F.Y.B.SC.IT – SEM II

OBJECT ORIENTED PROGRAMMING WITH C++ (PUSIT206T)

By,

Prof. Nikita Madwal


KEEP
CALM
AND
LOVE
PROGRAMMING

REFERENCE BOOKS :

Sr. No.	Title	Author/s	Publisher	Edition	Year
1.	Object-oriented Programming C++ Simplified	Hari Mohan Pandey	University Science Press	1st Edition	2017
2.	Object Oriented Programming in C++	E Balagurusamy	Tata McGraw-Hill	5th Edition	2011
3.	Object-Oriented Programming in C++	Robert Lafore	Sams	4th Edition	2002
4.	Programming with ANSI C++	Bhushan Trivedi	Oxford University Press	2nd Edition	2012
5.	Demystified Object-Oriented Programming with C++	Dorothy R. Kirk	Packt Publishing Lt	1st Edition	2021
6.	C++ Programming: An Object-Oriented Approach	Behrouz A. Forouzan , Richard F. Gilberg	McGraw-Hill Education	1st edition	2020
7.	C++ How to Program	Paul Deitel, Harvey Deitel	Pearson Education	10th Edition	2017


C++

UNIT 1

1. INTRODUCTION OF OBJECT-ORIENTED DESIGN

2. STARTING WITH C++

3. FEATURES OF C++

4. OPERATORS AND REFERENCES IN C++

1. INTRODUCTION OF OBJECT-ORIENTED DESIGN

Why Do We Need Object-Oriented Programming?

- Object-oriented programming was developed because limitations were discovered in earlier approaches to programming.
- To appreciate what OOP does, we need to understand what these limitations are and how they are different from traditional programming languages.

Procedure Oriented Programming

- C, Pascal, FORTRAN, and COBOL is commonly known as procedural languages
- In Procedure Oriented Programming, the problem is viewed as a sequence of a thing to be done such as reading , calculating and printing
- A program in a procedural language is a list of instructions
- The primary focus is on functions

- In this type of language the task is divided into smaller tasks or sub tasks called as procedure or function or methods.
- A procedure oriented programming language is one in which procedure i.e. method or function are more significance.

Division into Functions:

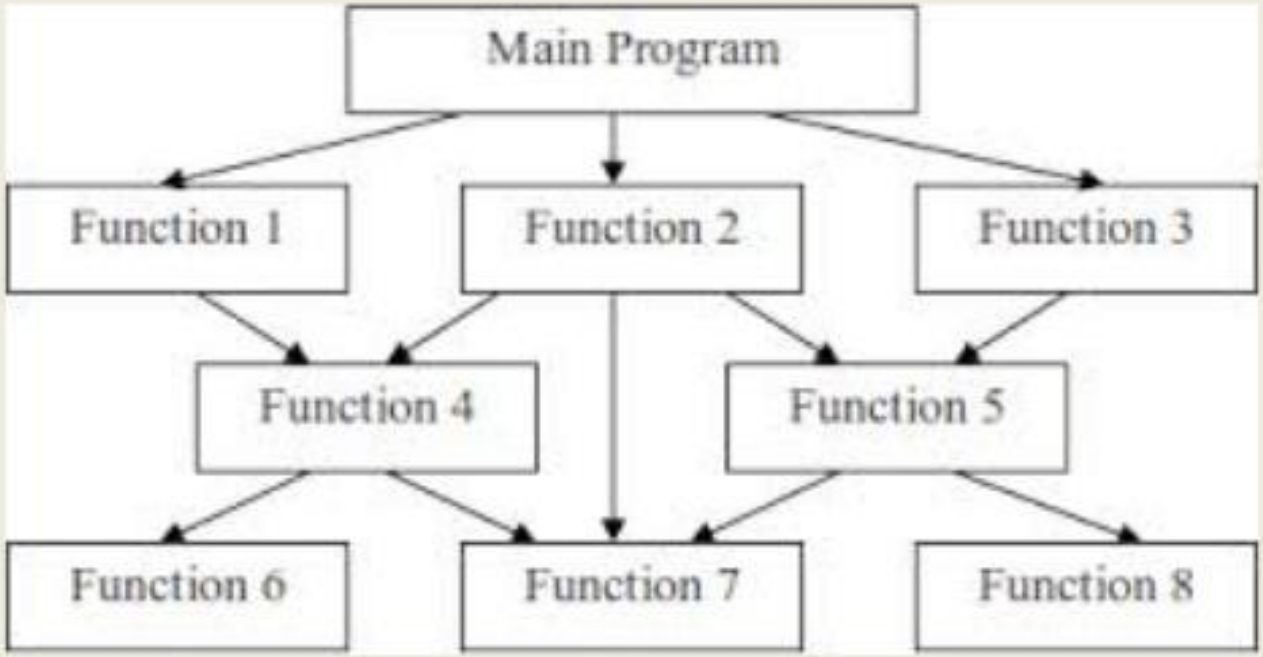


Fig 1: Structure of Procedure-Oriented Programming

Advantage and Disadvantage Of Procedure Oriented Language

- **Advantage**

- Division into Functions
- A procedural program is divided into **functions.**
- Each function has a clearly defined purpose and a clearly defined interface to the other functions in the program.
- The idea of breaking a program into functions can be further extended by grouping a number of functions together into a larger entity called a **module.**

Disadvantage

- Since every **function has complete access to the global variables,** the new programmer can **corrupt the data accidentally by creating function.**
- We can **access the data of one function from other** since**, there is no protection.**
- In **large program** it is very difficult to identify **what data is used by which function.**
- Similarly, if new data is to be added**, all the function needed to be modified to access the data.**

- In a large program, there are many functions and many global data items. The problem with the procedural paradigm is that this leads to an even larger number of potential connections between functions and data, as shown in Figure 1.2.

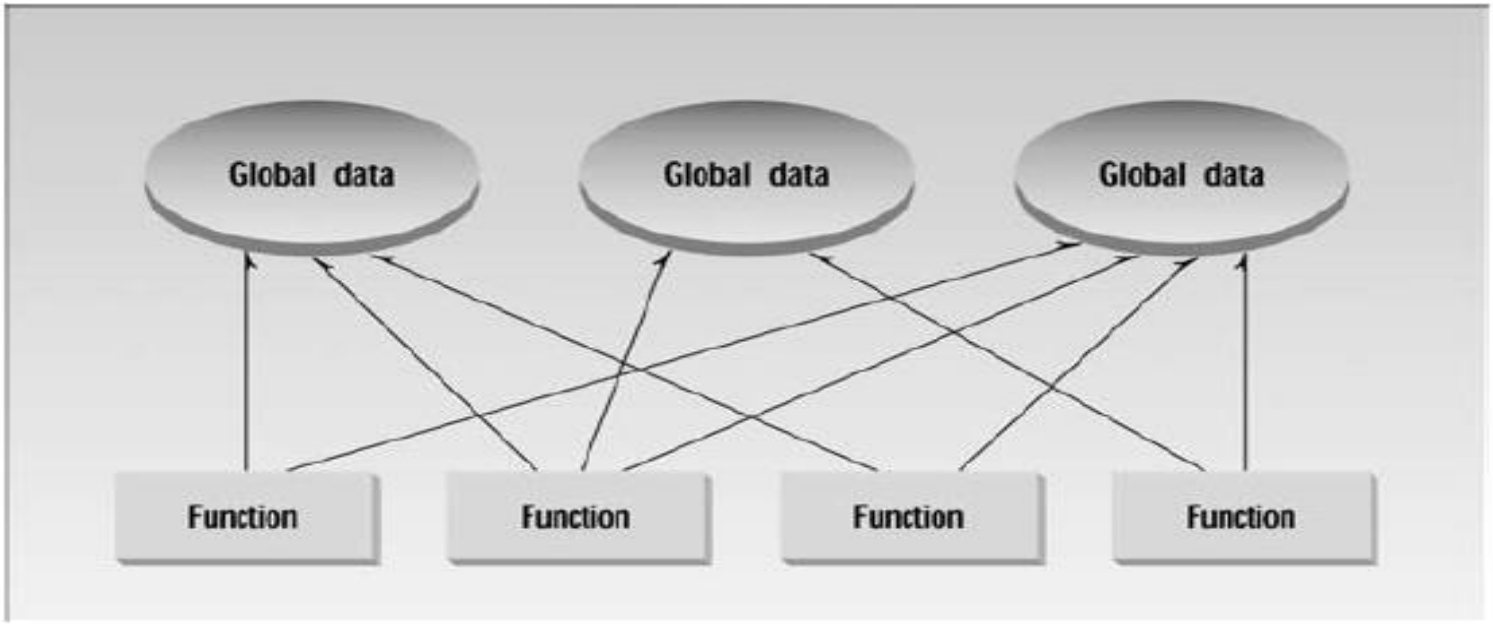


FIGURE 1.2
The procedural paradigm.

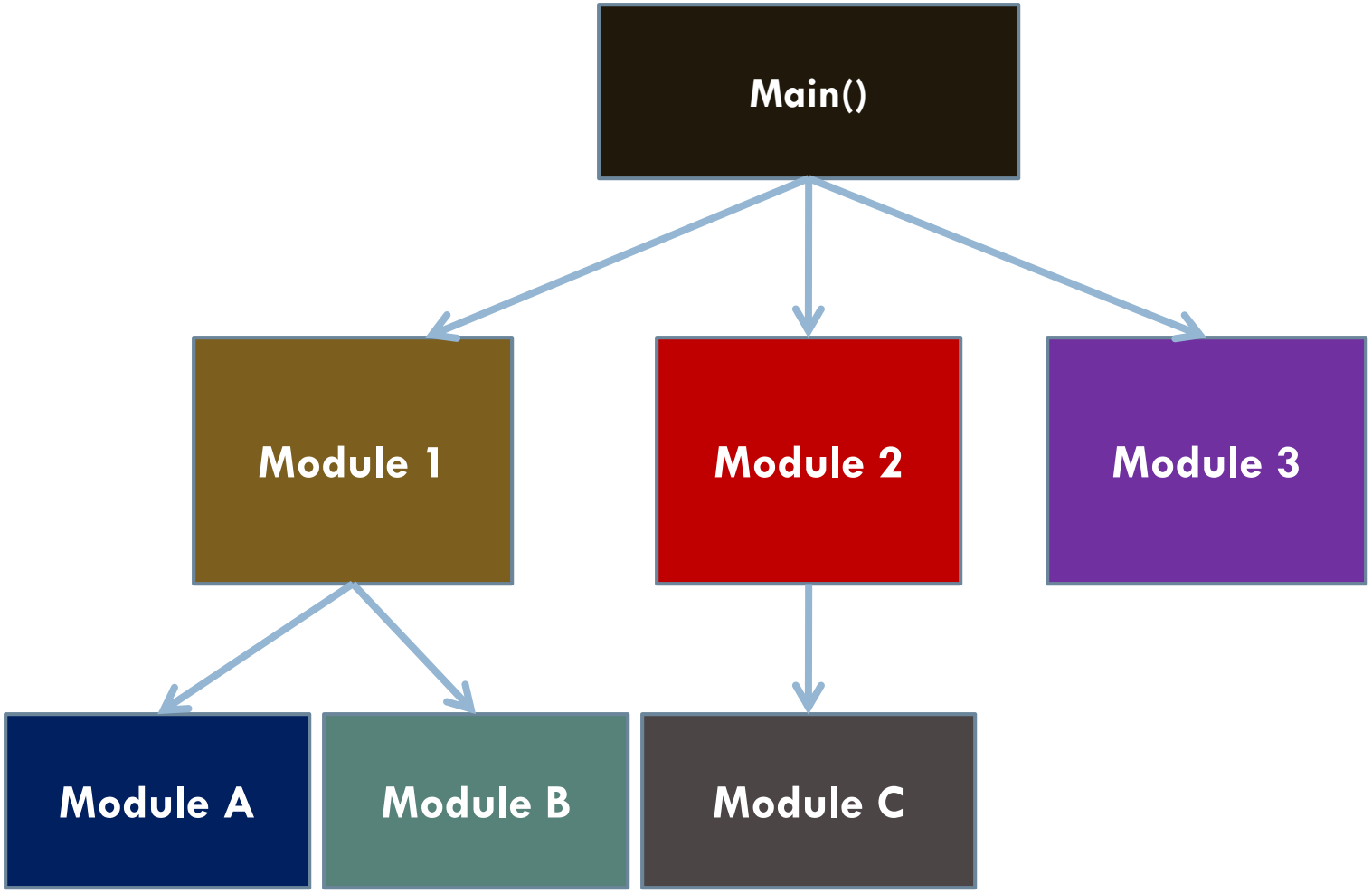
- This large number of connections causes problems in several ways.
  - **First,** it makes a program's structure **difficult to conceptualize.**
  - **Second,** it makes the program **difficult to modify.** A change made in a global data item may necessitate rewriting all the functions that access that item.
- When **data items** are modified in a **large program** it **may not be easy** to **tell which functions access the data,** and even when you figure this out, **modifications to the functions** may cause them to **work incorrectly with other global data items.**

Principles of Procedure-Oriented Programming

- **Characteristics**
- Large programs are divided into smaller programs known as **functions.**
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs **top-down approach** in program design.

Top- Down Approach

- Top-down decomposition is the process of **breaking the overall** procedure or task **into component parts(modules)** and then **subdivide each component module** until the **lowest level of detail has been reached.**

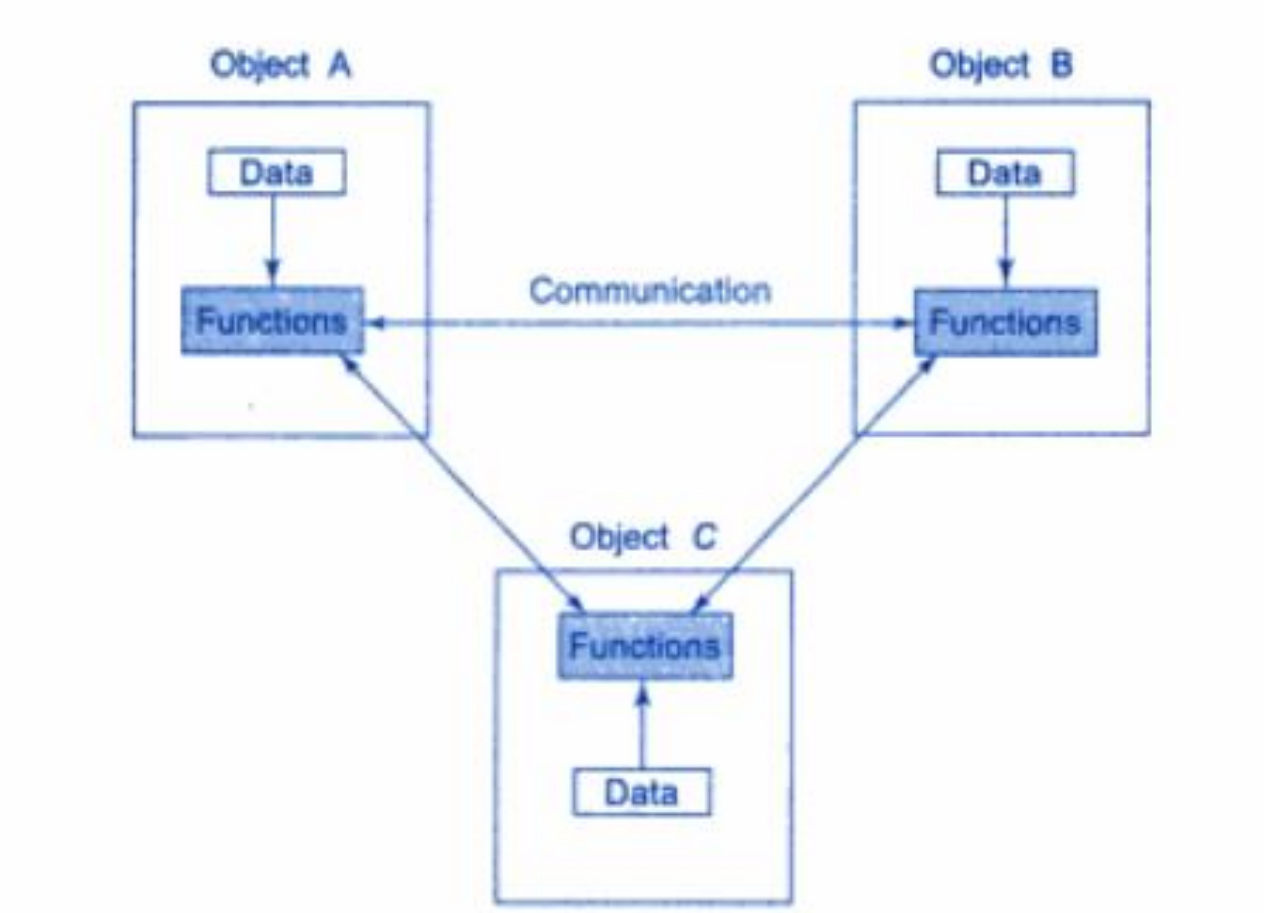

Main()
Module 1
Module 2
Module 3
Module A
Module B
Module C

Object Oriented Programming

- OOP was introduced to overcome flaws in the procedural approach programming.
- OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.
- OOP allows **decomposition of problem** into a **number of entities** called **Objects** and then builds data and function around these objects.
- The fundamental idea behind object-oriented languages is to combine into a single unit **both data and the functions** that operate on that data. Such a unit is called an **object.**

- An object's functions, called **member functions** in C++, typically provide the only way to access its data.
- The data of an object can be accessed only by the functions associated with it. The function of one object can access the data of another object trough the functions of that object
- If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can't access the data directly.

- If you want to modify the data in an object, you know exactly what functions interact with it i.e. the member functions in the object.
- The **data is hidden,** so it is safe from accidental alteration. Data and its functions are said to be encapsulated into a single entity.



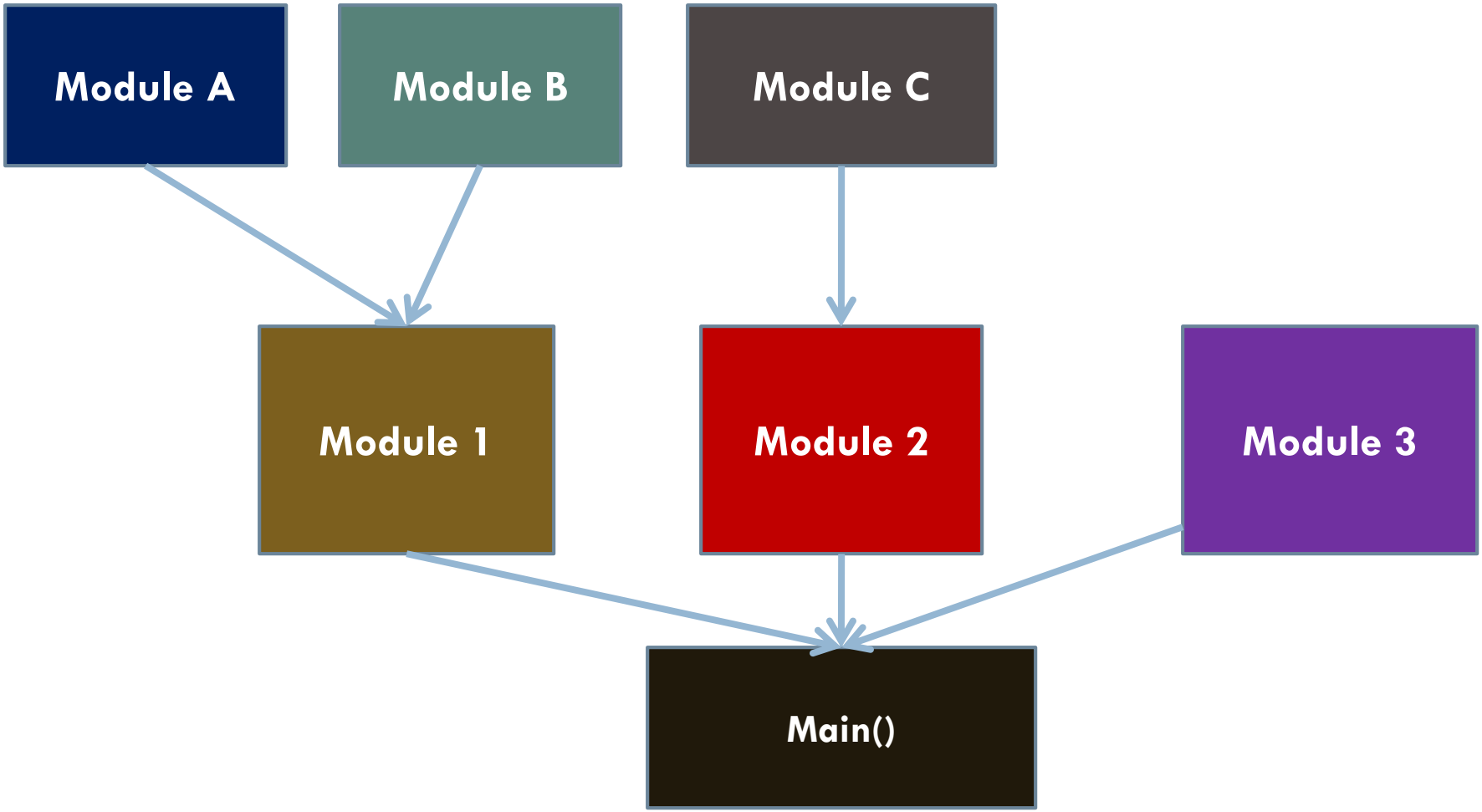
Structure of Object-Oriented Programming

Principles of Object-Oriented Programming

- **Characteristics**
- **Emphasis is on data** rather than procedure.
- **Programs** are **divided** into **entities** known as **objects.**
- Data is hidden and cannot be accessed by external functions.
- Objects communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Employs **bottom-up approach** in program design.

