

# Unit 2: Basics of C++ programming

# C++

- C++ is a statically typed, compiled, general-purpose, **case-sensitive**, free-form programming language that supports procedural, object-oriented, and generic programming.
- C++ is regarded as a **middle-level** language, as it comprises a combination of both **high-level** and **low-level** language features.
- C++ was developed by **Bjarne Stroustrup** starting in **1979** at Bell Labs, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
- C++ is a superset of C.

# Contd..

- Need of C++:

- In order to solve the complex problems and to model the real world perfectly, C++ was developed.
- The primary goal of C++ development was to provide **object-oriented facilities** for program organization together with C's efficiency and flexibility for system programming.

# Contd..

- Uses of C++:

- used to write device driver programs.
- Used to write applications in sensitive sectors such as banking, military , etc.
- C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
- Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.
- C++ was used to develop high performance software such as Mysql, Windows XP, Adobe products, Mozilla Firefox and many more.

# C++ Program Structure

- // First.Cpp

// First C++ program

← comment

#include <iostream>

← preprocessor directive

using namespace std;

← which namespace to use

int main()

← beginning of function named main

{ ← beginning of block for main

cout << "Hello World!";

← output statement

return 0;

↑ string literal

← send 0 to operating system

} ← end of block for main

//other codes- function definitions

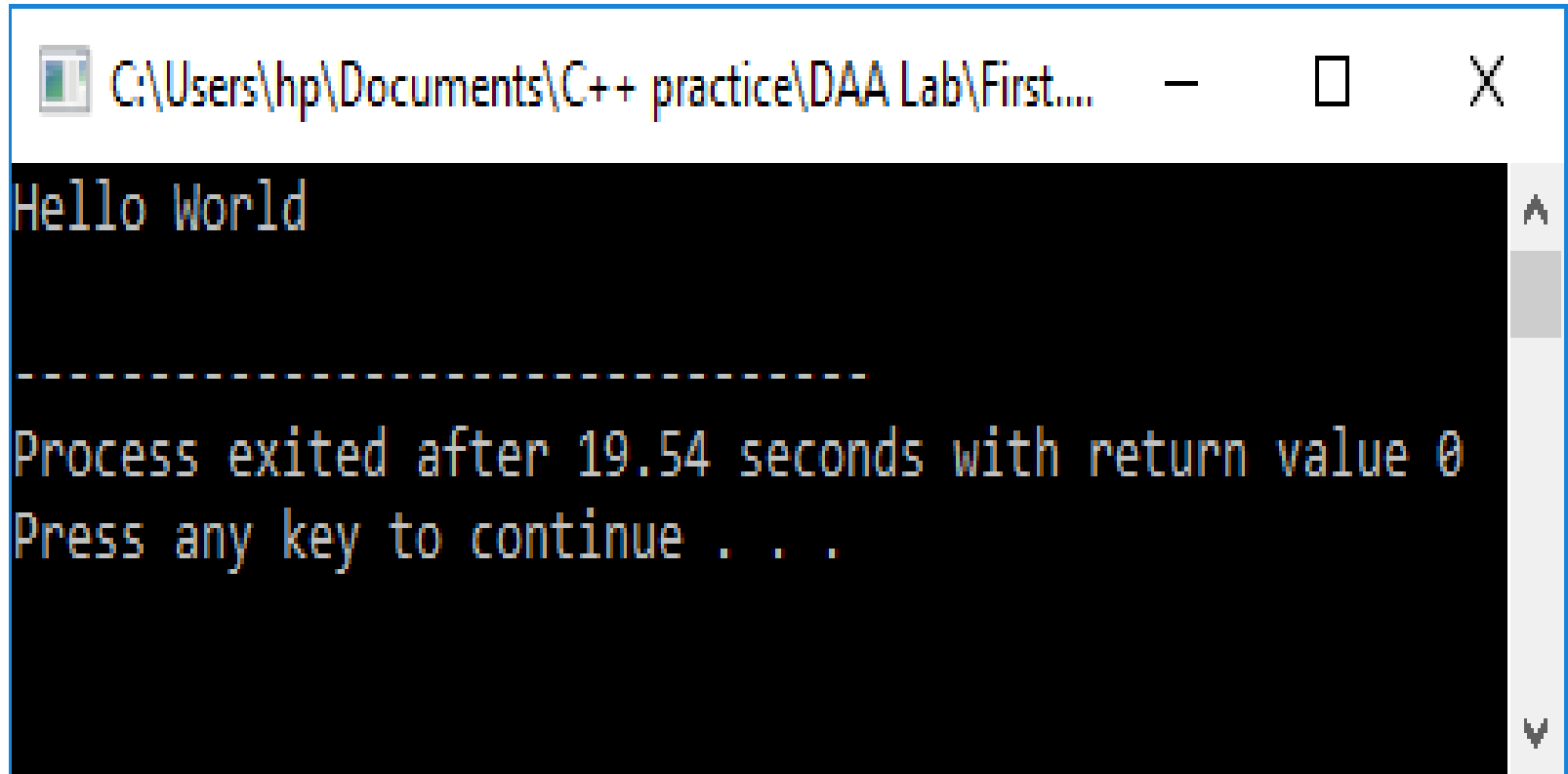
# Contd..

