamic_cast
- reinterpret_cast
- static_cast
- const_cast.

Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

dynamic_cast <new_type> (expression)

reinterpret_cast <new_type> (expression)

static_cast <new_type> (expression)

const_cast <new_type> (expression)

dynamic_cast

Dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class. dynamic_cast is always successful when we cast a class to one of its base classes but base-to-derived conversions are not allowed with dynamic_cast unless the base class is polymorphic. When a class is polymorphic, dynamic_cast performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class.

static_cast

Static_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic_cast is avoided.

static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;

int i = static_cast<int>(d);
```

reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed.

Summary

- Type conversion is conversion of data from one type to another type.
- Implicit type conversion is done automatically by the compiler whenever data from different types is intermixed.
- Implicit conversions also include constructor or operator conversions
- Explicit conversion can be done using type cast operator
- In C++ there are two notations used for explicit type conversion (C-style casting and C++-style casting)
- In an assignment statement, the type conversion rule is easy in this the value of the right side (expression side) of the assignment is converted to the type of the left side (target variable)
- When constants and variables of different types are mixed in an expression, they are all converted to the same type
- The compiler converts all operands up to the type of the largest operand, which is called type promotion
- Explicit type conversion operators cannot be applied indiscriminately on classes and pointers to classes as it may lead to code that will be syntactically correct but may cause runtime errors.
- To control the type conversions between classes, there are four specific casting operators: dynamic_cast, reinterpret_cast, static_cast, const_cast.

Self Check Exercise

1. What is meant by type conversion and why is it needed?
2. Differentiate between C-style casting and C++ -style casting?
3. How is expression type conversion done?
4. What is meant by implicit type casting?
5. Why explicit type casting cannot be used indiscriminately on classes?
6. What is the difference between the dynamic_cast and the static_cast?

Suggested Readings

- Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications,1994.
- E. Balaguruswamy, “Object Oriented Programming with C++”, Tata McGraw-Hill,2001. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Deitel and Deitel, “ C++ How to program” ,Pearson Education, 2001.
- Bjarne Strautrup, “ The C++ Programming Language”, Addison-Wesley Publication Co.,2001.
- Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.

Lesson No. 6 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

Input, Output Statements

Objectives

After reading this lesson you should be able to:

- Understand the general concept I/O system inherited by C++ from C
- Understand the two types of Library functions for I/O(console I/O functions, file I/O functions)
- Understand the two types of console I/O functions (formatted and unformatted)
- Understand the file I/O functions

Introduction

C++ supports two complete I/O systems: the one inherited from C and the other the object-oriented I/O system defined by C++ (hereafter called simply the C++ I/O system). The C-based I/O system will be discussed in this lesson and C++ I/O system in the Lesson 8. The C I/O is supported by C++ only for compatibility and due to some of the following reasons:

- Some times we are restricted to write code in the C subset. In this case, we are required to use the C-like I/O functions.
- Many programs which exist today and will exist in the near future will be hybrids of both C and C++ code. Further, it will be common for C programs to be "upgraded" into C++ programs. Thus, knowledge of both the C and the C++ is necessary.
- An understanding of the basic principles behind the C-like I/O system is crucial to an understanding of the C++ object-oriented I/O system. (Both share the same general concepts.)

- In certain situations (for example, in very short programs), it may be easier to use C's non-object-oriented approach to I/O than it is to use the object-oriented I/O defined by C++.

Library functions for I/O

There are numerous library functions available for I/O. These can be classified into two broad categories:

- Console I/O functions: Functions to receive input from keyboard and write output to VDU.
- File I/O functions: Functions to perform I/O operations on a floppy disk or hard disk. This topic is covered in next lesson.

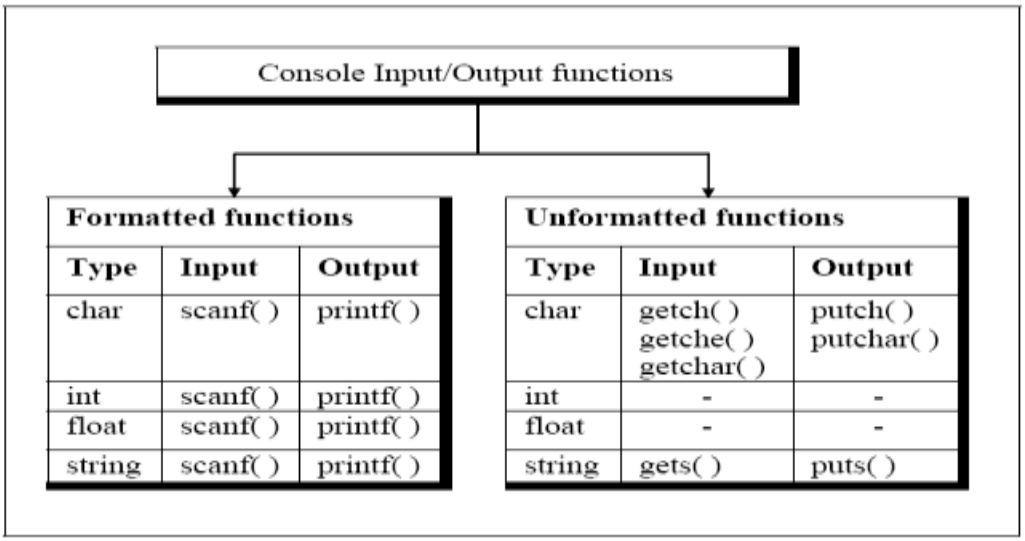
6.1 Console I/O Functions

The screen and the keyboard together are called a console. Console I/O functions can be further classified into two categories

- formatted console I/O functions
- unformatted console I/O functions

The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions.

The functions available under each of these two categories are shown below



Formatted functions

Type	Input	Output
char	scanf( )	printf( )
int	scanf( )	printf( )
float	scanf( )	printf( )
string	scanf( )	printf( )

Unformatted functions

Type	Input	Output
char	getch( ) getche( ) getchar( )	putch( ) putchar( )
int	-	-
float	-	-
string	gets( )	puts( )

6.1.1 Formatted Console I/O Functions: As can be seen from the above Figure the functions printf( ), and scanf( ) fall under the category of formatted console I/O functions. The printf() function writes data to the console. The scanf() function, its complement, reads

data from the keyboard. Both functions can operate on any of the built-in data types, including characters, strings, and numbers.

Printf()

The format of the printf() statement is given below

printf ("control string", list of variables) ;

The format string can contain:

- Characters that are simply printed as they are
- Conversion_specifications that begin with a % sign

Format specifiers

Printf() uses format specifiers that define the way in which the arguments will be displayed. A format specifier begins with a percent sign and is followed by the format code. There must be exactly the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, this **printf()** call

```
printf("I like %c%s", 'C', "++ very much!");
```

displays

I like C++ very much!

The **printf()** function accepts a wide variety of format specifiers, as shown in below.

Code	Format
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign
%c	Character
%d	Signed decimal integers

Printing Characters

To print an individual character, use %c. This causes its matching argument to be output, unmodified, to the screen. To print a string, use %s.

Printing Numbers

You may use either %d or %i to indicate a signed decimal number. These format specifiers are equivalent; both are supported for historical reasons. To output an unsigned value, use %u. The %f format specifier displays numbers in floating point. The %e and %E specifiers tell printf() to display a double argument in scientific notation. Numbers represented in scientific notation take this general form:

x.dddddE+/−yy

If you want to display the letter "E" in uppercase, use the %E format; otherwise use %e. You can tell printf() to use either %f or %e by using the %g or%Gformat specifiers. This causes printf() to select the format specifier that produces the shortest output. Where applicable, use%Gif you want "E" shown in uppercase; otherwise, use %g. You can display unsigned integers in octal or hexadecimal format using %o and %x, respectively. Since the hexadecimal number system uses the letters A through F to represent the numbers 10

through 15, you can display these letters in either upper- or lowercase. For uppercase, use the %X format specifier; for lowercase, use %x. If you wish to display an address, use %p. This format specifier causes printf() to display a machine address in a format compatible with the type of addressing used by the computer. The %n Specifier The %n format specifier is different from the others. Instead of telling printf() todisplay something, it causes printf() to load the variable pointed to by its corresponding argument with a value equal to the number of characters that have been output. In other words, the value that corresponds to the %n format specifier must be a pointer to a variable. After the call to printf() has returned, this variable will hold the number of characters output, up to the point at which the %n was encountered. The program below will help you to understand this somewhat unusual format code.

```
#include <stdio.h>

int main(void)

{

int count;

printf("this%n is a test\n", &count);

printf("%d", count);

return 0;

}
```

This program displays this is a test followed by the number 4. The %n format specifier is used primarily to enable your program to perform dynamic formatting.

Format Modifiers

Many format specifiers may take modifiers that alter their meaning slightly. For example, you can specify a minimum field width, the number of decimal places, and left justification. The format modifier goes between the percent sign and the format code.

- **The Minimum Field Width Specifier**: An integer placed between the % sign and the format code acts as a *minimum field width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you wish to pad with 0's, place a 0 before the field width specifier. For example, %05d will pad a number of less than five digits with 0's so that its total length is five. The minimum field width modifier is most commonly used to produce tables in which the columns line up.
- **The Precision Specifier:** The *precision specifier* follows the minimum field width specifier (if there is one). It consists of a period followed by an integer. Its exact

meaning depends upon the type of data it is applied to. When you apply the precision specifier to floating-point data using the %f, %e, or %E specifiers, it determines the number of decimal places displayed. For example, %10.4f displays a number at least ten characters wide with four decimal places. If you don't specify the precision, a default of six is used. When the precision specifier is applied to %g or %G, it specifies the number of significant digits. Applied to strings, the precision specifier specifies the maximum field length. For example, %5.7s displays a string at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated. When applied to integer types, the precision specifier determines the minimum number of digits that will appear for each number. Leading zeros are added to achieve the required number of digits.

The following program illustrates the precision specifier:

```
#include <stdio.h>

int main(void)

{

    printf("%.4f\n", 123.1234567);

    printf("%3.8d\n", 1000);

    printf("%10.15s\n", "This is a simple test.");

    return 0;

      }
```

It produces the following output:

```
123.1235

00001000

This is a simpl
```

- **Justifying Output**: By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For example, **%−10.2f** left justifies a floating-point number with two decimal places in a 10-character field.
    - The following program illustrates left justification:
    - #include <stdio.h>
    - int main(void)
    - {
        - printf("right-justified:%8d\n", 100);
        - printf("left-justified:%-8d\n", 100);

- return 0;
- }

- **Handling Other Data Types** There are two format modifiers that allow printf() to display short and long integers. These modifiers may be applied to the d, i, o, u, and x type specifiers. The l (*ell*)modifier tells printf() that a long data type follows. For example, %ld means that a long int is to be displayed. The h modifier instructs printf() to display a short integer. For instance, %hu indicates that the data is of type short unsigned int. The L modifier may prefix the floating-point specifiers e, f, and g, and indicates that a long double follows.
- **The * and # Modifiers:** The printf() function supports two additional modifiers to some of its format specifiers: * and #. Preceding g, G, f, E, or e specifiers with a # ensures that there will be a decimal point even if there are no decimal digits. If you precede the x or X format specifier with a #, the hexadecimal number will be printed with a 0x prefix. Preceding the o specifier with # causes the number to be printed with a leading zero. You cannot apply # to any other format specifiers.

Instead of constants, the minimum field width and precision pecifiers may be provided by arguments to printf() . To accomplish this, use an * as a placeholder. When the format string is scanned, printf() will match the * to an argument in the order in which they occur

The following program illustrates both **#** and *****:

```
#include <stdio.h>

int main(void)

{

        printf("%x %#x\n", 10, 10);

        printf("%*.*f", 10, 4, 1234.34);

        return 0;

        }
```

scanf( )

scanf() is the general-purpose console input routine. It can read all the built-in data types and automatically convert numbers into the proper internal format. It is much like the reverse of **printf()** . The general form of scanf( ) statement is as follows:

scanf ("control string", list of addresses of variables) ;

For example:

scanf ("%d %f %c", &c, &a, &ch) ;

Note that we are sending addresses of variables (addresses are obtained by using '&' the 'address of' operator) to scanf( ) function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s).

All the format specifications that we learnt in **printf( )** function are applicable to **scanf( )** function as well. The **scanf()** function returns the number of data items successfully assigned a value. If an error occurs, **scanf()** returns **EOF**.
Three classifications of characters are used in scanf():

- Format specifiers
- White-space characters
- Non-white-space characters

Format Specifiers: The input format specifiers are preceded by a % sign and tell **scanf()** what type of data is to be read next. The codes are given below.

Code	Meaning
%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, octal, or hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Read an unsigned decimal integer.
%[]	Scan for a set of characters.
%%	Read a percent sign.

The format specifiers are matched, in order from left to right, with the arguments in the argument list.

Inputting Numbers: To read an integer, use either the %d or %i pecifier. To read a floating-point number represented in either standard or scientific notation, use %e, %f, or %g. You can use scanf() to read integers in either octal or hexadecimal form by using the %o and %x format commands, respectively.

The following program reads an octal and hexadecimal

number:

```
#include <stdio.h>

int main(void)

{

int i, j;

scanf("%o%x", &i, &j);

printf("%o %x", i, j);

return 0;

}
```

The **scanf()** function stops reading a number when the first nonnumeric character is encountered.

Reading Individual Characters Using scanf( ): scanf() uses the %c format specifier for reading chararcters. It will generally line-buffer input when the %c specifier is used. This makes it somewhat troublesome in an interactive environment.

Although spaces, tabs, and newlines are used as field separators when reading other types of data, when reading a single character, white-space characters are read like any other character.

The scanf() function can be used to read a string from the input stream using the %s format specifier. Using %s causes scanf() to read haracters until it encounters a white-space character. The characters that are read are put into the character array pointed to by the corresponding argument and the result is null terminated. As it applies to scanf() , a white-space character is either a space, a newline, a tab, a vertical tab, or a form feed.

Using a Scanset : The scanf() function supports a general-purpose format specifier called a scanset. A *scanset* defines a set of characters. When scanf() processes a scanset, it will input characters as long as those characters are part of the set defined by the scanset. The characters read will be assigned to the character array that is pointed to by the scanset's corresponding argument. You define a scanset by putting the characters to scan for inside square brackets. The beginning square bracket must be prefixed by a percent sign. For example, the following scanset tells scanf() to read only the characters X, Y, and Z.

```
%[XYZ]
```

When you use a scanset, scanf() continues to read characters, putting them into the corresponding character array until it encounters a character that is not in the scanset.

Upon return from scanf() , this array will contain a null-terminated string that consists of the characters that have been read. To see how this works, try this program:

```
#include <stdio.h>

int main(void)

{

int i;

char str[80], str2[80];

scanf("%d%[abcdefg]%s", &i, str, str2);

printf("%d %s %s", i, str, str2);

return 0;

}
```

Enter 123abcdtye followed by ENTER. The program will then display 123 abcd tye. Because the "t" is not part of the scanset, scanf() stops reading characters into str when it encounters the "t." The remaining characters are put into str2. You can specify an inverted set if the first character in the set is a ^. The ^ instructs scanf() to accept any character that is *not* defined by the scanset. In most implementations you can specify a range using a hyphen. For example, this tells scanf() to accept the characters A through Z:

```
%[A-Z]
```

One important point to remember is that the scanset is case sensitive. If you want to scan for both upper- and lowercase letters, you must specify them individually.

Discarding Unwanted White Space

A white-space character in the control string causes scanf() to skip over one or more leading white-space characters in the input stream. A white-space character is either a space, a tab, vertical tab, form feed, or a newline. In essence, one white-space character in the control string causes scanf() to read, but not store, any number (including zero) of white-space characters up to the first non-white-space character.

Non-White-Space Characters in the Control String

A non-white-space character in the control string causes scanf() to read and discard matching characters in the input stream. For example, "%d,%d" causes scanf() to read an

integer, read and discard a comma, and then read another integer. If the specified character is not found, scanf() terminates. If you wish to read and discard a percent sign, use %% in the control string.

You Must Pass scanf( ) Addresses

All the variables used to receive values through scanf() must be passed by their addresses. This means that all arguments must be pointers to the variables used as arguments. Recall that this is one way of creating a call by reference, and it allows a function to alter the contents of an argument. For example, to read an integer into the variable count, you would use the following scanf() call:

```
scanf("%d", &count);
```

Strings will be read into character arrays, and the array name, without any index, is the address of the first element of the array. So, to read a string into the character array **str**, you would use

```
scanf("%s", str);
```

In this case, **str** is already a pointer and need not be preceded by the **&** operator.

Format Modifiers

As with printf() , scanf() allows a number of its format specifiers to be modified. The format specifiers can include a maximum field length modifier. This is an integer, placed between the % and the format specifier, that limits the number of characters read for that field. For example, to read no more than 20 characters into str, write

```
scanf("%20s", str);
```

If the input stream is greater than 20 characters, a subsequent call to input begins where this call leaves off. The other format specifiers are same as discussed in printf().

Suppressing Input

You can tell scanf() to read a field but not assign it to any variable by preceding that field's format code with an *. For example, given

```
scanf("%d%*c%d", &x, &y);
```

you could enter the coordinate pair 10,10. The comma would be correctly read, but not assigned to anything. Assignment suppression is especially useful when you need to process only a part of what is being entered.

6.1.2 Unformatted Console I/O Functions

Reading and Writing Characters

We often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. getch( ) and getche( ) are two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in getche( ) function means it echoes (displays) the character that you typed to the screen. As against this getch( ) just returns the character that you typed without echoing it on the screen. putch( ) and putchar( ) form the other side of the coin. They print a character on the screen. As far as the working of putch( ) putchar( ) and fputchar( ) is concerned it's exactly same.

These are the simplest of the console I/O functions, The prototypes for getchar() and putchar() are shown here:

int getchar(void);

int putchar(int *c*);

As its prototype shows, the getchar() function is declared as returning an integer. However, you can assign this value to a char variable, as is usually done, because the character is contained in the low-order byte. (The high-order byte is normally zero.) getchar() returns EOF if an error occurs. In the case of putchar(), even though it is declared as taking an integer parameter, you will generally call it using a character argument. Only the low-order byte of its parameter is actually output to the screen. The putchar() function returns the character written, or EOF if an error occurs. The following program illustrates getchar() and putchar(). It inputs characters from the keyboard and displays them in reverse case, that is, it prints uppercase as lowercase and lowercase as uppercase. To stop the program, a period has to be entered.

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   char ch;
   printf("Enter some text (type a period to quit).\n");
       do {
           ch = getchar();
           if(islower(ch)) ch = toupper(ch);
           else ch = tolower(ch);
           putchar(ch);
       } while (ch != '.');
   return 0;
}
```

Reading and Writing Strings

The functions gets() and puts() enable us to read and write strings of characters. The gets() function reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and gets() returns. In fact, you cannot use gets() to return a

carriage return (although getchar() can do so). You can correct typing mistakes by using the backspace key before pressing ENTER. The prototype for gets() is

char *gets(char **str*);

where *str* is a character array that receives the characters input by the user. gets() also returns *str*. The following program reads a string into the array str and prints its length:

```
#include <stdio.h>

#include <string.h>

int main(void)

{

char str[80];

gets(str);

printf("Length is %d", strlen(str));

return 0;

}
```

You need to be careful when using gets() because it performs no boundary checks on the array that is receiving input. Thus, it is possible for the user to enter more characters than the array can hold. While gets() is fine for sample programs and simple utilities that only you will use, you will want to avoid its use in commercial code. One alternative is the fgets() function, which allows you to prevent an array overrun. The puts() function writes its string argument to the screen followed by a newline. Its prototype is:

int puts(const char **str*);

puts() recognizes the same backslash codes as printf() , such as '\t' for tab. A call to puts() requires far less overhead than the same call to printf() because puts() can only output a string of characters□it cannot output numbers or do format conversions. Therefore, puts() takes up less space and runs faster than printf() . For this reason, the puts() function is often used when it is important to have highly optimized code. The puts() function returns EOF if an error occurs. Otherwise, it returns a nonnegative value. However, when writing to the console, you can usually assume that no error will occur, so the return value of puts() is seldom monitored. The following statement displays hello:

puts("hello");

Summary

- C++ supports two complete I/O systems: the one inherited from C and the other the object-oriented I/O system defined by C++
- There is no keyword available in I/O system inherited from C for doing input/output.
- All I/O is done using standard library functions.
- Library functions for I/O classified into two broad categories: Console I/O functions and File I/O functions
- The screen and the keyboard together are called a console. Console I/O functions can be further classified into two categories: formatted console I/O functions and unformatted console I/O functions

Self Check Exercise

1. Why is the C I/O supported by C++, give main reasons?
2. What is meant by console I/O functions explain any two in detail?
3. What are the main categorie of console I/O functions?
4. Which are the different format specifiers used in printf, explain each briefly?
5. What is the difference between format specifiers and format modifiers?
6. Explain any three format modifiers.

Suggested Readings

- Robert Lafore, “Object Oriented Programming in C++”, Galgotia Publications,1994.
- E. Balaguruswamy, “Object Oriented Programming with C++”, Tata McGraw-Hill,2001. Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Herbert Schildt, “ The Complete Reference C++”, Tata McGraw-Hill, 2001.
- Deitel and Deitel, “ C++ How to program” ,Pearson Education, 2001.
- Bjarne Strautrup, “ The C++ Programming Language”, Addison-Wesley Publication Co.,2001.
- Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

Lesson No. 7 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

FILE INPUTS/OUTPUTS

Objectives

After reading this lesson you should be able to:

- Understand the concept of streams and its different types
- Understand the concept of Files
- Understand File Operations in C type I/O operations supported by C++

Introduction

C++ supports the entire Standard C file system, and it also defines its own, object-oriented I/O system, which includes both I/O functions and I/O operators. Before beginning our discussion of the C file system, it is necessary to know the difference between the terms *streams* and *files*. The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed. The C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a *stream* and the actual device is called a *file*. It is important to understand how streams and files interact.

7.1 Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. Even though each device is very different, the buffered file system transforms each into a logical device called a stream. All streams behave similarly. Because streams are largely device independent, the same function that

can write to a disk file can also be used to write to another type of device, such as the console. There are two types of streams: text and binary.

7.1.1 Text Streams

A text stream is a sequence of characters. Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character. In a text stream, certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

7.1.2 Binary Streams

A binary stream is a sequence of bytes that have a one-to-one correspondence to those in the external device, that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device. However, an implementation-defined number of null bytes may be appended to a binary stream. These null bytes might be used to pad the information so that it fills a sector on a disk, for example.

7.1.3 Difference between Text and Binary File

There are three main areas where text and binary mode files are different.

- Handling of newlines: In text mode, a newline character is converted into the carriage return-linefeed combination before being written to the disk. Likewise, the carriage return-linefeed combination on the disk is converted back into a newline when the file is read by a C program. However, if a file is opened in binary mode, as opposed to text mode, these conversions will not take place.
- Representation of end of file : The second difference between text and binary modes is in the way the end-of-file is detected. In text mode, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function would return the EOF signal to the program. As against this, there is no such special character present in the binary mode files to mark the end of file. The binary mode files keep track of the end of file from the number of characters present in the directory entry of the file.
- Storage of numbers: The only function that is available for storing numbers in a disk file is the fprintf( ) function. It is important to understand how numerical data is stored on the disk by fprintf( ). Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, two bytes for an integer, four bytes for a float, and so on? No. Numbers are stored as strings of characters. Thus, 1234, even though it occupies two bytes in memory, when transferred to the disk using fprintf( ), would occupy four bytes, one byte per character. Similarly, the floating-point number 1234.56 would occupy 7 bytes on

disk. Thus, numbers with more digits would require more disk space. Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (fread) and fwrite( ) which are discussed later) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

7.2 Files

In C/C++, a file may be anything from a disk file to a terminal or printer. You associate a stream with a specific file by performing an open operation. Once a file is open, information may be exchanged between it and your program.

- Not all files have the same capabilities. For example, a disk file can support random access while some printers cannot. This brings up an important point about the C I/O system: All streams are the same but all files are not.
- If the file can support position requests, opening that file also initializes the file position indicator to the start of the file.
- As each character is read from or written to the file, the position indicator is incremented, ensuring progression through the file.
- You disassociate a file from a specific stream with a close operation. If you close a file opened for output, the contents, if any, of its associated stream are written to the external device. This process is generally referred to as flushing the stream, and guarantees that no information is accidentally left in the disk buffer.
- All files are closed automatically when your program terminates normally, either by main() returning to the operating system or by a call to exit() . Files are not closed when a program terminates abnormally, such as when it crashes or when it calls abort() . Each stream that is associated with a file has a file control structure of type FILE. Never modify this file control block.

7.2.1 File Operations

There are different operations that can be carried out on a file. These are:

- Creation of a new file
- Opening an existing file
- Reading from a file
- Writing to a file
- Moving to a specific location in a file (seeking)
- Closing a file

Thus the C file system is composed of several interrelated functions. The most common of these are shown below. They require the header **stdio.h**. C++ programs may also use the new-style header **<cstdio>**.

Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc()** .
getc( )	Reads a character from a file.
fgetc( )	Same as **getc()** .
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf()** is to the console.
fscanf( )	Is to a file what **scanf()** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

The brief descrption of each function is given below:

fopen()

The fopen() function opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file. The prototype for this function is given below:

```
FILE *fp;

fp = fopen("Filename", "w");
```

fclose()

The fclose() function closes a stream that was opened by a call to fopen() .