Bottom-Up Approach

- Reverse top-down approach.
- Lower level tasks are first carried out and are then integrated to provide the solution of a single program.
- Lower level structures of the program are evolved first then higher level structures are created i.e. (smaller tasks are first dealt in detail and gradually creating the entire huge system)
- It promotes code reuse.


Module A
Module B
Module C
Module 1
Module 2
Module 3
Main()

Difference Between POP & OOP

Procedure Oriented Programming

- POP is divided into small parts called as functions
- In POP, Importance is not given to data but to functions as well as sequence of actions to be done
- POP follows top-down approach
- POP does not have any access specifiers

Object Oriented Programming

- In OOP, program is divided into parts called objects
- In OOP, Importance is given to the data rather than procedures or functions
- OOP follows bottom-up approach
- OOP has access specifiers named Public, Private, Protected, etc

Procedure Oriented Programming

- To add new data and function in POP is not so easy
- POP does not have any proper way for hiding data so it is less secure
- In POP, overloading is not possible
- Example of POP are: C, VB, FORTRAN, Pascal

Object Oriented Programming

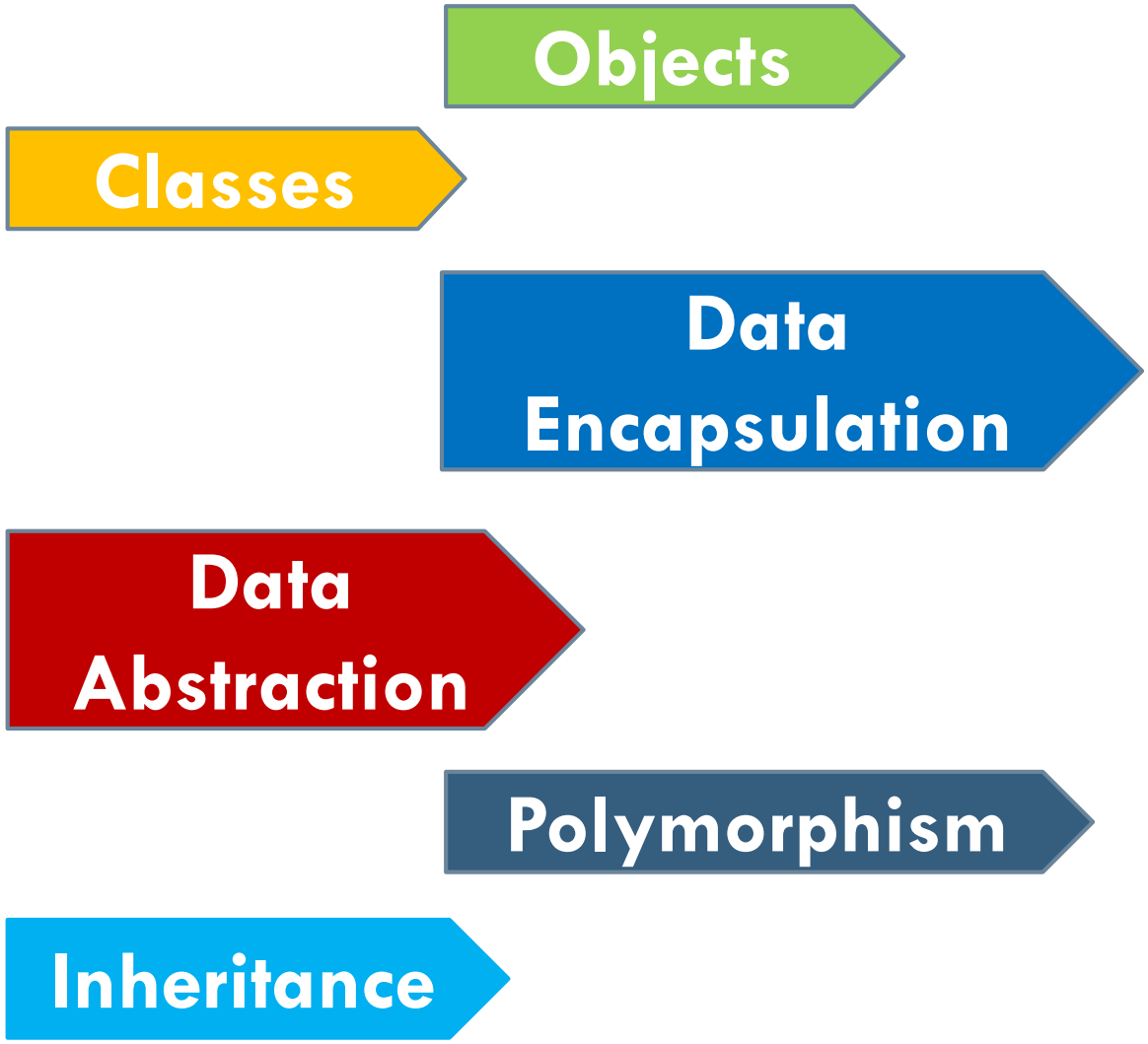
- OOP provides an easy way to add new data and function
- OOP provides data hiding so provides more security
- In OOP, overloading is possible in the form of Function Overloading and Operator Overloading
- Example of OOP are: C++, JAVA, VB.NET, C#.NET

Benefits of OOP's

- The projects executed using OOP techniques are more reliable
- Object oriented systems are easier to upgrade/modify
- OOP divides the problems into collection of objects to provide services for solving a particular problem
- **Inheritance:** Through this we can eliminate redundant code and extend the use of existing classes.
- **Data Hiding:** The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs

- **Reduced complexity of a problem:** The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
- **Easy to Maintain and Upgrade:** OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- **Modifiability:** it is easy to make minor changes in the data representation or the procedures in an OOP program. Changes inside a class do not affect any other part of a program

Basic Concepts of OOP's



Objects

- Objects are the **basic run-time entities** of an object oriented system.
- They may represent a person, a place or any item that the program must handle.
- Object is an instance of a class.
- It is also called as variable of class
- Syntax

  class-name object-name;
- E.g.

  student s1;

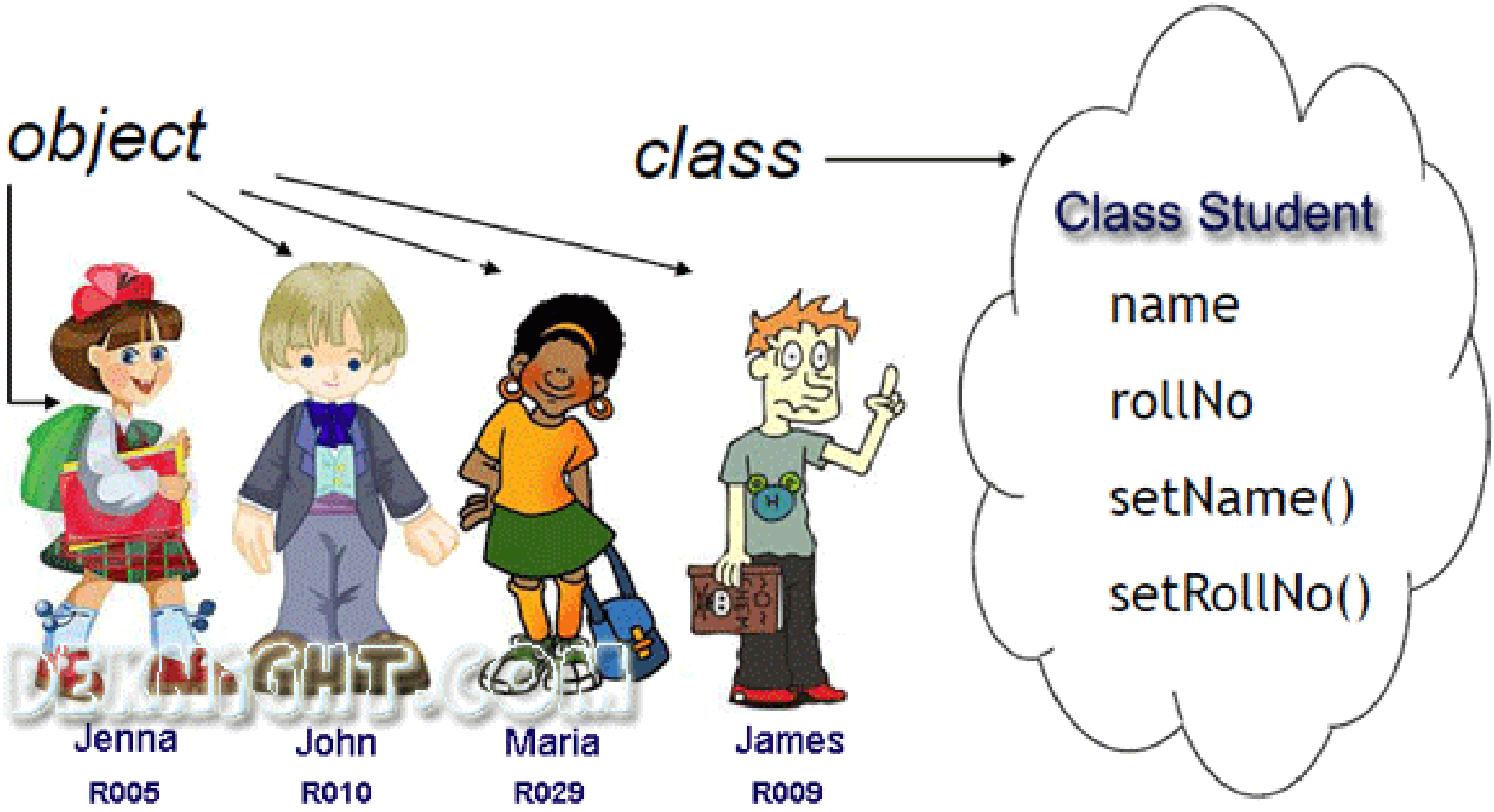
(Where student is a class name and s1 is its object)

Classes

- The most important of C++ is the class
- Classes are **user-defined data types** and it behaves like built-in types of programming language.
- Class holds its own data members and member functions , which can be accessed and used by creating instance of that class.
- It is similar to a structure with the difference that it can also have functions besides data items.
- A **class can have data members as well as function members**
- **The variables** inside class definition are called **as data members and the functions** are called **member functions.**
- **The class keyword is used**
- Each object is said to be an instance of its class

- **Syntax**

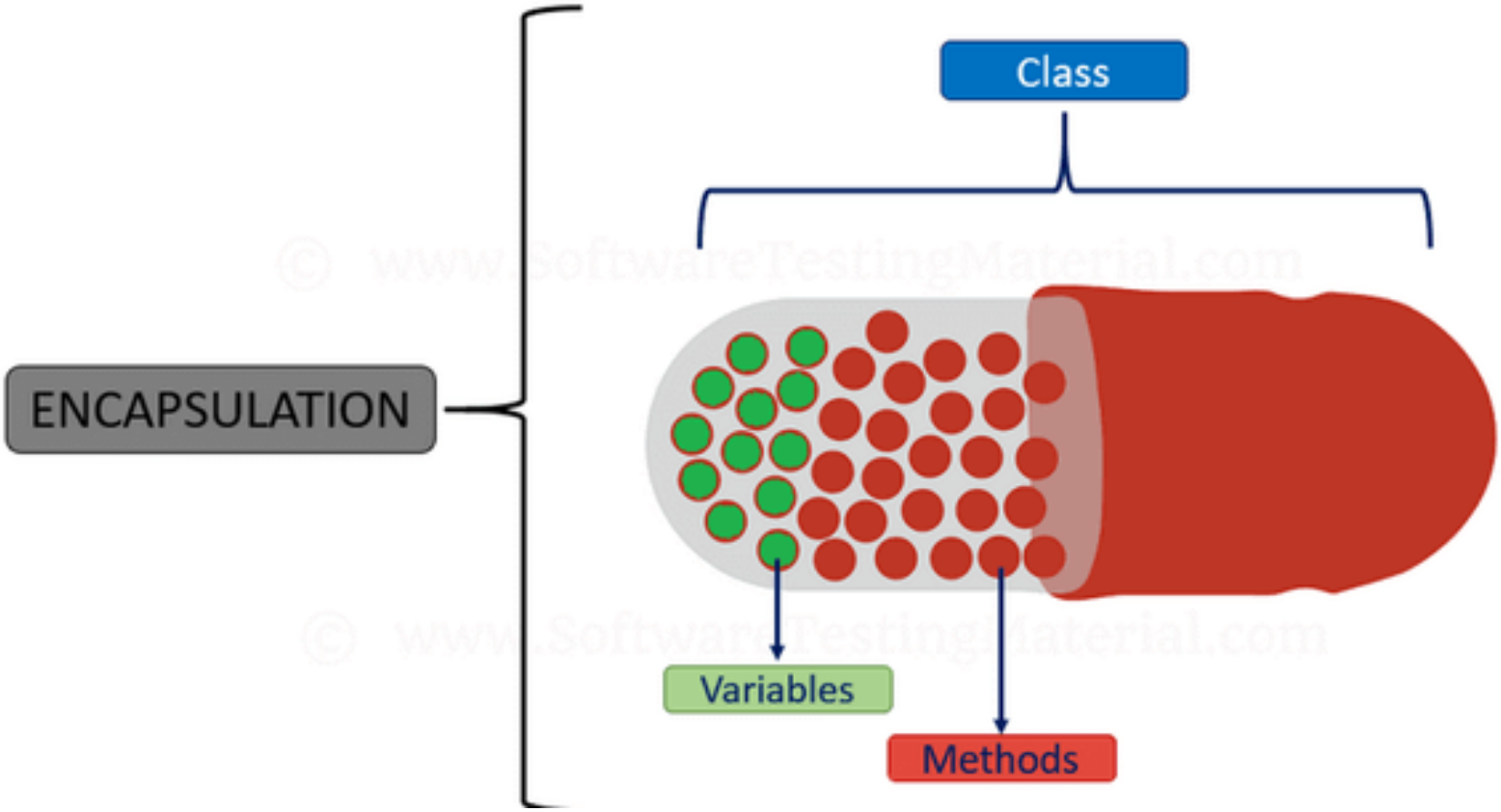
```
class class-name
{
public :
data & function;
private :
data & function;
protected :
data & function;
};
```


object
class
Class Student
name
rollNo
setName()
setRollNo()
Jenna
R005
John
R010
Maria
R029
James
R009

Data Encapsulation

- Encapsulation is the mechanism that binds together function and data in one compact form known as class.
- Encapsulation is a process of **binding data members** (variables, properties) and **member functions** (methods) into a single unit.
- Through encapsulation, Data is not accessible to the outside world, and only those functions which are wrapped in the class can access it.
- This can be done using access specifiers
- The data and function may be private or public. Private data/function can only be accessed only within the class. Public data/code can be accessed outside the class.
- Class is the best example of encapsulation.

Data Encapsulation



Data Abstraction

- **Abstraction** refers the representation of necessary features without including more details or explanations.
- It basically deals displaying only essential information and hiding the details
- **Abstraction** lets you focus on **what the object does** instead of how it does it.
- hidden part of a class acts like Encapsulation and exposed part of a class acts like Abstraction
- For example, **when you press a key on your keyboard the character appears on the screen,** you need to know only this, but how exactly it works electronically is not needed. This is called **Abstraction**.


Encapsulation
Abstraction
SAMSUNG
careerbaba.in

Polymorphism

- “Poly” means many and “Morphism” means form
- Thus, polymorphism means more than one form.
- Thus polymorphism provides a way for an entity to behave in several forms
- **Polymorphism defines the way in which a single task can be performed in multiple ways by the user.**
- For example, “+” is used to make sum of two numbers as well as it is used to combine two strings.
- This is known as operator overloading because same operator may behave differently on different instances

- Similarly, functions can be overloaded
- For example, sum() function takes two arguments or three arguments, etc. e.g. sum (8,4) or sum (4,8,9).

Polymorphism

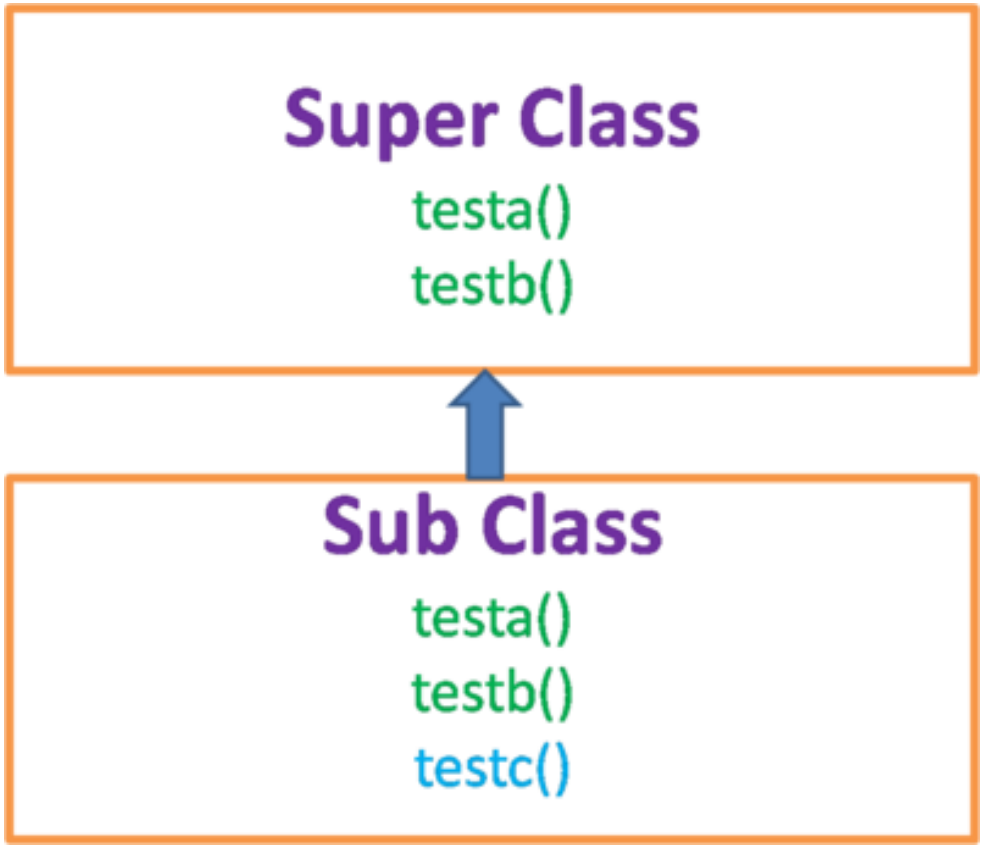
- In Shopping mall behave like customer
- In Bus behave like Passenger
- In School behave like Student
- At home behave like son

Inheritance

- The mechanism of **deriving** a **new class** from an **old class** is called **inheritance** or **derivation**
- Inheritance is the process by which **objects of one class *acquire*** the **properties** of **objects of another class**
- The **old class** and ***new class*** is called (given as pair) **base** - ***derived***, **parent** - ***child***, **super** - ***sub***
- **The basic motto of inheritance is creating a new class inheriting the properties/functionalities of an already existing class**
- The inheritance supports the idea of classification.
- In classification we can form hierarchies of different classes each of which having some special characteristics besides some common properties.

Inheritance

Inheritance



Object-Oriented Analysis

- The purpose of object-oriented analysis, as with all other analysis, is to obtain an understanding of the application : an understanding depending only on the system's functional requirements.
- Object-oriented analysis contains, in some order, the following activities :
  - Finding the object.
  - Organizations the objects.
  - Describing how the objects interact.
  - Defining the operations of the objects.
  - Defining the objects internally.

Finding the Objects

- The objects can be found as naturally occurring entries in the application domain.
- The **aim is to find the essential objects,** which are to **obtain which are to remain essential throughout the system life cycle.**
- Stability also depends on the fact that modifications often begin from some of these items and therefore are local.
- For example, in a application for controlling for water tank, typically objects would include contained water, regulator, valve and tank

Conceptual Modeling

- It has been used in several different contexts since it appeared in the 1970s; example analysis of information management system and organization theory.
- The **aim is to create models of the system or organization to be analyzed.**
- The concepts of conceptual modeling is often used as a synonym for data modeling and is often discussed with structuring and the use of the databases

Requirements Model

- The first transformation made is from the requirement specification to the requirement model.
- The requirements model consists of :

(a) A use case model.