- **Program 1:** Program to print a sentence on the screen.

```
1  //First program in C++
2  //It simply prints a sentence on the screen.
3
4  #include<iostream>
5
6  using namespace std;
7
8  int main()
9  {
10     cout<<"Hello World !" \n";
11
12     return 0;
13 }
```

# Contd...

- Output:



A screenshot of a Windows command prompt window. The title bar at the top shows the file path: C:\Users\hp\Documents\C++ practice\DAA Lab\First.... The window has standard minimize, maximize, and close buttons. The command prompt area has a black background with white text. It displays "Hello World" on the first line. The second line is empty. The third line consists of a dashed line. The fourth line says "Process exited after 19.54 seconds with return value 0". The fifth line says "Press any key to continue . . .". A vertical scrollbar is visible on the right side of the command prompt area.

```
C:\Users\hp\Documents\C++ practice\DAA Lab\First....  
Hello World  
  
-----  
Process exited after 19.54 seconds with return value 0  
Press any key to continue . . .
```

# Contd..

- **Program2:** A Program to add two number and output the sum

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a, b, c;
7
8      cout << "Enter two integers: ";
9      cin >> a;
10     cin >> b;
11
12     // sum of two numbers in stored in variable sumOfTwoNumbers
13     c = a + b;
14
15     // Prints sum
16     cout << a << " + " << b << " = " << c;
17
18     return 0;
19 }
```



# Contd....

- Output:

```
Enter two integers:  
6  
7  
6 + 7 = 13  
-----  
Process exited after 13.17 seconds with return value 0  
Press any key to continue . . .
```

# Character set and Tokens

- **Character set :**
  - Character set is a set of valid characters that a language can recognize. A character represents any **letter, digits**, or any other **sign**.
  - C++ has the following character set :

Character Set		
1.	Letters	Uppercase A-Z Lowercase a-z
2.	Digits	All digits 0-9
3.	Special Characters	All Symbols: , . : ; ? ' " !   \ / ~ _ \$ % # & ^ * - + < > ( ) { } [ ]
4.	White Spaces	Blank space, Horizontal tab, Carriage return, New line, Form feed

# Contd..

- **Tokens:**

- The **smallest individual unit** in a program is known as tokens.

C++ has the following tokens:

- Keywords
- Identifiers
- Constants
- Operators
- Special symbols.

# Contd..

## 1. Keywords:

- keywords are the **reserved words** for special purpose and must not be used as normal identifier names.
- Following table lists the Complete C++ Keywords :

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

# Contd..

## 2. Identifiers:

- Identifiers refers to the **names of variable, functions, arrays, classes, etc.**
- The following rule is used to create a identifier:
  - identifier must start with a letter or underscore symbol (\_), the rest of the characters should be letters, digits or underscores
  - legal identifiers:  
`x x1 x_1 _abc sum RateAveragE`
  - illegal identifiers, why?  
`13 3X %change data-1 my.identifier a(3)`
  - C++ is *case sensitive*:  
`MyVar` and `myvar` are different identifiers
- **Note:** A reserved keyword can not be used as a identifier.

# Contd..

## 3. Constants:

- Constants are the data items that **never change** their value during a program run.
- C++ allows the following kinds of literals :
  - integer-constant
  - character-constant
  - floating-constant
  - string-literal

# Contd..

**4. Operators:** An operator is a symbol that tells the compiler to perform specific **mathematical** or **logical** calculations on operands(variables).

- **Types of operators available in C++**

- Assignment operator
- Arithmetic / Mathematical operator
- Increment Decrement operator
- Relational operator
- Logical operator
- Conditional or Ternary operator
- Binary operator
- Unary operator

# Contd..

- Assignment operator :**

- Assignment operator is used to copy value from right to left variable.
- Suppose we have, float  $X = 5$ ,  $Y = 2$ ;

Operator	Name	Description	Example
=	Equal sign	Copy value from right to left.	$X = Y$ , Now both X and Y have 5
+=	Plus Equal to	Plus Equal to operator will return the addition of right operand and left operand.	$X += Y$ is similar to $X = X + Y$ , now X is 7
-=	Minus Equal to	Minus Equal to operator will return the subtraction of right operand from left operand.	$X -= Y$ is similar to $X = X - Y$ , now X is 3
*=	Multiply Equal to	Multiply Equal to operator will return the product of right operand and left operand.	$X *= Y$ is similar to $X = X * Y$ , now X is 10
/=	Division Equal to	Division Equal to operator will divide right operand by left operand and return the quotient.	$X /= Y$ is similar to $X = X / Y$ , now X is 2.5
%=	Modulus or Mod Equal to	Modulus Equal to operator will divide right operand by left operand and return the mod ( Remainder ).	$X \% = Y$ is similar to $X = X \% Y$ , now X is 1



# Contd..

**Arithmetic operators:** Arithmetic operators are used for mathematical operations.

Suppose we have, `int X = 5, Y = 2;`

Operator	Name	Description	Example
+	Plus	Return the addition of left and right operands.	(X + Y) will return 7
-	Minus	Return the difference b/w right operand from left operand.	(X - Y) will return 3
*	Multiply	Return the product of left and right operands.	(X * Y) will return 10
/	Division	Return the Quetiont from left operand by right operand.	(X / Y) will return 2(both are int, int doesn't support decimal)
%	Modulus or Mod	Return the Modulus ( Remainder ) from left operand by right operand.	(X % Y) will return 1

# Contd..

- **Relational operators:** Relational operators are used for checking conditions whether the given condition is true or false. If the condition is true, it will return non-zero value, if the condition is false, it will return 0.
- Suppose we have, `int X = 5, Y = 2;`

Operator	Name	Description	Example
>	Greater then	Check whether the left operand is greater then right operand or not.	(X > Y) will return true
<	Smaller then	Check whether the left operand is smaller then right operand or not.	(X < Y) will return false
>=	Greater then or Equal to	Check whether the left operand is greater or equal to right operand or not.	(X >= Y) will return true
<=	Smaller then or Equal to	Check whether the left operand is smaller or equal to right operand or not.	(X <= Y) will return false
==	Equal to	Check whether the both operands are equal or not.	(X == Y) will return false
!=	Not Equal to	Check whether the both operands are equal or not.	(X != Y) will return true

# Contd..

- **Logical operators:**

- Logical operators are used in situation when we have more then one condition in a single if statement.
- Suppose we have, int X = 5, Y = 2;

Operator	Name	Description	Example
&&	AND	Return true if all conditions are true, return false if any of the condition is false.	if(X > Y && Y < X) will return true
	OR	Return false if all conditions are false, return true if any of the condition is true.	if(X > Y    X < Y) will return true
!	NOT	Return true if condition is false, return false if condition is true.	if(!(X > Y)) will return false

# Contd..

- **Conditional Operator ?:**

- The conditional operator is also known as ternary operator. It is called ternary operator because it takes three arguments. First is condition, second and third is value. The conditional operator check the condition, if condition is true, it will return second value, if condition is false, it will return third value.

- **Syntax :**

- `val = condition ? val1 : val2;`

- **Example :**