The fclose() function has this prototype:

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

putc()

The putc() function writes characters to a file that was previously opened for writing using the fopen() function. The prototype of this function is int putc(int ch, FILE *fp);

where fp is the file pointer returned by fopen() and ch is the character to be output. The file pointer tells putc() which file to write to. If a putc() operation is successful, it returns the character written. Otherwise, it returns EOF.

getc()

The getc() function reads characters from a file opened in read mode by fopen() . The prototype of getc() is

int getc(FILE *fp);

where fp is a file pointer of type FILE returned by fopen() . getc() returns an integer, but the character is contained in the low-order byte. Unless an error occurs, the highorder byte is zero. The getc() function returns an EOF when the end of the file has been reached.

fgets() and fputs()

fgets() and fputs() , which read and write character strings from and to a disk file. These functions work just like putc() and getc() , but instead of reading or writing a single character, they read or write strings. They have the following prototypes:

int fputs(const char *str, FILE *fp);

char *fgets(char *str, int length, FILE *fp);

feof()

The function feof() , which determines when the end of the file has been encountered. The feof() function has this prototype:

int feof(FILE *fp);

feof() returns true if the end of the file has been reached; otherwise, it returns 0.

rewind()

The rewind() function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file. Its prototype is

void rewind(FILE *fp);

where fp is a valid file pointer.

ferror()

The ferror() function determines whether a file operation has produced an error. The ferror() function has this prototype:

int ferror(FILE *fp);

where fp is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns false.

remove()

The remove() function erases the specified file. Its prototype is

```
int remove(const char *filename);
```

fflush()

If you wish to flush the contents of an output stream, use the fflush() function, whose prototype is shown here:

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

This function writes the contents of any buffered data to the file associated with fp.

fread() and fwrite()

To read and write data types that are longer than one byte, the C file system provides two functions: fread() and fwrite() . These functions allow the reading and writing of blocks of any type of data. Their prototypes are

```
size_t fread(void *buffer, size_t num_bytes, size_t count, FILE *fp);

size_t fwrite(const void *buffer, size_t num_bytes, size_t count, FILE *fp);
```

For fread() , buffer is a pointer to a region of memory that will receive the data from the file. For fwrite() , buffer is a pointer to the information that will be written to the file. The value of count determines how many items are read or written, with each item being num_bytes bytes in length. Finally, fp is a file pointer to a previously opened stream.

fseek()

You can perform random-access read and write operations using the C I/O system with the help of fseek() , which sets the file position indicator. Its prototype is shown here:

```
int fseek(FILE *fp, long numbytes, int origin);
```

Here, fp is a file pointer returned by a call to fopen() . numbytes is the number of bytes from origin that will become the new current position, and origin is one of the following

macros:

Origin	Macro Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Therefore, to seek numbytes from the start of the file, origin should be SEEK_SET. To seek from the current position, use SEEK_CUR; and to seek from the end of the file, use SEEK_END. The fseek() function returns 0 when successful and a nonzero value if an error occurs.

ftell()

You can determine the current location of a file using ftell() . Its prototype is

long ftell(FILE *fp);

fprintf() and fscanf()

In addition to the basic I/O functions already discussed, the C I/O system includes fprintf() and fscanf() . These functions behave exactly like printf() and scanf() except that they operate with files. The prototypes of fprintf() and fscanf() are

int fprintf(FILE *fp, const char *control_string,. . .);

int fscanf(FILE *fp, const char *control_string,. . .);

where fp is a file pointer returned by a call to fopen() . fprintf() and fscanf() direct their I/O operations to the file pointed to by fp.

Consider the following program to read a file and display its contents on the screen.

```
/* Display contents of a file on screen. */

# include "stdio.h"

main( )

{

  FILE *fp ;

  char ch ;

  fp = fopen ( "PR1.C", "r" ) ;

  while ( 1 )

        {

        ch = fgetc ( fp ) ;

        if ( ch == EOF )

        break ;

        printf ( "%c", ch ) ;
```

```
    }
fclose ( fp ) ;
}
```

Now let us understand the various file functions used in the above program.

- The FILE structure has been defined in the header file "stdio.h" (standing for standard input/output header file). Therefore, it is necessary to #include this file.
- The File Pointer: The file pointer is the common thread that unites the C I/O system. A file pointer is a pointer to a structure of type FILE. It points to information that defines various things about the file, including its name, status, and the current position of the file. In essence, the file pointer identifies a specific file and is used by the associated stream to direct the operation of the I/O functions. In order to read or write files, your program needs to use file pointers. To obtain a file pointer variable, use a statement like this:

  FILE *fp;

- Opening a File: Before we can read (or write) information from (to) a file on a disk we must open the file. To open the file we have called the function fopen( ). It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Note that "r" is a string and not a character; hence the double quotes and not single quotes. The fopen() function opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file.
  The fopen() function has this prototype:

  FILE *fopen(const char *filename, const char *mode);

  where filename is a pointer to a string of characters that make up a valid filename and may include a path specification. The string pointed to by mode determines how the file will be opened. In our first program on disk I/O we have opened the file in read ("r") mode. However, "r" is but one of the several modes in which we can open a file. Following is a list of all possible modes in which a file can be opened. The tasks performed by fopen( ) when a file is opened in each of these modes are also mentioned. Table below shows the legal values for mode.

Mode	Meaning
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

File Opening Modes

- Reading from a File: Once the file has been opened for reading using fopen( ), as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer. This pointer is one of the elements of the structure to which fp is pointing. To read the file's contents from memory there exists a function called fgetc( ). This has been used in our program as,

  ch = fgetc (fp) ;

  fgetc( ) reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable ch. Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer fp. We have used the function fgetc( ) within an indefinite while loop. We break out when the end of file is reached. Where end of file is a special character, whose ASCII value is 26. This character is inserted beyond the last character in the file, when it is created.

- Closing the File: When we have finished reading from the file, we need to close it. This is done using the function fclose( ) through the statement,

  fclose (fp) ;

Summary

- C++ supports the entire Standard C file system
- The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed through the use of streams
- The C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a stream and the actual device is called a file.
- There are two types of streams: text and binary
- A text stream is a sequence of characters

- A binary stream is a sequence of bytes that have a one-to-one correspondence to those in the external device
- In C/C++, a file may be anything from a disk file to a terminal or printer. You associate a stream with a specific file by performing an open operation
- There are different operations that can be carried out on a file. Like creation of a new file, Opening an existing file, reading from a file, writing to a file, moving to a specific location in a file (seeking), Closing a file.

Self Check Exercise

1. What do you understand by streams? How many types of streams are there.
2. What is meant by files with respect to C type file I/O?
3. Name and explain any five file I/O functions?
4. Which are the different modes in which a file can be opened?
5. Explain fseek() in detail.
6. What is the difference between fgetc() and getc() functions?
7. Write a simple program to read and write a text file.

Suggested Readings

- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

Lesson No. 8 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

C++ I/O System

Objectives

After reading this lesson you should be able to:

- Understand the concept of C++ I/O system
- Understand the concept of predefined I/O streams
- Understand the formatted I/O
- Understand the concept of formatting using the ios members
- Understand the concept of formatting using manipulators

Introduction

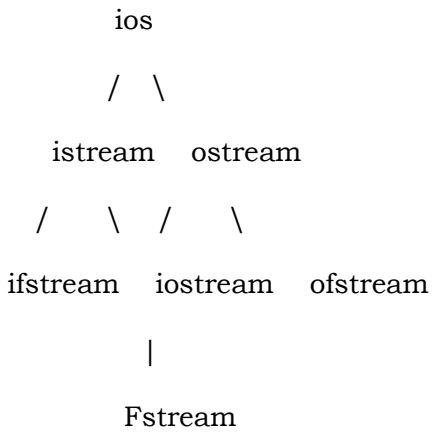
The I/O system inherited from C is extremely rich, flexible, and powerful, but C++ defines yet another system as the C's I/O system knows nothing about objects. Therefore, for C++ to provide complete support for object-oriented programming, it was necessary to create an I/O system that could operate on user-defined objects. In addition to support for objects, there are several benefits to using C++'s I/O system even in programs that don't make extensive (or any) use of user-defined objects.

8.1 Streams

C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen or the keyboard. A stream is an object where a program can either insert or extract characters to/from it. We do not really need to care about many specifications about the physical media associated with the stream - we only need to know it will accept or provide characters sequentialy. The standard C++ library includes the header file iostream, where the standard input and output stream objects are declared. Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters. In C++, I/O (input/output) occurs in streams of bytes. A stream is simply a sequence of bytes. Input streams receive bytes from any device (a file on a hard disk, the keyboard, or some other input device), while an output streams send a sequence of bytes to a device (the display, a file on the hard disk, the printer, etc). Any additional meaning or structure these bytes may have is imposed by the program. C++ provides both "low-level" and "high-level" ways for manipulating streams.

- Low level commands simply specify the number of bytes to be received from or sent to the stream.
- High level I/O, also known as formatted I/O, allows the programmer to to consider the bytes in the stream as grouped into meaningful units reprsenting ints, floats, chars, strings, etc.
- An input stream in C++ is called an istream.
- An output stream in C++ is called an ostream.

8.1.1 Class Hierarchy



The ios class contains the majority of the features you need to operate C++ streams. The three most important features are the formatting flags, the error-status bits, and the file operation mode. (The first two are discussed in this lesson and file operation is covered in next lesson)

As shown above the istream and ostream classes are derived from ios and are dedicated to input and output, respectively. The istream class contains such member functions as get(), getline(), read(), and the extraction (>>) operators, whereas ostream contains put() and write() and the insertion (<<) operators. The iostream class is derived from both istream and ostream by multiple inheritance. Classes derived from the iostream class can be used with devices, such as disk files, that may be opened for both input and output at the same time. Three classes— stream_withassign, ostream_withassign, and iostream_withassign—are inherited from istream, ostream, and iostream, respectively. They add assignment operators to these classes so that cin, cout, and so on can be assigned to other streams.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called basic_streambuf. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use basic_streambuf directly. The class hierarchy that you will most commonly be working with is derived from basic_ios. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for basic_ios is called ios_base, which defines several nontemplate traits used by basic_ios.) basic_ios is used as a base for several derived classes, including basic_istream, basic_ostream, and basic_iostream. These classes are used to create streams capable of input, output, and input/output, respectively.

8.2 C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams cin, cout, and cerr correspond to C's stdin, stdout, and stderr.

8.2.1 Standard Output (cout)

By default, the standard output of a program is the screen, and the C++ stream object defined to access it is cout. cout is used in conjunction with the insertion operator, which is written as << (two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;              // prints number 120 on screen
cout << x;               // prints the content of x on screen
```

The << operator inserts the data that follows it into the stream preceding it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and variable x into the standard output stream cout. Notice that the sentence in the first instruction is enclosed between double quotes (") because it is a constant string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes (") so that they can be clearly distinguished from variable names. For example, these two sentences have very different results:

```
cout << "Hello";  // prints Hello
cout << Hello;    // prints the content of Hello variable
```

The insertion operator (<<) may be used more than once in a single statement:

```
cout << "Hello, " << "I am " << "a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout << "Hello, I am " << age << " years old and my zipcode is " << zipcode;
```

If we assume the age variable to contain the value 24 and the zipcode variable to contain 90064 the output of the previous statement would be:

Hello, I am 24 years old and my zipcode is 90064

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.";
cout << "This is another sentence.";
```

will be shown on the screen one following the other without any line break between them:

This is a sentence. This is another sentence.

even though we had written them in two different insertions into cout. In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n):

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.

Second sentence.

Third sentence.

Additionally, to add a new-line, you may also use the endl manipulator. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

would print out:

First sentence.

Second sentence.

The end manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the \n escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

8.2.2 Standard Input (cin).

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the overloaded operator of extraction (>>) on the cin stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second one waits for an input from cin (the keyboard) in order to store it in this integer variable. cin can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character, the extraction from cin will not process the input until the user presses RETURN after the character has been introduced. You must always consider the type of the variable that you are using as a container with cin extractions. If you request an integer you will get an integer, if you request a character you

will get a character and if you request a string of characters you will get a string of characters.

Consider the following program:

```
#include <iostream>

using namespace std;

int main ()

{

  int i;

  cout << "Please enter an integer value: ";

  cin >> i;

  cout << "The value you entered is " << i;

  cout << " and its double is " << i*2 << ".\n";

  return 0;

}
```

Please enter an integer value: 702

The value you entered is 702 and its double is 1404.

The user of a program may be one of the factors that generate errors even in the simplest programs that use cin (like the one we have just seen). Since if you request an integer value and the user introduces a name (which generally is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by cin extractions you will have to trust that the user of your program will be cooperative and that he/she will not introduce his/her name or something similar when an integer value is requested. When we see the stringstream class we'll we will see a possible solution for the errors that can be caused by this type of user input. You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;

cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

cin and strings: We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, as it has been said, cin extraction stops reading as soon as if finds any blank space character, so in this case we will be able to get just one word for each extraction. In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin: The use of getline is shown in the following example

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

Notice how in both calls to getline we used the same string identifier (mystr). What the program does in the second call is simply to replace the previous content by the new one that is introduced.

8.2.3 standard error output (cerr)

cerr is a mechanism that is meant specifically for printing error messages. **cerr** is an output stream (just like cout) that is also defined in iostream.h. The cerr object is often used for error messages and program diagnostics. Output sent to cerr is displayed immediately, rather than being buffered, as output sent to cout is. Also, output to cerr cannot be redirected. For these reasons, you have a better chance of seeing a final output message from cerr if your program dies prematurely.

The following code illustrates how cerr can be used

```
void PrintString(char *strString)

{

   // Only print if strString is non-null

   if (strString)

      std::cout << strString;

   else

      std::cerr << "PrintString received a null parameter";

}
```

8.2.4 Buffered version of cerr (clog)

Another object, clog, is similar to cerr in that it is not redirected, but its output is buffered, whereas cerr's is not. It is a predefined stream that controls output to a stream buffer associated with the object stderr declared in <cstdio>.

Note: Standard C++ also defines these four additional streams: win, wout, werr, and wlog. These are wide-character versions of the standard streams. Wide characters are of type wchar_t and are generally 16-bit quantities.

8.3 Formatted I/O

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. This is done using formatting flags which are a set of enum definitions in ios. They act as on/off switches that specify choices for various aspects of input and output format and operation. There are two related but conceptually different ways that you can format data.

- Directly access members of the **ios** class. You can set various format status flags defined inside the **ios** class or call various **ios** member functions.
- You can use special functions called *manipulators* that can be included as part of an I/O expression.

There are several ways to set the formatting flags, and different flags can be set in different ways. Because they are members of the ios class, flags must usually be preceded

by the name ios and the scope-resolution operator (e.g., ios::skipws). All the flags can be set using the setf() and unsetf() ios member functions. For example,

```
cout.setf(ios::left);                    // left justify output text

cout >> "This text is left-justified";

cout.unsetf(ios::left);                  // return to default (right justified)
```

Many formatting flags can be set using manipulators.

8.3.1 Formatting using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted. The ios class declares a bitmask numeration called fmtflags in which the following values are defined.

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

- When the skipws flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When skipws is cleared, white-space characters are not discarded.
- When the left flag is set, output is left justified.
- When right is set, output is right justified.
- When the internal flag is set, a numeric value is padded to fill a field by
- Inserting spaces between any sign or base character.
  (If none of these flags are set, output is right justified by default.)

- By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the oct flag causes output to be displayed in octal.
- Setting the hex flag causes output to be displayed in hexadecimal.
- To return output to decimal, set the dec flag.
- Setting showbase causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.
- By default, when scientific notation is displayed, the e is in lowercase. Also, when a hexadecimal value is displayed, the x is in lowercase. When uppercase is set, these characters are displayed in uppercase.
- Setting showpos causes a leading plus sign to be displayed before positive values.

- Setting showpoint causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.
- By setting the scientific flag, floating-point numeric values are displayed using scientific notation.
- When fixed is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.
- When unitbuf is set, the buffer is flushed after each insertion operation.
- When boolalpha is set, Booleans can be input or output using the keywords true and false.
- Since it is common to refer to the oct, dec, and hex fields, they can be collectively referred to as basefield.
- Similarly, the left, right, and internal fields can be referred to as adjustfield.
- Finally, the scientific and fixed fields can be referenced as floatfield.

The following program displays the value 100 with the showpos and showpoint flags turned on.

```
#include <iostream>

using namespace std;

int main()

{

  cout.setf(ios::showpoint);

  cout.setf(ios::showpos);

  cout << 100.0; // displays +100.0

  return 0;

}
```

The ios class contains a number of functions that you can use to set the formatting flags and perform other tasks. Few important ones are discussed below

setf() and unsetf()

The setf() is used to set the flags The complement of setf() is unsetf(). This member function of ios is used to clear one or more format flags. Their general form is

```
void setf(fmtflags flags);

void unsetf(fmtflags flags);
```

flags()

- To know the current format settings (without altering the settings) ios includes the member function flags() , which simply returns the current setting of each format flag. Its prototype is shown here:

  fmtflags flags( );

- The flags() function has a second form that allows you to set all format flags associated with a stream. The prototype for this is given below

  fmtflags flags(fmtflags *f*);

The following example makes this more clear

```
#include <iostream>

using namespace std;

void showflags();

int main()

{

// show default condition of format flags

showflags();

// showpos, showbase, oct, right are on, others off

long f = ios::showpos | ios::showbase | ios::oct | ios::right;

cout.flags(f); // set all flags

showflags();

return 0;

}
```

8.3.2 Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called manipulators that can be included in an I/O expression. Manipulators are formatting instructions inserted directly into a stream. They are the most common way to control output formatting. Manipulators were added recently to C++ and will not be supported by older compilers. The following example shows the use of some of the manuplators like end1(which sends a new line to the stream and flushes it) and setiosflags() given below

```
cout << "To each his own." << endl;

cout << setiosflags(ios::fixed) // use fixed decimal point
```

<< setiosflags(ios::showpoint) // always show decimal point

<< var;

Manipulators are of two types

- which take an argument
- that do not take argument

The following table shows the various manipulators

Manipulator	Purpose	Input/Output
Boolalpha	Turns on boolapha flag.	Input/Output
dec	Turns on dec flag.	Input/Output
end1	Output a newline character and flush the stream	Output
ends	Output a null.	Output
fixed	Turns on fixed flag.	Output
flush	Flush a stream	Output
hex	Turns on hex flag.	Input/Output
Internal	Turns on internal flag.	Output
left	Turns on left flag.	Output
noboooalpha	Turns off boolalpha flag.	Input/Output
noshowbase	Turns off showbase flag.	Output
noshowpoint	Turns off showpoint flag.	Output
noshowpos	Turns off showpos flag.	Output

Manipulator	Purpose	Input/Output
noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To access manipulators that take parameters (such as setw()), you must include <iomanip> in your program.

Here is an example that uses some manipulators:

```
#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

cout << hex << 100 << endl;

cout << setfill('?') << setw(10) << 2343.0;
```

```
return 0;

}
```

This displays

64

??????2343

Notice how the manipulators occur within a larger I/O expression. Also notice that when a manipulator does not take an argument, such as endl() in the example, it is not followed by parentheses. This is because it is the address of the function that is passed to the overloaded << operator.

The two techniques for formatting using formatting flags discussed above are not only used for the keyboard and display, but, can also be used , for files as well.

When we compare C-type standard I/O and Stream I/O, the preferred approach is stream I/O because of the following two reasons

- First, it's easier to avoid mistakes using stream I/O. Some times we use wrong format specifier in printf() (e.g., %d instead of %f), and the that data is displayed incorrectly. This is not so if we use cin and cout
- Second, the stream I/O approach lets you use cout and cin with classes you write yourself. This allows you to perform I/O on any programming object the same way it's performed on basic data types. This is a very powerful technique, which is unavailable with standard C I/O

Summary

- C++ defines a second system of I/O as the C's I/O system knows nothing about objects.
- C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen or the keyboard.
- Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters.
- C++ provides both "low-level" and "high-level" ways for manipulating streams.
- When a C++ program begins execution, four built-in low level streams are automatically opened (cin, cout, cerr, clog)
- Each stream has associated with it a set of format flags that control the way information is formatted.

Self Check Exercise

1. Explain the four predefined streams in C++
2. What are the different ways of formatting I/O using C++ I/O system?

3. What is meant by manipulators?
4. How the various formatting flags set and cleared explain with an example?

Suggested Readings

- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

Lesson No. 9 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

C++ File I/O

Objectives

After reading this lesson you should be able to:

- Understand the concept of stream file I/O .
- Understand the two types formatted and unformatted file I/O
- Understand the different functions used for opening and closing files
- Understand the different functions used for reading and writing to files
- Understand some special functions like eof(),ignore( ), peek( ) and putback( ),flush( ), seekg() and seekp(),tellg() and tellp() etc which make the file I/O more flexible and verstile.
- Understand briefly the concept of object I/O

Introduction

C++ I/O forms an integrated I/O system, which treats the console and file I/O in the same way but still file I/O is subject to its own constraints and is treated separately in this lesson. We can say disk file I/O is simply a special case of the general I/O system

Disk files require a different set of classes than files used with the keyboard and screen. These are ifstream for input, fstream for input and output, and ofstream for output. Objects of these classes can be associated with disk files and you can use their member functions to read and write to the files. The ifstream is derived from istream, fstream is derived from iostream, and ofstream is derived from ostream. These ancestor classes are in turn derived from ios. Thus the file-oriented classes derive many of their member functions from more general classes. The file-oriented classes are also derived, by multiple inheritance, from the fstreambase class. This class contains an object of class filebuf, which is a file-oriented buffer with associated member functions derived from the more general streambuf class. For many file-based operations, streambuf is accessed automatically.

The approach to disk I/O used in C++ is quite different from that used in C. The old C functions, such as fread() and fwrite(), will work in C++, but they are not well suited to the object-oriented environment. As with printf() and scanf(), they aren't extensible, whereas you can extend the C++ iostream approach to work with your own classes.

There are two basic kinds of disk I/O in C++: formatted and unformatted. We will start our discussion with formatted file I/O.

9.1 Formatted File I/O

In formatted I/O, numbers are stored on disk as a series of characters. Thus 6.02, rather than being stored as a 4-byte type float or an 8-byte type double, is stored as the characters '6', '.', '0', and '2'. This can be inefficient for numbers with many digits, but it's appropriate in many situations and easy to implement. Characters and strings are stored more or less normally.

9.1.1 Opening and Closing a File

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream. There are three types of streams: input, output, and input/output.

- To create an input stream, you must declare the stream to be of class ifstream.
- To create an output stream, you must declare it as class ofstream.
- Streams that will be performing both input and output operations must be declared as class fstream.

For example, this fragment creates one input stream, one output stream, and one stream capable of both input and output:

```
ifstream in; // input

ofstream out; // output

fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using open(). This function is a member of each of the three stream classes. The prototype for each is shown here:

```
void ifstream::open(const char *filename, ios::openmode mode = ios::in);

void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);

void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

Here, filename is the name of the file; it can include a path specifier. The value of mode determines how the file is opened. It must be one or more of the following values defined by openmode, which is an enumeration defined by ios (through its base class ios_base).

```
ios::app

ios::ate

ios::binary

ios::in

ios::out

ios::trunc
```

You can combine two or more of these values by ORing them together.

- Including ios::app causes all output to that file to be appended to the end. This value can be used only with files capable of output.
- Including ios::ate causes a seek to the end of the file to occur when the file is opened. Although ios::ate causes an initial seek to end-of-file, I/O operations can still occur anywhere within the file.
- The ios::in value specifies that the file is capable of input.
- The ios::out value specifies that the file is capable of output.
- The ios::binary value causes a file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations may take place, such as carriage return/linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur.

- The ios::trunc value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length. When creating an output stream using ofstream, any preexisting file by that name is automatically truncated.

If open() fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded. You can do so by using a statement like this:

```
if(!mystream) {

cout << "Cannot open file.\n";

// handle error

}
```

9.1.2 Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using cin and cout, substitute a stream that is linked to a file.

For example, this program creates a short inventory file that contains each item's name and its cost:

```
#include <iostream>

#include <fstream>

using namespace std;

int main()

{

   ofstream out("INVNTRY"); // output, normal file

        if(!out) {

                cout << "Cannot open INVENTORY file.\n";

                return 1;

        }

  out << "Radios " << 39.95 << endl;

  out << "Toasters " << 19.95 << endl;

  out << "Mixers " << 24.80 << endl;
```

```
  out.close();
  return 0;
}
```

The following program reads the inventory file created by the previous program and displays its contents on the screen:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
ifstream in("INVNTRY"); // input
if(!in) {
cout << "Cannot open INVENTORY file.\n";
return 1;
}
char item[20];
float cost;
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in >> item >> cost;
cout << item << " " << cost << "\n";
in.close();
return 0;
}
```

In a way, reading and writing files by using >> and << is like using the C-based functions fprintf() and fscanf() functions.

9.1.3 getline( )

Another function that performs input is getline() . It is a member of each input stream class. Its prototypes are shown here:

istream &getline(char *_buf_, streamsize _num_);

istream &getline(char *_buf_, streamsize _num_, char _delim_);

Here is a program that demonstrates the getline() function. It reads the contents of a text file one line at a time and displays it on the screen.

```
// Read and display a text file line by line.