(b) Interface descriptions.

(c) A problem domain model.

- The use case model uses actors and use cases.
- **These concepts are simply an aid to defined what exits outside the system (actors) and what should be performed by the system (use case).**

Analysis Model

- We have seen that the requirements model aims to define the limitations of the system and to specify its behavior.
- When the requirement model has been developed and approved by the system users or orders, we can start to develop the actual system.
- This starts with development of the analysis model.
- Its model **aims to structure the system independently of the actual implementation environment.**
- This means that we focus on the **logical structure of the system.** It is here that we define the stable, robust and maintainable structure that is also extensible.

The Design Model

- In the construction process, we construct the system using both the analysis model and the requirements, model.
- First, we create a design model that is a refinement and formalization of the analysis model.
- **Design emphasizes a conceptual solution that fulfils the requirements, rather than its implementation.**
- For example, a description of a database schema and software objects.
- Ultimately, designs can be implemented.
- As with analysis, the term is best qualified, a in object design or database design.
- Analysis and design have been summarized in the phase do the right thing (analysis), and do the things right (design).

The Implementation Model

- **The implementation model consists of the annotated source code.**
- The information space is the one that the programming language uses.
- **Note** that we do not require an Object-oriented Programming language; the technique may be used with any programming language to obtain an object-oriented structure of the system.
- However, an Object-oriented-Programming language is desirable since all fundamental concepts can easily be mapped onto language constructs.

Test Model

- **The test model is the last model developed in the system development.**
- It describes, simply stated, **the result of the testing.**
- The fundamental concepts in **testing are mainly the test specification and the test result.**

The Evolution of Object Model

- The generation of programming languages as we look back upon the relatively briefly yet colorful history of software engineering.
- We cannot help but notice two sweeping trends.
- **The shift in focus from programming in the small to programming in the large.**
- The evolution of high order programming languages.
- Foundations of Object Model
- **Structured design methods evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks.**

- Similarly **object-oriented design methods have evolved to help developers exploit the expressive power of object based and object-oriented programming languages** using the **class and object** as basic building blocks.
- **Object-oriented analysis and design thus represents an evolutionary development,** not a revolutionary one; **it does not break with advances from the past, but builds upon proven ones.**

OBJECT-ORIENTED DESIGN

- Object-oriented design **is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.**
- There are two important parts to this definition :
- Object-oriented design leads to an object-oriented decomposition(**identifies individual autonomous objects in a system and the communication among these objects**).
- It uses different notations to express different models of the logical and physical design of a system, in addition to the static and dynamic aspects of the system.

Object-Oriented Analysis

- Object-oriented analysis **is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.**

How are OOA, OOD, and OOP related ?

- Basically, **the products of object-oriented analysis serve as the models from which we may start an object-oriented design; the products of object oriented design can then be used as blueprints for completely implementing a system using object-oriented programming methods.**

Elements of Object Model

- There are four **major elements** of its model:

(a) Abstraction

(b) Encapsulation

(c) Modularity

(d) Hierarchy

- There are three **minor elements** of the object model

(i) Typing

(ii) Concurrency

(iii) Persistence

Major Elements

(a) Abstraction

- An abstraction **denotes the essential characteristics** of an object that distinguish it from all other kinds of objects.
- Thus, it provides crisply defined conceptual boundaries, relative to the perspective of the viewer.

(b) Encapsulation

- Encapsulation is the **process of compartmentalizing the elements of an abstraction that constitutes its structure and behavior.**
- Encapsulation serves to separate the contractual interface of an abstraction and its implementation.
- **Abstraction and encapsulation are complimentary concepts.**

(c) Modularity

- Modularity **is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.**

(d) Hierarchy

- Hierarchy is a ranking of ordering of abstractions.
- The **two most important hierarchies** in a complex system are its class structure(‘**is a**’ or generalization/specialization) and object structure(‘**part of**’ or whole part)

- **"IS–A" hierarchy –**
- It defines the hierarchical relationship **in inheritance**, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on.
- For example, if we derive a class Rose from a class Flower, we can say that a rose "is–a" flower.
- **"PART–OF" hierarchy –**
- It defines the hierarchical relationship **in aggregation** by which **a class may be composed of other classes.**
- For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a "part–of" flower.
- **Inheritance is the most important most important hierarchy**
- Basically, **inheritance defines a relationship among classes wherein one class shares the structure or behavior defined in one or more classes.**

Minor Elements

(i) Typing

- **Concepts of typing derive primarily from theories of abstract data types.**
- A type is a precise characterization of structural or behavioral which a collection of entities all share.
- Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged or at the most, they may be interchanged only in very restricted ways.

(ii) Concurrency

- It is the property that distinguishes an active object from one that is not active.

(iii) Persistence

- **An object occupies a memory space and exists for a particular period of time.**
- In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it.
- In files or databases, the object lifespan is longer than the duration of the process creating the object.
- This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

The Role of OOAD in the Software Life Cycle

- We know that the object oriented modelling (OOM) technique visualizes things in an application by using models organized around objects.
- Any software development goes through the following stages –
  - **Analysis**
  - **Design and**
  - **Implementation**
- In object oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools

- **Phases in Object-Oriented Software Development**

The major phases of software development using object-oriented methodology are **object-oriented analysis, object-oriented design and object-oriented implementation.**

- **Object – Oriented Analysis**

- In this stage, the problem is formulated **,user – requirements are identified, and then a model is built based upon real-world objects.**
- The analysis produces models on how the desired system should function and how it must be developed.

- **Object – Oriented Design**
- Object – Oriented design **includes 2 main stages,** namely, **system design and object design**
- **System Design**
- In this stage, the **complete architecture of the desired system is designed.**
- The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes.

Object Design

- In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. **All the classes required are identified,**
- The designer decides whether –
- New classes are to be created from scratch,
- Any existing classes can be used in their original form, or
- New classes should be inherited from the existing classes.
- The developer designs the internal details of the classes and their associations.

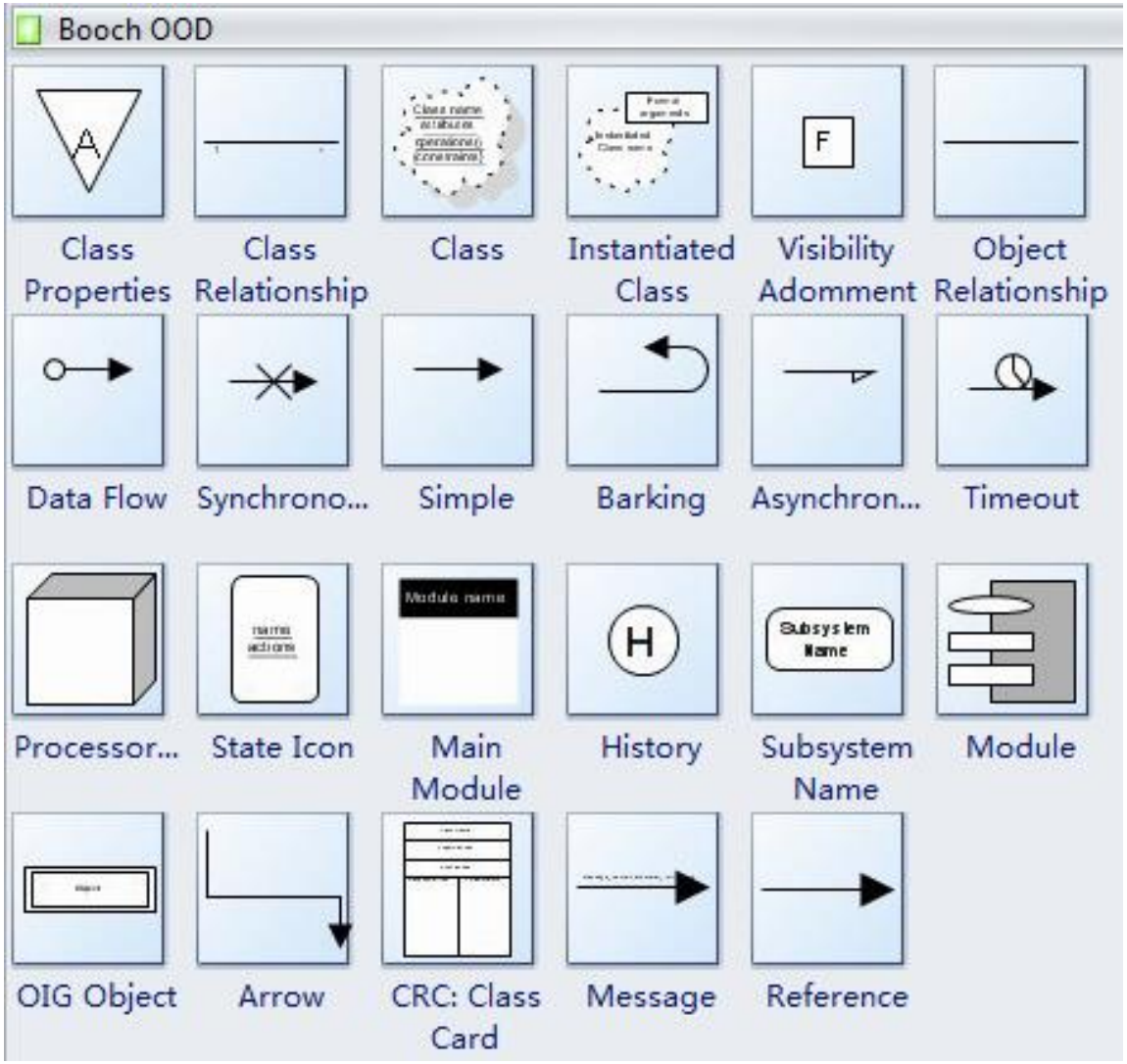
- **Object – Oriented Implementation and Testing**
- In this stage, **the design model developed in the object design is translated into code in an appropriate programming language or software tool.**
- The **database are created and the specific hardware requirements are ascertained.**
- **Once the code is in shape,** it is **tested using specialized techniques to identify and remove the errors in the code.**

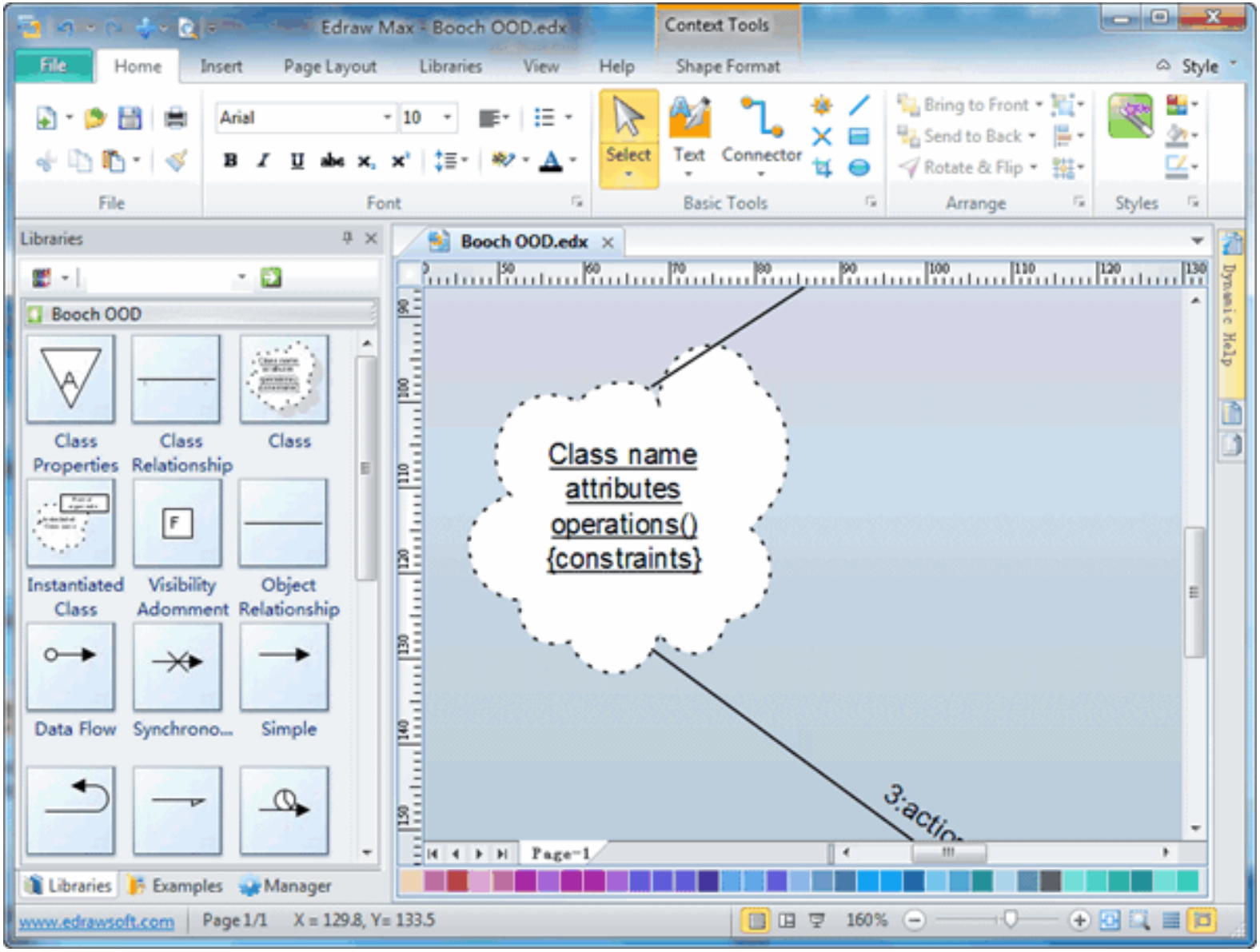
OOAD Methodologies

- OOAD methodologies fall into two basic types.
- The **ternary type** (or three-pronged) is the natural evolution of existing structured methods and has three separate notations for **data, dynamics, and process.**
- The **unary type** asserts that because objects combine processes (methods) and **data,** only one notation is needed.
- **The unary type is considered to be more object-like** and easier to learn from scratch, but has the disadvantage of producing output from analysis that may be impossible to review with users.
- **Dynamic modeling is concerned with events and states, and generally uses state transition diagrams.**
- **Process modeling or functional modeling is concerned with processes that transform data values, and traditionally uses techniques such as data flow diagrams.**

Grady Booch Approach

- Grady Booch's approach to OOAD is one of the most popular, and is supported by a variety of reasonably priced tools ranging from Visio to Rational Rose.
- Booch is the chief scientist at Rational Software, which produces Rational Rose. (Now that James Rumbaugh and Ivar Jacobson have joined the company, Rational Software is one of the major forces in the OOAD world).
- **While the Booch methodology covers requirements analysis and domain analysis;**
- **Booch's notation is generally regarded as the most complete one for the representing object- oriented systems.**
- its major strength has been in design.
- **Grady Booch's Object-Oriented Design (OOD),also known as Object-Oriented Analysis and Design (OOAD),is a precursor to the Unified Modelling (UML).**


Booch OOD
Class Properties
Class Relationship
Class
Instantiated Class
Visibility Adomment
F
Object Relationship
Data Flow
Synchrono...
Simple
Barking
Asynchron...
Timeout
Processor...
State Icon
Module name
Main Module
H
History
Subsystem Name
Module
OIG Object
Arrow
CRC: Class Card
Message
Reference


Edraw Max - Booch OOD.edx
Context Tools
File
Home
Insert
Page Layout
Libraries
View
Help
Shape Format
Style
Arial
10
Select
Text
Connector
Bring to Front
Send to Back
Rotate & Flip
File
Font
Basic Tools
Arrange
Styles
Libraries
Booch OOD
Class Properties
Class Relationship
Class
Instantiated Class
Visibility Adomment
Object Relationship
Data Flow
Synchrono...
Simple
Libraries
Examples
Manager
Booch OOD.edx
Class name
attributes
operations()
{constraints}
Page-1
Dynamic Help
www.edrawsoft.com
Page 1/1
X = 129.8, Y= 133.5
160%

- **Principles of Modeling**
- The use of modeling has a rich history in all the engineering disciplines.
- **That experience suggests four basic principles of modeling.**

1. The choice of what models to create has profound influence on how a problem is attacked and how a solution is shaped.

2. Every model may be expressed at different levels of precision.

3. The best models are connected to reality.

4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

2. STARTING WITH C++

C++ Overview

- C++ is an object-oriented programming language.
- It was developed by **Bjarne Stroustrup at AT & T's bell laboratories in the Murray Hill. New jersey, USA in the early 80's.**
- It's an extension to C with number of new features added.
- The named **C++** came from the increment operator + + of C as it is **considered a super set of C**, one version ahead of C, so it was named C++.

C++ Character Set

- A C++ program is a collection of number of instructions written in a meaningful order.
- Further instructions are made up of keywords, variables, functions, objects etc., which uses the C++ character set defined by C++.
- **Character set is a collection of various characters, digits and symbols which can be used in a C++ program.**
- It comprises followings :

Table 2.1

S.No.	*Elements of C++ character set*
1.	Upper Case letters : A to Z
2.	Lower Case letters : a to z
3.	Digits : 0 to 9
4.	Symbols (See below table)

Table 2.2 : C++ Character Set

Symbols	*Name*	*Symbols*	*Name*
~	Tilde	>	Greater than
<	Less than	&	Ampersand
|	Or/pipe	#	Hash
>=	Greater than equal	<=	Less than equal
= =	Equal	=	Assignment
!=	Not equal	^	Caret
{	Left brace	}	Right brace
(	Left parenthesis	)	Right parenthesis
[	Left square bracket	]	Right square bracket
/	Forward slash	\	Backward slash
:	Colon	;	Semicolon
+	Plus	–	Minus
*	Multiply	/	Division
%	Mod	,	Comma
‘	Single quote	“	Double quote
>>	Right shift	<<	Left shift
.	Period	_	Underscore

C++ TOKENS

- **Smallest individual unit in a C++ program is called C++ token.**
- **C++ defined six types of tokens.**

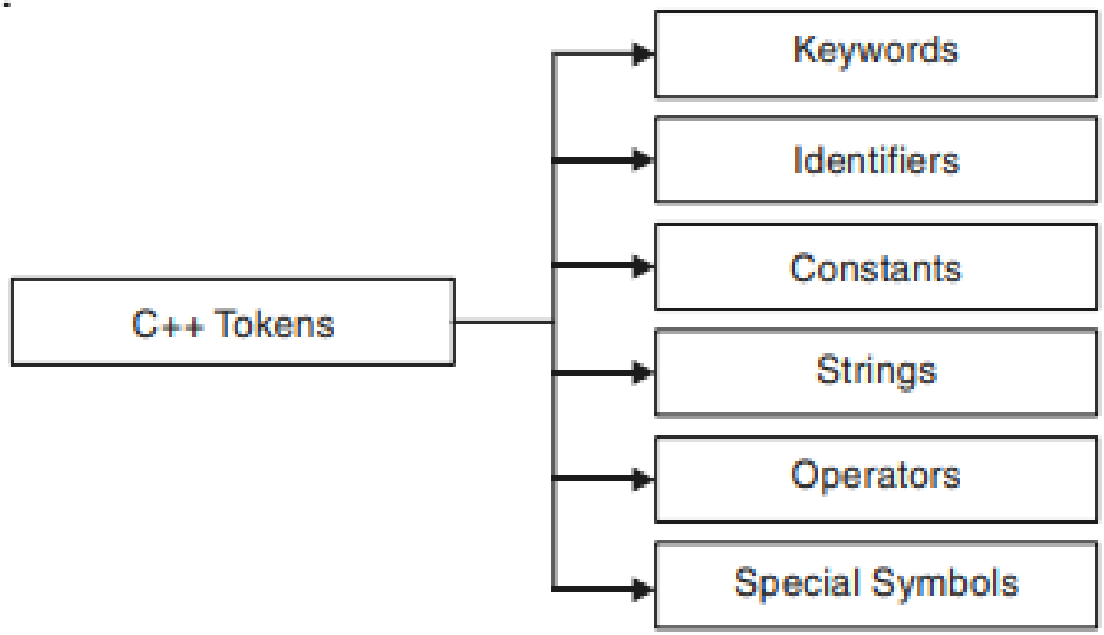


Figure 2.1. *Six types of tokens defined in C++.*

1. Keywords

- **Keywords are those words whose meaning has already been known to the compiler.**
- That is **meaning** of each keyword is **fixed.**
- The keywords are **used for** its **intended meaning.**
- The **meaning** of keywords **cannot be changed.**
- Also the keywords **cannot be used as names** for variables, function, array etc.
- All keywords are **written in small case letters.**
- There are **63 keywords** in C++.
- Following figure lists all keywords available in C++.

Table 2.3 : C++ keywords

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

2. Identifier

- **Identifiers are names given to various program elements like variables, array, functions, structures etc.**
- **Rules for Writing Identifiers :**
- Rule 1 : First letter must be an alphabet or underscore.
- Rule 2 : From second character onwards any combination of digits, alphabets or underscore is allowed.
- Rule 3 : Only digits, alphabets, underscore are allowed. No other symbol is allowed.
- Rule 4 : Keywords cannot be used as identifiers.