```
int main()
{
    int X=5,Y=2,lrg;
    lrg = (X>Y) ? X : Y;
    cout << "\nLargest number is : " << lrg;
}
```

Output : Largest number is : 5

# Contd..

- **Binary operator**

- Binary operators are those operators that works with at least two operands such as (Arithmetic operators)  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ .

- **Unary operator**

- Unary operators are those operators that works with single operands such as (Increment or Decrement operators)  $++$  and  $--$ .

# Contd..

**5. Special Symbols:** Escape characters are known as special symbols in C++. An escape sequence is represented by a backslash (\) followed by one or more characters.

Following table lists the C++ Escape Sequences :

Escape Sequence	Nongraphic Character
\a	Audible bell (alert)
\b	Backspace
\f	Formfeed
\n	Newline or linefeed
\r	Carriage Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quote
\"	Double quote
\?	Question mark
\On	Octal number (On represents the number in octal)
\xHn	Hexadecimal number (Hn represents the number in hexadecimal)
\0	Null

# C++ Data Types

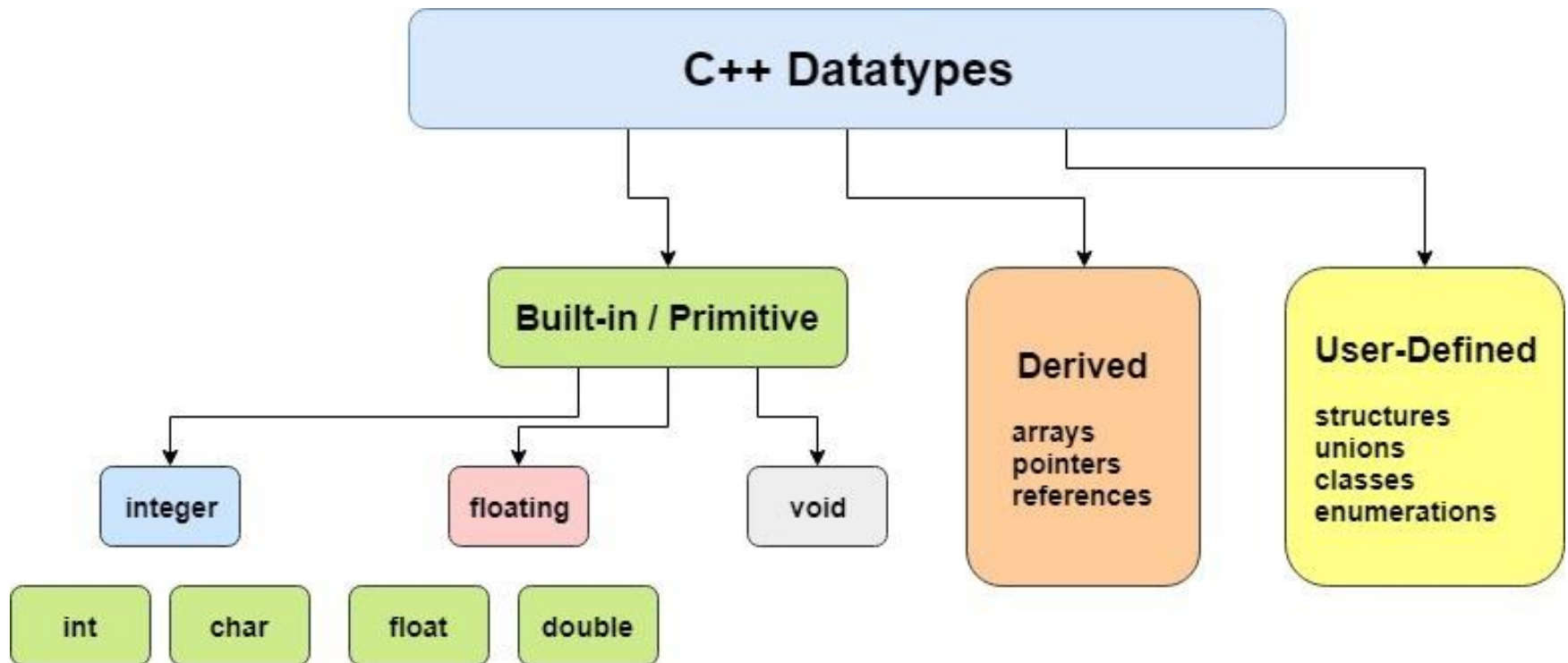