#include <iostream>

#include <fstream>

using namespace std;

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

{

if(argc!=2) {

cout << "Usage: Display <filename>\n";

return 1;

}

ifstream in(argv[1]); // input

(!in) {

cout << "Cannot open input file.\n";

return 1;

}

char str[255];

while(in) {

in.getline(str, 255); // delim defaults to '\n'
```

```
if(in) cout << str << endl;

}

in.close();

return 0;

}
```

9.2 Unformatted I/O

While reading and writing formatted text files is very easy, it is not always the most efficient way to handle files. Also, there will be times when you need to store unformatted (raw) binary data, not text. The functions that allow you to do this are described here.

- When performing binary operations on a file, be sure to open it using the **ios::binary** mode specifier.

Although the unformatted file functions will work on files opened for text mode, some character translations may occur. Character translations negate the purpose of binary file operations.

You can write a few numbers to disk using formatted I/O, but if you're storing a large amount of numerical data, it's more efficient to use binary I/O in which numbers are stored as they are in the computer's RAM memory rather than as strings of characters. In binary I/O an integer is always stored in 2 bytes, whereas its text version might be 12345, requiring 5 bytes. Similarly, a float is always stored in 4 bytes, whereas its formatted version might be 6.02314e13, requiring 10 bytes.

The next example shows how an array of integers is written to disk and then read back into memory using binary format. In this two new functions: write(), a member of ofstream, and read(), a member of ifstream are used. These functions think about data in terms of bytes (type char). They don't care how the data is formatted, they simply transfer a buffer full of bytes from and to a disk file. The parameters to write() and read() are the address of the data buffer and its length. The address must be cast to type char, and the length is the length in bytes (characters), *not* the number of data items in the buffer. The following example program shows how an integer file is read and written to using binary mode:

```
// binio.cpp

// binary input and output with integers

#include <fstream.h> // for file streams

const int MAX = 100; // number of ints

int buff[MAX]; // buffer for integers
```

```
void main()
{
  int j;
  for(j=0; j<MAX; j++) // fill buffer with data
    buff[j] = j; // (0, 1, 2, ...)
   // create output stream
  ofstream os("edata.dat", ios::binary);
  // write to it
  os.write( (char*)buff, MAX*sizeof(int) );
  os.close(); // must close it
  for(j=0; j<MAX; j++) // erase buffer
    buff[j] = 0; // create input stream
  ifstream is("edata.dat", ios::binary);
 // read from it
 is.read( (char*)buff, MAX*sizeof(int) );
 for(j=0; j<MAX; j++) // check data
   if( buff[j] != j )
    { cerr << "\nData is incorrect"; return; }
      cout << "\nData is correct";
    }
}
```

The ios::binary must be used as an argument in the second paramter to write() and read() when working with binary data. This is because the default is text mode, which can cause problems some times. For example, in text mode, the '\n' character is expanded into 2 bytes—a carriage return and a linefeed—before being stored to disk. This makes a formatted text file more readable for simple text editors, but causes confusion when applied to binary data because every byte that happens to have the ASCII value 10 is translated into 2 bytes. So far in the examples, there has been no need to close streams explicitly because they are closed automatically when they go out of scope; this invokes

their destructors and closes the associated file. However, in BINIO, because both the output stream os and the input stream is are associated with the same file, EDATA.DAT, the first stream must be closed before the second is opened. For the the close() member function has been used.

9.2.1 Reading and writing unformatted data

9.2.1.1 get() and put()

One way that you may read and write unformatted data is by using the member functions get() and put() . These functions operate on characters. That is, get() will read a character and put() will write a character. Of course, if you have opened the file for binary operations and are operating on a char (rather than a wchar_t stream), then these functions read and write bytes of data. The get() function has many forms, but the most commonly used version is shown here along with put():

```
istream &get(char &ch);

ostream &put(char ch);
```

The get() function reads a single character from the invoking stream and puts that value in ch. It returns a reference to the stream. The put() function writes ch to the stream and returns a reference to the stream.

The following program displays the contents of any file, whether it contains text or binary data, on the screen. It uses the get() function.

```
#include <iostream>

#include <fstream>

using namespace std;

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

{

char ch;

if(argc!=2) {

cout << "Usage: PR <filename>\n";

return 1;

}

ifstream in(argv[1], ios::in | ios::binary);

if(!in) {
```

```
cout << "Cannot open file.";

return 1;

}

while(in) { // in will be false when eof is reached

in.get(ch);

if(in) cout << ch;

}

return 0;

}
```

When the end-of-file is reached, the stream associated with the file becomes false. Therefore, when in reaches the end of the file, it will be false, causing the while loop to stop.

There is actually a more compact way to code the loop that reads and displays a file, as shown here:

```
while(in.get(ch))

cout << ch;
```

This works because get() returns a reference to the stream in, and in will be false when the end of the file is encountered.

9.2.2.2 read( ) and write( )

Another way to read and write blocks of binary data is to use C++'s read() and write() functions. Their prototypes are

```
istream &read(char *buf, streamsize num);

ostream &write(const char *buf, streamsize num);
```

The read() function reads num characters from the invoking stream and puts them in the buffer pointed to by buf. The write() function writes num characters to the invoking stream from the buffer pointed to by buf.

streamsize is a type defined by the C++ library as some form of integer. It is capable of holding the largest number of characters that can be transferred in any one I/O operation. The next program writes a structure to disk and then reads it back in:

```
#include <iostream>
```

```
#include <fstream>
#include <cstring>
using namespace std;
struct status {
char name[80];
double balance;
unsigned long account_num;
};
int main()
{
struct status acc;
strcpy(acc.name, "Ralph Trantor");
acc.balance = 1123.23;
acc.account_num = 34235678;
// write data
ofstream outbal("balance", ios::out | ios::binary);
        if(!outbal) {
                        cout << "Cannot open file.\n";
                        return 1;
                    }
outbal.write((char *) &acc, sizeof(struct status));
outbal.close();
// now, read back;
ifstream inbal("balance", ios::in | ios::binary);
        if(!inbal) {
                        cout << "Cannot open file.\n";
```

```
            return 1;
        }
inbal.read((char *) &acc, sizeof(struct status));
cout << acc.name << endl;
cout << "Account # " << acc.account_num;
cout.precision(2);
cout.setf(ios::fixed);
cout << endl << "Balance: $" << acc.balance;
inbal.close();
return 0;
}
```

As you can see, only a single call to **read()** or **write()** is necessary to read or write the entire structure. Each individual field need not be read or written separately. As this example illustrates, the buffer can be any type of object.

If the end of the file is reached before *num* characters have been read, then **read()** simply stops, and the buffer contains as many characters as were available.

9.3 Special functions

Next some functions which are helpful in making file I/O more verstile are discussed

9.3.1 Detecting EOF

You can detect when the end of the file is reached by using the member function **eof(),** which has this prototype:

```
bool eof( );
```

It returns true when the end of the file has been reached; otherwise it returns false. The following program uses **eof()** to display the contents of a file in both hexadecimal and ASCII.

9.3.2 The ignore( ) Function

You can use the **ignore()** member function to read and discard characters from the input stream. It has this prototype:

```
istream &ignore(streamsize num=1, int_type delim=EOF);
```

It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default). If the delimiting character is encountered, it is not removed from the input stream. Here, **int_type** is defined as some form of integer.

9.3.3 peek( ) and putback( )

You can obtain the next character in the input stream without removing it from that stream by using peek(). It has this prototype:

int_type peek( );

It returns the next character in the stream or EOF if the end of the file is encountered.

You can return the last character read from a stream to that stream by using putback(). Its prototype is

istream &putback(char c);

where c is the last character read.

9.3.4 flush( )

When output is performed, data is not necessarily immediately written to the physical device linked to the stream. Instead, information is stored in an internal buffer until the buffer is full. Only then are the contents of that buffer written to disk.

Its prototype is ostream &flush( );

9.3.5 Random Access

In C++'s I/O system, you perform random access by using the seekg() and seekp() functions. Their most common forms are istream &seekg(off_type *offset*, seekdir *origin*);

ostream &seekp(off_type *offset*, seekdir *origin*);

Here, off_type is an integer type defined by ios that is capable of containing the largest valid value that *offset* can have. seekdir is an enumeration defined by ios that determines how the seek will take place.

The seekg() function moves the associated file's current get pointer *offset* number of characters from the specified *origin*, which must be one of these three values:

ios::beg Beginning-of-file

ios::cur Current location

ios::end End-of-file

The seekp() function moves the associated file's current put pointer *offset* number of characters from the specified *origin*, which must be one of the values just shown.

9.3.6 Obtaining the Current File Position

You can determine the current position of each file pointer by using these functions:

pos_type tellg( );

pos_type tellp( );

Here, pos_type is a type defined by ios that is capable of holding the largest value that either function can return. You can use the values returned by tellg() and tellp() as arguments to the following forms of seekg() and seekp() , respectively.

istream &seekg(pos_type *pos)*;

ostream &seekp(pos_type *pos*);

These functions allow you to save the current file location, perform other file operations, and then reset the file location to its previously saved location.

9.3.7 I/O Status

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an object of type **iostate**, which is an enumeration defined by **ios** that includes the following members.

Name Meaning

ios::goodbit No error bits set

ios::eofbit 1 when end-of-file is encountered; 0 otherwise

ios::failbit 1 when a (possibly) nonfatal I/O error has occurred;

0 otherwise

ios::badbit 1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information. First, you can call the rdstate() function. It has this prototype:

iostate rdstate( );

It returns the current status of the error flags. As you can probably guess from looking at the preceding list of flags, rdstate() returns goodbit when no error has occurred. Otherwise, an error flag is turned on.

9.4 Object I/O

Because C++ is an object-oriented language, it's reasonable to wonder how objects can be written to and read from disk. The next examples show the process. The person class supplies the objects.

Writing an Object to Disk

When an object is written, generally binary mode is used. This writes the same bit configuration to disk that was stored in memory and ensures that numerical data contained in objects is handled properly. The program below creates an object PERS, which asks the user for information about an object of class person and then the information in this object is written to the disk file PERSON.DAT.

```
// saves person object to disk

#include <fstream.h> // for file streams

class person // class of persons

{

protected:

char name[40]; // person's name

int age; // person's age

public:

void getData(void) // get person's data

{

cout << "Enter name: "; cin >> name;

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

}

};

void main(void)

{

person pers; // create a person

pers.getData(); // get data for person

// create ofstream object
```

```
ofstream outfile("PERSON.DAT", ios::binary);

outfile.write( (char*)&pers, sizeof(pers) ); // write to it

}
```

The getData() member function of person is called to prompt the user for information, which it places in the pers object. The contents of the pers object are then written to disk using the write() function. The sizeof operator has been used to find the size of the pers object.

Reading an Object from Disk

In the same way reading an object back from the PERSON.DAT file requires the read() member function.

```
// ipers.cpp

// reads person object from disk

#include <fstream.h> // for file streams

class person // class of persons

{

  protected:

  char name[40]; // person's name

  int age; // person's age

  public:

    void showData(void) // display person's data

    {

      cout << "\n Name: " << name;

      cout << "\n Age: " << age;

    }

};

void main(void)

{

   person pers; // create person variable
```

```
   ifstream infile("PERSON.DAT", ios::binary); // create stream

   infile.read( (char*)&pers, sizeof(pers) ); // read stream

   pers.showData(); // display person

}
```

The output from tis program reflects whatever data the OPERS program placed in the PERSON.DATfile

Summary

- C++ I/O forms an integrated I/O system, which treats the console and file I/O in the same way
- Disk files require a different set of classes than files used with the keyboard and screen.
- There are two basic kinds of disk I/O in C++: formatted and unformatted.
- While reading and writing formatted text files is very easy, it is not always the most efficient way to handle files.
- In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream.
- It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using cin and cout, substitute a stream that is linked to a file.
- If you're storing a large amount of numerical data, it's more efficient to use binary I/O in which numbers are stored as they are in the computer's RAM memory rather than as strings of characters as in formatted I/O
- One way that you may read and write unformatted data is by using the member functions get() and put().
- Another way to read and write blocks of binary data is to use C++'s read() and write() functions.
- When an object is written, generally binary mode is used. This writes the same bit configuration to disk that was stored in memory

Self Check Exercise

1. Write the basic steps to read any text file usin stream I/O.
2. How will you differentiate between formatted and unformatted file I/O.
3. What is the advantage of using unformated file I/O?
4. How a file can be accessed randomly?
5. Explain how the object can be written and read into a file with an example.

Suggested Readings

- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

Lesson No. 10 **Author: Dr. Amandeep Kaur**
Converted into SLM by: Dr. Vishal Singh

Conditional, Iterative and Control Statement

Objectives

After reading this lesson you should be able to:

- Understand the concept decision control statements like if – else and switch
- Understand the concept of iterative statements
- Understand the concept of nested statements
- Understand the concept jump statements like goto, break etc

Introduction

A statement is a part of your program that can be executed. That is, a statement specifies an action. By default the statements in a program are executed sequentially. However, in serious programming situations, we want a set of statements to be executed in one situation, and an entirely different set of statements to be executed in another situation. This kind of situation is dealt by using a decision control statement (if and switch statements)

Also the versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. For example we may want to calculate gross salaries of ten different persons, or want to convert temperatures from centigrade to fahrenheit for 15 different cities. The mechanism, which meets this need, is the loop control statements or iteration statements (while, for, and do-while)
The fundamental C++ control statements that allow your program to do something more than once and to do different things in different circumstances (loops and decisions control statements) are covered in this lesson

10.1 Decision control Statements

C/C++ supports two types of decision statements: if and switch. In addition, the ? operator is an alternative to if in certain circumstances.

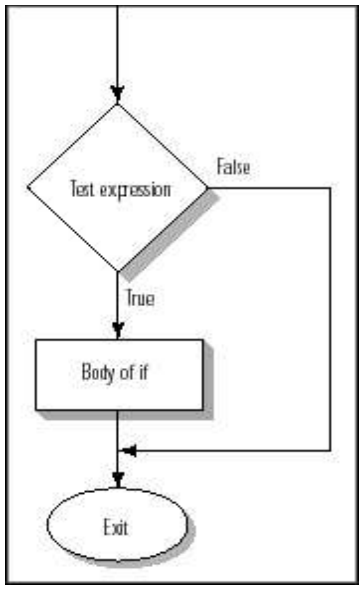
10.1.1 The if Statement

The simplest way to make a decision in C++ is with the if statement. The general form of the if statement is

if (expression) statement;

else statement;

If expression evaluates to true (anything other than 0), the statement or block that forms the target of if is executed; otherwise, the next statement after the target will be executed.



Thus as can be seen from the above figure an if statement consists of the keyword if followed by a test expression in parentheses. The loop body, which follows, consists of either a single statement or multiple statements surrounded by braces. The body of the loop follows immediately after the test expression. Consider the following example

```
if(denominator == 0)
        cout << “Division by zero is illegal”;
```

If a variable called denominator is 0, this fragment causes a message to be displayed. If denominator is not 0, then nothing happens. As with loops, if you use more than one statement in the body of an if statement, you need to surround them with braces.

```
if(choice == 's') // if user chooses “sold dog' option
  {
    cout << “Selling a dog”; // verify I sold the dog
    stand1.SoldOneDog(); // sell the dog
}
```

To check if the number is even?

```
#include<iostream.h>
#include<conio.h>
Int main() {
Int num;
Clrscr();
Cout<<”enter the number  to check”<<endl
Cin>>num;
```

```
If(num/2==0)
{
Cout<<”the number is even”<<endl;
Cout<<”end of the program”<<endl;
}
Getch();
Return();
}
```

10.1.2 The if...else Statement

In a simple if statement, something happens if the condition *is* true, but if the condition is *not* true, nothing happens at all. Suppose we want something to happen either way: one action if the condition is true and a different action if the condition is false. To do this, we will use an if...else statement. Consider the following code it takes the hour of the day, expressed in 24-hour time, and displays it in 12-hour time, with “am” or “pm” as appropriate:

```
if(hours < 12)
    cout << hours << “ am”; // e.g., “7 am” if hours is 7
else
    cout << hours-12 << “ pm”; // e.g., “3 pm” if hours is 15
```

As with other C++ constructions, the if body or the else body may consist of multiple statements surrounded by braces rather than the single statements. Consider another program which will make the if-else statement more clear:

```
/* Magic number program #2. */

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

  int magic; /* magic number */

  int guess; /* user's guess */

  magic = rand(); /* generate  a random magic number */

  printf("Guess the magic number: ");

  scanf("%d", &guess);
```

```
  if(guess == magic)
        printf("** Right **");
  else
        printf("Wrong");
  return 0;
}
```

To check if the number is even or Odd?

```
#include<iostream.h>
#include<conio.h>
Int main() {
Int num;
Clrscr();
Cout<<"enter the number  to check"<<endl;
Cin>>num;
If(num/2==0)
{
Cout<<"the number is even"<<endl;
}
Else
{
    Cout<<"the number is even"<<endl;
    }
    Cout<<"end of the program"<<endl;
    }
    Getch();
    Return();
    }
```

10.1.3 Nested if...else Statements

You can nest if…else statements within one another. Typically, the nested statements end up in the else body rather than in the if body. For example

```
if(age<2)
        cout << "\nInfant";
else
        if(age<18)
                cout << "\nChild";
```

```
else // age >= 18
        cout << "\nAdult";
```

This if…else "ladder" prints an appropriate description of a person's age. In the above example there is an if – else statement nested in the else body.

Program to calculate the greatest to three numbers using nested if-else statement.

```
#include<iostream.h>
#include<conio.h>
Int main() {
Int a,b,c;
Clrscr();
Cout<<"enter the number  to check"<<endl;
Cin>>a>>b>>c;
If(a>b)
{
If(a>c) {
Cout<<"A="<<a<<"is " greatest number"<<endl; }
Else {
Cout<<"C="<<c<<"is " greatest number"<<endl; }
Else if (b>c) {
Cout<<"B="<<b<<"is " greatest number"<<endl; }
Else
{
Cout<<"C="<<c<<"is " greatest number"<<endl; }
Getch();
Return 0;
}
```

10.1.4 The else if Construction

If we consider the previous example again.

```
if(hours < 12)
        cout << hours << " am"; // e.g., "7 am" if hours is 7
else
        cout << hours-12 << " pm"; // e.g., "3 pm" if hours is 15
```

We see that this statement does not handle the occasions when hour is 0 or 12. This problem can be solved using if – else – if ladder.

A common programming construct is the if-else-if ladder, sometimes called the if-else-if staircase because of its appearance. Its general form is

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

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final else is executed. That is, if all other conditional tests fail, the last else statement is performed. If the final else is not present, no action takes place if all other conditions are false.

Solving the above problem using an if…else ladder:

```
if(hours == 0) // first-level indent
  cout << “Midnight”;
else
   if(hours == 12) // second-level indent
     cout << “Noon”;
   else
      if(hours < 12) // third-level indent
        cout << hours << “ am”;
      else
       cout << hours-12 << “ pm”;
```

If the hours is 0, it's midnight, so the program displays an appropriate message and exits from the entire if...else ladder. If it isn't 0, the program goes on to the next if...else. It keep moving to more deeply nested if...else statements until it finds one whose test expression is true or it runs out of possibilities. This arrangement does what I want, but it's hard for humans to read the multiple indentations. Here's a somewhat more easily understood way to rewrite the same code:

```
if(hours == 0)
 cout << "Midnight";
  else if(hours == 12)
   cout << "Noon";
     else if(hours < 12)
       cout << hours << " am";
         else
           cout << hours-12 << " pm";
```

The if that follows each else is simply moved up onto the same line, thus creating a sort of artificial else if construction and removing the multiple levels of indentation. This arrangement not only saves space, it presents a clearer picture of the program's logic (at least, after you've gotten used to it). Notice that the else if construction is not really a part of the syntax of the C++ language; it's merely a way to rewrite an if...else ladder by rearranging the whitespace on the page. You can do this because—as you know—the compiler doesn't care about whitespace.

Program to calculate the division of student according to marks obtained in 4 subjects using if else ladder.

```
#include<iostream.h>
#include<conio.h>
Int main() {
Clrscr();
Int m1,m2,m3,m4;
Cout<<"enter the marks of students in four subjects";
Cin>>m1>>m2>>m3>>m4;
Float per;
Per=m1+m2+m3+m4/4;
If(per>=60)
Cout<<"First Divison"<<endl;
Else if (per>=50 && per<=60)
Cout<<"second Divison"<<endl;
Else if (per>=35 && per<=50)
```

```
 Cout<<"Third  Divison"<<endl;
Else
Cout<<"fail"<<endl;
Getch();
Return 0;
}
```

10.1.5 Ternary operator (?)

You can use the ? operator to replace if-else statements. The ? is called a ternary operator because it requires three operands. It takes the general form

Exp1 ? Exp2 : Exp3

where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. The value of a ? expression is determined as follows: Exp1 is evaluated. If it is true, Exp2 is evaluated and becomes the value of the entire ? expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression. For example,

consider

```
x = 10;

y = x>9 ? 100 : 200;
```

In this example, y is assigned the value 100. If x had been less than 9, y would have received the value 200. The same code written with the if-else statement would be

```
x = 10;

if(x>9) y = 100;

else y = 200;
```

10.1.6 Test Expression

The test expression in an if or an if...else statement can be just simple statements or complicated expressions. But the expression must simply evaluate to either a true or false (zero or nonzero) value. Consider the following example:

Example: To find if Feb has 29th day depending if the given year is a leap year or not and decide if the next day after 28th Feb will be 29th Feb or 1st March

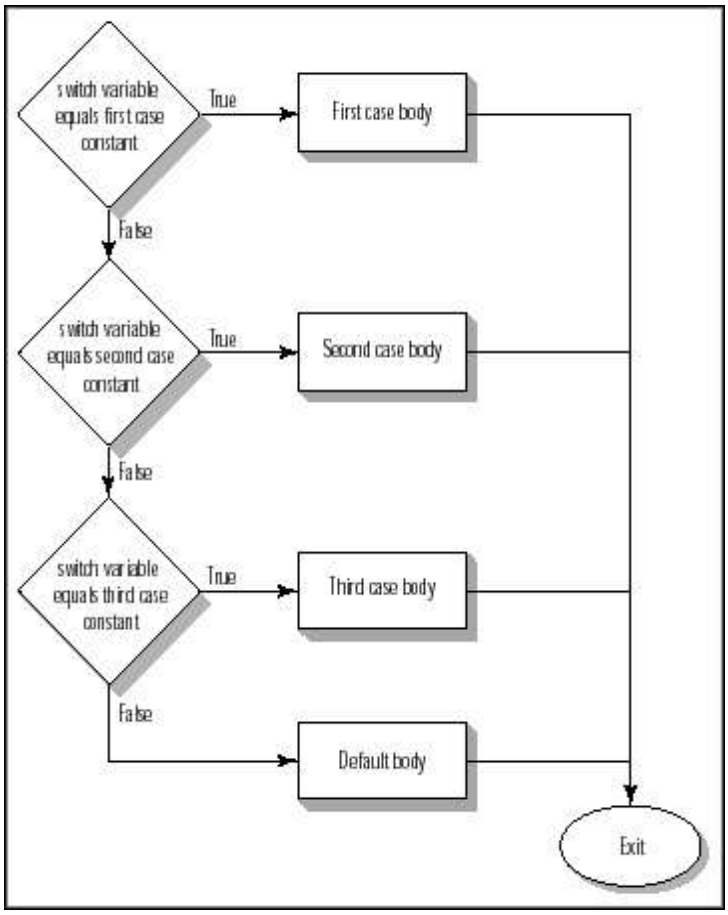
```
// if it's Feb 28 and it's a leap year

if(day==28 && month==2 && year%4==0 && year%100 != 0)

day = 29; // then the next day is the 29th

else // otherwise,
```

```
{
day = 1; // next day is March 1st
month = 3;
}
```

Leap years occur when the year is divisible by 4 (e.g., 1996 is a leap year), but not divisible by 100 (so 1900 is *not* a leap year, although it is divisible by 4. The remainder operator (%) is used to find if the year is divisible by 4 (and by 100) with no remainder. The AND operators make sure that February 29 occurs only when all the conditions are true at once.

10.1.7 The switch Statement

C/C++ has a built-in multiple-branch selection statement, called switch, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The *expression* must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of *expression* is tested, in order, against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with that case is executed until the break statement or the end of the switch statement is reached. The default statement is executed if no matches are found. The default is optional and, if it is not present, no action takes place if all matches fail. The general form of the switch statement is given in the figure below:



Here's an example:

```
switch(diskSpeed)
{
case 33: // if diskSpeed is 33
cout << "Long-playing album";
break;
case 45: // if diskSpeed is 45
cout << "Single-selection";
break;
case 78: // if diskSpeed is 78
cout << "Old single-selection";
```

```
break;

default: // if nothing matches

cout << “Unknown format”;

}
```

The switch statement consists of the keyword switch followed by a variable name in parentheses. The body of the switch statement, enclosed in braces, follows. Within the body are a number of labels, which consist of the keyword case followed by a constant and then the colon. When the value of the switch variable is equal to the constant following a particular case, control will go to the statements following this case label. The above section of code prints different messages depending on the value of the diskSpeed variable. If diskSpeed is 33, control jumps to the label case 33. If diskSpeed is 45, control jumps to the label case 45, and so on. If diskSpeed doesn’t match any of the cases, control jumps to the default label (or, if there is no default, falls through the bottom of the switch). The variable or expression used to determine which label is jumped to (diskSpeed, in this example) must be an integer or a character or must evaluate to an integer or a character. That is, you can use variable names or expressions, such as alpha, j+20, and ch+’0’, as long as alpha, j, and ch already have appropriate values. Once control gets to a label, the statements following the label are executed one after the other from the label onward. In this example, the cout statement will be executed. Then what? If the break weren’t there, control would continue down to the next cout statement, which is *not* what you want. Labels don’t delimit a section of code, they merely name an entry point. The break causes control to break out of the switch entirely.

Program to add, substract, multiply and divide two numbers using switch statement.