- Rule 5 : For ANSI (American National Standard Institute) C++ maximum length of identifier is 32, but many compiler support more than 32.
- Example of valid and invalid identifiers (on the basis of above given rules)

Valid Identifiers :

order_no	name	_err	_123
xyz	radius	a23	int_rate

Invalid Identifiers :

order-no	12name	err	int
x$	s name	hari+45	123

3. Constants

- **Constants in C++ refer to fixed values that do not change during the execution of a program.**
- There are various types of constants in C++.

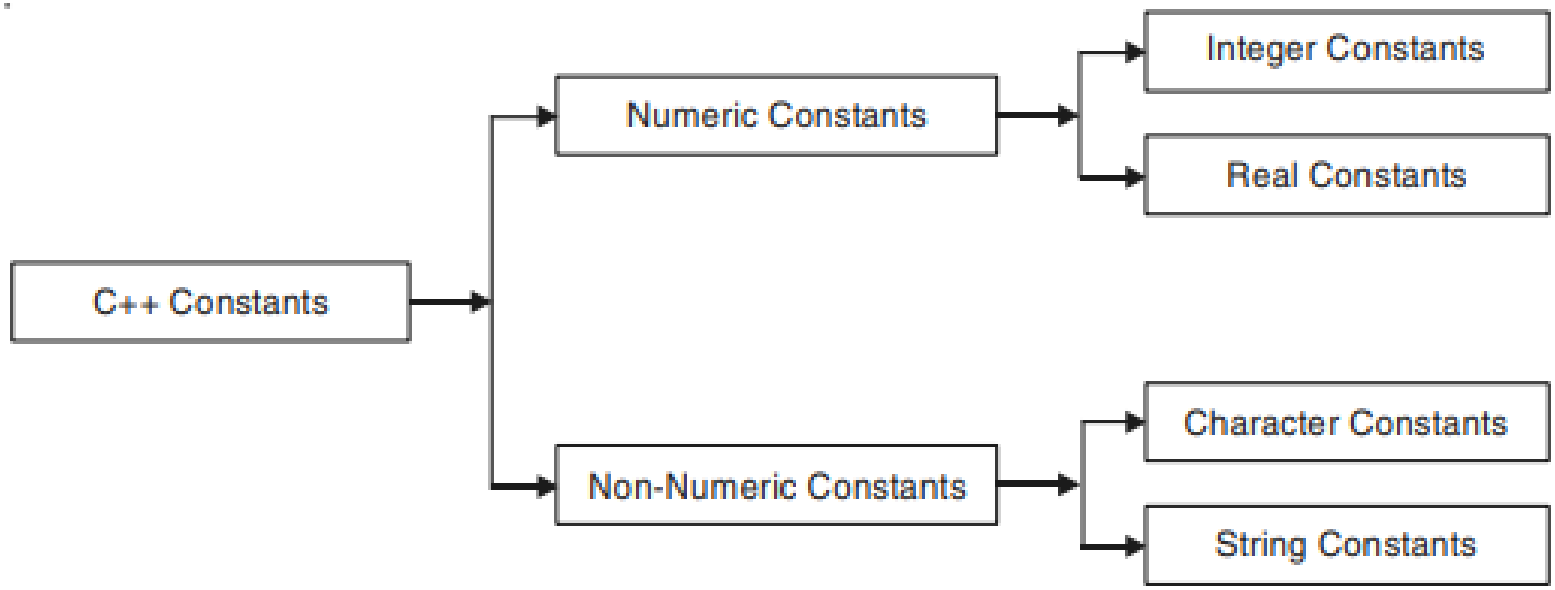


Figure 2.2. *C++ constants.*

(i) Integer Constants

- They are of three types :

(a) Decimal Constants :

- They are sequence of **digits from 0 to 9 without fractional part.** It may be negative, positive or zero.
- Example : 12, 455, -546, 0 etc.

(b) Octal Constant :

- They have sequence of numbers **from 0 to 7 and first digit must be 0**
- Example : 034, 0, 0564, 0123 etc.

(c) Hex Constant :

- They have sequence of digits **from 0 to 9 and A to F** (represents 10 to 15).
- They start with 0x or 0X.
- Example : 0x34, 0xab3, 0X3E etc.

(ii) Real Constant

- They are the number with fractional part.
- They also known as **floating point constants.**
- Example : 34.56, 0.67, 1.23 etc.
- Real constants **can also be represented in exponential or scientific notation which consists of two parts.**
- For example, the number 212.345 can be represented as 2.12345e + 2 where e + 2 means 10 to the power 2.
- Here the portion before the e that is 2.12345 is known as mantissa and +2 is the exponent.
- Exponent is always an integer number which can be written either in lower case or upper case.

- **(iii) Single Character Constants**
  - They **are enclosed in single quote.**
  - They **consist of single character or digit.**
  - Example : ‘4’, ‘A’, ‘\n’ etc.
- **(iv) String Constants**
  - They are **sequence of characters, digits or any symbol enclosed in double quote.**
  - Example : “hello”, “23twenty three”,”&^ABC”. “2.456” etc

(v) Backslash Constants

- **C++ defines several backslash constants which are used for special purpose.**
- Each backslash constant **starts with backslash (\).**
- They are represented with 2 characters whose general form is \char but treated as single character.
- They are **also called escape sequence.**
- Given below the list of backslash character (escape sequence) character constants :

Table 2.4: Table shows the Backslash Character Constant

S.No.	*BCC*	*Meaning*	*ASCII*
1	\b	Backspace	08
2	\f	Form feed	12
3	\n	New line	10
4	\r	Carriage return	13
5	\"	Double quote	34
6	\'	Single quote	39
7	\ ?	Question mark	63
8	\a	Alert	07
9	\t	Horizontal tab	09
10	\v	Vertical tab	11
11	\0	Null	00

- **4. Strings**
  - See string constants.
- **5. Operators**
  - They are discussed in chapters 3 and 4.
- **6. Special Symbols**
  - They are also known as separator and they are square brackets [], braces { }, parenthesis () etc.
  - They [ ] used in array and known as subscript operator, the symbol () is known as function symbols.

VARIABLES

- **A variable is a named location in memory that is used to hold a value that can be modified in the program by the instruction.**
- All variables must be declared before they can be used.
- They must be declared in the beginning of the function or block (except the global variables).
- The general form of variable declaration is :

```
data type variable [list];
```

- Here list denotes more than one variable separated by commas;
- Example :

```
int a;
float b,c;
char p,q;
```

- Here a is a variable of type int, b and c are variable of type float, and p, q are variables of type char, int, float and char are data types used in C.
- The rule for writing variables are same as for writing identifiers as a variables is nothing but an identifier.

- **C++ also allows you to initialize variable dynamically at the place of use.**
- That is you can write anywhere in the program like this.

```
int x = 23;
float s = 2345;
char name[ ] = “Hari”;
```

- **This is known as “Dynamic Initialization” of the variables.**

Counting Tokens

1. int a,b,c;

(*i*) int is a keyword (1)

(*ii*) a, b and c variables (3)

(*iii*) comma (,) is operator (2)

(*iv*) ; is delimiter (1)

So, total number of tokens (1 + 3 + 2 + 1= 7)

2. c = a + b – 10;

(*i*) a, b and c variables (3)

(*ii*) +, =, – are operators (3)

(*iii*) ; is delimiter (1)

(*iv*) 10 is integer constant (1)

So, total number of tokens (3 + 3 + 1 + 1 = 8)

Data Types

- A data type specifies the type of data that a variable can store such as integer, floating, character etc,.
- Based on the data type of a variable, the operating system allocated memory.

1. Built-in Type

- (*a*) Integral Type
  - (*i*) int
  - (*ii*) char
- (*b*) Floating Types
  - (*i*) float
  - (*ii*) double
- (*c*) void

2. User Defined Data Type

- (*a*) class
- (*b*) struct
- (*c*) union
- (*d*) enumeration

3. Derived Data Types

- (*a*) array
- (*b*) function
- (*c*) pointer
- (*d*) reference

QUALIFIERS

- **Qualifiers qualify the meaning of data types. Unsigned, signed, short and long are the 4 qualifiers available in C++.**
- The manner in which they are used as follows
  - signed int x, y, b;
  - unsigned int p,q;
  - long int num;
  - short int n;

RANGE OF DATA TYPES

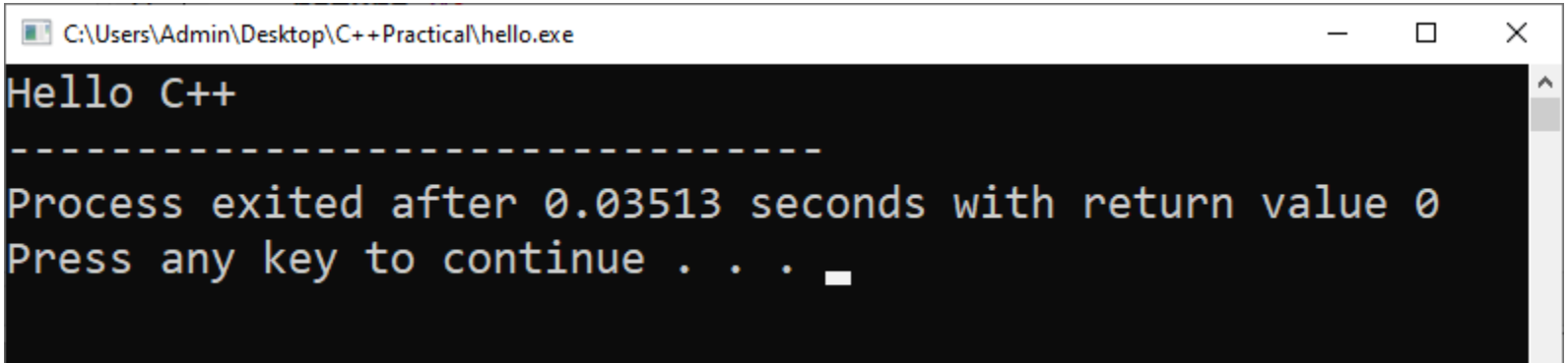
Table 2.5 : Data Type and Their Range

S.No.	*Data Type*	*Size(in byte)*	*Range*
1	int or short	2	–32768 to 32767
2	Unsigned int	2	0 to 65535
3	Long int	4	–2147483648 to 2147483647
4	Float	4	3.4e-38 to 3.4e+38
5	Char	1	–128 to 127
6	Unsigned char	1	0 to 255
7	Unsigned long	4	0 to 4294967295
8	Double	8	1.7e-308 to 1.7e+308
9	Long double	10	3.4e-4932 to 3.4e+4932

Your First C++ Program

hello.cpp

```cpp
// Simple C++ program to display "Hello C++"
// Header file for input output functions
#include<iostream>
using namespace std;

// main function -
// where the execution of program begins
int main()
{
    cout<<"Hello C++";/* prints hello C++*/
    return 0;
}
```



STRUCTURE OF A C++ PROGRAM

- The general structure of any C++ program is given below. It simply states that a standard C++ program may look like according to various sections presents in the structure.

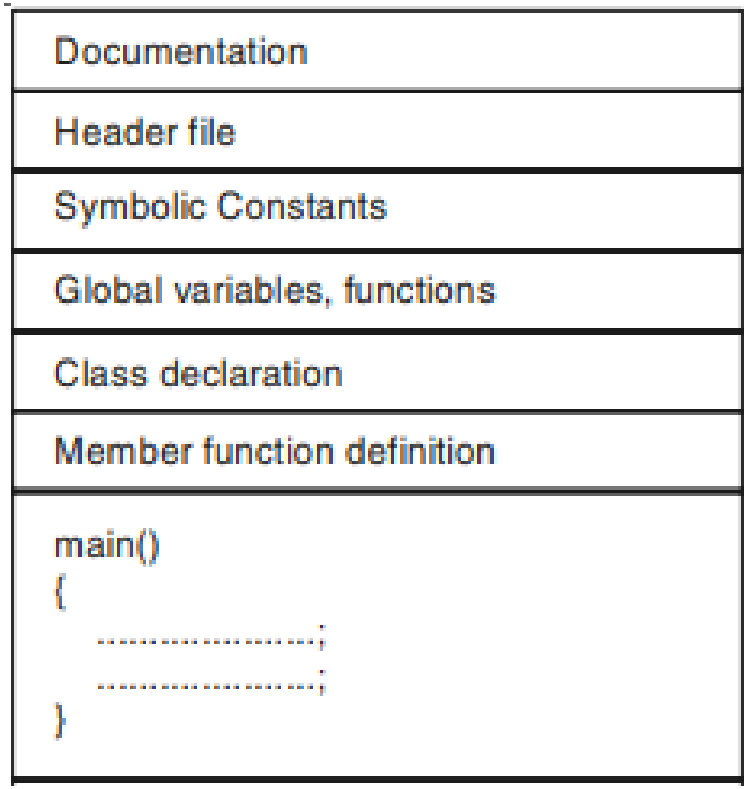


Figure 2.10. *General structure of a C++ program.*

- The **first** section **Documentation** is **optional** and is used to **put comments for the program** usually the program heading as we have given.
- In **second** section we include various **header files required by our C++ program.**
- **Symbolic constants and global variable,** functions are **defined after that if required.**
- The **class, main building block of C++ programming** is declared which consists of **declaration of data members and functions.** The member **function** *may be* **declared and defined in the class** *or* they are **declared in the class but defined outside the class** which is done outside the class declaration.

- The **main() function** must be present in every C++ program. It **contains all the statements which are to be executed**. Inside this main function **we create objects of class created earlier. All statements** are **put inside the braces** and **terminate with semicolon.**

STYLES OF WRITING C++ PROGRAMS

```
#include <iostream>
using namespace std;
void main()
{
    cout<<"hello C++"<<endl;
}

or
#include <iostream>
    int main()
 {
    std : :cout<<"hello C++"<<endl;
    return 0;
 }
```

- iostream.h is header file and iostream is a header file too.
- The difference is simply that **iostream provides global namespace std** under which **cout and cin are declared.**
- To access ***cout and cin you will have to write std : :cout and std : :cin if using namespace std is not written.***
- The **return type of function may be void or int.** In case of int type of main, the main must return a value.

3. FEATURES OF C++

INTRODUCTION

- There are lots of new features in C++ but there are features whi.ch are in common in both C and C++
- Most of the features like operators, if-else, loops, goto, arrays etc are discussed in this chapter.

OPERATORS AND EXPRESSIONS

- **Operators**
- **An operator is a symbol that operates on a value to perform specific mathematical or logical computations.**
- They form the foundation of any programming language.
- An operator operates the operands.
- E.g., int c=a+b;

Here, '**+**' is the addition **operator**. ***'a' and 'b'*** are the ***operands*** that are being added.

- **Expression**
- **Operator together with operands constitutes** an **expression *or*** a **valid combination of constants, variables, and operators forms** an **expression** is usually recognized by the type of operator used with in the expression. Depending upon there may have integer expression, floating point expression, relational expression etc.

Table 3.2 : Types of Expressions

S.No	*Expression*	*Type of Expression*
1.	2+3*4/6–7	Integer Arithmetic
2.	2.3*4.7/6.0	Real Arithmetic
3.	a>b!=c	Relational
4.	X && 10 || y	Logical
5.	2>3+x && y	Mixed

- Operators in C++ can be classified into following types
- Arithmetic Operators
- Relational Operator
- Logical Operators
- Assignment Operator
- Increment Operator (++) and Decrement Operator (− −)
- Conditional Operator
- Bitwise Operators
- size of Operator

- There are **two types of mathematical operators: binary** and **unary**
- **Binary Operators**
- **All the operators which require two operands to operate on are known as binary operators.**
- For example, all the arithmetic, relational, logical (except! (NOT) operator), assignment, bitwise (except ~ operator) operators etc., are binary operators.
- Example : $-2 + 4$, $35 > 45$, $x = 30$, 2 & 5.

- **Unary Operators**
- **Unary operators are those operators which require only one operand to operate on.**
- For example, as shown in the above table increment/decrement,! (NOT), ~ (1's complement operator), sizeof etc., are unary operators.
- C++ also provides unary plus and unary minus i.e., +20 and −35. Here + and – are known as **unary plus and unary minus** operators.
- Example : +4, −3, ++x, sizeof (int), sizeof(2.0).

Arithmetic Operators

Table 3.3 : Arithmetic Operators

S.No.	*Operator*	*Meaning/ used for*
1.	+	Addition
2.	–	Subtraction
3.	/	Division
4.	*	Multiplication
5.	%	Remainder

- The **reminder operator** can be **used only with integers.**
- For all **other operators** if **one** of the **operands is float** and **second** one **is int** then **result** will be **in float.**
- **The percentage symbol "%" is used for modulus of a number.**
- % operator **works only** with **integer operands.** Never use float operands.
- a/b produces quotient and a%b where % is called remainder operator produces remainder when a is divided by b.

Relational Operator

- All relational operators **yield Boolean values** i.e., **true or false.** A **true value** is represented by **1** and ***false value*** by ***0.***

Table 3.4 : Relational operators

S.No.	*Operator*	*Meaning/ Used for*
1.	>	Greater than
2.	<	Less than
3.	>=	Greater than equal to
4.	<=	Less than equal to
5.	= =	Equal to
6.	!=	Not equal to

- E.g.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

- **Logical Operators**
- **Logical operators are used to check logical relation between two expressions.**
- **Depending upon** the **truth or falsehood** of the expression they are assigned **value1 (true) and 0(false).**
- The expressions may be variables, constants, functions etc.
- See the table given below :

Table 3.5 : Logical Operators

S.No.	*Symbol*	*Meaning*
1.	&&	AND
2.	||	OR
3.	!	NOT

Operators	Example/Description
&& (logical AND)	(x>5)&&(y<5) It returns true when both conditions are true
|| (logical OR)	(x>=10)||(y>=10) It returns true when at-least one of the condition is true
! (logical NOT)	!((x>5)&&(y<5)) It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

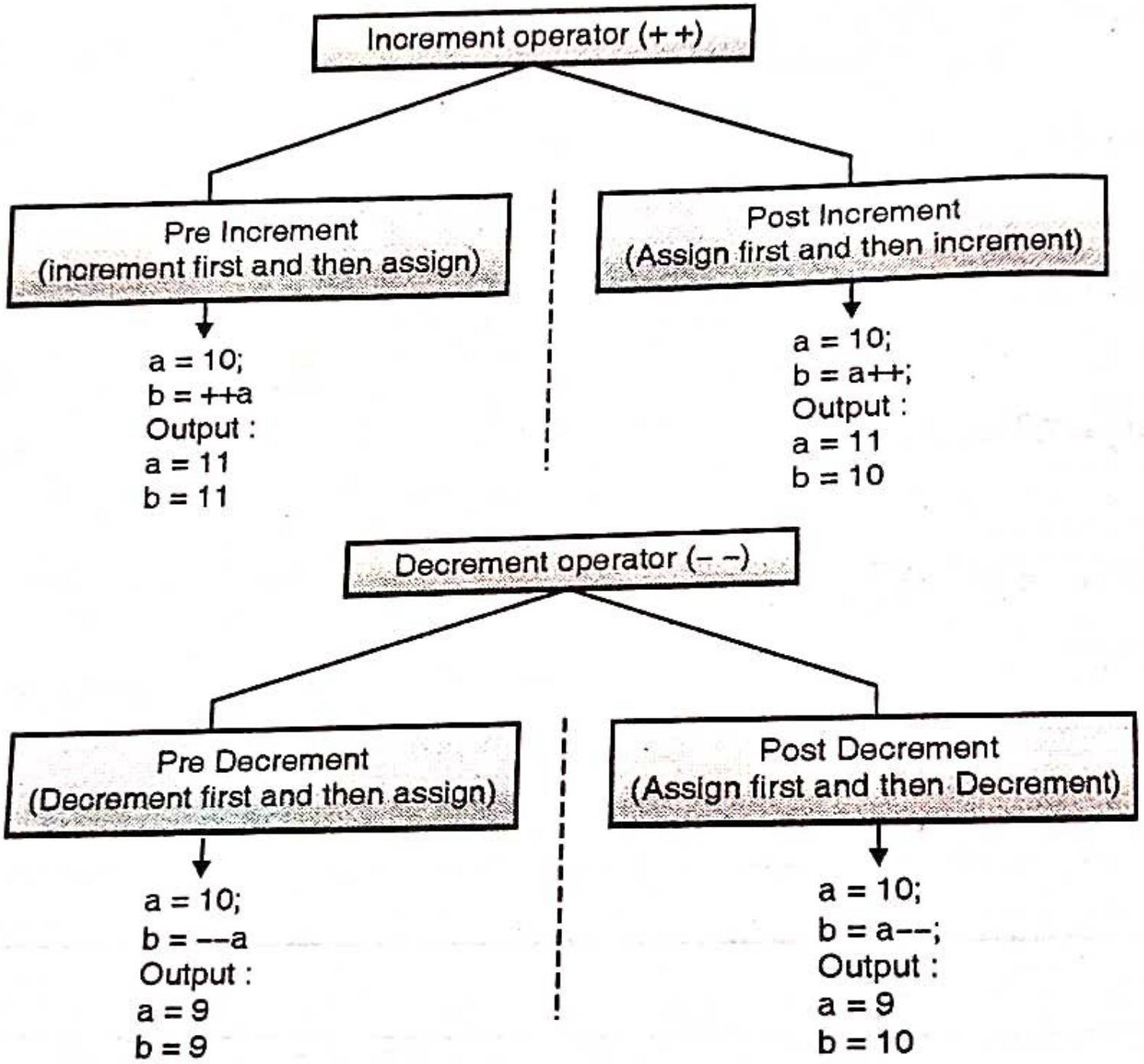
- The **&& and ||** are **binary operators**
- For **&&** to **return true** value ***both*** of its ***operand*** must yield **true value.**

- For **| |** to yield **true value** **at least one** of the **operand** yield **true value**
- The **NOT (!)** operator is **unary operator**
- It negates its operand i.e., if **operand is true** it **convert it into false** and **vice-versa.**

- **Assignment Operator**
- **These operators are used to assign value to a variable.**

Name	Symbol	Description	Example
Assignment Operator	=	Assigns the value on the right to the variable on the left	int a = 2; // a = 2
Add and Assignment Operator	+=	First adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left	int a = 2, b = 4; a+=b; // a = 6
Subtract and Assignment Operator	-=	First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left	int a = 2, b = 4; a-=b; // a = -2
Multiply and Assignment Operator	*=	First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left	int a = 2, b = 4; a*=b; // a = 8
Divide and Assignment Operator	/=	First divides the current value of the variable on left by the value on the right and then assign the result to the variable on the left	int a = 4, b = 2; a /=b; // a = 2

- **Increment Operator (++) and Decrement Operator (– –)**
- The operators ++ is known as increment operator and the operator – –is known as decrement operator.
- Both are unary operators.
- The ++ increment the value of its operand by 1 and – decrement the value of its operands by 1.