- A data type defines a set of **values** that a variable can **store** along with a set of operations that can be performed on that variable.
- Each and every variable and constants that we are using in our program must be declared before its use.
- The reason behind it is that, each and every variable/constant must be placed in some memory location. Such memory allocation is done at compile time by the compiler.

# C++ Data Types

- The various data types provided by C++ are:
  - *built-in data types,*
  - *derived data types and*
  - *user-defined data types*



# C++ Data Types



# Contd..

- Fundamental data types( built-in/basic data type):

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void

- Several of the basic types can be modified using one or more of these type modifiers –
  - signed
  - unsigned
  - short
  - long

# Contd..

- The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

# Contd..

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

## Contd..

- The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.
- Following is the example, which will produce correct size of various data types on your computer.

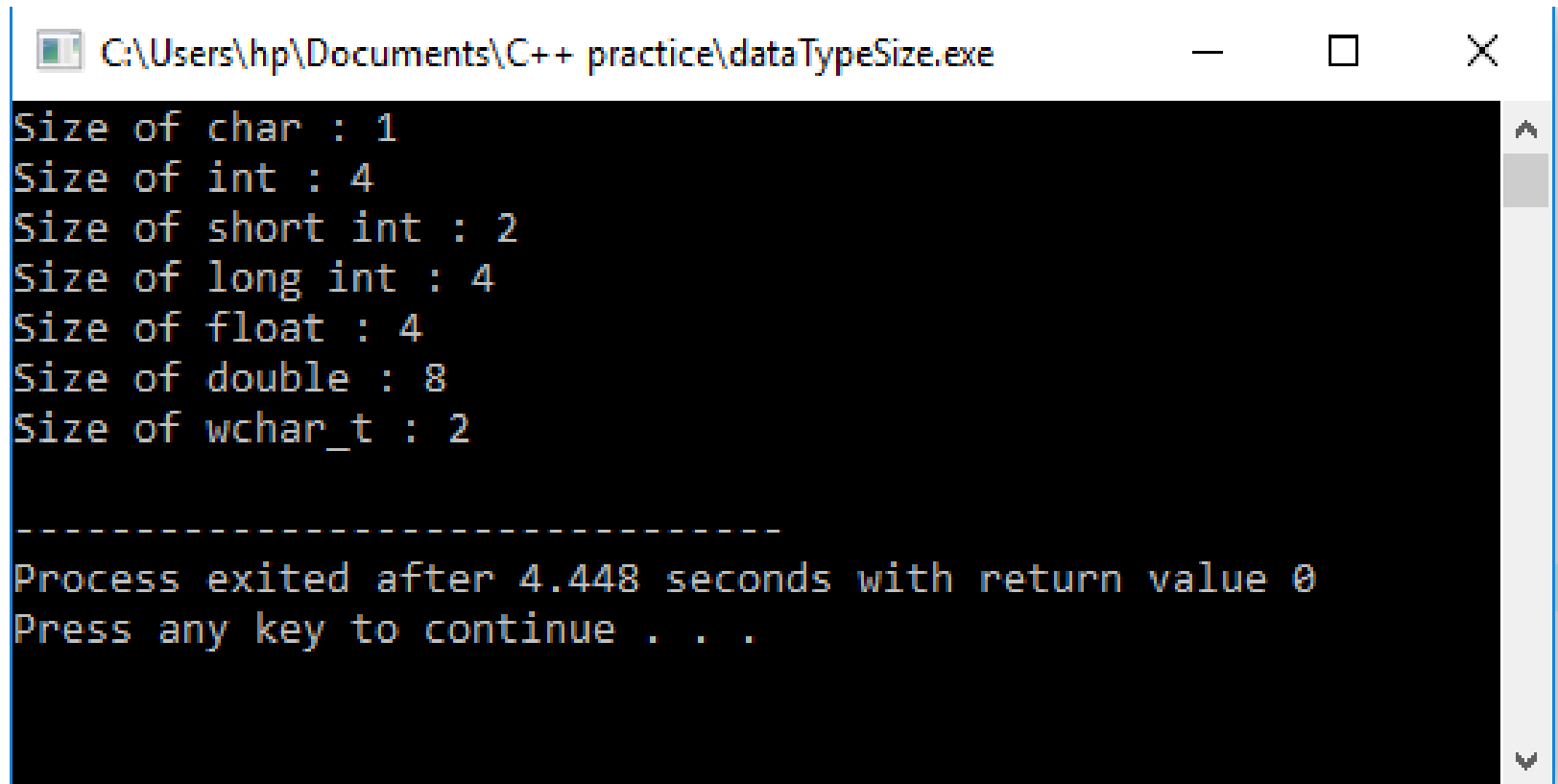
# Contd..

- **Program:**