```
#include<iostream.h>
#include<conio.h>
Int main() {
Clrscr();
Int a,b;
Cout<<”enter the numbers to process”<<endl;
Cin>>a>>b;
Cout<<”1: Addition”<<endl;
Cout<<”2: Substraction”<<endl;
Cout<<”3: Multiply”<<endl;
Cout<<”4: Divison”<<endl;
Int choice;
Cout<<”enter your choice”<<endl;
Cin>>choice;
Int c;
Switch(choice) {
```

```
Case 1: c=a+b;
Cout<<"the sum of numbers is="<<c<<endl;
Break;
Case 2: c=a-b;
Cout<<"the substradtion  of numbers is="<<c<<endl;
Break;
Case 1: c=a*b;
Cout<<"the Multplication of numbers is="<<c<<endl;
Break;
Case 1: c=a/b;
Cout<<"the divison of numbers is="<<c<<endl;
Break;
Default:
Cout<<"wrong choice"<<endl;
}
Getch();
Return 0;
}
```

Nested switch Statements

You can have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts arise. For example, the following code fragment is perfectly acceptable:

```
switch(x) {

case 1:

        switch(y) {

        case 0: printf("Divide by zero error.\n");

        break;

        case 1: process(x,y);

        }

break;

case 2:

.

.

.
```

10.2 Iteration Statements

In C/C++, and all other modern programming languages, iteration statements (also called loops) allow a set of instructions to be executed repeatedly until a certain condition is reached. This condition may be predefined (as in the for loop), or open-ended (as in the while and do-while loops).

10.2.1 while Loops

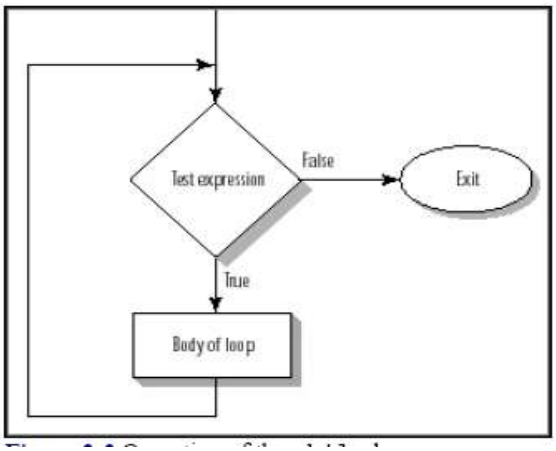
A while loop lets us do something over and over until a *condition* changes. The condition is something that can be expressed by a true/false value. For example, a while loop might repeatedly ask the user to enter a character. It would then continue to cycle until the user enters the character 'q' (for "quit").

Here's an example of a while loop that behaves just this way:

```
while(ch != 'q')

{

cout << "Enter a character: ";

cin >> ch;

}
```

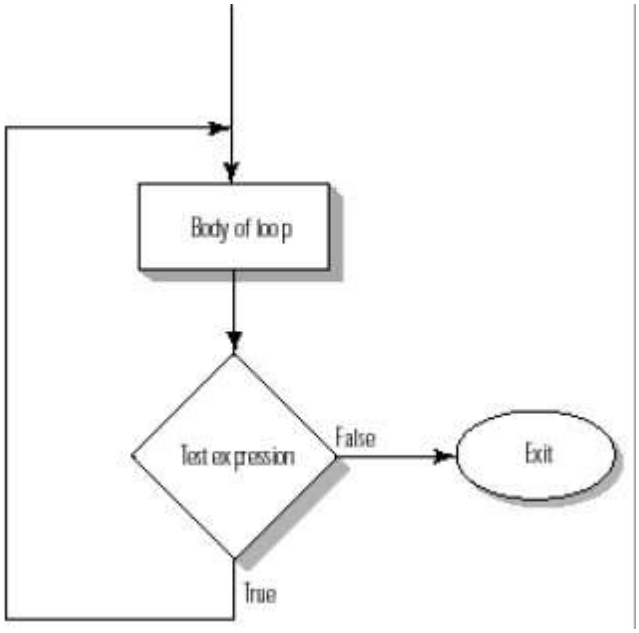
If the user does not press 'q', the loop continues.

A while loop consists of the keyword while followed by a *test expression* (also called a *conditional expression* or *condition*) enclosed in parentheses. The body of the loop is delimited by braces (but no semicolon), just like a function. The structure of while loop is shown in the below given figure:



10.2.2 do Loops

The do loop (often called the do while loop) operates like the while loop except that the test expression is checked *after* the body of the loop is executed. This is nice when you always want something (whatever is in the body of the loop) done at least once, no matter what the initial true/false state of the condition is. The structure of this loop is shown in the next figure:



A do loop begins with the keyword do, followed by the body of the loop in braces, then the keyword while, a test expression in parentheses, and finally a semicolon. The following code will make the concept of do-while more clear

```
do

{

cout << "\nEnter two numbers (to quit, set first to 0): "

cin >> x >> y;

cout << "The sum is " << x + y;

} while(x != 0);
```

10.2.3 for Loops

In both the while and do loops, we usually don't know, at the time of entering the loop, how many times the loop will be executed. The condition that terminates the loop arises spontaneously inside the loop: The user answers 'n'

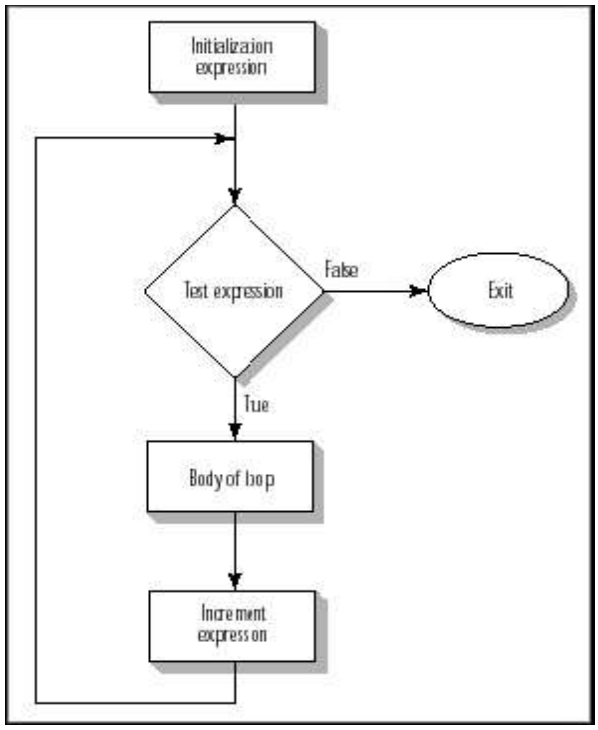
instead of 'y', for example. This is not the case with for loops. In a for loop, the number of times the loop will be executed is (usually) stated at the beginning of the loop. Here's a loop that prints 20 asterisks in a line across the page:

```
int j; // define the loop variable

for(j=0; j<20; ++j); // cycle 20 times

cout << '*'; // print asterisk
```

As we can see from the above example the parentheses following the keyword for contain three different expressions, separated by semicolons. In the most common situation, these three expressions operate on a variable called the *loop variable*, which in this example was j. These three expressions are

- The *initialization expression*, which usually initializes the value of a loop variable.
- The *test expression*, which usually checks the value of the loop variable to see whether to cycle again or to quit the loop.
- The *increment expression*, which usually increments (or decrements) the value of the loop variable.

In the example, the loop variable j is given an initial value of 0, then the test expression is evaluated. If this expression is true (if j is less than 20), the body of the loop is executed and the increment expression is executed (j's value is increased by 1). The concept of for loop will be clearer from the below given figure:



As in the while loop, the test expression is evaluated before the loop body is executed the first time. Thus, the loop body may not be executed at all if the test expression is false to begin with. Also there is no rule that says that the loop variable must be increased by 1 each time through the loop. You can also decrease it by 1:

```
for(j=10; j>0; --j)

cout << j << ' ';
```

which will display

10 9 8 7 6 5 4 3 2 1

or you can increase it or decrease it by any other amount. This code

```
for(j=0; j<100; j=j+10)

cout << j << ' ';
```

will display

0 10 20 30 40 50 60 70 80 90

There is a surprising amount of flexibility in what you can put in the three expressions in a for loop. For example, you can use multiple statements in the initialization expression and the test expression. Here's an example of a for loop with such multiple statements:

```
for(j=0, total=0; j<10; ++j, total=total+j)

cout << total << ' '; // display total
```

This loop prints

0 1 3 6 10 15 21 28 36 45

as in the earlier example. However, here the variable total is set to 0 in the initialization expression instead of before the loop, and increased by j in the increment expression instead of in the loop body. The individual statements in these expressions are separated by commas. Another option is to leave out any or all of the three for loop expressions entirely, retaining only the semicolons. When the conditional expression is absent, it is assumed to be true. Then it behaves like a infinite loop.

```
for( ; ; ) printf("This loop will run forever.\n");
```

Nested Loops

You can nest one loop inside another as shown in the following example

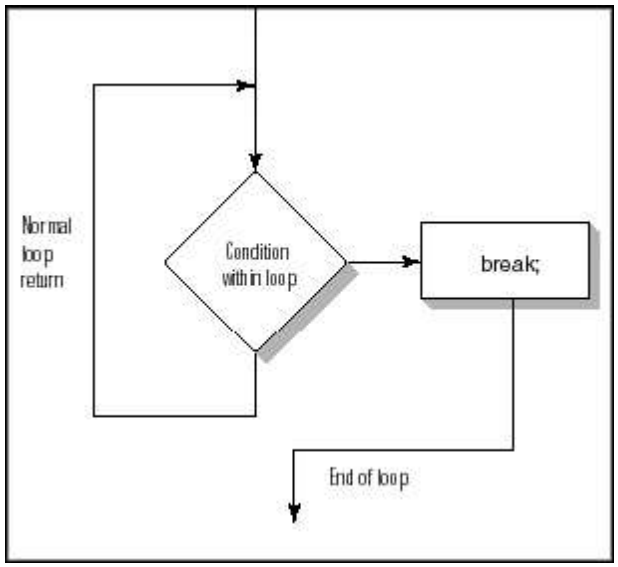
```
for(y=0; y<10; y++) // outer loop, drops down line-by-line

{

for(x=0; x<10; x++) // inner loop, goes across char-by-char

cout << 'X'; // print 'X'

cout << endl; // go to new line

}
```

10.3 Jump Statements

C/C++ has four statements that perform an unconditional branch: return, goto, break, and continue. Of these, return and goto maybe used anywhere in the program. The break and continue statements are used in conjunction with any of the loop statements.

10.3.1 The break Statement

The break statement causes immediate exit from a loop, as shown in following figure



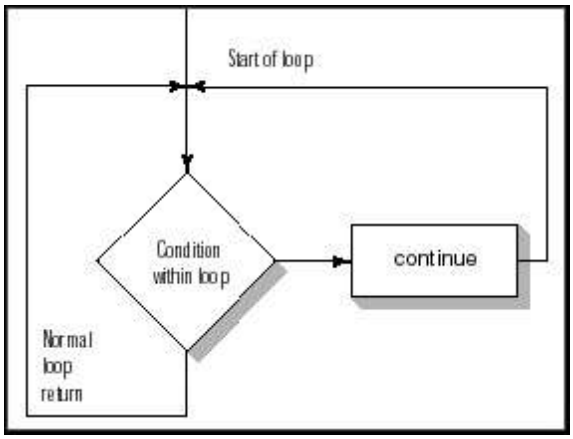
The break statement is often used to handle unexpected or nonstandard situations that arise within a loop. For example, consider the following code fragment that sets the variable isPrime to 1 if an integer n is a prime number or to 0 if n is not a prime number. (A prime number is divisible only by itself and 1.) To tell if n is prime, the straightforward approach is to divide it by all the numbers up to n-1. If any of them divide evenly (with no remainder), then it's not prime.

```
isPrime = 1; // assume n is prime

for(j=2; j<n; ++j) // divide by all integers from 2 to n-1

{

if(n%j == 0) // if evenly divisible,

{

isPrime = 0; // n is not a prime

break; // no point in looping again

}
```

To divide by all the numbers up to n-1, a for loop with appropriate expressions is used. However, if one of the j values does divide evenly, there's no use remaining in the loop and dividing by the remaining j values. So as soon as the program finds the first number that divides evenly, it sets isPrime to 0 and then immediately exit from the loop. The break statement allows you to exit from the loop at any time.

10.3.2 The continue Statement

The continue statement is similar to the break statement in that it is usually activated by an unexpected condition in a loop. However, it returns control to the top of the loop—causing the loop to continue—rather than causing an exit from the loop. Figure below shows how this looks.



Following a continue, control goes back to the top of the loop. Consider the following example:

```
do
{
cout << “Enter dividend: ”;
cin >> dividend;
cout << “Enter divisor: ”;
cin >> divisor;
if(divisor == 0) // if user error,
{
cout << “Divisor can't be zero\n”;
continue; // go back to top of loop
}
cout << “Quotient is ” << dividend / divisor;
cout “\nDo another (y/n)? ”;
```

```
cin >> ch;

} while(ch != 'n');
```

Division by zero is illegal, so if the user enters 0 for the divisor, control goes back to the top of the loop and the program prompts for a new dividend and divisor so the user can try again. To exit from the loop, the user must answer 'n' to the "Do another" question.

10.3.3 The goto Statement

Since C/C++ has a rich set of control structures and allows additional control using break and continue, there is little need for goto. Most programmers' chief concern about the goto is its tendency to render programs unreadable. Nevertheless, although the goto statement fell out of favor some years ago, it occasionally has its uses.

The goto statement requires a label for operation. (A *label* is a valid identifier

followed by a colon.) Furthermore, the label must be in the same function as the goto

that uses it—you cannot jump between functions. The general form of the goto

statement is

goto *label*;

..

.

label:

where *label* is a

10.3.4 The return Statement

The return statement is used to return from a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A return may or may not have a value associated with it. If return has a value associated with it, that value becomes the return value of the function.

Summary

- By default the statements in a program are executed sequentially.
- The fundamental C++ control statements that allow your program to do something more than once and to do different things in different circumstances (loops and decisions control statements) are important for serious programming.
- C/C++ supports two types of decision statements: if and switch.
- The test expression in an if or an if…else statement must simply evaluate to either a true or false (zero or nonzero) value.

- Iteration statements allow a set of instructions to be executed repeatedly until a certain condition is reached,this condition may be predefined, or open-ended.
- C/C++ has four statements that perform an unconditional branch: return, goto, break, and continue.

Self Check Exercise

1. Explain else-if statement with an example.
2. Why be need to use break statement in a switch statement?
3. What is the difference between the continue aand break statement?
4. If all the three for loop expressions are left out and only three semicolons are left how the for loop is going to behave?

Suggested Readings

- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications, 1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

ARRAYS

Objectives

After reading this lesson you should be able to:
Understand the syntax of declaring single dimensional arrays
Understand the concept of accessing the elements of an array
Understand the concept of initializing the arrays
Understand the general concept of passing arrays to functions
Understand the general concept of doing calculations using arrays
Understand the concept of two and multidimensional arrays
Understand the concept of Indexing Pointers

Introduction

Arrays are the most common data structures tused o combine a number of data items into a single unit and are the fundamental data storage mechanism which can be defined as a collection of variables of the same type that are referred to through a common name. Consider the example of an array that creates storage for four integers values to store the ages of four people.

```
int age[4];
```

The int specifies the type of data to be stored, age is the name of the array, and 4 is the size of the array; that is, the maximum number of variables of type int that it will hold.

- A specific element in an array is accessed by an index.
- All arrays consist of contiguous memory locations.

- The lowest address corresponds to the first element and the highest address to the last element.
- The lowest index is zero
- An arrays may be single or multidimensional. Thus we will start our discussion with single dimensional arrays

Arrays are widely used in programming. Sometimes a single-dimensioned array is called a **list** or a **table**. A number of actions can be commonly done on lists and with lists. Lists may be unsorted (unordered) or they may be sorted into some order, numerically low to high or perhaps alphabetically. An unordered list may need to be sorted into some order. A new item may be added to either an ordered or unordered list. A list may be searched for a matching item; this is sometimes called a **table lookup**. Two lists may be related; these are called **parallel arrays**. For example, one array might hold the student id number while the parallel array holds the student grades for a course. In such a case, a specific element of one array corresponds to that same element in another array. That is, element 0 of the id array contains the id of the student whose grade is in the corresponding element 0 of the grades array. In this chapter, we explore these different uses of arrays.

1.1 Single-Dimension Arrays

1.1.1 Declaring an array

An array declaration is very similar to a variable declaration.The general form for declaring a single-dimension array is as follows:

datatype variable_name[size];

Consider the following example

double temps[240];

This defines **temps** to be an array of 240 **double**s.

Here, datatype declares the base type of the array, which is the type of each element in the array, and size defines how many elements the array will hold. For example, to declare a 100-element array called balance of type double, use this statement:

double balance[100];

Each variable stored in an array is called an element. The elements are numbered. These numbers are called index numbers or indexes. Some people also refer to them as subscripts. The index of the first array element is 0, the index of the second is 1, and so on. If the size of the array is n, the last element has the index n-1. For example, in the following example

char p[10];

you are declaring a character array that has ten elements, p[0] through p[9].

11.1.2 Acessing an array

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example,

balance[3] = 12.23;

assigns element number 3 in balance the value 12.23.

Thus an array is accessed by an integer type subscript that can be a constant, variable or an expression. The subscript is also enclosed inside of [ ] brackets. The following are valid array references

temps[0] temps[i] temps[j + 1]

where **i** and **j** are both integer types.

Consider some more examples:

The following are valid.

```
temps[i] = 42; // stores 42 in the ith element
temps[i]++; // adds 1 to the ith element of temps
cin >> temps[i]; // input the ith element of temps
cout << temps[i]; // output the ith element of temps
```

The following are invalid.

```
cin >> temps;
cout << temps;
```

An array cannot be input as a group in a single input statement. All assignment and math operations must be done on an individual element basis. So these are invalid as well.

```
temps += count;
temps++;
```

One array cannot be assigned to another array; instead, a loop must be written to copy each element.

```
double save_temps[10];
save_temps = temps; // illegal - use a loop instead
```

Rather, if a copy of the array is needed, then the following loop accomplishes it.

```
for (i=0; i<10; i++) {
save_temps[i] = temps[i];
}
```

When using subscripts, it is vital that the subscript be within the range defined by the array. In the case of the 240 temperatures, valid subscripts are in the range from 0 to 239. Note a common cause of errors is to attempt to access temps[240], which does not exist as it would be the 241 element. This attempt to access an element t st hat is beyond the bounds of the array is called subscript out of bounds. Thus C/C++ has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code. As the programmer, it is your job to provide bounds checking where needed. For example, this code will compile without error, but is incorrect because the for loop will cause the array count to be overrun.

```
int count[10], i;
/* this causes count to be overrun */
for(i=0; i<100; i++) count[i] = i;
```

The following example will help you to understand arrays in a better way. The following program loads an integer array with the numbers 0 through 99:

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

```
int x[100]; /* this declares a 100-integer array */
int t;
/* load x with values 0 through 99 */
for(t=0; t<100; ++t) x[t] = t;
/* display contents of x */
        for(t=0; t<100; ++t) printf("%d ", x[t]);
        return 0;
}
```

The amount of storage required to hold an array is directly related to its type and size. For a single-dimension array, the total size in bytes is computed as shown here:

total bytes = sizeof(base type) x size of array

Single-dimension arrays are essentially lists of information of the same type that are stored in contiguous memory locations in index order. For example, You can generate a pointer to the first element of an array by simply specifying the array name, without any index. Thus, the following program fragment assigns p the address of the first element of sample:

```
int *p;
int sample[10];
p = sample;
```

You can also specify the address of the first element of an array using the & operator. For example, sample and &sample[0] both produce the same results. However, in professionally written C/C++ code, you will almost never see &sample[0].

11.1.3 Initializing Array Elements

When an array is defined or declared, it may also be initialized if desired. Suppose that one needed to define an array of **totals** to hold the total sales from each of ten departments within a store. One could code

```
double totals[10];
```

However, in practice, the first usage of any element in the array is likely to be of the form

```
total[j] = total[j] + sales;
```

This requires that each element in the array be initialized to zero. You can set array elements to a value when you first define the array. The initialization syntax in general is

data type variable-name[limit] = {value0, value1, value2, … valuen};

Initialization of arrays begins with a begin brace { and ends with an end brace }. The syntax provides for giving each element its unique initial value. In the example of store **totals**, one could code

```
double totals[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

Here's another example:

```
int coins[6] = { 1, 5, 10, 25, 50, 100 };
```

The first element is initialized to 1, the second to 5, and so on. The equal sign connects the list of values to the array definition, the values are separated by commas, and the list is delimited by braces.

However, C++ provides a default value of 0 for all elements not specifically initialized. In reality, one would define totals this way

```
double totals[10] = {0};
```

This tells the compiler to place a 0 in the first element and then to place a default value (that happens to also be 0) in all remaining elements. Once you stop providing initial values, all remaining elements are defaulted to a 0 starting value. Values neither can be omitted nor can there be more values than array elements. The following coding is illegal, since once you stop providing values (after element zero in this case) you cannot later on resume providing explicit values.

```
double totals[10] = {0, , , 0};// error - cannot omit values
```

An important feature of array initialization is that you don't need to count how many items are being initialized unless you want to. The definition

```
int coins[ ] = { 1, 5, 10, 25, 50, 100 };
```

works inspite of not specifying the array size. The compiler counts how many values there are and uses this for the array size. If the specified array size disagrees with initialization values that are actually on the list in that case:

- If there are more values than the array size specified, the compiler gives error.
- If there are fewer values in the list than the array size, the compiler will simply fill out the balance of the array with 0s.

Thus if you want to initialize an array—of *any* size—with all 0s, then the following statement can be used

```
int anarray[10] = { 0 }; // initialize 10 ints to 0
```

The first element is initialized to 0 explicitly and the remaining 9 are initialized to 0 because no value is given for them. If they are not initialized, the elements of arrays declared *inside* a function will have random (*garbage*) values. Arrays declared outside of a function or class—that is, as external variables—are initialized to zero automatically.

11.2 Passing Single-Dimension Arrays to Functions

Very commonly, arrays are passed to functions whose purpose is to encapsulate the entire array's input operation. The main() function is then left with just a single function call to function which loads the array, thereby streamlining main().The function to load an array must be passed two values the array to fill up and the maximum number of elements in the array also it is good if it returns the actual number of elements inputted. Thus, if we were to write a loadArray() function to load the array of temperatures, the only coding in main() would now be

```
int numTemps = loadArray (temps, MAXTEMPS);
```

When coding the prototype for **loadArray()**, two different notations can be used.

```
int loadArray (double temps[ ], int limit);
int loadArray (double* temps, int limit);
```

In the first version, the **[ ]** tells the compiler that the preceding symbol **temps** is not just a **double** but is also an array of **double**s, which therefore means that this parameter is really just the memory address where the array begins. Notice one significant detail when setting up the prototype of a function that has an array as a parameter. The actual

array bound is not used as it was when defining the array. Compare the two statements in **main()**:

```
// defines storage for the array
double temps[MAXTEMPS];
// prototype for the loadArray function
int loadArray (double temps[], int limit);
```

The first defines the array and allocates space for MAXTEMPS doubles. The second line says that within **loadArray()**, the **temps** parameter is an array of **double**s. The compiler never cares how many elements are in this passed array, only that it is an array of **double**s. One could also have coded the prototype as

```
int loadArray (double temps[MAXTEMPS], int limit);
```

However, the compiler simply ignores the array bounds; only the **[ ]** is important, for the **[ ]** tells the compiler that the symbol is an array. As usual, the programmer must be vigilant in making sure that the array bound is not exceeded by passing in that maximum number of elements as the second parameter and using it to avoid inputting too many values.

In the case of second version a pointer to the first location of array temp is passed as a parameter to the function.

Next, let's see how the **loadArray()** function could be implemented. In this array values are read from a text file named myfile as show below:

```
int loadArray (double temps[], int limit) {
  ifstream infile;
  infile.open ("myfile.txt");
    if (!infile) {
    cerr << "Cannot open myfile.txt\n";
    exit (1);
    }
     int i = 0;
   while (i<limit && infile >> temps[i]) {
      i++;
   }
 // guard against too many temperatures in the file
    if (i == limit && infile >> ws && infile.good()) {
      cerr << "Error: too many temperatures\n";
      infile.close ();
      exit (2);
    }
// guard against bad data in the input file
    else if (!infile.eof() && infile.fail ()) {
    cerr << "Error: bad data in the file\n";
    infile.close ();
    exit (3);
    }
```

```
// set the number of temps in the array on this run
  infile.close ();
  return i;
  }
```

In case of error situations, such as, being unable to open the file, or having too much data. An error message is displayed, program termination is invoked by calling the **exit()** function, which takes one parameter, the integer return code to give to DOS.

Thus in case of C/C++, you cannot pass an entire array as an argument to a function. You can, however, pass to the function a pointer to an array by specifying the array's name without an index. Also if a function receives a single-dimension array, you may declare its formal parameter in one of three ways: as a pointer, as a sized array, or as an unsized array. For example, to receive **i**, a function called **func1()** can be declared as

```
void func1(int *x) /* pointer */
{
.
.
.
}
```

- or

```
void func1(int x[10]) /* sized array */
{
.
.
.
}
```

- or finally as

```
void func1(int x[]) /* unsized array */
{
.
.
.
}
```

All three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. The first declaration actually uses a pointer. The second employs the standard array declaration. In the final version, a modified version of an array declaration simply specifies that an array of type **int** of some length is to be received. As you can see, the length of the array doesn't matter as far as the function is concerned because C/C++ performs no bounds checking. In fact, as far as the compiler is concerned,

```
void func1(int x[32])
{
.
```

```
.
.
}
```

also works because the compiler generates code that instructs **func1()** to receive a pointer—it does not actually create a 32-element array.