Increment operator (+ +)
Pre Increment
(increment first and then assign)
a = 10;
b = ++a
Output :
a = 11
b = 11
Post Increment
(Assign first and then increment)
a = 10;
b = a++;
Output :
a = 11
b = 10
Decrement operator (– –)
Pre Decrement
(Decrement first and then assign)
a = 10;
b = --a
Output :
a = 9
b = 9
Post Decrement
(Assign first and then Decrement)
a = 10;
b = a--;
Output :
a = 9
b = 10

- **Conditional Operator**

A conditional expression is written in the form

expression 1 ? expression 2 : expression 3

- The two symbols ? : together are called ternary or conditional operator.
- This operator takes **three operands** (One before ?, Second after ? and third after :); therefore it is known as **ternary operator**.
- Before? condition is specified with the help of relational operators
- The operators return the value based on the condition.

- the ternary operator determines the answer on the basis of the evaluation of expression 1.
- If ***expression 1 is true*** (i.e., if its value is nonzero), then ***expression 2 is evaluated*** and this becomes the answer for the expression.
- if **expression 1 is false** (i.e., if its value is zero), then **expression 3 is evaluated** and this becomes the answer for the expression.

- **Bitwise Operators**
- They are called so because they **operate on bits.**
- They can be used for the manipulation of bits.
- All these operators are extensively used when **interfacing with the hardware and for setting of bits in registers of the device.**
- C++ provides total **6 types of bitwise operators.**

Table 3.6 : Bitwise Operators

S.No.	*Operator*	*Meaning/Used for*
1.	&	Bitwise AND
2.	|	Bitwise OR
3.	^	Bitwise XOR
4.	~	One's complement
5.	>>	Right shift
6.	<<	Left shift

- **Bitwise And (&)**
  - It takes two bits as operand and returns the **value 1** if **both are 1**. If **either** of them **is 0**, the **result is 0**.

Table 3.7 : Truth Table of Bitwise AND

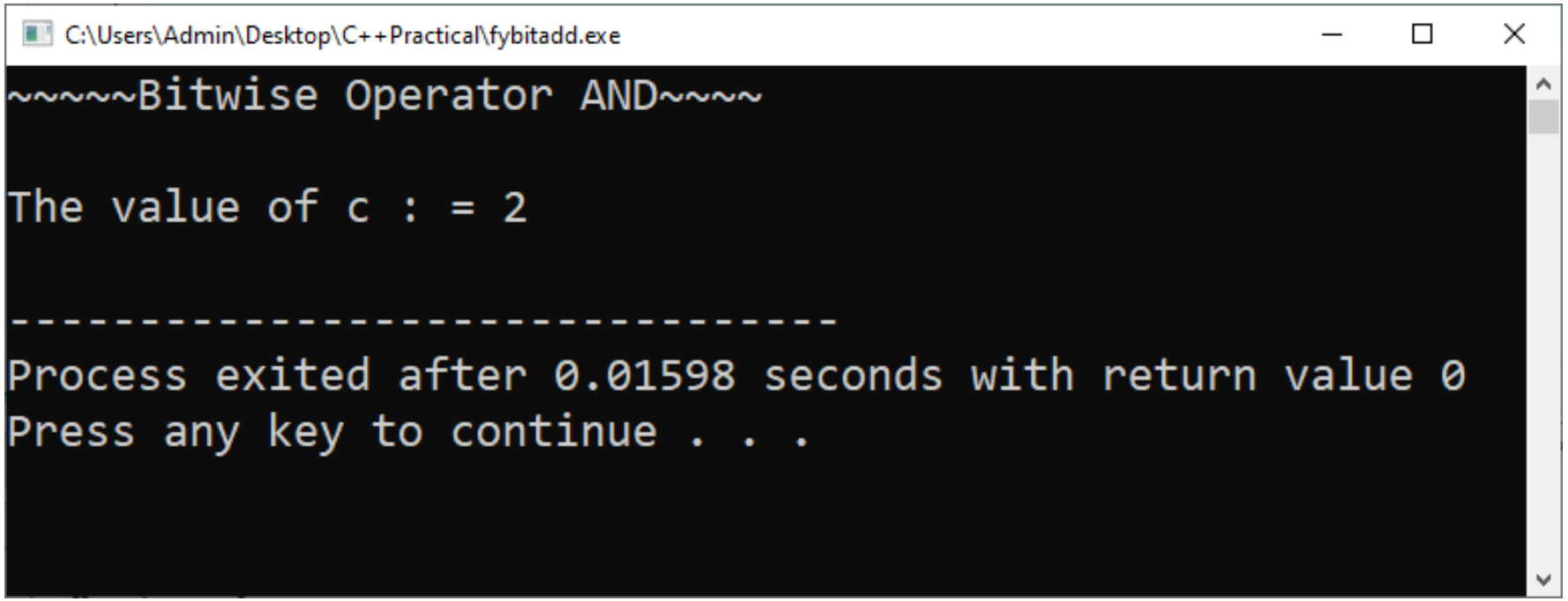
First bit	*Second bit*	*Result*
0	0	0
0	1	0
1	0	0
1	1	1

- E.g.

fybitadd.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main( )
4    {
5    int a,b;
6    a=2;
7    b=3;
8    int c=a&b;
9    cout<<"~~~~~Bitwise Operator AND~~~~"<<endl<<endl;
10   cout<<"The value of c : = "<<c<<endl;
11   return 0;
12   }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\fybitadd.exe
~~~~~Bitwise Operator AND~~~~
The value of c : = 2

Process exited after 0.01598 seconds with return value 0
Press any key to continue . . .
~~~~~

- Explanation

EXPLANATION : Binary values of a=2 is 0010 and b=3 is 0011

Bitwise AND of those values is performed as follows :

```
  0 0 1 0
  0 0 1 1
---------
  0 0 1 0    (output in c will be 2 in decimal)
---------
```

If both bit are **1** output bit will be 1 using **&** operator else **0.**

- **Bitwise OR (|)**
  - It takes two bits as operand and returns the **value 1** if **at least one is 1**. If both are **0 only then result is 0 else it is 1**.

Table 3.8 : Truth Table of Bitwise OR

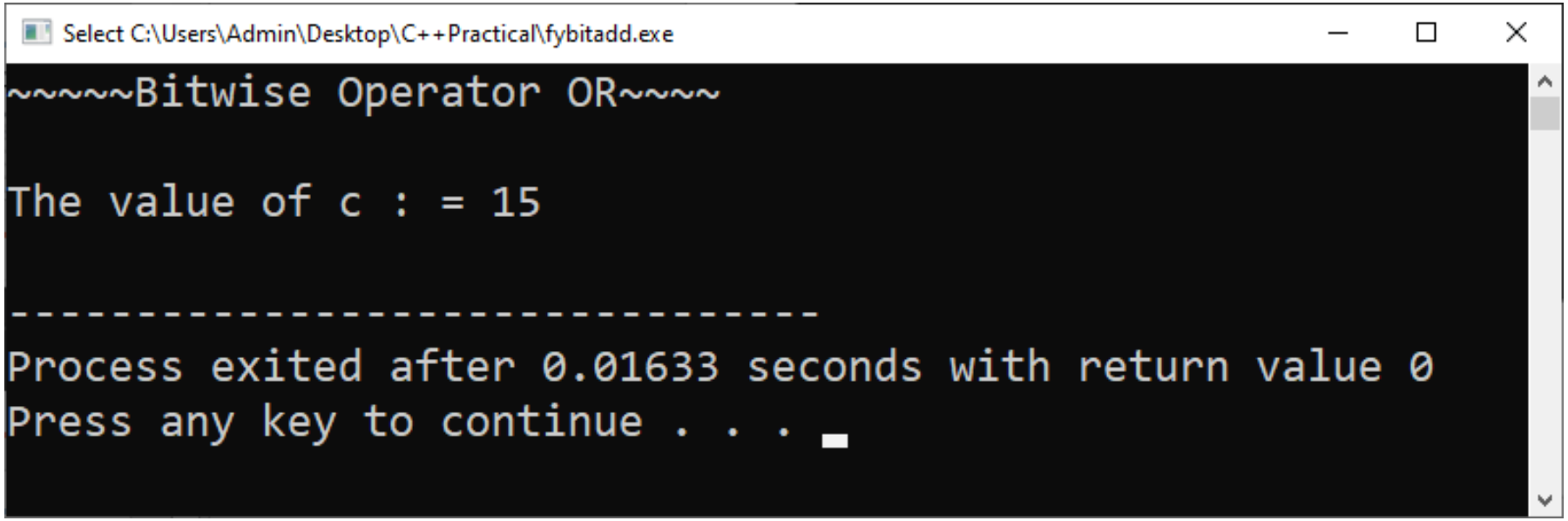
First bit	*Second bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	1

- E.g.

fybitadd.cpp

```
1  #include<iostream>
2  using namespace std;
3  int main( )
4   {
5   int a,b;
6   a=12;
7   b=7;
8   int c=a|b;
9   cout<<"~~~~~Bitwise Operator OR~~~~"<<endl<<endl;
10  cout<<"The value of c : = "<<c<<endl;
11  return 0;
12  }
```

- Output:


Select C:\Users\Admin\Desktop\C++Practical\fybitadd.exe
~~~~~Bitwise Operator OR~~~~
The value of c : = 15

Process exited after 0.01633 seconds with return value 0
Press any key to continue . . .
~~~~~

- Explanation :

EXPLANATION : Binary value of **a=12 is 1100 and b =7 is 0111**
OR of these two values is performed as follows :

```
  1 1 0 0
  0 1 1 1
 ---------
  1 1 1 1    (output in c will be 15 in decimal)?
 ---------
```

If any of the bit **1** output will be **1** using **OR** operator.

- **Bitwise XOR (^)**
- This operator takes at least two bits (may be more than two).
- If number of **1's are odd then result is 1 else result is 0.**

Table 3.9 : Truth Table of Bitwise XOR

First bit	*Second bit*	*Result*
0	0	0
0	1	1
1	0	1
1	1	0

- Explanation

EXPLANATION : Binary values of **a =5 is 0101 and b = 6 is 0110**

XOR of these two values is performed as follows :

```
  0 1 0 1
  0 1 1 0
---------
  0 0 1 1    (output in c will be 3 in decimal)
---------
```

If odd number of 1's are there output will be one otherwise output will be 0 using XOR operator.

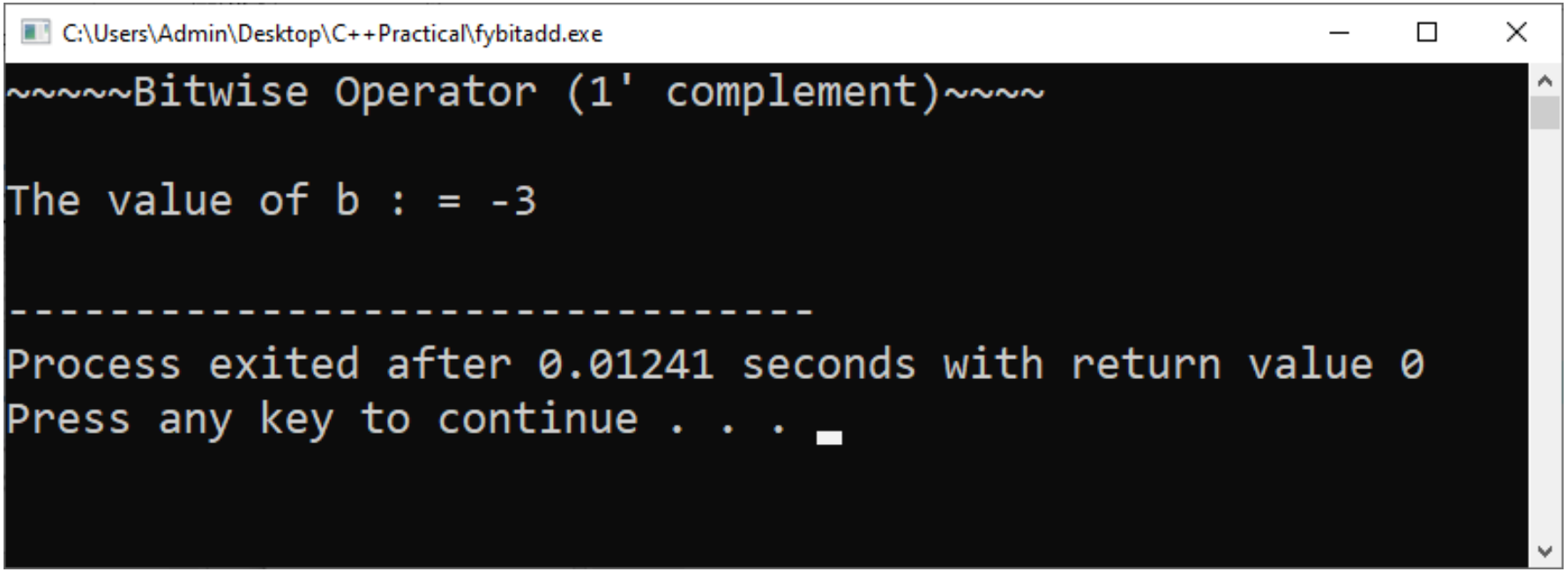
- **1'S Complement (~)**
- **The symbol (~) denotes one's complement.**
- It is a unary operator and complements the bits in its operand i.e., **1 is converted to 0** and **0 is converted to 1**.

- E.g.

fybitadd.cpp

```
1  #include<iostream>
2  using namespace std;
3  int main( )
4   {
5   int a,b;
6   a=2;
7   b=~a;
8   cout<<"~~~~~Bitwise Operator (1' complement)~~~~"<<endl<<endl;
9   cout<<"The value of b : = "<<b<<endl;
10  return 0;
11  }
```

- Output :


C:\Users\Admin\Desktop\C++Practical\fybitadd.exe
~~~~~Bitwise Operator (1' complement)~~~~
The value of b : = -3

Process exited after 0.01241 seconds with return value 0
Press any key to continue . . .
~~~~~

- Explanation

Bitwise complement Operation of 2 (~ 0010): *1101*

Calculate 2's complement of 3:

Binary form of 3 = *0011*

1's Complement of 3 = *1100*

Adding 1 to 1's complement = *1100 +1*

2's complement of 3 = *1101*

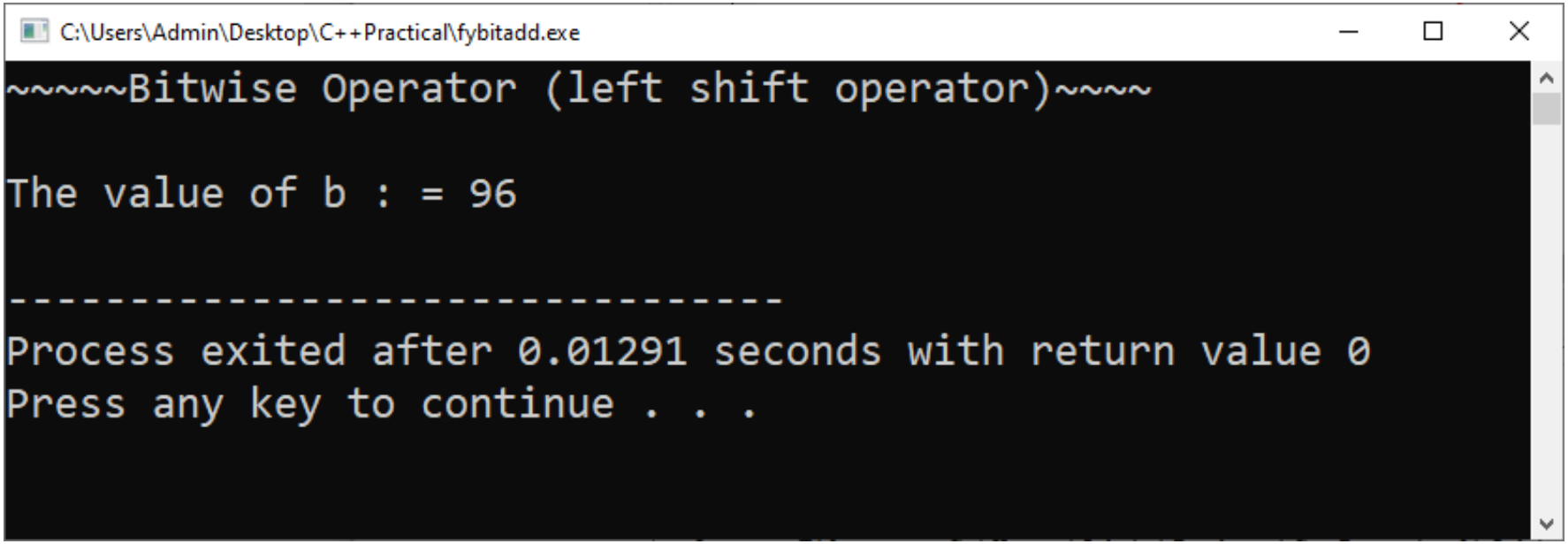
- **Left Shift Operator (<<)**
- The operator is used to shift the bits of its operand.
- It is written as x<<num;
- which means **shifting the bits of** x **towards left** by num number of times.
- In general, if we shift a number by n position to left, the output will be **number * (2^n).**

- E.g.

fybitadd.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main( )
4    {
5    int a,b;
6    a=12;
7    b=a<<3;
8    cout<<"~~~~~Bitwise Operator (left shift operator)~~~~"<<endl<<endl;
9    cout<<"The value of b : = "<<b<<endl;
10   return 0;
11   }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\fybitadd.exe
~~~~~Bitwise Operator (left shift operator)~~~~
The value of b : = 96

Process exited after 0.01291 seconds with return value 0
Press any key to continue . . .
~~~~~

- Let's assume the number as 12

If we shift the number 3 position to the left.

Then the output will be,

$12 << 3$

$= 12 * (2^3)$

$= 12 * 8$

$= 96.$

- **Right Shift Operator (>>)**
- The operator is used to shift the bits of its operand.
- It is written as x>> num;
- which means shifting the **bits of** x **towards right** by num number of times.
- In general, if we shift a number by n times to right, the output will be **number / (2^n)** .

- **Example**

Let's assume number as 128.

If we shift the number 5 position to the right, the output will be

128 >> 5

= 128 / (2^5)

=128/32

=4.