```
1 //program to find the size of data types in your computer
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     cout << "Size of char : " << sizeof(char) << endl;
7     cout << "Size of int : " << sizeof(int) << endl;
8     cout << "Size of short int : " << sizeof(short int) << endl;
9     cout << "Size of long int : " << sizeof(long int) << endl;
10    cout << "Size of float : " << sizeof(float) << endl;
11    cout << "Size of double : " << sizeof(double) << endl;
12
13
14    return 0;
15 }
```

# Contd..

- Output:



```
C:\Users\hp\Documents\C++ practice\dataTypeSize.exe
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 2

-----
Process exited after 4.448 seconds with return value 0
Press any key to continue . . .
```

# Contd..

- **Derived data types:**
  - Data types that are derived from the built in data types are known as derived data types.
  - The various derived data types provided by C++ are:
    - Arrays
    - Functions
    - Pointers and etc.



# Contd..

- User defined data types:
  - The data types that are defined by the user as per their need is called user defined data types,
  - Various user defined data types provided by C++ are
    - Structures
    - Unions
    - Enumerations and
    - classes

# Contd..

- Example Program :

```
1  // Program to Swap two Numbers (Using Temporary Variable)
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      int a = 5, b = 10, temp;
8
9      cout << "Before swapping." << endl;
10     cout << "a = " << a << ", b = " << b << endl;
11
12     temp = a;
13     a = b;
14     b = temp;
15
16     cout << "\nAfter swapping." << endl;
17     cout << "a = " << a << ", b = " << b << endl;
18
19     return 0;
20 }
```



C:\Users\hp\Documents\C++ practice\DAA Lab\Swap.e...



Before swapping.

a = 5, b = 10

After swapping.

a = 10, b = 5

-----

Process exited after 3.837 seconds with return value 0

Press any key to continue . . .



# Type Conversion

- Converting an expression of a given type into another type is known as type conversion.
- When data (constants and variables) of different types are mixed in an expression, they are converted to the same type.
- In C++ there are two types of type conversion:
  - Implicit type conversion and
  - Explicit type conversion

# Contd...

- **Implicit (Automatic) Type conversion:**
  - An implicit type conversion is a conversion performed by the compiler **without programmer's intervention.**
  - There are two basic types of implicit type conversion:
    - **promotions** and
    - **conversions.**

# Contd..

- **Promotion**

- Whenever a value from one type is converted into a value of a larger similar data type, this is called a promotion (or **widening**) For example, an int can be widened into a long, or a float promoted into a double:
- They are always safe, and no data loss will result.
- E.g., `int x = 3;`

`float y = 2.5, z;`

`z = x + y;`

# Contd..

- **Conversion:**

- When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion**. For example:
  - `double d = 3; // convert integer 3 to a double (between different types)`
  - `short s = 2; // convert integer 2 to a short (from larger to smaller type)`
- conversions **may or may not result in a loss of data.**

# Contd..

- **Explicit type conversion(Type casting):**

- Type casting is the forceful conversion from one type to another type.
- Sometimes a programmer needs to convert a value from one type to another type forcefully in a situation where compiler will not do it automatically.
- For this C++ permits explicit type conversion of variables or expressions as follows:
  - (Type-name) expression // C notation
  - Type- name (expression) // C++ notation



# Contd..

- For example:

```
short int a = 10000;
```

```
Short int b= long(a)* 5/2; // correct
```

Here if we do not cast a into long, the result of sub-expression  $a*5$  will be 50000 which is outside of the range of data type short int and causes the program to produce wrong result. But this case automatic type conversion is not done by the compiler and hence programmer needs to use explicit type conversion to produce the right result.

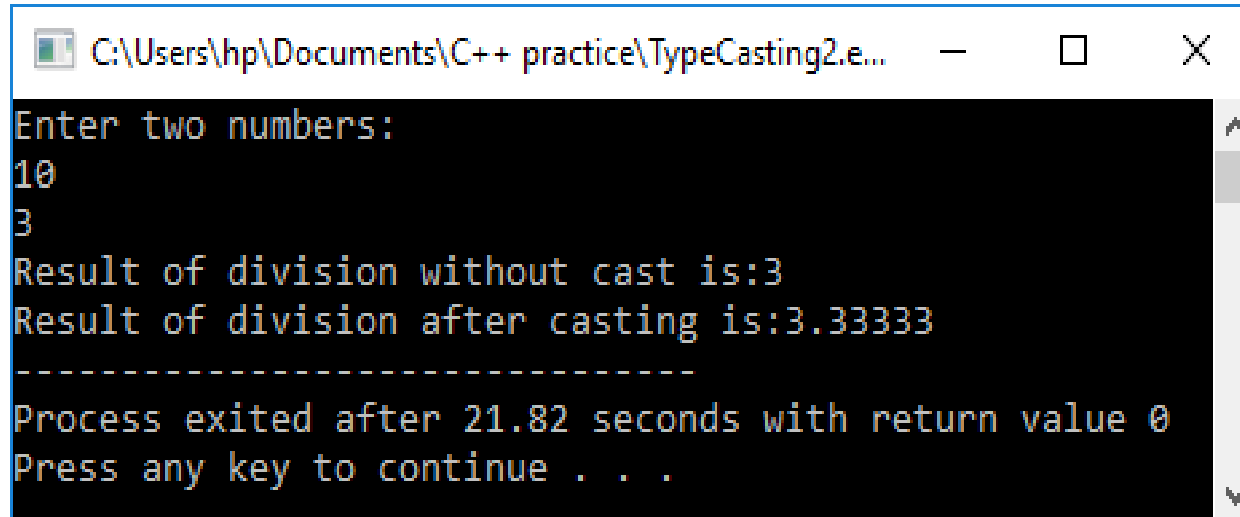
# Contd..

- Program:

```
1 // program to illustrate the type casting
2 #include<iostream>
3 using namespace std;
4 int main()
5 {
6     int x, y;
7
8     float z;
9
10    cout<<"Enter two numbers:"<<endl;
11
12    cin>>x>>y;
13
14    z= x/y;
15
16    cout<<"Result of division without cast is:"<<z<<endl;
17
18    z= float(x)/y;
19
20    cout<<"Result of division after casting is:"<<z;
21
22    return 0;
23 }
```

# Contd..

- Output:



```
C:\Users\hp\Documents\C++ practice\TypeCasting2.e...  
Enter two numbers:  
10  
3  
Result of division without cast is:3  
Result of division after casting is:3.33333  
-----  
Process exited after 21.82 seconds with return value 0  
Press any key to continue . . .
```

# Preprocessor Directives

- preprocessor directives are lines included in the code of programs preceded by a hash sign (#).
- These lines are not program statements but **directives** for the **preprocessor**.
  - The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.
- These preprocessor directives extend only across a **single line** of code.
- can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).
- No semicolon (;) is expected at the end of a preprocessor directive.

# Contd..

- The **#define** directive:

- The **#define** preprocessor directive creates symbolic constants called **macro**.
- **Syntax:** **#define** **macro-name** **replacement**
- When the preprocessor encounters this directive, it replaces any occurrence of macro in the rest of the code by replacement before the program compiled.

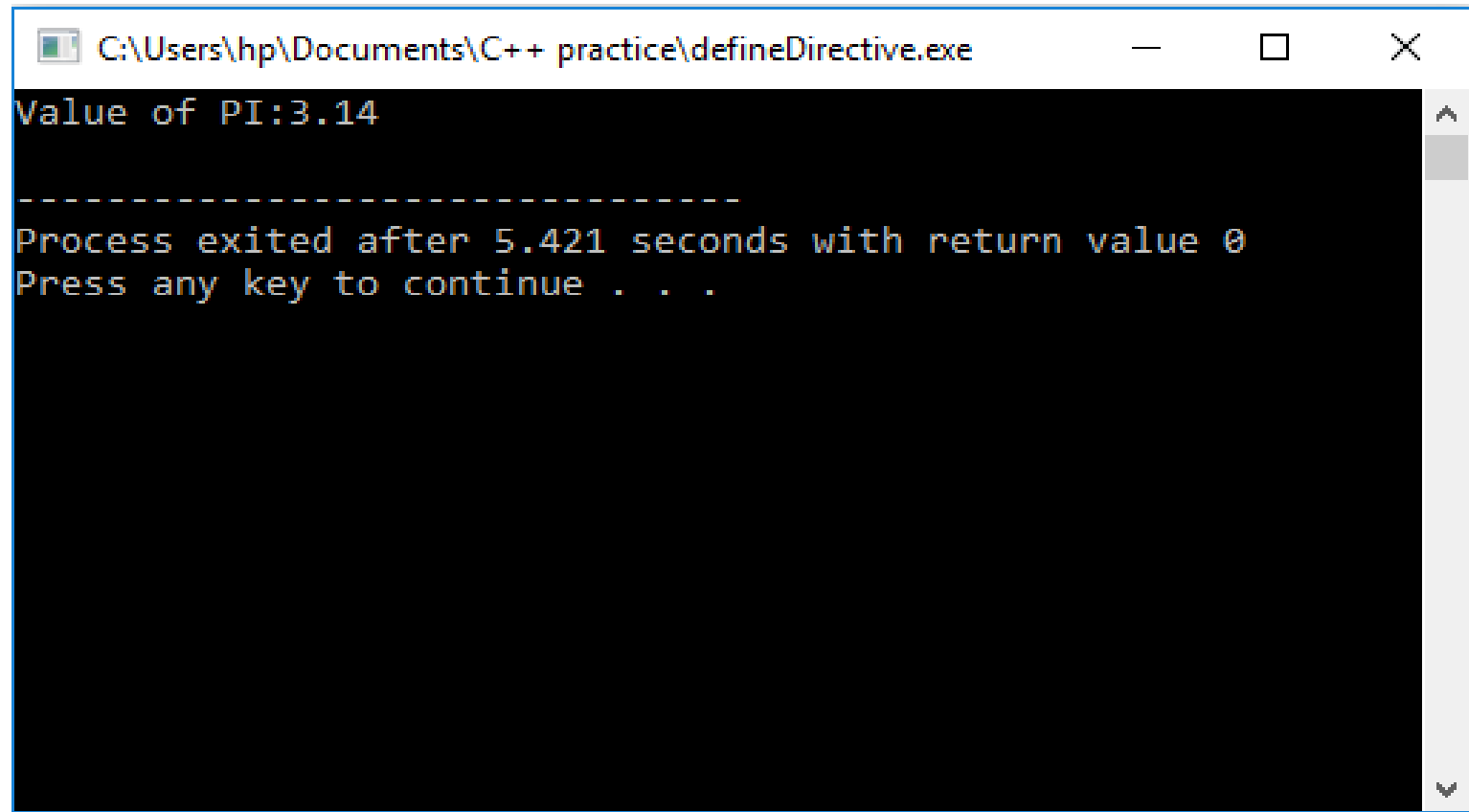
# Contd.

- e.g.,

```
1 // program to illustrate the define directive
2 #include<iostream>
3 #define PI 3.14
4 using namespace std;
5 int main()
6 {
7     cout<<"Value of PI:"<< PI << endl;
8     return 0;
9 }
```

# Contd...

- Output:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Users\hp\Documents\C++ practice\defineDirective.exe" and includes standard minimize, maximize, and close buttons. The command prompt has a black background with white text. The output displayed is: "Value of PI:3.14", followed by a line of ten dashes "-----", then "Process exited after 5.421 seconds with return value 0", and finally "Press any key to continue . . .". A vertical scrollbar is visible on the right side of the window.