11.3 Working with Arrays

A program often does one or more calculations using the array element values. Quite often,every element in the array must be used. Consider the example of finding the average temperature when temperatures are stored in an array. If the program adds each temperature to a **sum** and then divides that **sum** by the number of temperatures, the average can be found. In the following coding, it is assumed that **numTemps** holds the actual number of temperatures inputted in this execution of the program.

```
double sum = 0;
for (i=0; i< numTemps; i++) {
   sum += temps[i];
}
double average = sum / numTemps;
```

Next consider how the highest and the lowest temperatures are found? Assuming that high and low variables are to contain the results, each element in the array, that is, each temperature, must be compared to these two values. If the next temperature is greater than or less than the corresponding high or low temperature, then that value replaces the current high or low value. This can be done as follows.

```
double high = temps[0];
double low = temps[0];
for (i=1; i<numTemps; i++) {
        if (temps[i] > high)
                high = temps[i];
        if (temps[i] < low)
                low = temps[i];
}
// here high and low have the largest and smallest
// temperatures in the array
```

Here is a common coding error that yields a mostly-working program. Can you spot the error?

```
double high = 0;
double low = 0;
for (i=0; i<numTemps; i++) {
        if (temps[i] > high)
                high = temps[i];
        if (temps[i] < low)
                low = temps[i];
}
```

Frequently all the elements of the array must be displayed. Writing another loop, outputting each element in turn does this.

```
cout << setprecision (1);
for (i=0; i<numTemps; i++) {
        cout << setw(12) << temps[i] << endl;
}
```

11.4 Two-Dimensional Arrays

C/C++ supports multidimensional arrays. You can create arrays of as many dimensions as you like, and each dimension can be a different size. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array can be looked at as an array of arrays. which we can say is, essentially, an array of one-dimensional arrays. To declare a two-dimensional integer array **d** of size 10,20, you would write

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

Pay careful attention to the declaration. Some other computer languages use commas to separate the array dimensions; C/C++, in contrast, places each dimension in its own set of brackets. Similarly, to access point 1,2 of array **d**, you would use d[1][2]

The following example loads a two-dimensional array with the numbers 1 through 12 and prints them row by row.

```
#include <stdio.h>
int main(void)
{
int t, i, num[3][4];
for(t=0; t<3; ++t)
  for(i=0; i<4; ++i)
    num[t][i] = (t*4)+i+1;
    /* now print them out */
  for(t=0; t<3; ++t) {
     for(i=0; i<4; ++i)
         printf("%3d ", num[t][i]);
         printf("\n");
     }
 return 0;
}
```

In this example, num[0][0] has the value 1, num[0][1] the value 2, num[0][2] the value 3, and so on. The value of num[2][3] will be 12. You can visualize the num array as shown here:

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column. This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory. In the case of a two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:

bytes = size of 1st index x size of 2nd index x sizeof(base type)

Therefore, assuming 4-byte integers, an integer array with dimensions 10,5 would have 10 x 5 x 4 or 200 bytes allocated.

When a two-dimensional array is used as an argument to a function, only a pointer to the first element is actually passed. However, the parameter receiving a two-dimensional array must define at least the size of the rightmost dimension. (You can specify the left dimension if you like, but it is not necessary.) The rightmost dimension is needed because the compiler must know the length of each row if it is to index the array correctly. For example, a function that receives a two-dimensional integer array with dimensions 10,10 is declared like this:

```
void func1(int x[][10])
    {
    .
    .
    .
    }
```

The compiler needs to know the size of the right dimension in order to correctly execute expressions such as x[2][4] inside the function. If the length of the rows is not known, the compiler cannot determine where the third row begins. The following short program uses a two-dimensional array to store the numeric grade for each student in a teacher's classes. The program assumes that the teacher has three classes and a maximum of 30 students per class. Notice the way the array **grade** is accessed by each of the functions.

```
/* A simple student grades database. */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#define CLASSES 3
#define GRADES 30
int grade[CLASSES][GRADES];
void enter_grades(void);
int get_grade(int num);
void disp_grades(int g[][GRADES]);
int main(void)
{
char ch, str[80];
for(;;) {
do {
printf("(E)nter grades\n");
printf("(R)eport grades\n");
printf("(Q)uit\n");
gets(str);
ch = toupper(*str);
} while(ch!='E' && ch!='R' && ch!='Q');
```

```
switch(ch) {
case 'E':
enter_grades();
break;
case 'R':
disp_grades(grade);
break;
case 'Q':
exit(0);
}
}
return 0;
}
/* Enter the student's grades. */
void enter_grades(void)
{
int t, i;
for(t=0; t<CLASSES; t++) {
printf("Class # %d:\n", t+1);
for(i=0; i<GRADES; ++i)
grade[t][i] = get_grade(i);
}
}
/* Read a grade. */
int get_grade(int num)
{
char s[80];
printf("Enter grade for student # %d:\n", num+1);
gets(s);
return(atoi(s));
}
/* Display grades. */
void disp_grades(int g[][GRADES])
{
int t, i;
for(t=0; t<CLASSES; ++t) {
printf("Class # %d:\n", t+1);
for(i=0; i<GRADES; ++i)
printf("Student #%d is %d\n", i+1, g[t][i]);
}
}
```

11.5 Multidimensional Arrays

C/C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler. The general form of a multidimensional array declaration is *type name*

[*Size1*][*Size2*][*Size3*]. . .[*SizeN*];

Arrays of more than three dimensions are not often used because of the amount of memory they require. For example, a four-dimensional character array with dimensions 10,6,9,4 requires 10 * 6 * 9 * 4 or 2,160 bytes. If the array held 2-byte integers, 4,320 bytes would be needed. If the array held **double**s (assuming 8 bytes per **double**), 17,280 bytes would be required. The storage required increases exponentially with the number of dimensions. For example, if a fifth dimension of size 10 was added to the preceding array, then 172, 800 bytes would be required.

In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array. When passing multidimensional arrays into functions, you must declare all but the leftmost dimension. For example, if you declare array **m** as int m[4][3][6][5];a function, **func1()** , that receives **m**, would look like this:

```
void func1(int d[][3][6][5])
{
.
.
.
}
```

Of course, you can include the first dimension if you like.

11.6 Indexing Pointers

In C/C++, pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array. For example, consider the following array.

```
char p[10];
```

The following statements are identical:

```
p
&p[0]
```

Put another way,

```
p == &p[0]
```

evaluates to true because the address of the first element of an array is the same as the address of the array. As stated, an array name without an index generates a pointer. Conversely, a pointer can be indexed as if it were declared to be an array. For example, consider this program fragment:

```
int *p, i[10];
p = i;
p[5] = 100; /* assign using index */
*(p+5) = 100; /* assign using pointer arithmetic */
```

Both assignment statements place the value 100 in the sixth element of **i**. The first statement indexes **p**; the second uses pointer arithmetic. Either way, the result is the same. This same concept also applies to arrays of two or more dimensions. The cast of the pointer to the array into a pointer of its base type is necessary in order for the pointer arithmetic to operate properly. Pointers are sometimes used to access arrays because pointer arithmetic is often faster than array indexing. A two-dimensional array can be reduced to a pointer to an array of one dimensional arrays. Therefore, using a separate pointer variable is one easy way to use pointers to access elements within a row of a two-dimensional array. The following function illustrates this technique. It will print the contents of the specified row for the global integer array **num**:

```
int num[10][10];
.
.
.
void pr_row(int j)
{
int *p, t;
p = (int *) &num[j][0]; /* get address of first
element in row j */
for(t=0; t<10; ++t) printf("%d ", *(p+t));
}
```

You can generalize this routine by making the calling arguments be the row, the row length, and a pointer to the first array element, as shown here:

```
void pr_row(int j, int row_dimension, int *p)
{
int t;
p = p + (j * row_dimension);
for(t=0; t<row_dimension; ++t)
printf("%d ", *(p+t));
}
.
.
.
void f(void)
{
int num[10][10];
pr_row(0, 10, (int *) num); /* print first row */
}
```

Arrays of greater than two dimensions may be reduced in a similar way. For example, a three-dimensional array can be reduced to a pointer to a two-dimensional array, which can be reduced to a pointer to a single-dimension array. Generally, an *n*-dimensional array can be reduced to a pointer and an (*n*-1)-dimensional array. This new

array can be reduced again with the same method. The process ends when a single-dimension array is produced.

Summary

- Arrays are the most common way to combine a number of data items into a single unit
- can be defined as a collection of variables of the same type that are referred to through a common name.
- All arrays consist of contiguous memory locations
- An array cannot be input as a group in a single input statement
- One array cannot be assigned to another array
- The amount of storage required to hold an array is directly related to its type and size
- C/C++ has no bounds checking on arrays
- Single-dimension arrays are essentially lists of information of the same type
- You can generate a pointer to the first element of an array by simply specifying the array name
- You can also specify the address of the first element of an array using the & operator.
- C/C++ supports multidimensional arrays. You can create arrays of as many dimensions as you like, and each dimension can be a different size.
- Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column.
- When a two-dimensional array is used as an argument to a function and if the parameter receiving a two-dimensional array is used then it must define at least the size of the rightmost dimension.
- Arrays of more than three dimensions are not often used because of the amount of memory they require.
- Accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array because of higher time taken to compute each index

Self Check Exercise

1. Why arrays are needed?
2. How are array elements initialized explain in detail?
3. What is meant by indexing pointers?
4. What is the disadvantage of using multidimensional arrays?

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- Vic Broquard, C++ for Computer Science and Engineering, 4th Edition.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.

- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

ARRAYS AS CHARACTER STRINGS

Contents

Objective

After reading this lesson you should be able to:

- Understand the general concept of null terminated strings
- Understand how to input and output the strings
- Understand how to pass strings to functions
- Understand the different functions that manipulate null-terminated strings
- Understand the concept of array of strings
- Understand the general concept of the string class

- Understand the concept of constructors , operators and member functions related to string class

Introduction

By far the most common use of the one-dimensional array is as a character string. C++ supports two types of strings. The first is the null-terminated string, which is a null-terminated character array. (A null is zero.) Thus a null-terminated string contains the characters that comprise the string followed by a null. This is the only type of string defined by C, and it is still the most widely used. Sometimes null-terminated strings are called C-strings. C++ also defines a string class, called string, which provides an object-oriented approach to string handling. When declaring a character array that will hold a null-terminated string, you need to declare it to be one character longer than the largest string that it is to hold. For example, to declare an array str that can hold a 10-character string, you would write

char str[11];

This makes room for the null at the end of the string. When you use a quoted string constant in your program, you are also creating a null-terminated string. A string constant is a list of characters enclosed in double quotes. For example,

"hello there" You do not need to add the null to the end of string constants manually—the compiler does this for you automatically. In C++, nearly all strings ever used are null-terminated. However, the language provides for non-null terminated strings as well.

12.1 Null terminated Strings

Consider the following example the string literal value “Sam” is used

```
cout << "Sam";
```

then the output stream displays

Sam

But how does the computer know how long the literal string is and where it ends? The answer is that the literal “Sam” is a null-terminated string consisting of four bytes containing 'S', 'a', 'm', and 0. This null-terminator can be represented by a numerical 0 or by the escape sequence \0. Most all of the C++ functions that take a character string as an argument expect that string to be null-terminated. The null-terminator marks the end of the characters in the variable. For example, suppose that a variable is defined to hold a person's name as follows

```
char name[21];
```

This definition is saying that the maximum number of characters that can be stored is twenty plus one for the null-terminator. This maximum length is different from the number of characters actually stored when a person's name is entered. For example, assume that the program has inputted the name as follows

```
cin >> name;
```

Assume that the user has entered “Sam” from the keyboard. In this instance, only four of the possible twenty-one are in use with the null terminator in the 4th character. Make sure you understand the distinction between the maximum size of a string and the actual size in a specific instance. When defining a string variable, it makes good programming sense not to hard-code the array bounds but to use a const int, just as is

done with other kinds of arrays. Thus, the name variable ought to have been coded like this

```
const int NAMELENGTH = 21;
int main () {
char name[NAMELENGTH];
```

If at a later date you decide that twenty character names are too short, it is a simple matter of changing the constant value and recompiling.

12.2 Initializing strings

When character string variables are defined, they can also be initialized. Two forms are possible. Assume that when the name variable is defined, it should be given the value of "Sam."

- Following the initialization syntax for any other kind of array, one could code
  ```
  char name[NAMELENGTH] = {'S', 'a', 'm', '\0'};
  ```
  Here each specific character is assigned its starting value; do not fail to include the null terminator.
- However, the compiler allows a string to be initialized with another literal string as follows
  ```
  char name[NAMELENGTH] = "Sam";
  ```

Clearly this second form is much more convenient. With all forms of arrays, when defining and initializing an array, it is permissible to omit the array bounds and let the compiler determine how many elements the array must have based on the number of initial values you provide. Thus, the following is valid.

```
char name[] = "Sam";
```

However, in general, this approach is error prone because the compiler allocates an array just large enough to hold the literal "Sam." That is, the **name** array is only four characters long. What would happen if later on one attempted to input another name that needed more characters? Thus always provide the array bounds whenever possible.

12.3 Inputting Character Strings

The extraction operator can be used to input character strings. The specific rules of string extraction follow those for the other data types we have covered. It skips over whitespace to the first non-whitespace character, inputs successive characters storing them into successive bytes in the array until the extraction operator encounter whitespace or the end of file. Lastly, it stores the null terminator. There are two aspects of this input operation that frequently make the use of the extraction operator useless. Notice that the extraction operator does not permit a blank to be in the string. Suppose that you prompted the user to input their name and age and then used cin to input them as follows

```
cin >> name >> age;
```

What results if the user enters the following data?

```
Sam Spade 25
```

The input stream goes into the bad or fails state. It inputs the characters "Sam" and stores them along with the trailing null-terminator into the name field. It skips over the blank and attempts to input the character S of Spade into the age integer and goes immediately into the bad state. One way to get around the extraction operator's

disadvantages is to use either the get() or getline() function. The get() function can be used in one of two ways. Note: while I am using cin in these examples, any ifstream instance can be used as well.

```
cin.get (string variable, sizeof (string variable));
cin.get (string variable, sizeof (string variable), delimiter character);
```

These input all characters from the current position in the stream until either the maximum number of characters including the null terminator has been read or EOF or the delimiter is found. By default the delimiter is a new line code. The delimiter is not extracted but remains in the input stream.

```
cin.getline (string variable, sizeof (string variable));
cin.getline (string variable, sizeof (string variable), delimiter character);
```

This function works the same way except the delimiter is removed from the input stream but never stored in the string variable. It also defaults to the new line code.

In the input set of data or file, all character strings are the same length, the maximum. Shorter strings have blanks added onto the end of the character series to fill out the maximum length.

12.4 Outputting Character Strings

Outputting strings presents a different set of problems, one is of spacing and alignment. In most of the cases, the insertion operator handles the output of strings quite well. In the most basic form one might output a line of the cost record as follows

```
cout << setw (10) << itemnumber
<< setw (10) << quantity
<< description
<< setw (10) << cost << endl;
```

If the entire program output consisted of one line, the above is fine. Usually, the output consists of many lines, columnarly aligned. If so, the above fails utterly. With a string, the insertion operator outputs all of the characters up to the null terminator. It does not output the null terminator.

Left alignment must be used when displaying strings. Right alignment must be used when displaying numerical data. The alignment is easily changed by using the setf() function.

```
cout << setw (10) << itemnumber
<< setw (10) << quantity;
cout.setf (ios::left, ios::adjustfield);
cout << setw (30) << description;
cout.setf (ios::right, ios::adjustfield);
cout << setw (10) << cost << endl;
```

In the call to setf(), the second parameter ios::adjustfield clears all the justification flags — that is, turns them off. Then left justification is turned on. Once the string is output, the second call to setf() turns right justification back on for the other numerical data. It is vital to use the ios::adjustfield second parameter. The Microsoft implementation of the ostream contains two flags, one for left and one for right justification. If the left justification flag is on, then left justification occurs. Since there are two separate flags,

when setting justification, failure to clear all the flags can lead to the weird circumstance in which both left and right justification flags are on.

12.5 Passing a String to a Function

When passing a string to a function, the prototype of the string is just like that of any other array. Suppose that we have a PrintRecord() function whose purpose was to display one cost record. The description string must be passed. The prototype of the PrintRecord() function is void PrintRecord (const char description[],... and the main() function could invoke it as

PrintRecord (description,...

Recall that the name of an array is always the memory address of the first element, or a pointer. Sometimes you may see the prototype for a string using pointer notation instead of array notation.

void PrintRecord (const char* description, ...

These are entirely equivalent notations when passing a string to a function. Remember, if a function is not going to alter the caller's character string, it should have the const qualifier.

12.6 Working with Strings

C/C++ supports a wide range of functions that manipulate null-terminated strings. There are many bult-in string functions. The prototypes of all of these are in <string>. The data type size_t is really an unsigned integer. The most common are as given in the table below:

Name	**Function**
strcpy(*s1*, *s2*)	Copies *s2* into *s1*.
strcat(*s1*, *s2*)	Concatenates *s2* onto the end of *s1*.
strlen(*s1*)	Returns the length of *s1*.
strcmp(*s1*, *s2*)	Returns 0 if *s1* and *s2* are the same; less than 0 if *s1*<*s2*; greater than 0 if *s1*>*s2*.
strchr(*s1*, *ch*)	Returns a pointer to the first occurrence of *ch* in *s1*.
strstr(*s1*, *s2*)	Returns a pointer to the first occurrence of *s2* in *s1*.

12.6.1 Copying Strings

The older function to copy a string is strcpy(). Its prototype is char* strcpy (char* destination, const char* source); It copies all characters including the null terminator of the source string, placing them in the destination string. In the previous example where we wanted to copy the currentName into the previousName field, we code

strcpy (previousName, currentName);

Of course, the destination string should have sufficient characters in its array to store all the characters contained in the source string. If not, a memory overlay occurs. For example, if one has defined the following two strings

char source[20] = "Hello World";

char dest[5];
If one copies the source string to the destination string, memory is overlain.
strcpy (dest, source);
In the above case only first five bytes will be written into dest and memory overlay will take place in case of the characters “World” and the null terminator. Consider the following example
char* strcpy (char* desString, size_t maxDestSize, const char* srcString);
All bytes of the srcString are copied into the destination string, including the null terminator. The function returns the desString memory address. It aborts the program if destination is too small.

12.6.2 Getting the Actual Number of Characters Currently in a String

The next most frequently used string function is strlen(), which returns the number of bytes that the string currently contains. Suppose that we had defined
char name[21] = "Sam";
If we code the following
int len = strlen (name); // returns 3 bytes
int size = sizeof (name); // returns 21 bytes
then the strlen(name) function would return 3. Notice that strlen() does NOT count the null terminator. The sizeof(name) gives the defined number of bytes that the variable contains or 21 in this case. Notice the significant difference. Between these two operations.

12.6.3 Concatenating or Joining Two Strings into One Larger String

Again, there is a new version of this function in .NET 2005. The older function is the strcat() function which appends one string onto the end of another string forming a concatenation of the two strings. Suppose that we had defined
char drive[3] = "C:";
char path[_MAX_PATH] = "\\Programming\\Samples";
char name[_MAX_PATH] = "test.txt";
char fullfilename[_MAX_PATH];
In reality, when users install an application, they can place it on nearly any drive and nearly any path. However, the application does know the filename and then has to join the pieces together. The objective here is to join the filename components into a complete file specification so that
the fullfilename field can then be passed to the ifstream open() function. The sequence would be
strcpy (fullfilename, drive); // copy the drive string
strcat (fullfilename, path); // append the path

12.6.4 For Comparing String (strcmp and stricmp)

strcmp and stricmp (string compare, case sensitive and case insensitive respectively)
Prototype: int strcmp (const char* string1, const char* string2);
int stricmp (const char* string1, const char* string2);
strcmp does a case sensitive comparison of the two strings, beginning with the first character of each string. It returns 0 if all characters in both strings are the same. It

returns a negative value if the different character in string1 is less than that in string2. It returns a positive value if it is larger.

12.6.5 To Search String for First Occurrence of the Character (strchr)

Prototype: char* strchr (const char* srcString, int findChar);

It returns the memory address or char* of the first occurrence of the findChar in the srcString. If findChar is not in the srcString, it returns NULL or 0.

Example: char s1[10] = "Burr";

char* found = strchr (s1, 'r');

returns the memory address of the first letter r character, so that found[0] would give you that 'r'.

12.6.6 To Search String1 for the First Occurrence of Particular String (strstr)

Prototype: char* strstr (const char* string1, const char* findThisString);

It returns the memory address (char*) of the first occurrence of findThisString in string1 or NULL (0) if it is not present.

Example: char s1[10] = "abcabc";

char s2[10] = "abcdef";

char* firstOccurrence = strstr (s1, "abc");

It finds the first abc in s1 and firstOccurrence has the same memory address as s1, so that s1[0] and firstOccurrence[0] both contain the first letter 'a' of the string

char* where = strstr (s2, "def");

Here where contains the memory address of the 'd' in the s2

12.6.7 To Convert String to Lowercase or Upper case (strlwr and strupr)

Strlwr convert a string to lowercase

Prototype: char* strlwr (char* string);

All uppercase letters in the string are converted to lowercase letters. All others are left untouched.

Example: char s1[10] = "Hello 123";

strlwr (s1);

Yields "hello 123" in s1 when done.

Strupr convert a string to uppercase

Any lowercase letters in the string are converted to uppercase; all others are untouched.

Example: char s1[10] = "Hello 123";

strupr (s1);

When done, s1 contains "HELLO 123"

12.6.8 To Reverse the String (strrev)

Prototype: char* strrev (char* string);

Reverses the characters in a string.

Example: char s1[10] = "Hello";

strrev (s1);

When done, string contains "olleH"

These functions use the standard header file string.h. (C++ programs can also use the new-style header <cstring>)

The following program illustrates the use of these string functions:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
char s1[80], s2[80];
gets(s1);
gets(s2);
printf("lengths: %d %d\n", strlen(s1), strlen(s2));
if(!strcmp(s1, s2)) printf("The strings are equal\n");
strcat(s1, s2);
printf("%s\n", s1);
strcpy(s1, "This is a test.\n");
printf(s1);
if(strchr("hello", 'e')) printf("e is in hello\n");
if(strstr("hi there", "hi")) printf("found hi");
return 0;
}
```

If you run this program and enter the strings "hello" and "hello", the output is

```
lengths: 5 5
The strings are equal
hellohello
This is a test.
e is in hello
found hi
```

Be sure to use the logical operator **!** to reverse the condition, as just shown, if you are testing for equality. Although C++ now defines a string class, null-terminated strings are still widely used in existing programs. They will probably stay in wide use because they offer a high level of efficiency and afford the programmer detailed control of string operations. However, for many simple string-handling chores, C++'s string class provides a convenient alternative.

12.7 Arrays of Strings

It is not uncommon in programming to use an array of strings. For example, the input processor to a database may verify user commands against an array of valid commands. To create an array of null-terminated strings, use a two-dimensional character array. The size of the left index determines the number of strings and the size of the right index specifies the maximum length of each string. The following code declares an array of 30 strings, each with a maximum length of 79 characters char str_array[30][80];

It is easy to access an individual string: You simply specify only the left index. For example, the following statement calls gets() with the third string in str_array.