- **sizeof Operator**
- This unary operator is used to **find size in bytes** a particular variable of a particular type or a constant takes up in memory.
- E.g.

```
int b;
sizeof(b); //returns 4
```

DECLARING CONSTANTS

- As the name suggests the name constants are given to such variables or values in C++ programming language **which cannot be modified once they are defined.**
- A **constant** is a value which **cannot be changed**.
- They are **fixed values** in the program.
- For example, value of π (pi) is 22/7 or 3.14 which is a constant.
- To use constants in our programs C++ defines mainly **two ways of declaring constants.**

1. Use of #define Directive

- **#define preprocessor** directive can be **used** to declare a constant.
- It's general syntax is given as :

#define constant_name value

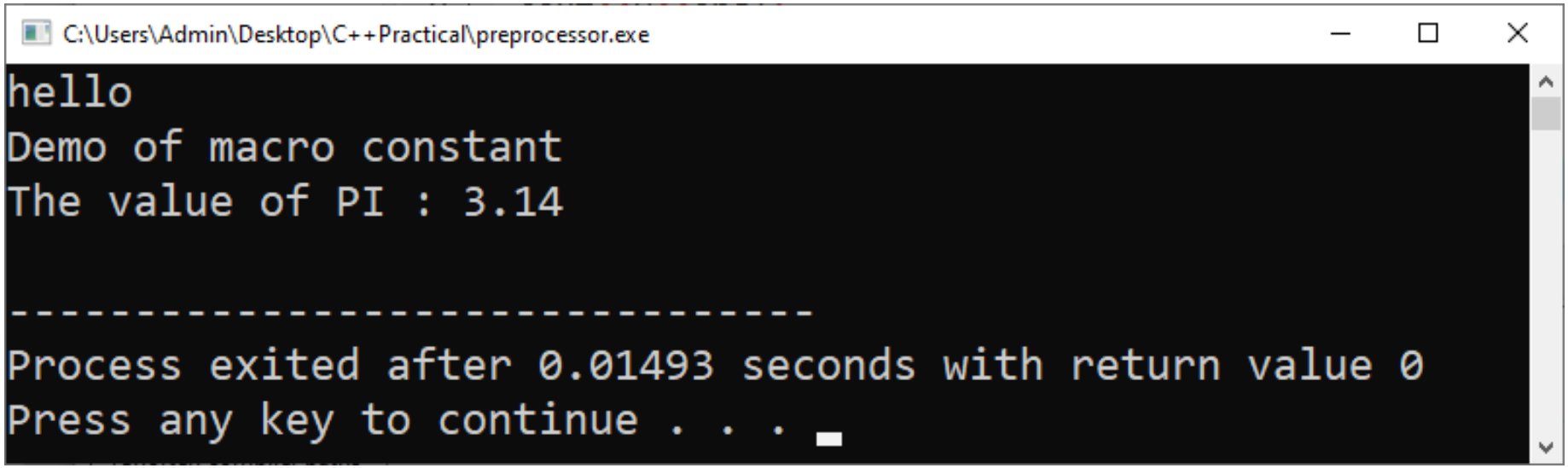
- For example : #define PI 3.14 which declares a constant (actually it is a macro constant) named PI with value 3.14.
- There should be **no assignment operator (=)** between PI and 3.14.
- Each macro constant must be **declared on separate line.**

- E.g.

preprocessor.cpp

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

#define PI 3.14
#define H "hello"
#define DMC "Demo of macro constant"
 int main()
 {
 cout<<H<<endl;
 cout<<DMC<<endl;
 cout<<"The value of PI : "<<PI<<endl;
 return 0;
 }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\preprocessor.exe
hello
Demo of macro constant
The value of PI : 3.14

Process exited after 0.01493 seconds with return value 0
Press any key to continue . . .

2. Use of Const Keyword to Define Constant

- The second way of **declaring constant** is to use **const keyword.**
- Its syntax is given as follows :

const data_type var_name = value;

- Example :

const int x = 100;

1 2 3 4

1. const ? KEYWORD
2. int ? data type
3. x ? variable
4. 100 ? value

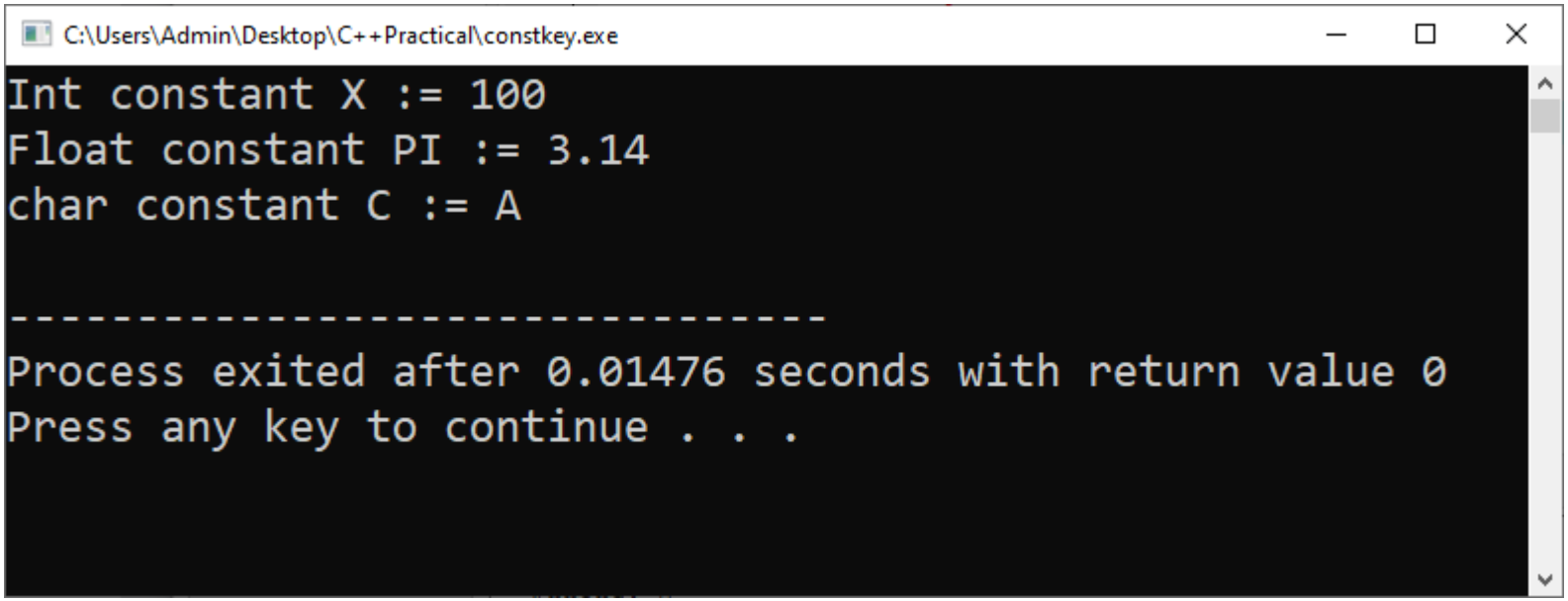
That is x as a constant and assign a value 100 to it. Later in the program we cannot change the value of x.

- E.g.

constkey.cpp

```cpp
#include<iostream>
using namespace std;
 int main( )
 {
 const int x=100;
 const float PI=3.14;
 const char c='A';
 cout<<"Int constant X := "<<x<<endl;
 cout<<"Float constant PI := "<<PI<<endl;
 cout<<"char constant C := "<<c<<endl;
 return 0;
 }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\constkey.exe
Int constant X := 100
Float constant PI := 3.14
char constant C := A

Process exited after 0.01476 seconds with return value 0
Press any key to continue . . .

TYPE CONVERSION

- **Type conversion is the process that converts from one data type to another data type**
- Type Conversion of two types:

1. Implicit type conversion.
2. Explicit type conversion.

1. Implicit type conversion

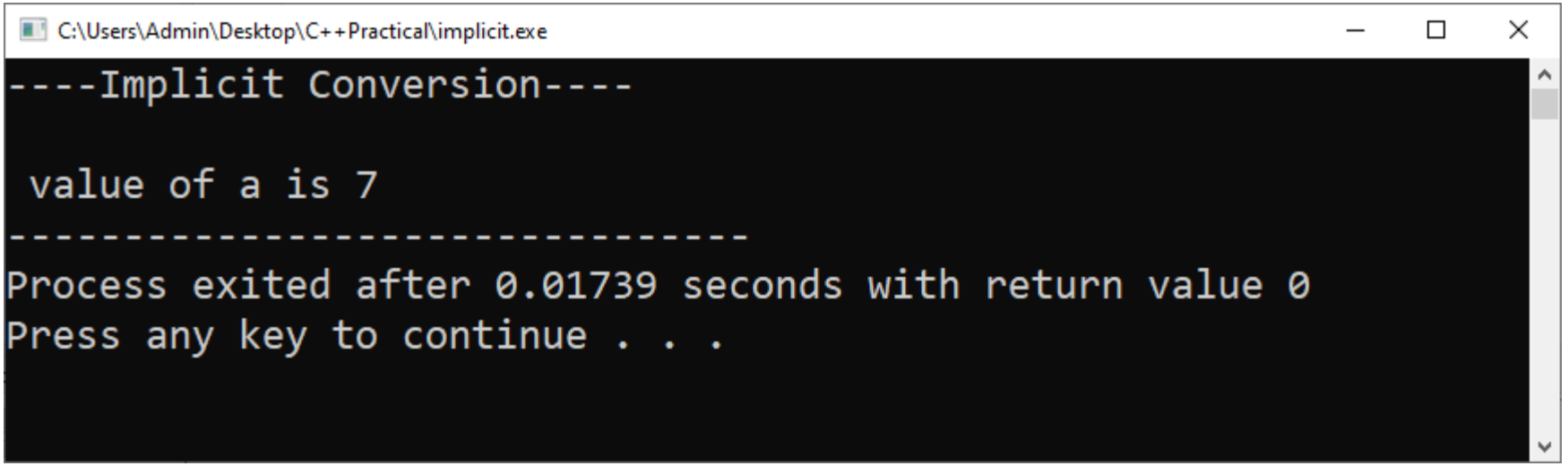
- It also known as “automatic type conversion”.
- In this type of conversion the c**ompiler internally converts** the type depending upon expression without letting user to know.

- E.g.

implicit.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int a;
6       a=7.8;
7       cout<<"----Implicit Conversion----"<<endl;
8       cout<<"\n value of a is "<<a;
9       return 0;
10  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\implicit.exe
----Implicit Conversion----
value of a is 7

Process exited after 0.01739 seconds with return value 0
Press any key to continue . . .

2. Explicit type conversion

- Conversion **that require user innervation** to change the data type of one variable to another is called the explicit type conversion.
- This process is **also called type casting** and it is user defined.
- The **explicit conversion** is divided into **two ways**

1. Explicit conversion using the cast operator

Syntax:

static_cast<data_type> (expression)

Where data_type indicated the data type which the final result is converted.

2. Explicit conversion using the assignment operator

Syntax:

(type) expression

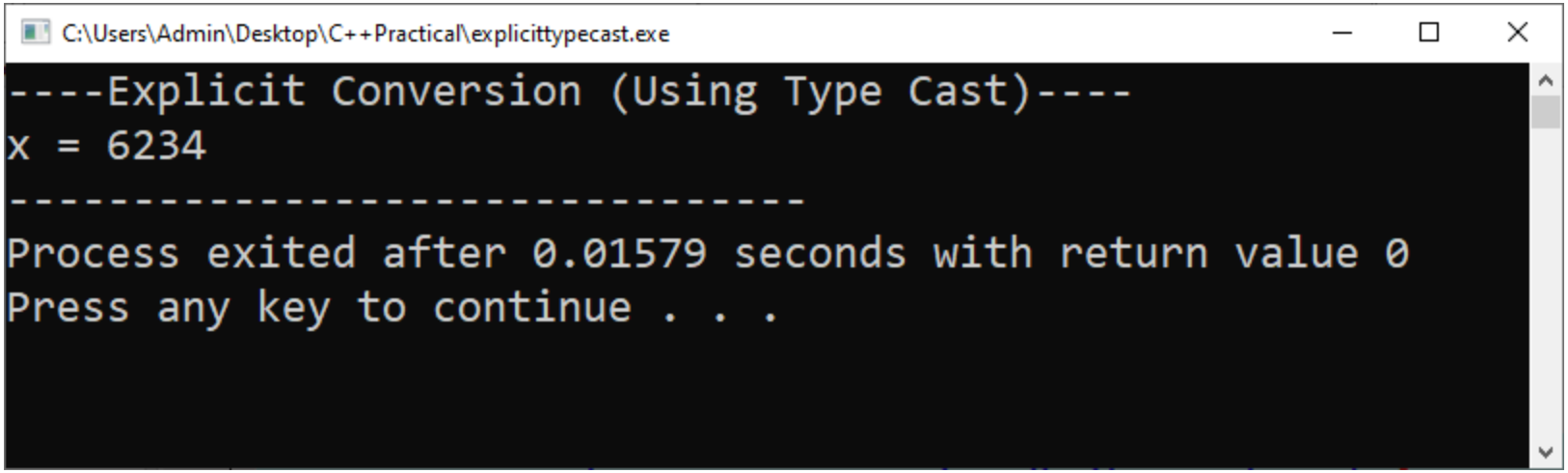
Where type indicates the data type to which the final result is converted.

- E.g. Explicit conversion using the cast operator

explicittypecast.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5
6       double f=6234.7;
7       int x;
8       x=static_cast<int> (f);
9       cout<<"----Explicit Conversion (Using Type Cast)----\n";
10      cout<<"x = "<<x;
11      return 0;
12  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\explicittypecast.exe
----Explicit Conversion (Using Type Cast)----
x = 6234

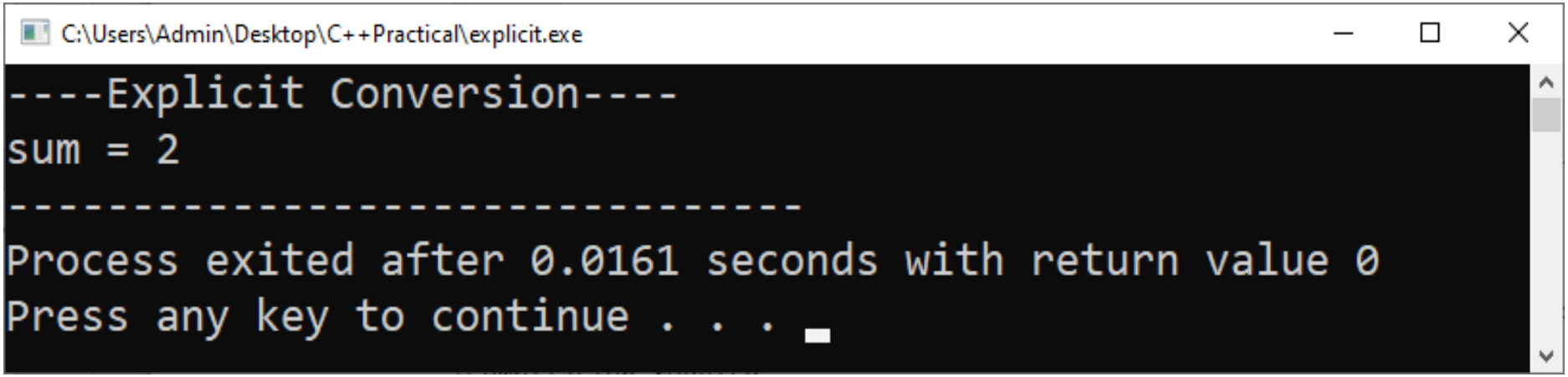
Process exited after 0.01579 seconds with return value 0
Press any key to continue . . .

- E.g. Explicit conversion using the assignment operator

explicit.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       double sum,x = 1.2;
6       sum = (int)x + 1;  // Explicit conversion from double to int
7       cout<<"----Explicit Conversion----\n";
8       cout<<"sum = "<<sum;
9       return 0;
10  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\explicit.exe
----Explicit Conversion----
sum = 2

Process exited after 0.0161 seconds with return value 0
Press any key to continue . . .

Decision Making: An Introduction

- Decision making statements are needed to **alter the sequence of the statements** in the program depending upon certain circumstances.
- In the **absence of decision making statements** a program executes in the **serial fashion statement basis.**
- **Decision can be made on the basis of success or failure of some condition.**
- They **allow us to control the flow of our program.**

- C++ language supports the following decision making control statements :

(a) The if statement

(b) The if-else statement

(c) The if-else-if ladder statement

(d) The switch – case statement.

- All these decision making statements **checks the given condition** and **then executes its sub block** if the **condition happens to be true**. On **falsity of condition** the **block is skipped**.
- All control statement uses a combination of relational and logical operators to form conditions as per the requirement of the programmer.

- **The if statement**
- The if statement is used to execute/skip a block of statements on the basis of truth or falsity of a condition.
- The condition to be checked is put inside the parenthesis which is preceded by keyword if.
- Syntax:

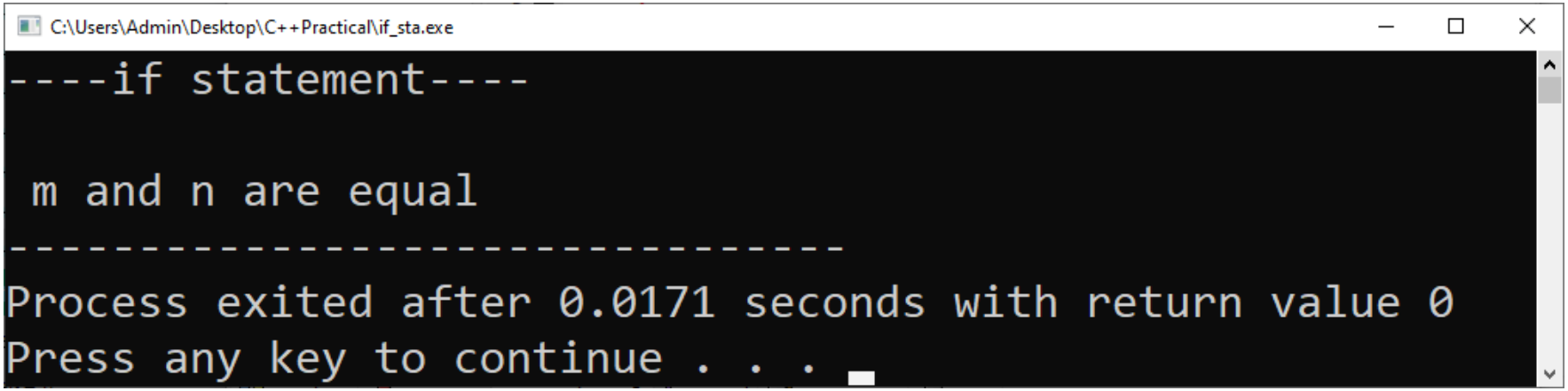
```
if (condition)
{
Statements;
 }
```

- E.g.

if_sta.cpp

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

int main()
{
    int m=20,n=20;
    cout<<"----if statement----";
    if(m==n)
    {
        cout<<"\n\n m and n are equal";
    }
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\if_sta.exe
----if statement----
m and n are equal

Process exited after 0.0171 seconds with return value 0
Press any key to continue . . .

- **The if-else Statement**
- The if-else statement is used to express decisions
- In these type of statements, group of **statements are executed** when **condition is true.** If **condition is false**, then **else part statements are executed**.
- Syntax:

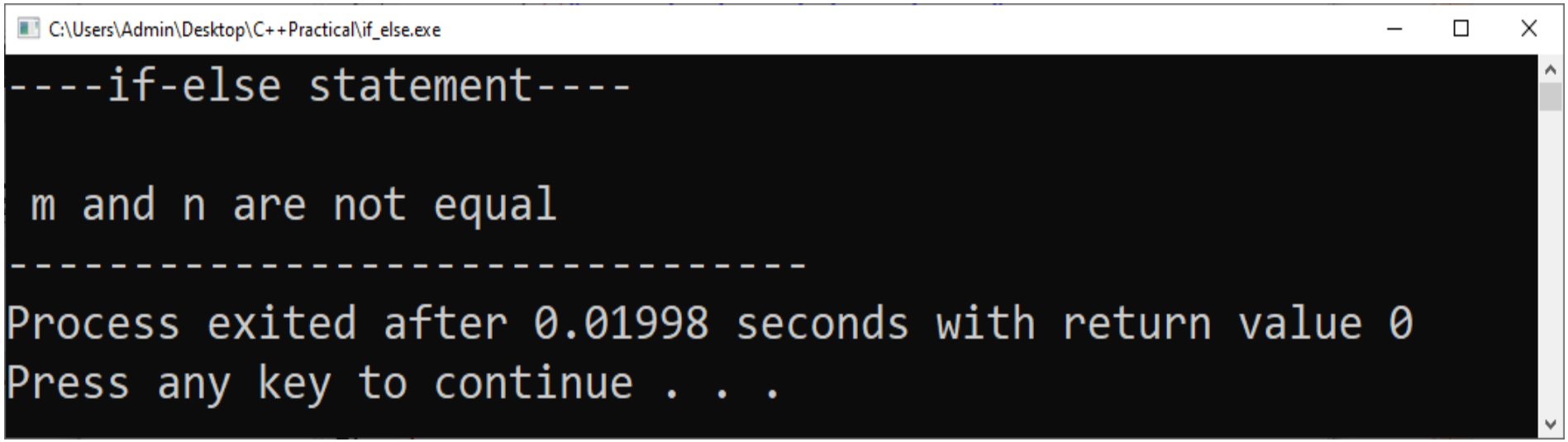
```
if (condition)
{
Statements;
}
else
{
Statements;
}
```

- E.g.

if_else.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int m=20,n=30;
6       cout<<"----if-else statement----";
7
8       if(m==n)
9       {
10          cout<<"\n\n m and n are equal";
11      }
12      else
13      {
14          cout<<"\n\n m and n are not equal";
15      }
16      return 0;
17  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\if_else.exe
----if-else statement----
m and n are not equal

Process exited after 0.01998 seconds with return value 0
Press any key to continue . . .

- **The if-else-if ladder statement(***Nesting of if–else's***)**
- Nesting of if-else means one if-else or simple if as the part of another if or else.
- There may be various syntaxes of nesting of if-else we present few of them :

Syntax : 1)

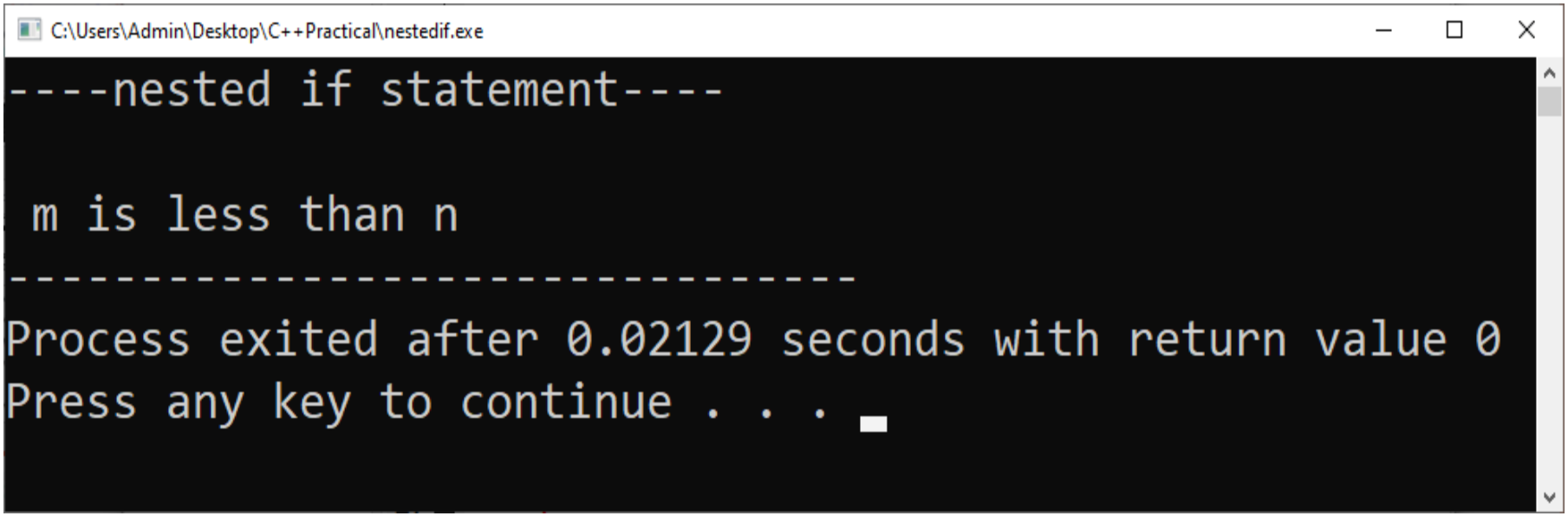
```
if(condition)
{
        Statements;
}
else if(condition)
{
        Statements;
}
else
{
        Statements;
}
```

E.g.

nestedif.cpp

```cpp
#include<iostream>
using namespace std;
int main()
{
    int m=20,n=30;
    cout<<"----nested if statement----";

    if(m>n)
    {
        cout<<"\n\n m is greater than n";
    }
    else if(m<n)
    {
        cout<<"\n\n m is less than n";
    }
    else
    {
        cout<<"\n\n m is equal to n";
    }

    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\nestedif.exe
----nested if statement----
m is less than n

Process exited after 0.02129 seconds with return value 0
Press any key to continue . . .

Syntax : 2)

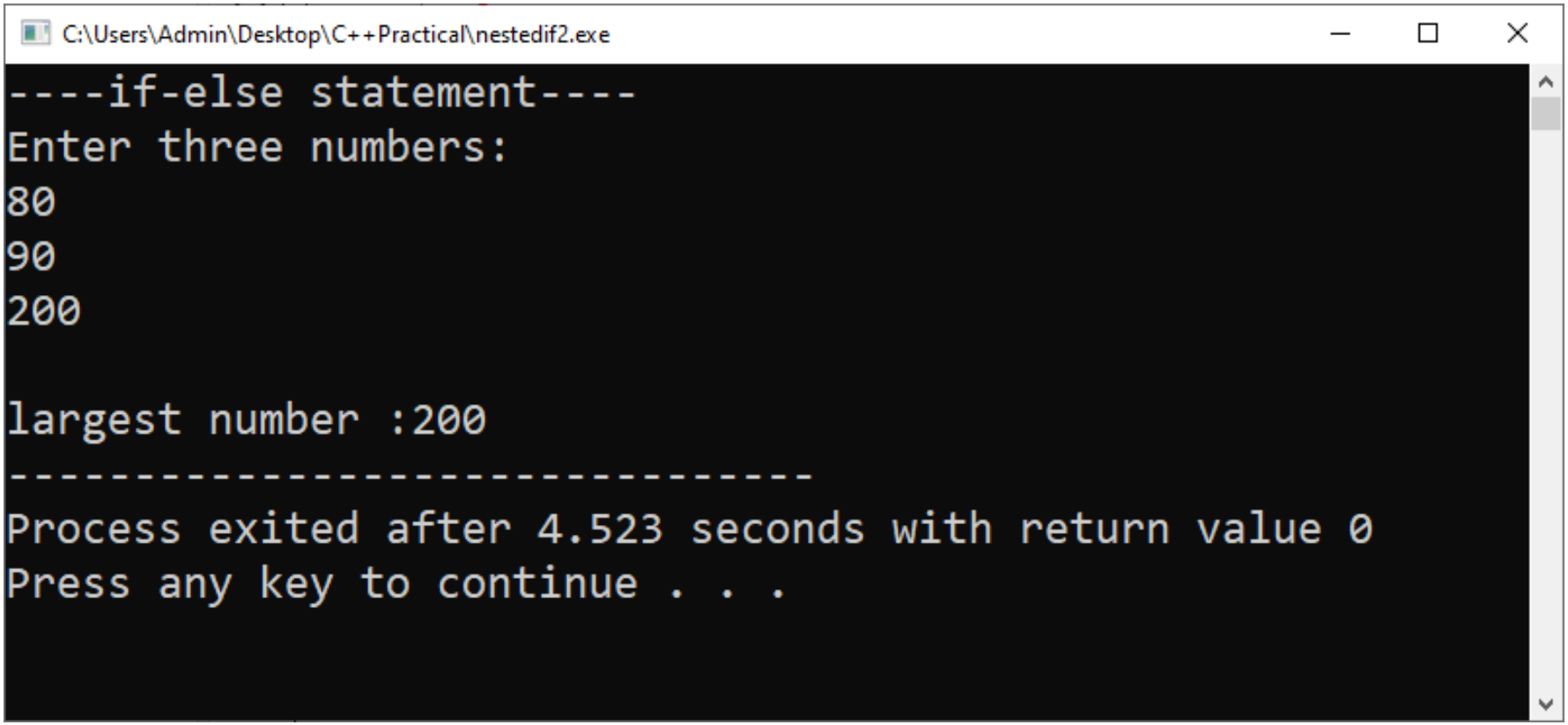
```
if(condition)
{
    if(condition)
    {
        Statements;
    }
    else
    {
        Statements;
    }
}
else
{
    if(condition)
    {
        Statements;
    }
    else
    {
        Statements;
    }
}
```

E.g.

nestedif2.cpp

```
#include<iostream>
using namespace std;
int main()
{
     cout<<"----if-else statement----";
     int m,n,o;
     cout<<"\nEnter three numbers: ";
     cin>>m>>n>>o;
     if(m>n)
     {
       if(m>o)
       {
         cout<<"\nlargest number :"<<m;

       }
       else
       {
         cout<<"\nlargest number :"<<o;
       }
     }
     else
     {
        if(n>o)
        {
         cout<<"\nlargest number :"<<n;
        }
        else
        {
         cout<<"\nlargest number :"<<o;
        }
     }
     return 0;    }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\nestedif2.exe
----if-else statement----
Enter three numbers:
80
90
200
largest number :200

Process exited after 4.523 seconds with return value 0
Press any key to continue . . .

- **switch-case Statement**

- Switch-case statement can be used **to replace else-if ladder construct.**
- The switch statement is a **multi-way decision.**
- The switch statement **causes a particular group of statements** to be **chosen from several available groups**
- The control statement that **allows us to make** a **decision from the number of choices** is called a **switch,** or more correctly a switch case-default

- The switch statement tests the value of a given variable (or expression) against a list of values and when a match is found, a block of statements associated with that is executed.
- The switch statement is an alternate to if-else-if ladder statement which allows us to execute multiple operations
- The **case value must be an integer or character constant.**
- The **case labeled default** is **executed if none of the other cases are satisfied**

- Syntax:

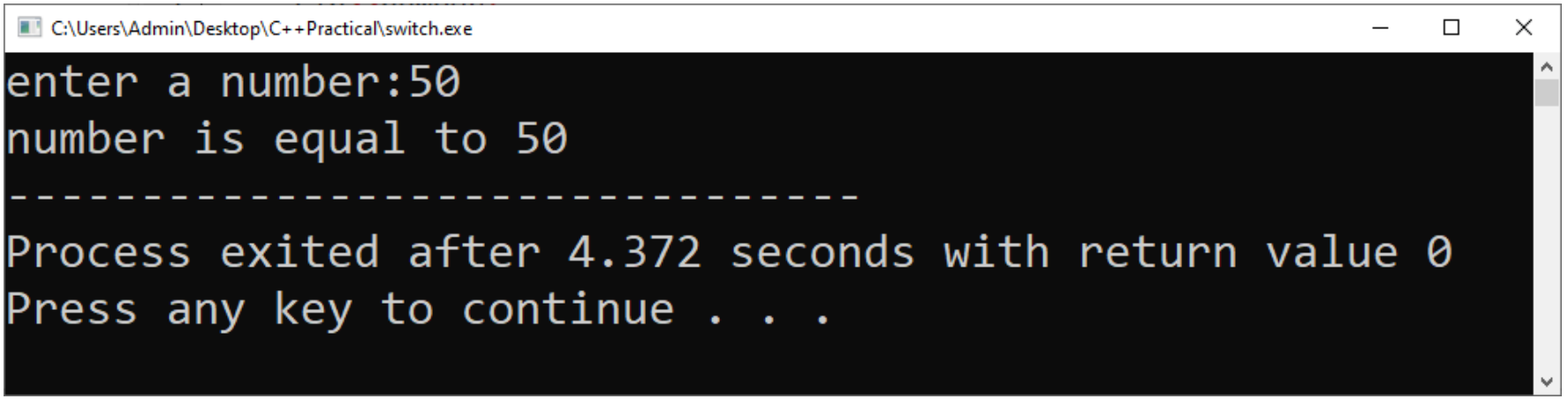
```
switch(expression)
{
        case choice1 :
                        statements;
                        break;
        case choice2 :
                        statements;
                        break;
        case choice3 :
                        statements;
                        break;
                        ……;
                        ……;
        default :
}
```

E.g.

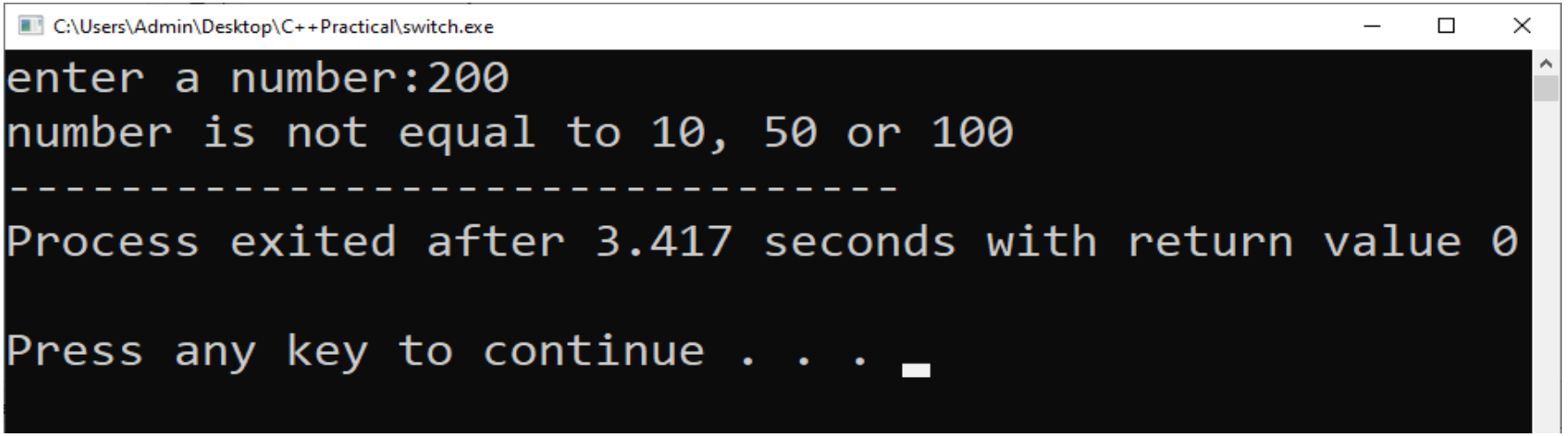
switch.cpp

```cpp
#include<iostream>
using namespace std;
int main()
{
    int number;
    cout<<"enter a number:";
    cin>>number;

    switch(number)
    {
        case 10:
        cout<<"number is equals to 10";
        break;
        case 50:
        cout<<"number is equal to 50";
        break;
        case 100:
        cout<<"number is equal to 100";
        break;

        default:
        cout<<"number is not equal to 10, 50 or 100";
    }
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\switch.exe
enter a number:50
number is equal to 50

Process exited after 4.372 seconds with return value 0
Press any key to continue . . .


C:\Users\Admin\Desktop\C++Practical\switch.exe
enter a number:200
number is not equal to 10, 50 or 100

Process exited after 3.417 seconds with return value 0
Press any key to continue . . .

UNCONDITIONAL BRANCHING USING GOTO

- The **goto statement** is used to **transfer control of program from one point to other.**
- In fact it **make program control to jump to the statement specified by a label.**
- This type of **unconditional transfer of control using goto** is called **branching.**
- **The label must be in the same function in which goto is used.**
- The target statement must be labeled, and the **label** must be **followed** by a **colon (:)**
- The general syntax for goto statement is as follows :

goto label;

- The target statement will be written as follows

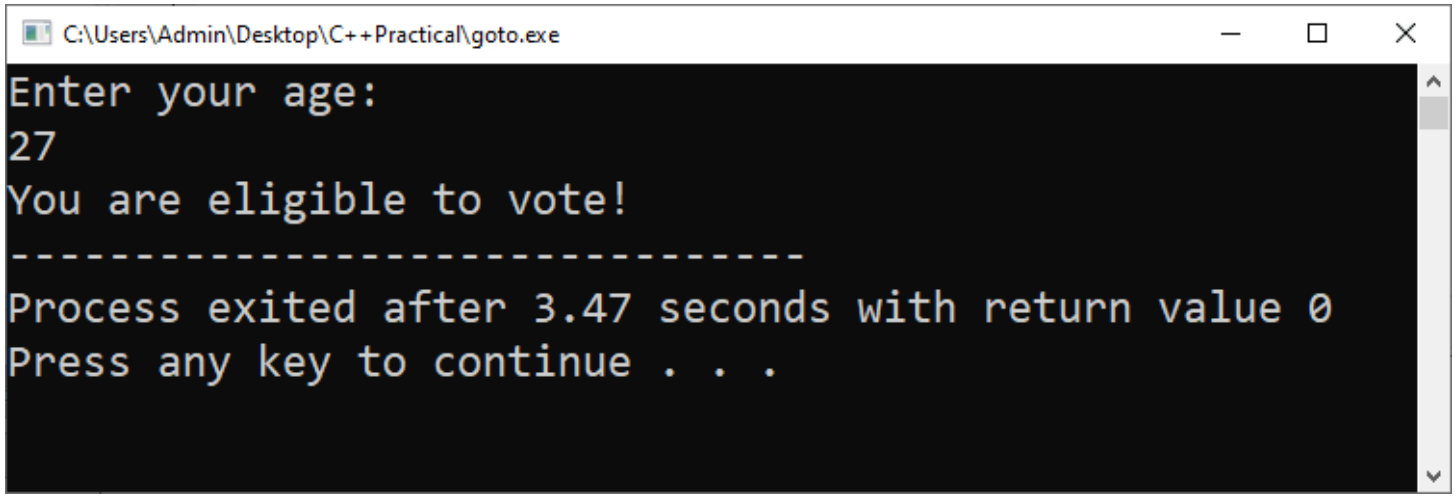
label : statement

E.g.

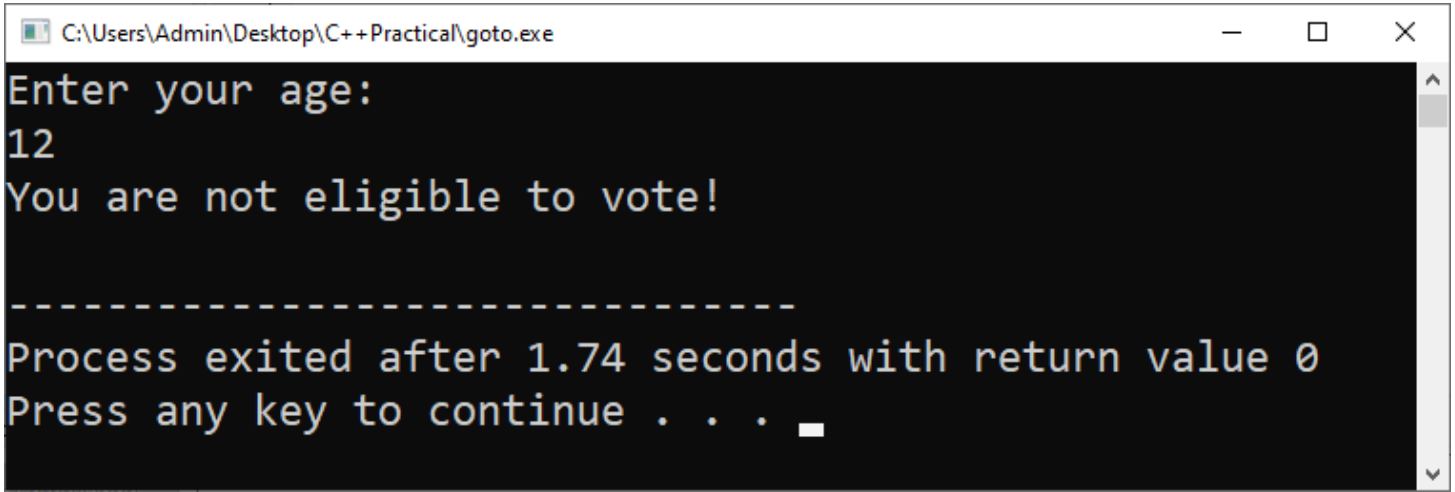
goto.cpp

```cpp
#include <iostream>
using namespace std;
int main()
{
    int age;
    cout<<"Enter your age:\n";
    cin>>age;

    if (age < 18)
    {
      goto ineligible;

      ineligible:
      cout<<"You are not eligible to vote!\n";
    }
    else
    {
      cout<<"You are eligible to vote!";
    }
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\goto.exe
Enter your age:
27
You are eligible to vote!

Process exited after 3.47 seconds with return value 0
Press any key to continue . . .


C:\Users\Admin\Desktop\C++Practical\goto.exe
Enter your age:
12
You are not eligible to vote!

Process exited after 1.74 seconds with return value 0
Press any key to continue . . .

INTRODUCTION TO LOOPING

- **Looping is a process in which set of statements are executed repeatedly for a finite or infinite number of times.**
- **C++ provides** three ways to performs looping by providing **three different types of loop.**
- **Looping** can be **called** synonymously **iteration of repetition.**
- A loop is a block of statements with which are **executed again and again till** a **specific condition is satisfied.**

- C++ provides three loops to perform repetitive action.

1. while
2. for
3. do-while

- To work with any types of loop three things have to be performed :

- Loop control variable and its initialization.
- Condition for controlling the loop.
- Increment / decrement of control variable.

- **for Loop**
- The for loop is most frequently used by programmers just because of its simplicity.
- In for loop control statement, loop is executed until a particular condition is satisfied.
- The **for loop is another entry-controlled loop** (*entry controlled loop is where the test condition is tested before executing the body of a loop*)
- initialization, condition and increment/decrement is part of for loop.

- **initialization** : This step allows to declare and initialize any loop control variables
- **condition** : If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute
- **increment/decrement** : This statement allows to update any loop control variables

- Syntax:

```
for (initialization; condition; increment/decrement)
{
        Statements;
        Statements;
        ..............;
 }
```

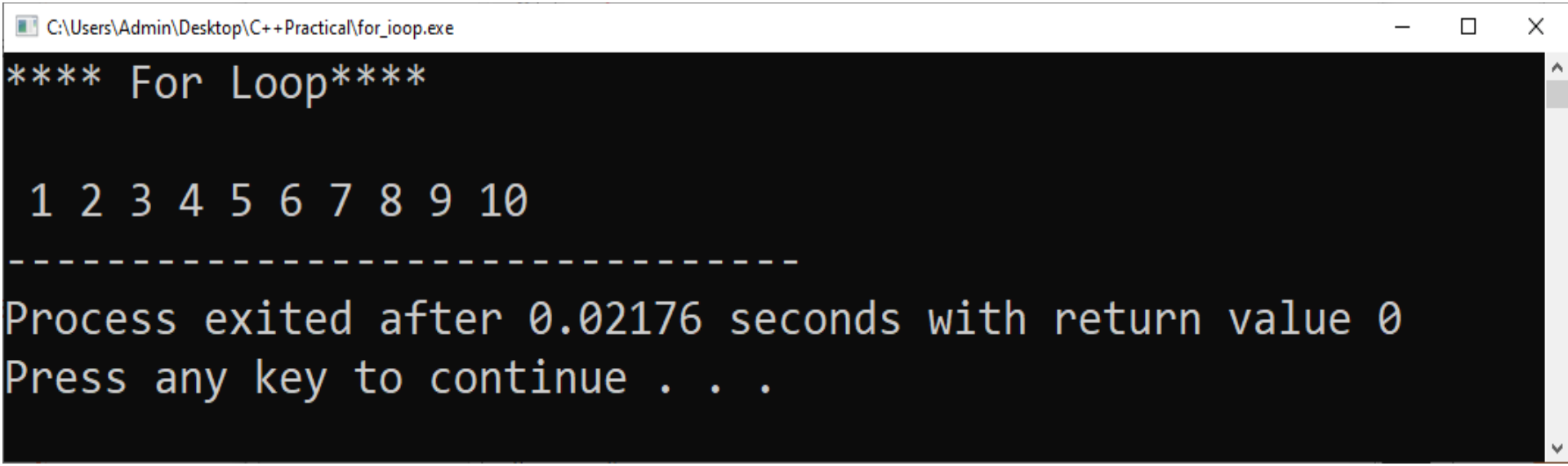
- Where,
  exp1 – initialization
  (e.g. i=0, j=2)
  exp2 – condition
  (e.g. i>5, j<3)
  exp3 – increment/decrement
  (e.g. ++i, j--)

- E.g.

for_ioop.cpp

```cpp
#include<iostream>
using namespace std;
int main()
{
   int i;
   cout<<"**** For Loop****\n\n";
   for (i = 1; i<=10;i++)
   {
      cout<<" "<<i;
   }
   return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\for_ioop.exe
**** For Loop****
1 2 3 4 5 6 7 8 9 10

Process exited after 0.02176 seconds with return value 0
Press any key to continue . . .