```
gets(str_array[2]);
```

The preceding statement is functionally equivalent to gets(&str_array[2][0]);
but the first of the two forms is much more common in professionally written
C/C++ code.

To better understand how string arrays work, study the following short program, which uses a string array as the basis for a very simple text editor:

```
/* A very simple text editor. */
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
int main(void)
{
register int t, i, j;
printf("Enter an empty line to quit.\n");
for(t=0; t<MAX; t++) {
printf("%d: ", t);
gets(text[t]);
if(!*text[t]) break; /* quit on blank line */
}
for(i=0; i<t; i++) {
for(j=0; text[i][j]; j++) putchar(text[i][j]);
putchar('\n');
}
return 0;
}
```

This program inputs lines of text until a blank line is entered. Then it redisplays each line one character at a time.

12.8 The String Class

This is the second way of handling strings by using class object of type string. Actually, the string class is a specialization of a more general template class called basic_string. In fact, there are two specializations of basic_string: string, which supports 8-bit character strings, and wstring, which supports wide-character strings. Since 8-bit characters are by far the most commonly used in normal programming, string is the version of basic_string examined here. Null-terminated strings cannot be manipulated by any of the standard C++ operators. Nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];
s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings.

To have access to the string class, you must include <string> in your program.

The string class is very large, with many constructors and member functions. Also, many member functions have multiple overloaded forms. For this reason, it is not possible to

look at the entire contents of string in this chapter. nstead, we will examine several of its most commonly used features. Once you have a general understanding of how string works, you can easily explore the rest of it on your own.

12.8.1 Reasons for the Inclusion of the Standard String Class

- consistency (a string now defines a data type)
- convenience (you may use the standard C++ operators)
- safety (array boundaries will not be overrun).

But null-terminated strings are still the most efficient way in which to implement strings. However, when speed is not an overriding concern, using the new string class gives you access to a safe and fully integrated way to manage strings.

12.8.2 String Class Constructors

The string class supports several constructors. The prototypes for three of its most commonly used ones are shown here:

```
string( );
string(const char *str);
string(const string &str);
```

The first form creates an empty string object. The second creates a string object from the null-terminated string pointed to by str. This form provides a conversion from null-terminated strings to string objects. The third form creates a string from another string.

12.8.3 String Class Operators

A number of operators that apply to strings are defined for **string** objects through the following table which gives the operator name and meaning:

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[ ]	Subscripting
<<	Output
>>	Input

These operators allow the use of string objects in normal expressions and eliminate the need for calls to functions such as strcpy() or strcat() , for example. In general, you can mix string objects with normal, null-terminated strings in expressions. For example, a string object can be assigned a null-terminated string. The + operator can be used to concatenate a string object with another string object or a string object with a C-style string.

12.8.4 String Member Functions

Although most simple string operations can be accomplished using the string operators, more complex or subtle ones are accomplished using string member functions. While string has far too many member functions to discuss them all, we will examine several of the most common.

- **String Manipulations**
  - **assign()** function

    To assign one string to another, use the assign() function. Two of its forms are shown here:

    string &assign(const string &strob, size_type start, size_type num);

    string &assign(const char *str, size_type num);

    In the first form, num characters from strob beginning at the index specified by start will be assigned to the invoking object. In the second form, the first num characters of the null-terminated string str is assigned to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the **=** to assign one entire string to another. You will need to use the **assign()** function only when assigning a partial string.
  - **append()** function

    You can append part of one string to another using the append() member function.

    Two of its forms are shown here:

    string &append(const string &strob, size_type start, size_type num);

    string &append(const char *str, size_type num);

    Here, num characters from strob beginning at the index specified by start will be appended to the invoking object. In the second form, the first num characters of the null-terminated string str are appended to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the + to append one entire string to another. You will need to use the append() function only when appending a partial string.
  - **insert()** and **replace() function**

    You can insert or replace characters within a string using insert() and replace() function .

    The prototypes for their most common forms are shown here:

    string &insert(size_type start, const string &strob);

    string &insert(size_type start, const string &strob,

    size_type insStart, size_type num);

    string &replace(size_type start, size_type num, const string &strob);

string &replace(size_type start, size_type orgNum, const string &strob,
size_type replaceStart, size_type replaceNum);

The first form of insert() inserts strob into the invoking string at the index specified by start. The second form of insert() function inserts num characters from strob beginning at insStart into the invoking string at the index specified by start. the first form of replace() replaces num characters from the invoking string, with strob. The second form replaces orgNum characters, beginning at start, in the invoking string with the replaceNum characters from the string specified by strob beginning at replaceStart.

- **erase() function**

You can remove characters from a string using erase() function . One of its forms is shown here:

string &erase(size_type start = 0, size_type num = npos);

It removes num characters from the invoking string beginning at start. A reference to the invoking string is returned.

- **Searching a String**

The string class provides several member functions that search a string, including find() and rfind() . Here are the prototypes for the most common versions of these functions:

size_type find(const string &strob, size_type start=0) const;
size_type rfind(const string &strob, size_type start=npos) const;

Beginning at start, find() searches the invoking string for the first occurrence of the string contained in strob. If found, find() returns the index at which the match occurs within the invoking string. If no match is found, then npos is returned. rfind() is the opposite of find() . Beginning at start, it searches the invoking string in the reverse direction for the first occurrence of the string contained in strob (i.e, it finds the last occurrence of strob within the invoking string). If found, rfind() returns the index at which the match occurs within the invoking string. If no match is found, npos is returned.

- **Comparing Strings**

To compare the entire contents of one string object to another, you will normally use the overloaded relational operators described earlier. However, if you want to compare a portion of one string to another, you will need to use the compare() member function, shown here:

int compare(size_type start, size_type num, const string &strob) const;

Here, num characters in strob, beginning at start, will be compared against the invoking string. If the invoking string is less than strob, compare() will return less than zero. If the invoking string is greater than strob, it will return greater than zero. If strob is equal to the invoking string, compare() will return zero.

- **Obtaining a Null-Terminated String**

Although string objects are useful in their own right, there will be times when you will need to obtain a null-terminated character-array version of the string. For example, you might use a string object to construct a filename. However, when opening a file, you will need to specify a pointer to a standard, null-terminated string. To solve this problem, the member function c_str() is provided. Its prototype is shown here:

```
const char *c_str( ) const;
```

This function returns a pointer to a null-terminated version of the string contained in the invoking string object. The null-terminated string must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the string object.

Summary

- The most common use of the one-dimensional array is as a character string
- C++ supports two types of strings (null-terminated strings and string class)
- When character string variables are defined, they can also be initialized.
- The extraction operator can be used to input character strings.
- The insertion operator handles the output of strings quite well.
- C/C++ supports a wide range of functions that manipulate null-terminated strings.
- To create an array of null-terminated strings, use a two-dimensional character array.
- This is the second way of handling strings by using class object of type string.
- Reasons for the Inclusion of the Standard String Class is consistency, convenience, and safety
- A number of operators that apply to strings are defined for string objects
- Although most simple string operations can be accomplished using the string operators, more complex or subtle ones are accomplished using string member functions.

Self Check Exercise

1. How many types of strings are supported by C++
2. What are the various techniques to initialize null terminated strings
3. How can extraction operator be used to input character strings
4. Can the string be reversed if yes how?
5. Give main reasons for the Inclusion of the standard string Class
6. Explain some member functions used for string manipulation

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- Vic Broquard, C++ for Computer Science and Engineering, 4th Edition.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.

- Bjarne Strautrup, “ The C++ Programming Language”, Addison-Wesley Publication Co.,2001.
- Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

STRUCTURES AND UNIONS

Objectives

After reading this lesson you should be able to:
Understand the general concept structure and unions
Understand the concept enumeration and user defined data type
Understand the concept of bit-fields

Introduction

The C++ language gives six ways to create a custom data type. Out of these five ways are inherited from C and one is added by C++.The six methods given below:

- The structure, which is a grouping of variables under one name and is called a compound data type. (The terms aggregate or conglomerate are also commonly used.)
- The bit-field, which is a variation on the structure and allows easy access to individual bits.
- The union, which enables the same piece of memory to be defined as two or more different types of variables.
- The enumeration, which is a list of named integer constants.

- The typedef keyword, which defines a new name for an existing type.
- C++ supports all of the above and adds classes.

This lesson concerns the detailed study of structures and unions. In C++, structures and unions have both object-oriented and non-object-oriented attributes. Also bit fields, enumerations are discussed.

13.1 Structure

A structure is a collection of variables referenced under one name, providing a Convenient means of keeping related information together. Or it can be defined as follows:

A data structure is a set of diverse types of data that may have different lengths grouped together under a unique declaration. A structure declaration forms a template that may be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called members. (Structure members are also commonly referred to as elements or fields.) It has the following form:

```
struct struc_name {
data_type1 element1;
data_type2 element2;
data_type3 element3;
.
.
} object_name;
```

Notice that the declaration is terminated by a semicolon. This is because a structure declaration is a statement. where struc_name is a name for the the structure type and the optional parameter object_name is a valid identifier (or identifiers) for structure object instantiations. Within curly brackets { } are the variables of different data types which the structure compose of. Consider the following example

```
struct furniture {
char name [30];
float price;
} ;
furniture chair, table;
```

We have first defined the structure furniture with two fields: name and price, each of a different type. We have then used the name of the structure type to declare two objects of that type: chair and table. Once declared, furniture has become a new valid data type like the fundamental ones int, char or short. And since then we have been able to declare objects (variables) of that type. The optional field object_name that can go at the end of the structure declaration serves to directly declare objects of the structure type. For example, to declare the structure variables chair, table

```
struct {
char name [30];
float price;
} chair, table ;
```

But in the above case it will not be possible to declare more objects of this type struture later on as we have not given any name to the structure.

Notice that the declaration is terminated by a semicolon. This is because a structure declaration is a statement. The type name of the structure is addr. As such, addr identifies this particular data structure and is its type specifier. At this point, no variable has actually been created. Only the form of the data has been defined. When you define a structure, you are defining a compound variable type, not a variable. Not until you declare a variable of that type does one actually exist. In C, to declare a variable (i.e., a physical object) of type addr, write struct addr addr_info;
This declares a variable of type addr called addr_info. In C++, you may use this shorter form.
addr addr_info;

As you can see, the keyword struct is not needed. In C++, once a structure has been declared, you may declare variables of its type using only its type name, without preceding it with the keyword struct. The reason for this difference is that in C, a structure's name does not define a complete type name. In fact, Standard C refers to a structure's name as a tag. In C, you must precede the tag with the keyword struct when declaring variables. However, in C++, a structure's name is a complete type name and may be used by itself to define variables. Keep in mind, however, that it is still perfectly legal to use the C-style declaration in a C++ program.

When a structure variable (such as addr_info) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members. You may also declare one or more structure variables when you declare a structure.
For example,

```
struct addr {
char name[30];
char street[40];
char city[20];
char state[3];
unsigned long int zip;
} addr_

info, binfo, cinfo;
```

defines a structure type called addr and declares variables addr_info, binfo, and cinfo of that type.

If you only need one structure variable, the structure type name is not needed. That means that

```
struct {
char name[30];
char street[40];
char city[20];
char state[3];
unsigned long int zip;
} addr_info;
```

declares one variable named addr_info as defined by the structure preceding it.

The general form of a structure declaration is
struct struct-type-name {
type member-name;
type member-name;
type member-name;
.. .
} structure-variables;
where either struct-type-name or structure-variables may be omitted, but not both.

13.1.1 Accessing Structure Members

Individual members of a structure are accessed through the use of the operator (usually called the dot operator). For example, the following code assigns the ZIP code 12345 to the zip field of the structure variable addr_info declared earlier:

```
addr_info.zip = 12345;
```

The structure variable name followed by a period and the member name references that individual member. The general form for accessing a member of a structure is structure-name.member-name

Therefore, to print the ZIP code on the screen, write printf("%d", addr_info.zip);

This prints the ZIP code contained in the zip member of the structure variable addr_info. In the same fashion, the character array addr_info.name can be used to call gets() , as shown here:

```
gets(addr_info.name);
```

This passes a character pointer to the start of name. Since name is a character array, you can access the individual characters of addr_info.name by indexing name. For example, you can print the contents of addr_info.name one character at a time by using the following code:

```
register int t;
        for(t=0; addr_info.name[t]; ++t)
        putchar(addr_info.name[t]);
```

Structure Assignments

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the value of each member separately. The following program illustrates structure assignments:

```
#include <stdio.h>
int main(void)
{
struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
printf("%d", y.a);
return 0;
```

}
After the assignment, y.a will contain the value 10.

13.1.2 Arrays of Structures

Perhaps the most common usage of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type addr, defined earlier, write

```
struct addr addr_info[100];
```

This creates 100 sets of variables that are organized as defined in the structure addr. To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
printf("%d", addr_info[2].zip);
```

Like all array variables, arrays of structures begin indexing at 0.

13.1.3 Passing Structures to Functions

This section discusses passing structures and their members to functions.

Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable (unless, of course, that element is compound, such as an array). For example, consider this structure:

```
struct fred
{
char x;
int y;
float z;
char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x); /* passes character value of x */
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

If you wish to pass the address of an individual structure member, put the & operator before the structure name. For example, to pass the address of the members of the structure mike, write

```
func(&mike.x); /* passes address of character x */
func2(&mike.y); /* passes address of integer y */
func3(&mike.z); /* passes address of float z */
func4(mike.s); /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

Remember that the & operator precedes the structure name, not the individual member name. Note also that s already signifies an address, so no & is required.

13.1.4 Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

When using a structure as a parameter, remember that the type of the argument must match the type of the parameter. For example, in the following program both the argument arg and the parameter parm are declared as the same type of structure.

```
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
printf("%d", parm.a);
}
```

As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had struct_type been declared inside main() (for example), then it would not have been visible to f1().

As just stated, when passing structures, the type of the argument must match the type of the parameter. It is not sufficient for them to simply be physically similar; their type names must match. For example, the following version of the preceding program is incorrect and will not compile because the type name of the argument used to call f1() differs from the type name of its parameter.

```
/* This program is incorrect and will not compile. */
#include <stdio.h>
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
/* Define a structure similar to struct_type,
```

```
but with a different name. */
struct struct_type2 {
int a, b;
char ch;
} ;
void f1(struct struct_type2 parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg); /* type mismatch */
return 0;
}
void f1(struct struct_type2 parm)
{
printf("%d", parm.a);
}
```

13.1.5 Structure Pointers

C/C++ allows pointers to structures just as it allows pointers to any other type of variable. However, there are some special aspects to structure pointers that you should know.

- **Declaring a Structure Pointer**

Like other pointers, structure pointers are declared by placing * in front of a structure variable's name. For example, assuming the previously defined structure addr, the following declares addr_pointer as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Remember, in C++ it is not necessary to precede this declaration with the keyword struct.

- **Using Structure Pointers**

There are two primary uses for structure pointers: to pass a structure to a function using call by reference, and to create linked lists and other dynamic data structures that rely on dynamic allocation. This chapter covers the first use. There is one major drawback to passing all but the simplest structures to functions: the overhead needed to push the structure onto the stack when the function call is executed. (Recall that arguments are passed to functions on the stack.) For simple structures with few members, this overhead is not too great. If the structure contains many members, however, or if some of its members are arrays, run-time performance may degrade to unacceptable levels. The solution to this problem is to pass only a pointer to the function. When a pointer to a structure is passed to a function, only the address of the structure is pushed on the stack. This makes for very fast function calls. A second advantage, in some cases, is when a function needs to reference the actual structure used as the argument, instead of a copy. By passing a pointer, the function can modify the contents of the structure used in the call. To find the address of a structure variable, place the & operator before the structure's name. For example, given the following fragment:

```
struct bal {
float balance;
char name[80];
} person;
struct bal *p; /* declare a structure pointer */
```
then
```
p = &person;
```
places the address of the structure person into the pointer p.

To access the members of a structure using a pointer to that structure, you must use the −> operator. For example, this references the balance field:

```
p->balance
```
The −> is usually called the arrow operator, and consists of the minus sign followed by a greater-than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointer to the structure. To see how a structure pointer can be used, examine this simple program, which prints the hours, minutes, and seconds on your screen using a software timer.
```
/* Display a software timer. */
#include <stdio.h>
#define DELAY 128000
struct my_time {
int hours;
int minutes;
int seconds;
} ;
void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);
int main(void)
{
struct my_time systime;
systime.hours = 0;
systime.minutes = 0;
systime.seconds = 0;
for(;;) {
update(&systime);
display(&systime);
}
return 0;
}
void update(struct my_time *t)
{
```

```
t->seconds++;
if(t->seconds==60) {
t->seconds = 0;
t->minutes++;
}
if(t->minutes==60) {
t->minutes = 0;
t->hours++;
}
if(t->hours==24) t->hours = 0;
delay();
}
void display(struct my_time *t)
{
printf("%02d:", t->hours);
printf("%02d:", t->minutes);
printf("%02d\n", t->seconds);
}
void delay(void)
{
long int t;
/* change this as needed */
for(t=1; t<DELAY; ++t) ;
}
```

The timing of this program is adjusted by changing the definition of DELAY. As you can see, a global structure called my_time is defined but no variable is declared. Inside main() , the structure systime is declared and initialized to 00:00:00. This means that systime is known directly only to the main() function. The functions update() (which changes the time) and display() (which prints the time) are passed the address of systime. In both functions, their arguments are declared as a pointer to a my_time structure.

Inside update() and display() , each member of systime is accessed via a pointer. Because update() receives a pointer to the systime structure, it can update its value. For example, to set the hours back to 0 when 24:00:00 is reached, update() contains this line of code:

```
if(t->hours==24) t->hours = 0;
```

This tells the compiler to take the address of t (which points to systime in main() and to reset hours to zero. Remember, use the dot operator to access structure elements when operating on the structure itself. When you have a pointer to a structure, use the arrow operator.

13.1.6 Arrays and Structures Within Structures

A member of a structure may be either a simple or compound type. A simple member is one that is of any of the built-in data types, such as integer or character. You have already seen one type of compound element: the character arrays used in addr. Other

compound data types include one-dimensional and multidimensional arrays of the other data types and structures. A member of a structure that is an array is treated as you might expect from the earlier examples. For example, consider this structure:

```
struct x {
int a[10][10]; /* 10 x 10 array of ints */
float b;
} y;
```

To reference integer 3,7 in a of structure y, write y.a[3][7]

When a structure is a member of another structure, it is called a nested structure. For example, the structure address is nested inside emp in this example:

```
struct emp {
struct addr address; /* nested structure */
float wage;
} worker;
```

Here, structure emp has been defined as having two members. The first is a structure of type addr, which contains an employee's address. The other is wage, which holds the employee's wage. The following code fragment assigns 93456 to the zip element of address.

```
worker.address.zip = 93456;
```

As you can see, the members of each structure are referenced from outermost to innermost. Standard C specifies that structures may be nested to at least 15 levels. Standard C++ suggests that at least 256 levels of nesting be allowed.

13.2 Bit-Fields

Unlike some other computer languages, C/C++ has a built-in feature called a bit-field that allows you to access a single bit. To access individual bits, C/C++ uses a method based on the structure. In fact, a bit-field is really just a special type of structure member that defines how long, in bits, the field is to be. Bit-fields can be useful for a number of reasons, such as:

- If storage is limited, you can store several Boolean (true/false) variables in one byte.
- Certain devices transmit status information encoded into one or more bits within a byte.
- Certain encryption routines need to access the bits within a byte.

Although these tasks can be performed using the bitwise operators, a bit-field can add more structure (and possibly efficiency) to your code. To access individual bits, C/C++ uses a method based on the structure. In fact, a bit-field is really just a special type of structure member that defines how long, in bits, the field is to be. The general form of a bit-field definition is

```
struct struct-type-name {
type name1 : length;
type name2 : length;
..
.
type nameN : length;
```

} variable_list;

Here, type is the type of the bit-field and length is the number of bits in the field. A bit-field must be declared as an integral or enumeration type. Bit-fields of length 1 should be declared as unsigned, because a single bit cannot have a sign. Bit-fields are frequently used when analyzing input from a hardware device. For example, the status port of a serial communications adapter might return a status byte organized like this:

Bit	**Meaning When Set**
0	Change in clear-to-send line
1	Change in data-set-ready
2	Trailing edge detected
3	Change in receive line
4	Clear-to-send
5	Data-set-ready
6	Telephone ringing
7	Receiving signal

You can represent the information in a status byte using the following bit-field:

```
struct status_type {
unsigned delta_cts: 1;
unsigned delta_dsr: 1;
unsigned tr_edge: 1;
unsigned delta_rec: 1;
unsigned cts: 1;
unsigned dsr: 1;
unsigned ring: 1;
unsigned rec_line: 1;
} status;
```

You might use a routine similar to that shown here to enable a program to determine when it can send or receive data.

```
status = get_port_status();
if(status.cts) printf("clear to send");
if(status.dsr) printf("data ready");
```

To assign a value to a bit-field, simply use the form you would use for any other type of structure element. For example, this code fragment clears the ring field:

```
status.ring = 0;
```

As you can see from this example, each bit-field is accessed with the dot operator. However, if the structure is referenced through a pointer, you must use the –> operator. You do not have to name each bit-field. This makes it easy to reach the bit you want, bypassing unused ones. For example, if you only care about the cts and dsr bits, you could declare the status_type structure like this:

```
struct status_type {
unsigned : 4;
unsigned cts: 1;
unsigned dsr: 1;
```

} status;

Also, notice that the bits after dsr do not need to be specified if they are not used. It is valid to mix normal structure members with bit-fields. For example,

```
struct emp {
struct addr address;
float pay;
unsigned lay_off: 1; /* lay off or active */
unsigned hourly: 1; /* hourly pay or wage */
unsigned deductions: 3; /* IRS deductions */
};
```

defines an employee record that uses only 1 byte to hold three pieces of information: the employee's status, whether the employee is salaried, and the number of deductions. Without the bit-field, this information would have taken 3 bytes. Bit-fields have certain restrictions. You cannot take the address of a bit-field. Bitfields cannot be arrayed. They cannot be declared as static. You cannot know, from machine to machine, whether the fields will run from right to left or from left to right; this implies that any code using bit-fields may have some machine dependencies. Other restrictions may be imposed by various specific implementations, so check the user manual for your compiler.

13.3 Unions

A union is a memory location that is shared by two or more different variables,generally of different types, at different times. Declaring a union is similar to declaring a structure. Its general form is union union-type-name

```
{
type member-name;
type member-name;
type member-name;
.
..
} union-variables;
```

For example:

```
union u_type {
int i;
char ch;
};
```

This declaration does not create any variables. You may declare a variable either by placing its name at the end of the declaration or by using a separate declaration statement. In C, to declare a union variable called cnvt of type u_type using the definition just given, write

```
union u_type cnvt;
```

When declaring union variables in C++, you need use only the type name, you don't need to precede it with the keyword union. For example, this is how cnvt is declared in C++:

```
u_type cnvt;
```

In C++, preceding this declaration with the keyword union is allowed, but redundant. In C++, the name of a union defines a complete type name. In C, a union name is its tag and it must be preceded by the keyword union. (This is similar to the situation with structures described earlier. In cnvt, both integer i and character ch share the same memory location. Of course, i occupies 2 bytes (assuming 2-byte integers) and ch uses only 1.

When a union variable is declared, the compiler automatically allocates enough storage to hold the largest member of the union. For example (assuming 2-byte integers), cnvt is 2 bytes long so that it can hold i, even though ch requires only 1 byte.

To access a member of a union, use the same syntax that you would use for structures: the dot and arrow operators. If you are operating on the union directly, use the dot operator. If the union is accessed through a pointer, use the arrow operator. For example, to assign the integer 10 to element i of cnvt, write

```
cnvt.i = 10;
```

In the next example, a pointer to cnvt is passed to a function:

```
void func1(union u_type *un)
{
un->i = 10; /* assign 10 to cnvt using
function */
}
```

Using a union can aid in the production of machine-independent (portable) code. Because the compiler keeps track of the actual sizes of the union members, no unnecessary machine dependencies are produced. That is, you need not worry about the size of an int, long, float, or whatever. Unions are used frequently when specialized type conversions are needed because you can refer to the data held in the union in fundamentally different ways. For example, you may use a union to manipulate the bytes that comprise a double in order to alter its precision or to perform some unusual type of rounding.

To get an idea of the usefulness of a union when nonstandard type conversions are needed, consider the problem of writing a short integer to a disk file. The C/C++ standard library defines no function specifically designed to write a short integer to a file. While you can write any type of data to a file using fwrite(), using fwrite() incurs excessive overhead for such a simple operation. However, using a union you can easily create a function called putw() , which writes the binary representation of a short integer to a file one byte at a time. (This example assumes that short integers are 2 bytes long.) To see how, first create a union consisting of one short integer and a 2-byte character array:

```
union pw {
short int i;
char ch[2];
};
```

Now, you can use pw to create the version of putw() shown in the following program.

```
#include <stdio.h>
union pw {
```

```
short int i;
char ch[2];
};
int putw(short int num, FILE *fp);
int main(void)
{
FILE *fp;
fp = fopen("test.tmp", "wb+");
putw(1000, fp); /* write the value 1000 as an integer */
fclose(fp);
return 0;
}
int putw(short int num, FILE *fp)
{
union pw word;
word.i = num;
putc(word.ch[0], fp); /* write first half */
return putc(word.ch[1], fp); /* write second half */
}
```

Although putw() is called with a short integer, it can still use the standard function putc() to write each byte in the integer to a disk file one byte at a time.

C++ supports a special type of union called an anonymous union

13.4 Enumerations and User Defined Data Types

An enumeration is a set of named integer constants that specify all the legal values a variable of that type may have. Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is penny, nickel, dime, quarter, half-dollar, dollar. Enumerations are defined much like structures; the keyword enum signals the start of an enumeration type. The general form for enumerations is

enum enum-type-name { enumeration list } variable_list;

Here, both the type name and the variable list are optional. (But at least one must be present.) The following code fragment defines an enumeration called coin:

```
enum coin { penny, nickel, dime, quarter,
half_dollar, dollar};
```

The enumeration type name can be used to declare variables of its type. In C, the following declares money to be a variable of type coin.

```
enum coin money;
```

In C++, the variable money may be declared using this shorter form:

```
coin money;
```

In C++, an enumeration name specifies a complete type. In C, an enumeration name is its tag and it requires the keyword enum to complete it. (This is similar to the situation as it applies to structures and unions, described earlier.) Given these declarations, the following types of statements are perfectly valid:

```
money = dime;
```

```
if(money==quarter) printf("Money is a quarter.\n");
```

The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they may be used anywhere that an integer may be used. Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore,

```
printf("%d %d", penny, dime);
```

displays 0 2 on the screen.

You can specify the value of one or more of the symbols by using an initializer.

Do this by following the symbol with an equal sign and an integer value. Symbols that appear after initializers are assigned values greater than the previous initialization value. For example, the following code assigns the value of 100 to quarter:

```
enum coin { penny, nickel, dime, quarter=100,
half_dollar, dollar};
```

Now, the values of these symbols are

```
penny 0
nickel 1
dime 2
quarter 100
half_dollar 101
dollar 102
```

One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case. For example, the following code fragment will not perform as desired:

```
/* this will not work */
money = dollar;
printf("%s", money);
```

Remember, dollar is simply a name for an integer; it is not a string. For the same reason, you cannot use this code to achieve the desired results:

```
/* this code is wrong */
strcpy(money, "dime");
```

That is, a string that contains the name of a symbol is not automatically converted to that symbol. Actually, creating code to input and output enumeration symbols is quite tedious (unless you are willing to settle for their integer values). For example, you need the following code to display, in words, the kind of coins that money contains:

```
switch(money) {
case penny: printf("penny");
break;
case nickel: printf("nickel");
break;
case dime: printf("dime");
break;
case quarter: printf("quarter");
break;
```

```
case half_dollar: printf("half_dollar");
break;
case dollar: printf("dollar");
}
```

Sometimes you can declare an array of strings and use the enumeration value as an index to translate that value into its corresponding string. For example, this code also outputs the proper string:

```
char name[][12]={
"penny",
"nickel",
"dime",
"quarter",
"half_dollar",
"dollar"
printf("%s", name[money]);
```

Of course, this only works if no symbol is initialized, because the string array must be indexed starting at 0. Since enumeration values must be converted manually to their human-readable string values for I/O operations, they are most useful in routines that do not make such conversions. An enumeration is often used to define a compiler's symbol table, for example. Enumerations are also used to help prove the validity of a program by providing a compile-time redundancy check confirming that a variable is assigned only valid values.

13.5 Using sizeof to Ensure Portability

You have seen that structures and unions can be used to create variables of different sizes, and that the actual size of these variables may change from machine to machine. The sizeof operator computes the size of any variable or type and can help eliminate machine-dependent code from your programs. This operator is especially useful where structures or unions are concerned. For the following discussion, assume an implementation, common to many C/C++ compilers, that has the sizes for data types shown here:

Type Size in Bytes
char 1
int 4
double 8

Therefore, the following code will print the numbers 1, 4, and 8 on the screen:

```
char ch;
int i;
double f;
printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

The size of a structure is equal to or greater than the sum of the sizes of its members. For example,

```
struct s {
char ch;
int i;
double f;
} s_var;
```

Here, sizeof(s_var) is at least 13 (8 + 4 + 1).However, the size of s_var might be greater because the compiler is allowed to pad a structure in order to achieve word or paragraph alignment. (A paragraph is 16 bytes.) Since the size of a structure may be greater than the sum of the sizes of its members, you should always use sizeof when you need to know the size of a structure. Since sizeof is a compile-time operator, all the information necessary to compute the size of any variable is known at compile time. This is especially meaningful for unions, because the size of a union is always equal to the size of its largest member.

For example, consider

```
union u {
char ch;
int i;
double f;
} u_var;
```

Here, the sizeof(u_var) is 8. At run time, it does not matter what u_var is actually holding. All that matters is the size of its largest member, because any union must be as large as its largest element.

13.6 typedef

You can define new data type names by using the keyword typedef. You are not actually creating a new data type, but rather defining a new name for an existing type. This process can help make machine-dependent programs more portable. If you define your own type name for each machine-dependent data type used by your program, then only the typedef statements have to be changed when compiling for a new environment. typedef also can aid in self-documenting your code by allowing descriptive names for the standard data types. The general form of the typedef statement is

```
typedef type newname;
```

where type is any valid data type and newname is the new name for this type. The new name you define is in addition to, not a replacement for, the existing type name. For example, you could create a new name for float by using

```
typedef float balance;
```

This statement tells the compiler to recognize balance as another name for float. Next, you could create a float variable using balance:

```
balance over_due;
```

Here, over_due is a floating-point variable of type balance, which is another word for float. Now that balance has been defined, it can be used in another typedef. For example,

```
typedef balance overdraft;
```

tells the compiler to recognize overdraft as another name for balance, which is another name for float. Using typedef can make your code easier to read and easier to port to a new machine, but you are not creating a new physical type.

Summary

- The C++ language gives six ways to create a custom data type. Out of these five ways are inherited from C and one is added by C++.
- A structure is a collection of variables referenced under one name.
- Individual members of a structure are accessed through the use of the . operator.
- The most common usage of structures is in arrays of structures.
- When you pass a member of a structure to a function, you are actually passing
- The value of that member to the function.
- C/C++ allows pointers to structures just as it allows pointers to any other type of variable.
- To access the members of a structure using a pointer to that structure, you must use the –> operator.
- A member of a structure may be either a simple or compound type.
- Unlike some other computer languages, C/C++ has a built-in feature called a bit-field that allows you to access a single bit.
- A union is a memory location that is shared by two or more different variables.
- An enumeration is a set of named integer constants that specify all the legal values a variable of that type may have.
- The sizeof operator computes the size of any variable or type and can help eliminate machine-dependent code from your programs.
- You can define new data type names by using the keyword typedef.

Self Check Exercise

1. In C++ how many custom data types are there, describe each briefly.
2. How structures differ from unions?
3. Write the syntax of structure
4. How user defined data types created in C++?
5. Write a small program to show how unions work.
6. What is the use of bit fields
7. How to define structure within a structure, explain?

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.

ARRAYS OF OBJECTS, ALLOCATING ARRAYS AND THE VECTOR CONTAINER

Objectives

- After reading this lesson you should be able to:
- Understand the general concept of arrays of objects and its syntax
- Understand what is meant by Initialized vs. Uninitialized Arrays and where each is used.
- Understand the vector container which is behaves just like an array but is more powerful and safer to use than the standard C++ array.

Introduction

This section covers three topics which describes those features of the arrays which are specific to C++. These features are designed to support object-oriented programming (OOP),

14.1 Arrays of Objects

All of whose elements are of the same class, can be declared just as an array of any built-in type. Each element of the array is an object of that class. Being able to declare arrays of objects in this way underscores the fact that a class is similar to a type. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
 class cl {
 int i;
 public:
```

```
 void set_i(int j) { i=j; }
 int get_i() { return i; }
 };

int main()
{
 cl ob[3];
 int i;
 for(i=0; i<3; i++) ob[i].set_i(i+1);
   for(i=0; i<3; i++)
     cout << ob[i].get_i() << "\n";
 return 0;
}
```

This program displays the numbers **1, 2,** and **3** on the screen. If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructor function. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the list is passed to the constructor's parameter. For example, here is a slightly different version of the preceding program that uses an initialization:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen. Actually, the initialization syntax shown in the preceding program is shorthand for this longer form:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Here, the constructor for **cl** is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that

applies to constructors taking only one argument . Thus, the short form can only be used to initialize object arrays whose constructors only require one argument. If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example,

```
#include <iostream>
using namespace std;
class cl {
int h;
int i;
public:
cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
cl(5, 6)
};
int i;
for(i=0; i<3; i++) {
cout << ob[i].get_h();
cout << ", ";
cout << ob[i].get_i() << "\n";
}
return 0;
}
```

Here, **cl**'s constructor has two parameters and, therefore, requires two arguments. This means that the shorthand initialization format cannot be used and the long form, shown in the example, must be employed.

14.2 Allocating Arrays

C++ supports dynamic allocation and deallocation of objects using the new and delete operators. These operators allocate memory for objects from a pool called the free store. The new operator calls the special function operator new, and the delete operator calls the special function operator delete. If you allocate an array of memory using new, you must remember to use the special delete[ ] operator. If you forget to use the delete[ ] operator and instead call delete (with no brackets), you will deallocate only the first object, and the rest of the memory will be lost. Operators new and delete are exclusive of C++. They are not available in the C language. But using pure C language and its library,

dynamic memory can also be used through the functions malloc, calloc, realloc and free, which are also available in C++ including the <cstdlib> header file

14.2.1 Allocating Arrays

You can initialize allocated memory to some known value by putting an initializer after the type name in the **new** statement. Here is the general form of **new** when an initialization is included:

p_var = new var_type (initializer);

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

This program gives the allocated integer an initial value of 87:

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int (87); // initialize to 87
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```

14.2.2 De-allocating Arrays

To free an array, use this form of **delete:**

delete [ ] p_var;

Here, the **[ ]** informs **delete** that an array is being released. For example, the next program allocates a 10-element integer array.

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p, i;
try {
p = new int [10]; // allocate 10 integer array
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
```

```
}
for(i=0; i<10; i++ )
p[i] = i;
for(i=0; i<10; i++)
cout << p[i] << " ";
delete [] p; // release the array
return 0;
}
```

Notice the **delete** statement. As just mentioned, when an array allocated by **new** is released, **delete** must be made aware that an array is being freed by using the **[ ]**. One restriction applies to allocating arrays: They may not be given initial values. That is, you may not specify an initializer when allocating arrays.

14.2.3 Creating Initialized vs. Uninitialized Arrays

The array allocation through new has a potential performance problem: it initializes every element in the array unconditionally. If your application deals with large arrays, this isn't the most efficient way. In some applications only a portion of the array is actually used, and in other applications the elements are assigned a different value immediately after their construction. In these cases, you want to postpone, or even completely avoid, the automatic initialization of array elements. Thus a special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
```

Here, the constructor function defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration:

```
cl a[9]; // error, constructor requires initializers
```

The reason that this statement isn't valid (as **cl** is currently defined) is that it implies that **cl** has a parameterless constructor because no initializers are specified. However, as it stands, **cl** does not have a parameterless constructor. Because there is no valid constructor that corresponds to this declaration, the compiler will report an error. To solve this problem, you need to overload the constructor function, adding one that takes no parameters. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
int i;
public:
cl() { i=0; } // called for non-initialized arrays
cl(int j) { i=j; } // called for initialized arrays
int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:
cl a1[3] = {3, 5, 6}; // initialized
cl a2[34]; // uninitialized

14.3 The Vector Container

You often use arrays to store and access a number of elements. Elements in an array are of the same type and are accessed with an index. The STL provides a container class vector that behaves just like an array but that is more powerful and safer to use than the standard C++ array. **Vectors** are a C++ implementation of the dynamic array data structure. Their interface emulates the behavior of a C array (i.e., capable of fast random access) but with the additional ability to automatically resize itself when inserting or removing an object.

Thus a vector is a template class that is a part of the C++ Standard Template Library. They can store any type, but are limited to storing only one type per instance. Therefore, they could not store data in the form of both char and int within the same vector instance. Vectors can, however, store pointers to several class objects provided they are all from the same class. Vectors provide a standard set of methods for accessing elements, adding elements to the start or end, deleting elements and finding how many elements are stored.

14.3.1 Main features of Vectors

- One of the features of a vector is that if you use the at() function, an exception will be thrown if you try to access an element that doesn't exist.
- A vector is a template class, a generic array of whatever type you want it to be. This gives vectors a great deal of flexibility, as they can be used as an array of anything. For instance, you can even have a vector of vectors.
- Vectors can automatically detect out of bounds errors (see Bounds checking) by using the at() method, or skip bounds checking by using operator[].
- They are very fast for random access.
- Vectors also have a number of useful functions which can tell you certain properties of the vector.
- Like other STL containers, vectors may contain both primitive data types and complex, user-defined classes or structs. Unlike other STL containers, such as deques and lists, vectors allow the user to denote an initial capacity for the container, which, if used correctly can allow for faster execution of the program.

14.4 Vector Initialization

A C++ program that is to use vectors must #include <vector>. A vector is initialized using:

std::vector<type> instance;

Where "type" is the data type the vector is to store, and "instance" is the variable name of the vector. "Type" can be a primitive or any user defined class or struct. However care must be taken to avoid object slicing.

Summary

- All of whose elements are of the same class, can be declared just as an array of any built-in type.

- The syntax for declaring and using an object array is exactly the same as it is for any other type of array.
- If you allocate an array of memory using new, you must remember to use the special delete[ ] operator.
- The STL provides a container class vector that behaves just like an array but that is more powerful and safer to use than the standard C++ array.
- A vector is a template class, a generic array of whatever type you want it to be.
- Vectors can automatically detect out of bounds errors.
- Vectors are very fast for random access

Self Check Exercise

1. How dynamic memory allocation takes place in C++?
2. What care must be taken to de-allocate memory allocated to arrays?
3. The array allocation through new has a potential performance problem, what is that and how it can be solved?
4. What are the main advantages of using vectors over arrays in C++ discuss.

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.
- Bruce Eckel, " Thinking In C++" Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

FUNCTIONS

Objectives

After reading this lesson you should be able to:
Understand the general concept of what a function is and why we need functions.
Understand the concept of different types of functions
Understand what is meant by function prototypes
Understand the general concept of function header
Understand scope rules of functions
Understand how arguments are passed to functions
Understand the general concept of returning from functions
Understand the concept of void function
Understand recursive call to functions
Understand the concept of improving efficiency and usability of function.
Understand the general concept of function overloading

Understand the general concept of reference parameters

Introduction

Functions are the building blocks of C and C++ programs and is the place where all program activity occurs. A function is a self-contained block of statements that perform a coherent task of some kind. Writing functions avoids rewriting the same code over and over. We have been using many of C++'s built-in functions, such as sqrt(), sin(), acos(), pow(), and so on. C++ encourages you to develop your own functions to break the more complex activities down into more manageable units. In fact, this idea of breaking the total work load down into smaller pieces is a fundamental design principle of modern programming. The general term to describe this process of breaking a complex problem down into more manageable units is functional decomposition. One of the simplest design tool to assist functionally decomposing a problem is Top-down Design.

- If a program contains only one function, it must be main( ).
- If a C/C++ program contains more than one function, then one (and only one) of these functions must be main( ), because program execution always begins with main( ).
- There is no limit on the number of functions that might be present in a C++ program.
- Each function in a program is called in the sequence specified by the function calls in main( ).
- After each function has done its thing, control returns to main( ).When main( ) runs out of function calls, the program ends.
- A function can call itself. Such a process is called 'recursion'. We would discuss this aspect of C functions later in this chapter.
- A function can be called from other function, but a function cannot be defined in another function.
- The order in which the functions are defined in a program and the order in which they get called need not necessarily be same.
- Every function in C++ must have a prototype or model or blueprint for the compiler to use when it needs to call or invoke that function. Knowing a function's prototype enables the compiler to handle any necessary data conversions. For example, in this problem, both the two numbers are doubles.

15.1 Types of Functions

When you write programs, your functions generally will be of three types.

- The first type is simply computational. These functions are specifically designed to perform operations on their arguments and return a value based on that operation. Examples are the standard library functions sqrt() and sin()
- The second type of function manipulates information and returns a value that simply indicates the success or failure of that manipulation. An example is the library function fclose() , which is used to close a file. If the close operation is successful, the function returns 0; if the operation is unsuccessful, it returns EOF.
- The last type of function has no explicit return value. In essence, the function is strictly procedural and produces no value. An example is exit() , which terminates a program. All functions that do not return values should be declared as returning

type void. By declaring a function as void, you keep it from being used in an expression, thus preventing accidental misuse.

15.2 Function prototype

In C++ all functions must be declared before they are used. This is normally accomplished using a *function prototype*. Prototypes enable both C and C++ to provide stronger type checking, somewhat like that provided by languages such as Pascal. When you use prototypes, the compiler can find and report any illegal type conversions between the type of arguments used to call a function and the type definition of its parameters. The general form of a function prototype is

type func_name(type parm_name1, type parm_name2,. . .,
type parm_nameN);

The use of parameter names is optional. However, they enable the compiler to identify any type mismatches by name when an error occurs, so it is a good idea to include them. Also any standard library function used by your program must be prototyped. To accomplish this, you must include the appropriate header for each library function. All necessary headers are provided by the C/C++ compiler. In C, all headers are files that use the .H extension. In C++, headers may be either separate files or built into the compiler itself. In either case, a header contains two main elements: any definitions used by the library functions and the prototypes for the library functions. For example, stdio.h is included in almost all programs in this part of the book because it contains the prototype for printf() .

15.3 Function Header

The actual coding of the function begins with a function header. The function header follows the exactly same format as the prototype except that the ending semicolon is replaced with a beginning { and ending } indicating here come the actual statements that the function represents. The general form of a function is

```
ret-type function-name(parameter list)
{
body of the function
}
```

The ret-type specifies the type of data that the function returns. A function may return any type of data except an array. The parameter list is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. A function may be without parameters, in which case the parameter list is empty. However, even if there are no parameters, the parentheses are still required. In variable declarations, you can declare many variables to be of a common type by using a comma-separated list of variable names. In contrast, all function parameters must be declared individually, each including both the type and name. That is, the parameter declaration list for a function takes the following general form:

f(type varname1, type varname2, . . . , type varnameN)

For example, here are correct and incorrect function parameter declarations:

```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

Each function in a program including main() must be outside of any other function. While there are no limits on where you can call or invoke functions, their actual definitions cannot be nested within other functions like an inner while loop contained within an outer loop. Each function definition must be by itself. One of the sequences which can be followed is :

First main(), then the functions that main() directly calls and next the functions that those functions call and so on.

Consider the following code and try finding the error in it

```
#includes
const ints
prototypes
int main () {
...
double higher (double x, double y) {
...
}
...
}
```

The code is illegal because the function definition itself is within the body of the main() function. If compiled, it often generates an error message that local functions are not supported.

15.4 Scope Rules of Functions

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. For parameter type variables, the scope is from the point of their definition within the function header to the end brace of the function. Another way of looking at this is that a local or parameter type of variable belongs exclusively to the function in which it is defined. Its name is not known outside of that function or earlier in the same function before it is defined.

Each function is a discrete block of code. A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use goto to jump into the middle of another function.) The code that constitutes the body of a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither affect nor be affected by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope.

Variables that are defined within a function are called local variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the static storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. In C (and C++) you cannot define a function

within a function. This is why neither C nor C++ are technically block-structured languages.

15.5 Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. As shown in the following function, the parameter declarations occur after the function name:

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
  while(*s)
        if(*s==c) return 1;
        else s++;
  return 0;
}
```

The function is_in() has two parameters: s and c. This function returns 1 if the character c is part of the string s; otherwise, it returns 0. As with local variables, you may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, you can use them as you do any other local variable.

15.5.1 Passing Arguments to Functions

In a computer language, there are two ways that arguments can be passed to a subroutine. The first is known as *call by value* and the second is known as *Call by reference.*

call by value

This method copies the *value of* an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument. In the following example the values of two variables x and y are passed to function higher.

```
#include <iostream>
#include <iomanip>
using namespace std;
double higher (double x, double y);
int main () {
  double x; // a number entered by user
  double y; // a second number entered by user
  double bigger; // a place to store the larger of x and y
  cout << "Enter two numbers: ";
  cin >> x >> y;
  bigger = higher (x, y);//call higher &store return val in bigger
  cout << bigger << endl;
  return 0;
```

```
}
```
Consider another example

```
#include <stdio.h>
int sqr(int x);
int main(void)
{
  int t=10;
  printf("%d %d", sqr(t), t);
  return 0;
}

int sqr(int x)
{
  x = x*x;
  return(x);
}
```

In this example, the value of the argument to sqr() , 10, is copied into the parameter x. When the assignment x = x*x takes place, only the local variable x is modified. The variable t, used to call sqr() , still has the value 10. Hence, the output is 100 10. Remember that it is a copy of the value of the argument that is passed into the function. What occurs inside the function has no effect on the variable used in the call.

Call by reference

Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing. First, you can explicitly pass a pointer to the argument. Second, you can use a reference parameter. For most circumstances the best way is to use a reference parameter. To fully understand what a reference parameter is and why it is valuable, we will begin by reviewing how a call-by-reference can be generated using a pointer parameter. The following program manually creates a call-by-reference parameter using a pointer in the function called neg(), which reverses the sign of the integer variable pointed to by its argument.

```
// Manually create a call-by-reference using a pointer.
#include <iostream>
using namespace std;
void neg(int *i);
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(&x);
cout << x << "\n";
```

```
return 0;
}
void neg(int *i)
{
*i = -*i;
}
```

In this program, neg() takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, neg() must be explicitly called with the address of x. Further, inside neg() the * operator must be used to access the variable pointed to by i. This is how you generate a "manual" call-by-reference in C++, and it is the only way to obtain a call-by-reference using the C subset. Fortunately, in C++ you can automate this feature by using a reference parameter. To create a reference parameter, precede the parameter's name with an &. For example, here is how to declare neg() with i declared as a reference parameter: void neg(int &i); For all practical purposes, this causes i to become another name for whatever argument neg() is called with. Any operations that are applied to i actually affect the calling argument. In technical terms, i is an implicit pointer that automatically refers to the argument used in the call to neg() . Once i has been made into a reference, it is no longer necessary (or even legal) to apply the * operator. Instead, each time i is used, it is implicitly a reference to the argument and any changes made to i affect the argument. Further, when calling neg(), it is no longer necessary (or legal) to precede the argument's name with the & operator. Instead, the compiler does this automatically.

In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Even though C/C++ uses call by value for passing parameters, you can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value. Of course, you need to declare the parameters as pointer types. For example, the function swap() , which exchanges the values of the two integer variables pointed to by its arguments, shows how.

```
void swap(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
}
```

swap() is able to exchange the values of the two variables pointed to by x and y because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the

variables used to call the function are swapped. Remember that swap() (or any other function that uses pointer parameters) must be called with the addresses of the arguments. The following program shows the correct way to call swap() :

```
void swap(int *x, int *y);
int main(void)
{
  int i, j;
  i = 10;
  j = 20;
  swap(&i, &j); /* pass the addresses of i and j */
  return 0;
}
```

In this example, the variable i is assigned the value 10 and j is assigned the value 20. Then swap() is called with the addresses of i and j. (The unary operator & is used to produce the address of the variables.) Therefore, the addresses of i and j, not their values, are passed into the function swap() . C++ allows you to fully automate a call by reference through the use of reference parameters.

15.5.2 Command Line Argument

A command line argument is the information that follows the program's name on the command line of the operating system. For example, when you compile a program, you might type something like the following after the command prompt:

cc program_name where program_name is a command line argument that specifies the name of the program you wish to compile. There are two special built-in arguments, argv and argc, that are used to receive command line arguments. The argc parameter holds the number of arguments on the command line and is an integer. It is always at least 1 because the name of the program qualifies as the first argument. The argv parameter is a pointer to an array of character pointers. Each element in this array points to a command line argument. All command line arguments are strings—any numbers will have to be converted by the program into the proper internal format. For example, this simple program prints Hello and your name on the screen if you type it directly after the program name.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
        if(argc!=2) {
                printf("You forgot to type your name.\n");
                exit(1);
        }
  printf("Hello %s", argv[1]);
  return 0;
}
```

If you called this program name and your name were Tom, you would type name Tom to run the program. The output from the program would be Hello Tom. You must declare argv properly. The most common method is

```
char *argv[];
```

The empty brackets indicate that the array is of undetermined length. You can now access the individual arguments by indexing argv. For example, argv[0] points to the first string, which is always the program's name; argv[1] points to the first argument, and so on.

15.5.3 Declaring Variable-Length Parameter Lists

You can specify a function that has a variable number of parameters. The most common example is printf() . To tell the compiler that an unknown number of arguments may be passed to a function, you must end the declaration of its parameters using three periods. For example, this prototype specifies that func() will have at least two integer parameters and an unknown number (including 0) of parameters after that.

```
int func(int a, int b, ...);
```

This form of declaration is also used by a function's definition. Any function that uses a variable number of parameters must have at least one actual parameter. For example, this is incorrect:

```
int func(...); /* illegal */
```

15.6 Returning from a Function

All functions, except those of type void, return a value. When a function has no parameters, nothing is coded in the prototype or header — just the pair of parentheses (). When a function returns nothing, the keyword void must be used for the return data type. The prototype for a headings function that took no parameters and returned no value is as follows.

```
void headings ();
```

In case the functions return a value this value is specified by the return statement. In C, if a non-void function does not explicitly return a value via a return statement, then a garbage value is returned. In C++, a non-void function must contain a return statement that returns a value. That is, in C++, if a function is specified as returning a value, any return statement within it must have a value associated with it. However, if execution reaches the end of a non-void function, then a garbage value is returned. Although this condition is not a syntax error, it is still a fundamental error and should be avoided. As long as a function is not declared as void, you may use it as an operand in an expression.

There are two ways that a function terminates execution and returns to the caller. The first occurs when the last statement in the function has executed and, conceptually, the function's ending curly brace (}) is encountered. (Of course, the curly brace isn't actually present in the object code, but you can think of it in this way.) For example, the pr_reverse() function in this program simply prints the string "I like C++" backwards on the screen and then returns.

```
#include <string.h>
#include <stdio.h>
void pr_reverse(char *s);
```

```
int main(void)
{
pr_reverse("I like C++");
return 0;
}
void pr_reverse(char *s)
{
register int t;
for(t=strlen(s)-1; t>=0; t--) putchar(s[t]);
}
```

Once the string has been displayed, there is nothing left for pr_reverse() to do, so it returns to the place from which it was called. Actually, not many functions use this default method of terminating their execution. Most functions rely on the return statement to stop execution either because a value must be returned or to make a function's code simpler and more efficient. A function may contain several return statements. For example, the find_substr() function in the following program returns the starting position of a substring within a string, or returns −1 if no match is found.

```
#include <stdio.h>
int find_substr(char *s1, char *s2);
int main(void)
{
if(find_substr("C++ is fun", "is") != -1)
printf("substring is found");
return 0;
}
/* Return index of first match of s2 in s1. */
int find_substr(char *s1, char *s2)
{
register int t;
char *p, *p2;
for(t=0; s1[t]; t++) {
p = &s1[t];
p2 = s2;
while(*p2 && *p2==*p) {
p++;
p2++;
}
if(!*p2) return t; /* 1st return */
}
return -1; /* 2nd return */
}
```

As a general rule, a function cannot be the target of an assignment. A statement such as `swap(x,y) = 100; /* incorrect statement */` is wrong. The C/C++ compiler will flag

it as an error and will not compile a program that contains it. (As is discussed in Part Two, C++ allows some interesting exceptions to this general rule, enabling some types of functions to occur on the left side of an assignment.)

15.7 Returning Pointers

Although functions that return pointers are handled just like any other type of function, a few important concepts need to be discussed. Pointers to variables are neither integers nor unsigned integers. They are the memory addresses of a certain type of data. The reason for this distinction is because pointer arithmetic is relative to the base type. For example, if an integer pointer is incremented, it will contain a value that is 4 greater than its previous value (assuming 4-byte integers). In general, each time a pointer is incremented (or decremented), it points to the next (or previous) item of its type. Since the length of different data types may differ, the compiler must know what type of data the pointer is pointing to. For this reason, a function that returns a pointer must declare explicitly what type of pointer it is returning. To return a pointer, a function must be declared as having a pointer return type. For example, this function returns a pointer to the first occurrence of the character c in string s:

```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
{
while(c!=*s && *s) s++;
return(s);
}
```

If no match is found, a pointer to the null terminator is returned. Here is a short program that uses match() :

```
#include <stdio.h>
char *match(char c, char *s); /* prototype */
int main(void)
{
char s[80], *p, ch;
gets(s);
ch = getchar();
p = match(ch, s);
if(*p) /* there is a match */
printf("%s ", p);
else
printf("No match found.");
return 0;
}
```

This program reads a string and then a character. If the character is in the string, the program prints the string from the point of match. Otherwise, it prints No match found.

15.8 Functions of Type void

One of void's uses is to explicitly declare functions that do not return values. This prevents their use in any expression and helps avert accidental misuse. For example, the function print_vertical() prints its string argument vertically down the side of the screen. Since it returns no value, it is declared as void.