- **while Loop**
  - The while statement is used to carry out looping operations, in which a group of statements is executed repeatedly, until some condition has been satisfied.
  - The while **tests the condition <u>before</u> executing any of the statements within the while loop**(*First checked the condition and then statement is executed*)
  - It is known as **entry-controlled loop**

- Syntax:

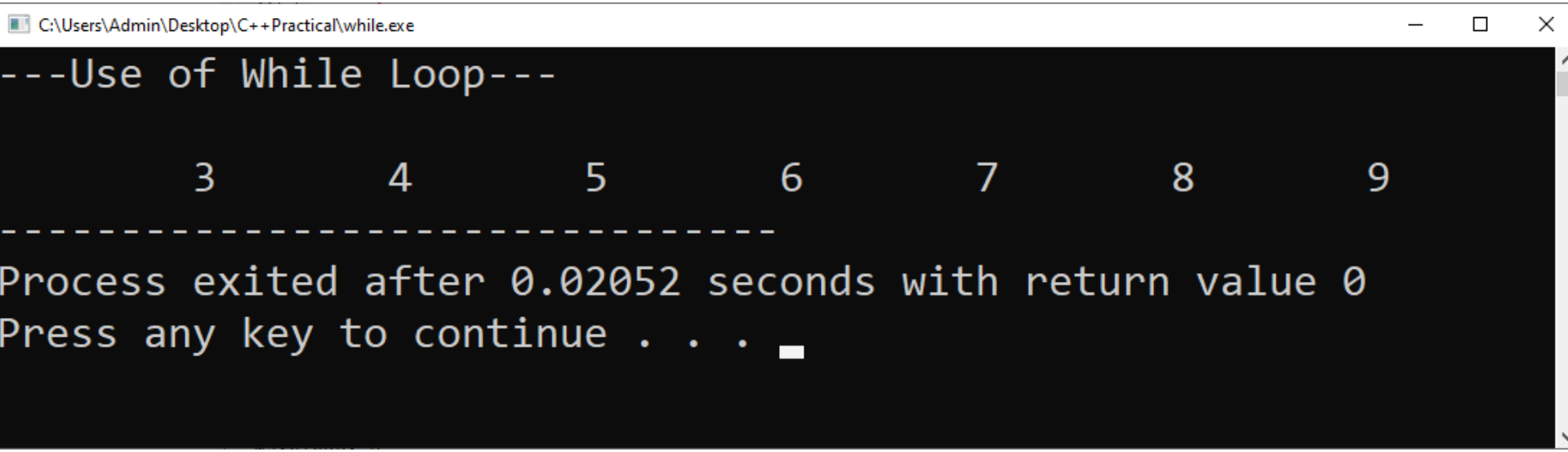
```
while (condition)
{
statements;
 }
```

- E.g.

while.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int i = 3;
6       cout<<"---Use of While Loop---\n\n";
7       while(i<10)
8       {
9           cout<<"\t"<<i;
10          i++;
11      }
12      return 0;
13  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\while.exe
---Use of While Loop---
3 4 5 6 7 8 9

Process exited after 0.02052 seconds with return value 0
Press any key to continue . . .

- **do…while Loop**
- **do…while loop is similar to while loop** in that the loop continues as long as the specified loop condition remains true.
- The **main difference** is that **do..while** loop **checks for the condition after executing the statements** (*i.e. it tests the condition after executed the statement*)
- In do-while, the block of **statements is executed at least once, even** if the **condition fails for the first time.**
- It is also called as **exit-controlled loop** *(exit controlled loop is where the condition is checked after the loop's body is executed)*

- Syntax:

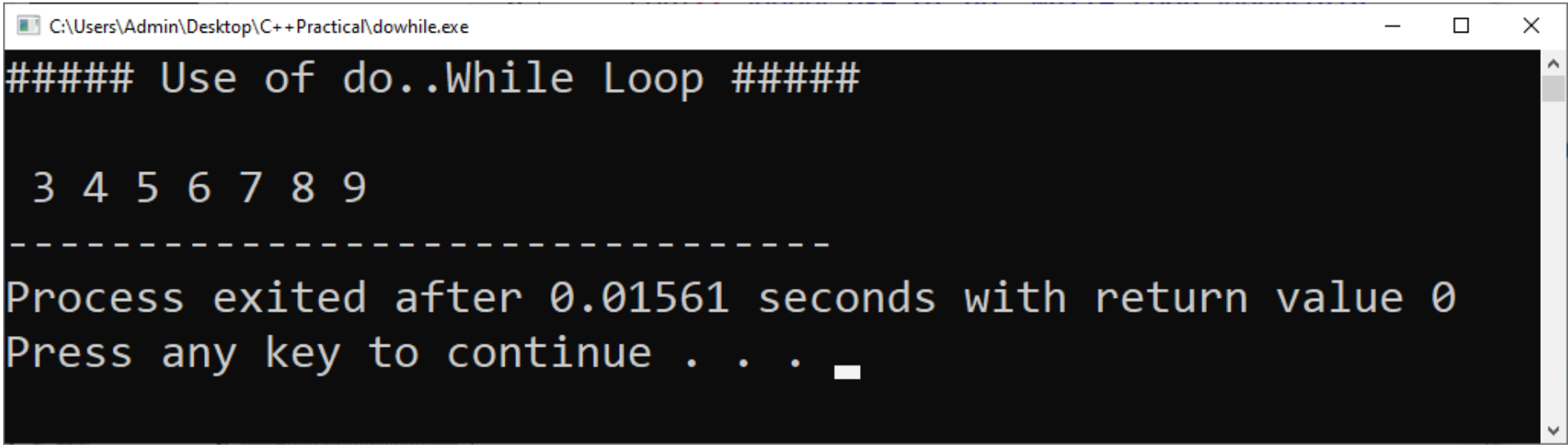
```
do
{
statements;
 }while (condition);
```

- E.g.

dowhile.cpp

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int i =3;
6       cout<<"##### Use of do..While Loop #####\n\n";
7       do
8       {
9           cout<<" "<<i;
10          i++;
11      }while(i<10);
12      return 0;
13  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\dowhile.exe
Use of do..While Loop
3 4 5 6 7 8 9

Process exited after 0.01561 seconds with return value 0
Press any key to continue . . .

- **Nested loop**
- Nesting of loops is the feature in C++ that allows the looping of statements inside another loop
- Syntax:

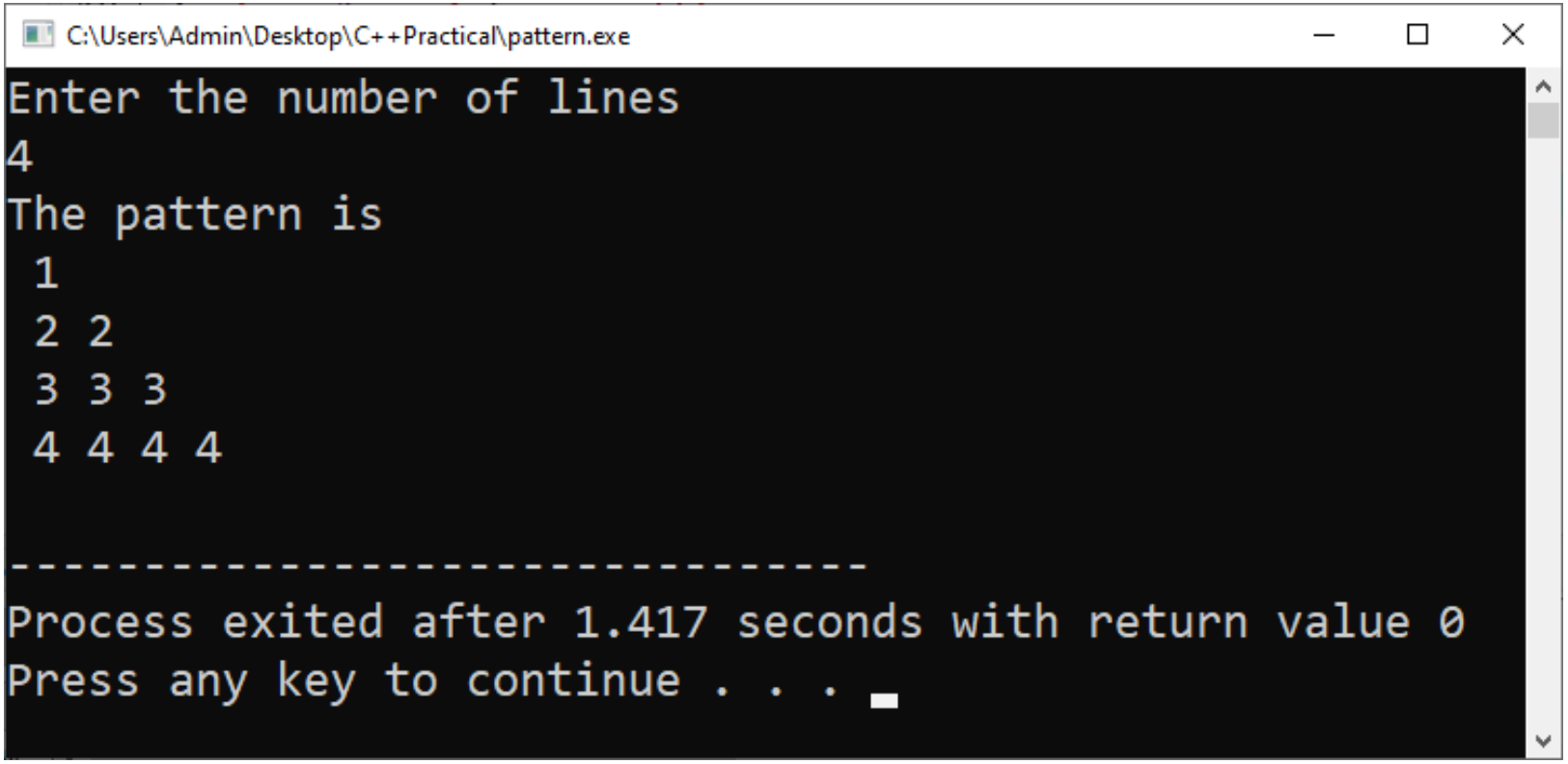
```
for(initialization; condition; increment/decrement)
{
        for(initialization; condition; increment/decrement)
        {
                Statements;
        }
        Statements;
}
```

- E.g.

pattern.cpp

```
 1  #include<iostream>
 2  using namespace std;
 3  int main( )
 4  {
 5  int line,row,col;
 6  cout<<"Enter the number of lines"<<endl;
 7  cin>>line;
 8  cout<<"The pattern is"<<endl;
 9
10  for(row=1;row<=line;row++)
11  {
12  for(col=1;col<=row;col++)
13  cout<<" "<<row;
14  cout<<"\n";
15  }
16  return 0;
17  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\pattern.exe
Enter the number of lines
4
The pattern is
1
2 2
3 3 3
4 4 4 4

Process exited after 1.417 seconds with return value 0
Press any key to continue . . .

- **Break statement**
  - The break statement is **used to terminate loops or to exit from a switch**
  - It can be used within a for, while, do -while, or switch statement
  - If a break statement is included in a while, do - while or for loop, then **control will immediately be transferred out of the loop when the break statement is encountered.** This **provides a convenient way to terminate the loop.**
  - The syntax of continue statement is simply

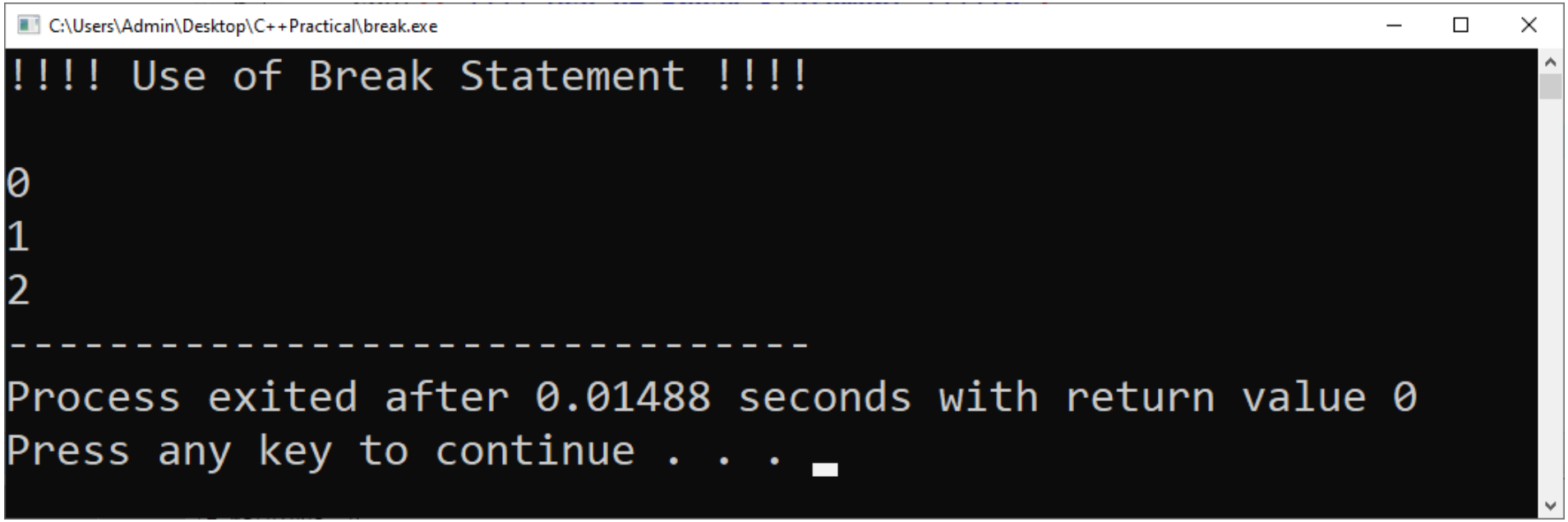
break;

- E.g.

break.cpp

```
 1  #include<iostream>
 2  using namespace std;
 3  int main()
 4  {
 5      int i=0;
 6      cout<<"!!!! Use of Break Statement !!!!\n";
 7
 8      for (i=0; i<5; i++)
 9      {
10          cout<<"\n"<<i;
11            if (i==2)
12
13                break;
14      }
15      return 0;
16  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\break.exe
!!!! Use of Break Statement !!!!
0
1
2

Process exited after 0.01488 seconds with return value 0
Press any key to continue . . .

- **Continue Statement**
- The continue statement **passes control to the next iteration**(i.e. ***it is useful to continue with the next iteration***(repetition) of a loop)
- The **loop does not terminate** when a continue statement is encountered
- The continue statement **skips** some **lines of code inside the loop** and **continues with the next iteration**
- The syntax of continue statement is simply

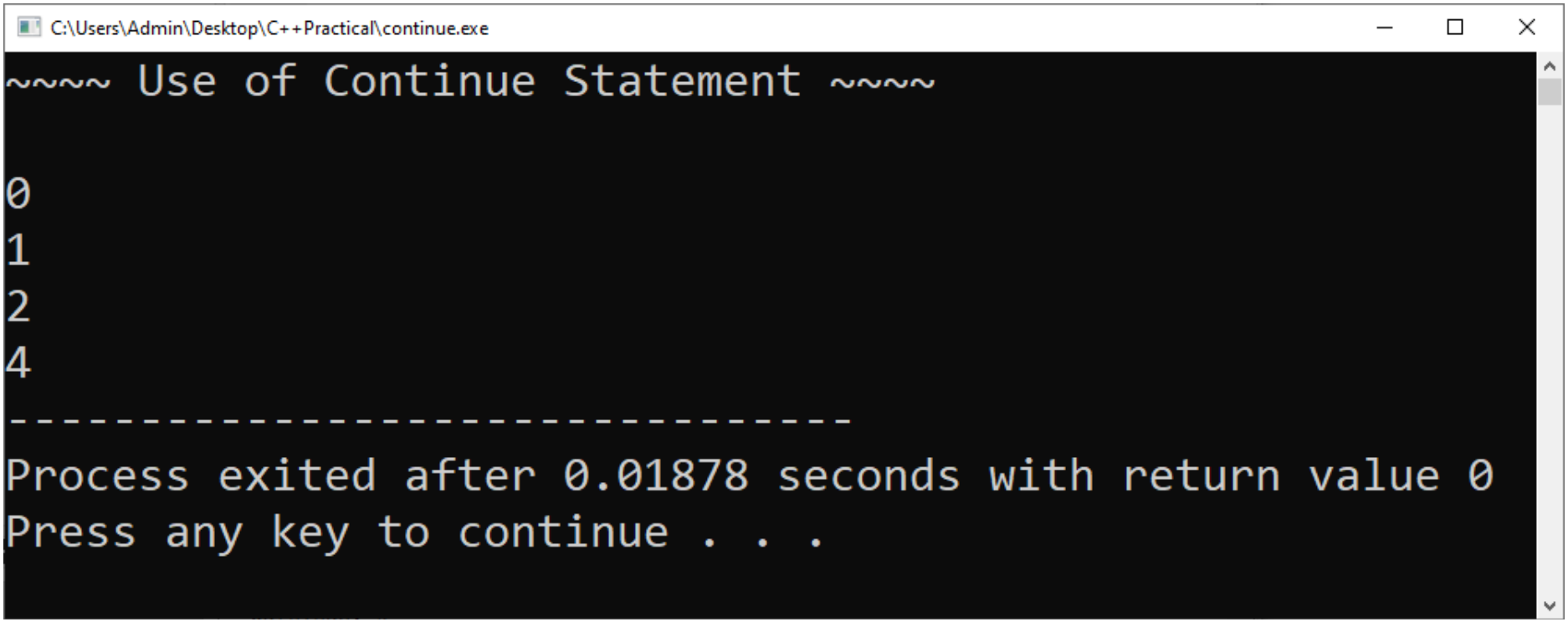
continue;

- E.g.

continue.cpp

```cpp
#include<iostream>
using namespace std;
int main()
{
    int i=0;
    cout<<"~~~~ Use of Continue Statement ~~~~\n";

    for (i=0; i<5; i++)
    {
        if (i==3)
        {
            continue;
        }
        cout<<"\n"<<i;
    }
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\continue.exe
~~~~ Use of Continue Statement ~~~~
0
1
2
4

Process exited after 0.01878 seconds with return value 0
Press any key to continue . . .
~~~~