```
void print_vertical(char *str)
{
while(*str)
printf("%c\n", *str++);
}
```

Here is an example that uses print_vertical() .

```
#include <stdio.h>
void print_vertical(char *str); /* prototype */

int main(int argc, char *argv[])
{
if(argc > 1) print_vertical(argv[1]);
return 0;
}
void print_vertical(char *str)
{
while(*str)
printf("%c\n", *str++);
}
```

One last point: Early versions of C did not define the void keyword. Thus, in early C programs, functions that did not return values simply defaulted to type int. Therefore, don't be surprised to see many examples of this in older code. What Does main( ) Return?

The main() function returns an integer to the calling process, which is generally the operating system. Returning a value from main() is the equivalent of calling exit() with the same value. If main() does not explicitly return a value, the value passed to the calling process is technically undefined. In practice, most C/C++ compilers automatically return 0, but do not rely on this if portability is a concern.

15.9 Recursive call to functions

In C/C++, a function can call itself. A function is said to be recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called circular definition.

A simple example of a recursive function is factr() , which computes the factorial of an integer. The factorial of a number n is the product of all the whole numbers between 1 and n. For example, 3 factorial is 1 x 2 x 3, or 6. B oth factr() and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
int answer;
```

```
if(n==1) return(1);
answer = factr(n-1)*n; /* recursive call */
return(answer);
}
/* non-recursive */
int fact(int n) {
int t, answer;
answer = 1;
for(t=1; t<=n; t++)
answer=answer*(t);
return(answer);
}
```

The nonrecursive version of fact() uses a loop that runs from 1 to n and progressively multiplies each number by the moving product. The operation of the recursive factr() is a little more complex. When factr() is called with an argument of 1, the function returns 1. Otherwise, it returns the product of factr(n−1)*n. To evaluate this expression, factr() is called with n−1. This happens until n equals 1 and the calls to the function begin returning. Computing the factorial of 2, the first call to factr() causes a second, recursive call with the argument of 1. This call returns 1, which is then multiplied by 2 (the original n value). The answer is then 2.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function. Most recursive routines do not significantly reduce code size or improve memory utilization. Also, the recursive versions of most routines may execute a bit slower than their iterative equivalents because of the overhead of the repeated function calls. In fact, many recursive calls to a function could cause a stack overrun. Because storage for function parameters and local variables is on the stack and each new call creates a new copy of these variables, the stack could be overrun.

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the quicksort algorithm is difficult to implement in an iterative way. Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively.

15.10 Implementation Issues

There are a few important things to remember about functions that affect their efficiency and usability.

15.10.1 Parameters and General-Purpose Functions

A general-purpose function is one that will be used in a variety of situations, perhaps by many different programmers. Typically, you should not base general-purpose

functions on global data. All of the information a function needs should be passed to it by its parameters. When this is not possible, you should use static variables. Besides making your functions general purpose, parameters keep your code readable and less susceptible to bugs resulting from side effects.

15.10.2 Efficiency

Functions are the building blocks of C/C++ and are crucial to all but the simplest programs. However, in certain specialized applications, you may need to eliminate a function and replace it with *inline* code. Inline code performs the same actions as a function, but without the overhead associated with a function call. For this reason, inline code is often used instead of function calls when execution time is critical. Inline code is faster than a function call for two reasons. First, a CALL instruction takes time to execute. Second, if there are arguments to pass, these have to be placed on the stack, which also takes time. For most applications, this very slight increase in execution time is of no significance. But if it is, remember that each function call uses time that would be saved if the function's code were placed in line. For example, the following are two versions of a program that prints the square of the numbers from 1 to 10. The inline version runs faster than the other because the function call adds time. in line function call

```
#include <stdio.h> #include <stdio.h>
int sqr(int a);
int main(void) int main(void)
{{
int x; int x;
for(x=1; x<11; ++x) for(x=1; x<11; ++x)
printf("%d", x*x); printf("%d", sqr(x));
return 0; return 0;
}}
int sqr(int a)
{
return a*a;
}
```

Inline functions are an important component of the C++ language.

15.11 Function Overloading

One way that C++ achieves polymorphism is through the use of function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different. In this situation, the functions that share the same name are said to be overloaded, and the process is referred to as function overloading. To see why function overloading is important, first consider three functions defined by the C subset: abs(), labs(), and fabs(). The abs() function returns the absolute value of an integer, labs() returns the absolute value of a long, and fabs() returns the absolute value of a double. Although these functions perform almost identical actions, in C three slightly different names must be used to represent these essentially similar tasks. This makes the situation more complex, conceptually, than it actually is. Even though the underlying concept of each function is the same, the programmer has to remember three things, not

just one. However, in C++, you can use just one name for all three functions, as this program illustrates:

```
#include <iostream>
using namespace std;
// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);
int main()
{
cout << abs(-10) << "\n";
cout << abs(-11.0) << "\n";
cout << abs(-9L) << "\n";
return 0;
}
int abs(int i)
{
cout << "Using integer abs()\n";
return i<0 ? -i : i;
}
double abs(double d)
{
cout << "Using double abs()\n";
return d<0.0 ? -d : d;
}
long abs(long l)
{
cout << "Using long abs()\n";
return l<0 ? -l : l;
}
```

This program creates three similar but different functions called abs(), each of which returns the absolute value of its argument. The compiler knows which function to call in each situation because of the type of the argument. The value of overloaded functions is that they allow related sets of functions to be accessed with a common name. Thus, the name abs() represents the general action that is being performed. It is left to the compiler to choose the right specific method for a particular circumstance. You need only remember the general action being performed. Due to polymorphism, three things to remember have been reduced to one. This example is fairly trivial, but if you expand the concept, you can see how polymorphism can help you manage very complex programs. In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction when overloading a function: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the

types or number of their parameters. (Return types do not provide sufficient information in all cases for the compiler to decide which function to use.) Of course, overloaded functions may differ in their return types, too.

Summary

- A function is a self-contained block of statements that perform a coherent task.
- Every function in C++ must have a prototype for the compiler to use when it needs to call or invoke that function.
- Each function in a program including main() must be outside of any other function.
- A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function.
- There are two ways that arguments can be passed to a subroutine. (call by value and Call by reference)
- A command line argument is the information that follows the program's name on the command line of the operating system.
- You can specify a function that has a variable number of parameters. The most common example is printf().
- A function that returns a pointer must declare explicitly what type of pointer it is returning
- In C/C++, a function can call itself. A function is said to be recursive if a statement in the body of the function calls itself.
- When a function calls itself, a new set of local variables and parameters are allocated storage on the stack
- Inline code performs the same actions as a function, but without the overhead associated with a function call.
- In C++, two or more functions can share the same name as long as their parameter declarations are different.

Self Check Exercise

1. Why is function prototype needed?
2. What are the scope rules funcions?
3. How are arguments passed to functions, explain with an example program?
4. How can recursive calls be made to a function, explain with an example?
5. What is meant by function overloading and how is it useful?

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications,1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.

- Bruce Eckel, “ Thinking In C++” Second Edition, Prentice Hall.
- Yashavant P. Kanetkar,"Let Us C", Fifth Edition, BPB publications

POINTERS

Objectives

After reading this lesson you should be able to:
Understand the general concept of pointer variables, operators and expressions
Understand the concept of multiple indirections
Understand the concept of pointers to functions
Understand how pinters and arrays are related

Introduction

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second. For a type T , T * is the type "pointer to T ." That is, a variable of type T * can hold the address of an object of type T . For example:

```
c h a r c = ´a ´;
c h a r * p = &c ; // p holds the address of c
```

The correct understanding and use of pointers is critical to successful C/C++ programming. There are three reasons for this: First, pointers provide the means by which functions can modify their calling arguments. Second, pointers support dynamic allocation. Third, pointers can improve the efficiency of certain routines. Pointers take on

additional roles in C++. Pointers are one of the strongest but also one of the most dangerous features in C/C++. For example, uninitialized pointers (or pointers containing invalid values) can cause your system to crash. Perhaps worse, it is easy to use pointers incorrectly, causing bugs that are very difficult to find.

16.1 Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

*type *name;*

where type is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by name. The base type of the pointer defines what type of variables the pointer can point to.Technically, any type of pointer can point anywhere in memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly.

16.2 Pointer Operators

There are two special pointer operators: * and &.

- The & is a unary operator that returns the memory address of its operand.

For example,

m = &count;

places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. You can think of & as returning "the address of." Therefore, the preceding assignment statement means "m receives the address of count." To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also assume that count has a value of 100. Then, after the preceding assignment, m will have the value 2000.

- The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows. For example, if m contains the memory address of the variable count,

q = *m;

places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m. You can think of * as "at address." In this case, the preceding statement means "q receives the value at address m."

Both & and * have a higher precedence than all other arithmetic operators except the unary minus, with which they are equal. You must make sure that your pointer variables always point to the correct type of data. For example, when you declare a pointer to be of type int, the compiler assumes that any address that it holds points to an integer variable—whether it actually does or not. Because C allows you to assign any address to a pointer variable, the following code fragment compiles with no error messages (or only warnings, depending upon your compiler), but does not produce the desired result:

```
#include <stdio.h>
int main(void)
```

```
{
double x = 100.1, y;
int *p;
/* The next statement causes p (which is an
integer pointer) to point to a double. */
p = &x;
/* The next statement does not operate as
expected. */
y = *p;
printf("%f", y); /* won't output 100.1 */
return 0;
}
```

This will not assign the value of x to y. Because p is declared as an integer pointer, only 2 or 4 bytes of information will be transferred to y, not the 8 bytes that normally make up a double. In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast. For this reason, the preceding program will not even compile if you try to compile it as a C++ (rather than as a C) program. However, the type of error described can still occur in C++ in a more roundabout manner.

16.3 Pointer Expressions

In general, expressions involving pointers conform to the same rules as other expressions. This section examines a few special aspects of pointer expressions.

16.3.1 Pointer Assignments

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example,

```
#include <stdio.h>
int main(void)
{
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
printf(" %p", p2); /* print the address of x, not x's value! */
return 0;
}
```

Both p1 and p2 now point to x. The address of x is displayed by using the %p printf() format specifier, which causes printf() to display an address in the format used by the host computer.

Program to demonstrate the use of pointer variable and address of operator(&).

```
#include<iostream.h>
#include<conio.h>
Int main() {
Clrscr();
Int I,*ptr;
```

```
I=5;
ptr=&I;
cout<<"value of I is"<<&I<<endl;
cout<<"the value of Ptr is"<<ptr<<endl;
getc();
return 0;
}
```

16.3.2 Pointer Arithmetic

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long. After the expression

```
p1++;
```

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression

```
p1--;
```

causes p1 to have the value 1998.

The following rules govern pointer arithmetic

- Each time a pointer is incremented, it points to the memory location of the next element of its base type.
- Each time it is decremented, it points to the location of the previous element. When applied to character pointers, this will appear as "normal" arithmetic because characters are always 1 byte long.
- All other pointers will increase or decrease by the length of the data type they point to. This approach ensures that a pointer is always pointing to an appropriate element of its base type. You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

  ```
  p1 = p1 + 12;
  ```

  Makes p1 point to the twelfth element of p1's type beyond the one it currently points to.
- Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You may subtract one pointer from another in order to find the number of objects of their base type that separate the two.
- All other arithmetic operations are prohibited. Specifically, you may not multiply or divide pointers; you may not add two pointers; you may not aypply the bitwise operators to them; and you may not add or subtract type float or double to or from pointers.

16.3.3 Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers p and q, the following statement is perfectly valid:

```
if(p<q) printf("p points to lower memory than q\n");
```

Generally, pointer comparisons are used when two or more pointers point to a common object, such as an array. Consider the following example, a pair of stack routines are developed that store and retrieve integer values. A stack is a list that uses first-in, last-out accessing. It is often compared to a stack of plates on a table—the first one set down is the last one to be used. Stacks are used frequently in compilers, interpreters, spreadsheets, and other system-related software. To create a stack, you need two functions: push() and pop() . The push() function places values on the stack and pop() takes them off. These routines are shown here with a simple main() function to drive them. The program puts the values you enter into the stack. If you enter 0, a value is popped from the stack. To stop the program, enter -1.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 60
void push(int i);
int pop(void);
int *tos, *p1, stack[SIZE];

int main(void)
{
int value;
tos = stack; /* tos points to the top of stack */
p1 = stack; /* initialize p1 */
do {
printf("Enter value: ");
scanf("%d", &value);
if(value!=0) push(value);
else printf("value on top is %d\n", pop());
} while(value!=-1);
return 0;
}

void push(int i)
{
p1++;
if(p1==(tos+SIZE)) {
printf("Stack Overflow.\n");
exit(1);
}
*p1 = i;
}
int pop(void)
{
if(p1==tos) {
```

```
printf("Stack Underflow.\n");
exit(1);
}
p1--;
return *(p1+1);
}
```

You can see that memory for the stack is provided by the array stack. The pointer p1 is set to point to the first element in stack. The p1 variable accesses the stack. The variable tos holds the memory address of the top of the stack. It is used to prevent stack overflows and underflows. Once the stack has been initialized, push() and pop() may be used. Both the push() and pop() functions perform a relational test on the pointer p1 to detect limit errors. In push() , p1 is tested against the end of stack by adding SIZE (the size of the stack) to tos. This prevents an overflow. In pop() , p1 is checked against tos to be sure that a stack underflow has not occurred. In pop() , the parentheses are necessary in the return statement. Without them, the statement would look like this:

```
return *p1 +1;
```

which would return the value at location p1 plus one, not the value of the location p1+1.

16.3.4 Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, p1 has been set to the address of the first array element in str. To access the fifth element in str, you could write

```
str[4]
```

or

```
*(p1+4)
```

Both statements will return the fifth element. Remember, arrays start at 0. To access the fifth element, you must use 4 to index str. You also add 4 to the pointer p1 to access the fifth element because p1 currently points to the first element of str.

Thus C/C++ provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C/C++ programmers commonly use pointers to access array elements. These two versions of putstr()— one with array indexing and one with pointers— illustrate how you can use pointers in place of array indexing. The putstr() function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t) putchar(s[t]);
}
```

```
/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

Most professional C/C++ programmers would find the second version easier to read and understand. In fact, the pointer version is the way routines of this sort are commonly written in C/C++.

Pointers may be arrayed like any other data type. The declaration for an int pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called var to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of var, write

```
*x[2]
```

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays—simply call the function with the array name without any indexes. For example, a function that can receive array x looks like this:

```
void display_array(int *q[])
{
int t;
for(t=0; t<10; t++)
printf("%d ", *q[t]);
}
```

Remember, q is not a pointer to integers, but rather a pointer to an array of pointers to integers. Therefore you need to declare the parameter q as an array of integer pointers, as just shown. You cannot declare q simply as an integer pointer because that is not what it is. Pointer arrays are often used to hold pointers to strings. You can create a function that outputs an error message given its code number, as shown here:

```
void syntax_error(int num)
{
static char *err[] = {
"Cannot Open File\n",
"Read Error\n",
"Write Error\n",
"Media Failure\n"
};
printf("%s", err[num]);
}
```

The array err holds pointers to each string. As you can see, printf() inside syntax_error() is called with a character pointer that points to one of the various error messages indexed by the error number passed to the function. For example, if num is

passed a 2, the message Write Error is displayed. As a point of interest, note that the command line argument argv is an array of character pointers.

16.3.5 Multiple Indirection

You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection, or pointers to pointers. The value of a normal pointer is the address of the object that contains the value desired. In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the value desired. Multiple indirection can be carried on to whatever extent rquired, but more than a pointer to a pointer is rarely needed. In fact, excessive indirection is difficult to follow and prone to conceptual errors. Do not confuse multiple indirection with high-level data structures, such as linked lists, that use pointers. These are two fundamentally different concepts. A variable that is a pointer to a pointer must be declared as such. You do this by placing an additional asterisk in front of the variable name. For example, the following declaration tells the compiler that newbalance is a pointer to a pointer of type float:

```
float **newbalance;
```

You should understand that newbalance is not a pointer to a floating-point number but rather a pointer to a float pointer.

To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include <stdio.h>
int main(void)
{
int x, *p, **q;
x = 10;
p = &x;
q = &p;
printf("%d", **q); /* print the value of x */
return 0;
}
```

Here, p is declared as a pointer to an integer and q as a pointer to a pointer to an integer. The call to printf() prints the number 10 on the screen.

16.4 Initializing Pointers

After a local pointer is declared but before it has been assigned a value, it contains an unknown value. (Global pointers are automatically initialized to null.) Should you try to use the pointer before giving it a valid value, you will probably crash your program—and possibly your computer's operating system as well. There is an important convention that most C/C++ programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero). By convention, any pointer that is null implies that it points to nothing and should not be used. The use of null is simply a convention that programmers follow. It is not a rule enforced by the C or C++ languages. You can use the null pointer to make many of your pointer routines easier to code and more efficient. For example, you could use a null

pointer to mark the end of a pointer array. Another variation on the initialization theme is the following type of string declaration:

```
char *p = "hello world";
```

As you can see, the pointer p is not an array. The reason this sort of initialization works is because of the way the compiler operates. All C/C++ compilers create what is called a *string table*, which is used to store the string constants used by the program. Therefore, the preceding declaration statement places the address of hello world, as stored in the string table, into the pointer p. Throughout a program, p can be used like any other string. For example, the following program is perfectly valid:

```
#include <stdio.h>
#include <string.h>
char *p = "hello world";
int main(void)
{
register int t;
/* print the string forward and backwards */
printf(p);
for(t=strlen(p)-1; t>-1; t--) printf("%c", p[t]);
return 0;
}
```

In Standard C++, the type of a string literal is technically const char *. But C++ provides an automatic conversion to char *. Thus, the preceding program is still valid. However, this automatic conversion is a deprecated feature, which means that you should not rely upon it for new code. For new programs, you should assume that string literals are constants and the declaration of p in the preceding program should be written like this.

```
const char *p = "hello world";
```

16.5 Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions. You obtain the address of a function by using the function's name without any parentheses or arguments. (This is similar to the way an array's address is obtained when only the array name, without indexes, is used.) To see how this is done, study the following program, paying close attention to the declarations:

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
```

```
char s1[80], s2[80];
int (*p)(const char *, const char *);
p = strcmp;
gets(s1);
gets(s2);
check(s1, s2, p);
return 0;
}
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
{
printf("Testing for equality.\n");
if(!(*cmp)(a, b)) printf("Equal");
else printf("Not Equal");
}
```

When the check() function is called, two character pointers and one function pointer are passed as parameters. Inside the function check() , the arguments are declared as character pointers and a function pointer. Notice how the function pointer is declared. You must use a similar form when declaring other function pointers, although the return type and parameters of the function may differ. The parentheses around the *cmp are necessary for the compiler to interpret this statement correctly. Inside check() , the expression

```
(*cmp)(a, b)
```

calls strcmp() , which is pointed to by cmp, with the arguments a and b. The parentheses around *cmp are necessary. This is one way to call a function through a pointer. A second, simpler syntax, as shown here, may also be used.

```
cmp(a, b);
```

The reason that you will frequently see the first style is that it tips off anyone reading your code that a function is being called through a pointer. (That is, that cmp is a function pointer, not the name of a function.) Other than that, the two expressions are equivalent. Note that you can call check() by using strcmp() directly, as shown here:

```
check(s1, s2, strcmp);
```

This eliminates the need for an additional pointer variable. You may wonder why anyone would write a program in this way. Obviously, nothing is gained and significant confusion is introduced in the previous example. However, at times it is advantageous to pass functions as parameters or to create an array of functions. For example, when a compiler or interpreter is written, the parser (the part that evaluates expressions) often calls various support functions, such as those that compute mathematical operations (sine, cosine, tangent, etc.), perform I/O, or access system resources. Instead of having a large switch statement with all of these functions listed in it, an array of function pointers can be created. In this approach, the proper function is selected by its index. You can get the flavor of this type of usage by studying the expanded version of the previous example.

In this program, check() can be made to check for either alphabetical equality or numeric equality by simply calling it with a different comparison function.

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b);
int main(void)
{
   char s1[80], s2[80];
   gets(s1);
   gets(s2);
   if(isalpha(*s1))
   check(s1, s2, strcmp);
   else
   check(s1, s2, numcmp);
   return 0;
}
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
{
  printf("Testing for equality.\n");
  if(!(*cmp)(a, b)) printf("Equal");
  else printf("Not Equal");
}
int numcmp(const char *a, const char *b)
{
   if(atoi(a)==atoi(b)) return 0;
   else return 1;
}
```

In this program, if you enter a letter, strcmp() is passed to check() . Otherwise, numcmp() is used. Since check() calls the function that it is passed, it can use different comparison functions in different cases.

16.6 Problems with Pointers

Pointers give you tremendous power and are necessary for many programs. At the same time, when a pointer accidentally contains a wrong value, it can be the most difficult bug to find. An erroneous pointer is difficult to find because the pointer itself is not the problem. The problem is that each time you perform an operation using the bad pointer, you are reading or writing to some unknown piece of memory. If you read from it, the worst that can happen is that you get garbage. However, if you write to it, you might be writing over other pieces of your code or data. This may not show up until later in the

execution of your program, and may lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. To help you avoid them, a few of the more common errors are discussed here.

- The classic example of a pointer error is the *uninitialized pointer*. Consider this program.

```
/* This program is wrong. */
int main(void)
{
   int x, *p;
   x = 10;
  *p = x;
  return 0;
}
```

This program assigns the value 10 to some unknown memory location. Here is why: Since the pointer p has never been given a value, it contains an unknown value when the assignment *p = x takes place. This causes the value of x to be written to some unknown memory location. This type of problem often goes unnoticed when your program is small because the odds are in favor of p containing a "safe" address—one that is not in your code, data area, or operating system. However, as your program grows, the probability increases of p pointing to something vital. Eventually, your program stops working. The solution is to always make sure that a pointer is pointing at something valid before it is used.

- A second common error is caused by a simple misunderstanding of how to use a pointer. Consider the following:

```
/* This program is wrong. */
#include <stdio.h>
int main(void)
{
int x, *p;
x = 10;
p = x;
printf("%d", *p);
return 0;
}
```

The call to printf() does not print the value of x, which is 10, on the screen. It prints some unknown value because the assignment p = x; is wrong. That statement assigns the value 10 to the pointer p. However, p is supposed to contain an address, not a value. To correct the program, write p = &x;

- Another error that sometimes occurs is caused by incorrect assumptions about the placement of variables in memory. You can never know where your data will be placed in memory, or if it will be placed there the same way again, or whether each compiler will treat it in the same way. For these reasons, making any comparisons

between pointers that do not point to a common object may yield unexpected results.

For example,

```
char s[80], y[80];
char *p1, *p2;
p1 = s;
p2 = y;
if(p1 < p2) . . .
```

is generally an invalid concept. (In very unusual situations, you might use something like this to determine the relative position of the variables. But this would be rare.)

- A related error results when you assume that two adjacent arrays may be indexed as one by simply incrementing a pointer across the array boundaries. For example,

```
int first[10], second[10];
int *p, t;
p = first;
for(t=0; t<20; ++t) *p++ = t;
```

This is not a good way to initialize the arrays first and second with the numbers 0 through 19. Even though it may work on some compilers under certain circumstances, it assumes that both arrays will be placed back to back in memory with first first. This may not always be the case.

Summary

- A pointer is a variable that holds a memory address.
- The correct understanding and use of pointers is critical to successful C/C++ programming.
- Pointers can improve the efficiency of certain routines.
- There are two special pointer operators: * and &.
- There are only two arithmetic operations that you may use on pointers: addition and subtraction.
- You can compare two pointers in a relational expression.
- C/C++ provides two methods of accessing array elements: pointer arithmetic and array indexing.
- You can have a pointer point to another pointer that points to the target value. This situation is called multiple indirection
- Excessive indirection is difficult to follow and prone to conceptual errors
- By convention, any pointer that is null implies that it points to nothing and should not be used
- Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer

Self Check Exercise

1. Why is the use of pointer critical for C++ Programming, explain?
2. How many methods of accessing array elements are there in C++, illustrate with the help of a program

3. Which arithmetic operations are allowed in pointers?
4. What are the main problems in the use of pointers?

Suggested Readings

- Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Robert Lafore, "Object Oriented Programming in C++", Galgotia Publications, 1994.
- E. Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill,2001. Herbert Schildt, " The Complete Reference C++", Tata McGraw-Hill, 2001.
- Deitel and Deitel, " C++ How to program" ,Pearson Education, 2001.
- Bjarne Strautrup, " The C++ Programming Language", Addison-Wesley Publication Co.,2001.

Last Updated on April 2023

Mandatory Student Feedback Form

https://forms.gle/KS5CLhvpwrpgjwN98

Note: Students, kindly click this google form link, and fill this feedback form once.