```
C:\Users\hp\Documents\C++ practice\defineDirective.exe
Value of PI:3.14
-----
Process exited after 5.421 seconds with return value 0
Press any key to continue . . .
```

# Contd...

- The `#include` directive:
  - When the preprocessor finds an `# include` directive it replaces the header by the entire content of the specified header file or file.
  - To include headers provided by the implementation use the following syntax:
    - `#include<header>`
  - To include a file use the following syntax:
    - `#include "file"`



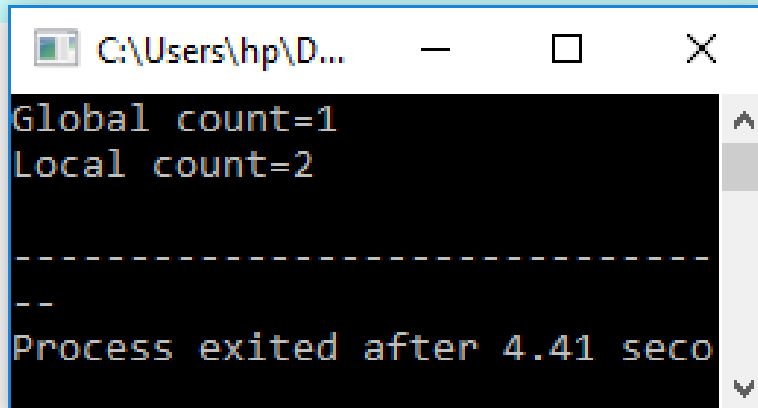
# Scope Resolution operator(::)

- The scope resolution (::) operator is used to qualify hidden names so that you can still use them.
- a namespace scope or global scope name is hidden by the declaration of the same name in a block.

# Contd..

- E.g.,

```
1 // program to illustrate the use of scope resolution(::) operator
2 #include<iostream>
3 using namespace std;
4 int count =0;
5 int main()
6 {
7     int count=0;
8     :: count = 1; // set global count to 1
9     cout<<"Global count="<<::count<<endl;
10    count =2; // set local count to 2
11    cout<<"Local count="<<count<<endl;
12    return 0;
13 }
```



```
C:\Users\hp\D...
Global count=1
Local count=2
-----
--
Process exited after 4.41 seco
```

# Namespace

- Namespaces allow to **group entities** like classes, objects and functions under a name.
- In this way the **global scope can be divided in "sub-scopes"**, each one with its own name.
- The purpose of namespace is to localize a name of identifiers **to get rid of naming conflicts across different modules** designed by different members of programming team.
- Syntax for defining Namespace:
  - **namespace** namespace\_name
    - {
    - entities; // declaration of variables, functions, classes, etc
    - }

# Contd..

- For example:

- namespace myNamespace

- {

- int a, b; // scope of variable a and b is within this namespace only

- }

- In order to access these variables from outside the myNamespace namespace we have to use the scope resolution operator :: as follows:

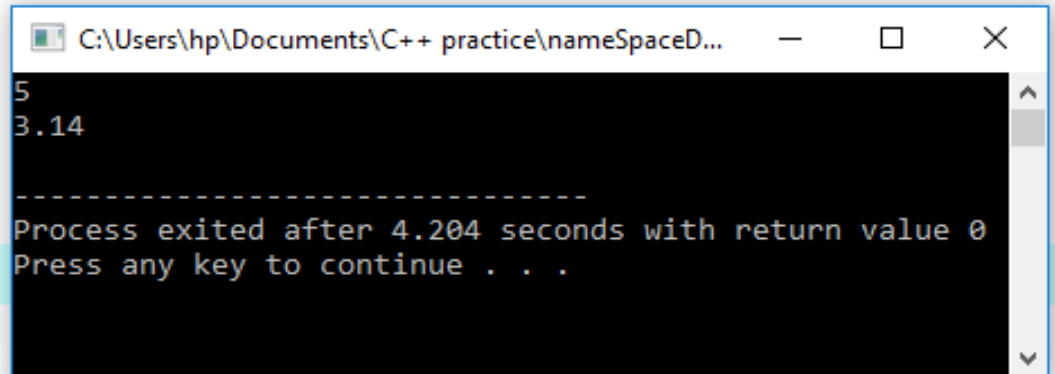
- myNamespace::a

- myNamespace::b

# Contd..

- Example1:

```
1  // program to illustrate the concept of namespace
2  #include<iostream>
3  using namespace std;
4  namespace first
5  {
6      int var=5;
7  }
8  namespace second
9  {
10     double var = 3.14;
11 }
12 int main()
13 {
14     cout<<first::var<<endl;
15     cout<<second::var<<endl;
16     return 0;
17 }
```



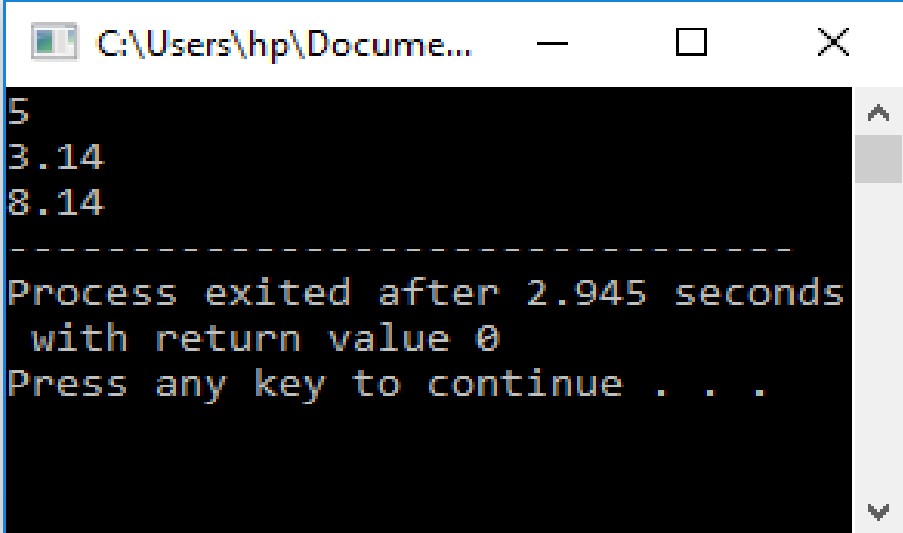
The screenshot shows a Windows command prompt window titled "C:\Users\hp\Documents\C++ practice\nameSpaceD...". The output of the program is displayed as follows:

```
5
3.14
-----
Process exited after 4.204 seconds with return value 0
Press any key to continue . . .
```

# Contd..

- Repeated use of variable in the name space

```
1 // Repeated use of the variable in the namespace
2 #include<iostream>
3 using namespace std;
4 namespace first
5 {
6     int var=5;
7 }
8 namespace second
9 {
10    double var = 3.14;
11 }
12 int main()
13 {
14    cout<<first::var<<endl;
15    cout<<second::var<<endl;
16    double sum =first::var+ second::var;
17    cout<<sum;
18    return 0;
19 }
```



```
C:\Users\hp\Docume...
5
3.14
8.14
-----
Process exited after 2.945 seconds
with return value 0
Press any key to continue . . .
```

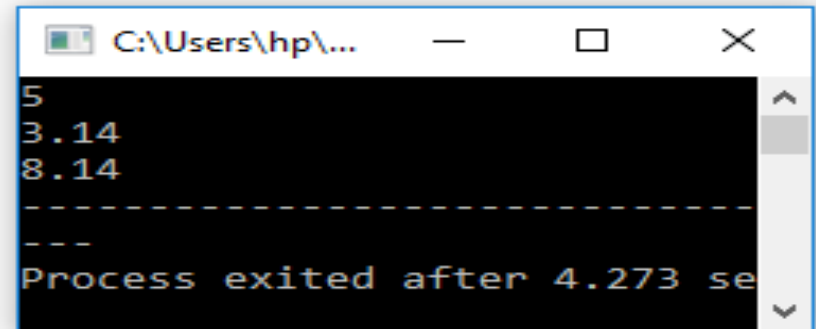
# Contd..

- Repetitive use of namespace name can be eliminated by including the namespace in the source file as:

– **Syntax:** using namespace namespace\_name;

– **Example:**

```
1 // Repeated use of the variable in the namespace
2 #include<iostream>
3 using namespace std;
4 namespace first
5 {
6     int var1=5;
7 }
8 namespace second
9 {
10    double var2 = 3.14;
11 }
12 using namespace first;
13 using namespace second;
14 int main()
15 {
16     cout<<var1<<endl;
17     cout<<var2<<endl;
18     double sum =var1+ var2;
19     cout<<sum;
20     return 0;
21 }
```



```
C:\Users\hp\...
5
3.14
8.14
-----
---
Process exited after 4.273 se
```

same variable name vayo  
vane, pahillai error  
aauxa

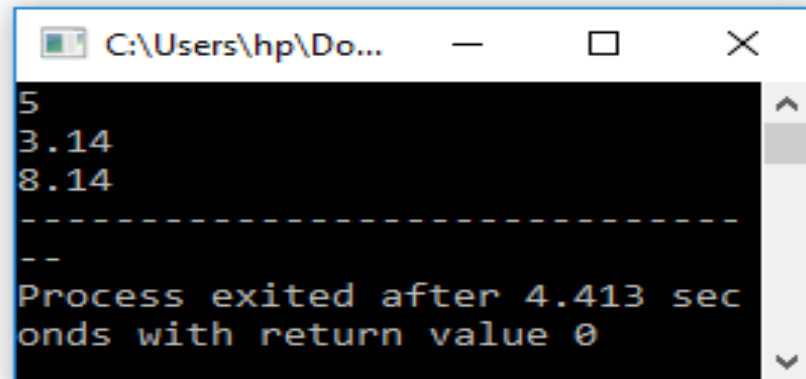
# Contd..

- Including the specific component from the name space as:

- Syntax: Using namespace\_name::component\_name

- Example:

```
2  #include<iostream>
3  using namespace std;
4  namespace first
5  {
6      int var1=5;
7  }
8  namespace second
9  {
10     int var1 = 3;
11     double var2 = 3.14;
12 }
13 using namespace first;
14 using second::var2;
15 int main()
16 {
17     cout<<var1<<endl;
18     cout<<var2<<endl;
19     double sum =var1+ var2;
20     cout<<sum;
21     return 0;
22 }
```



```
C:\Users\hp\Do...
5
3.14
8.14
-----
--
Process exited after 4.413 sec
onds with return value 0
```



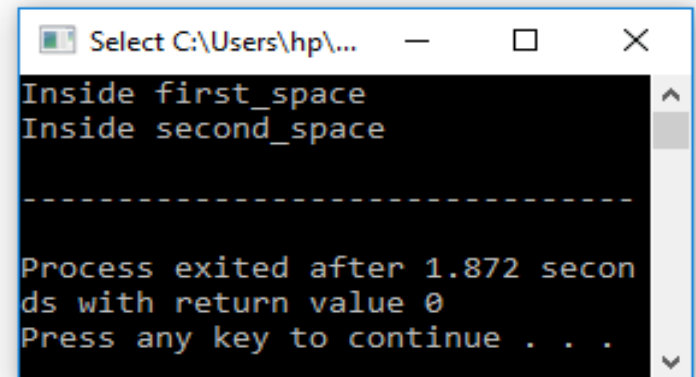
# Note the following fact about the namespace:

- We can define namespace with the same namespace name which has already been used.
- **Nesting namespace:** Namespaces can be nested as:
  - namespace outer
    - { entities;
    - namespace inner
      - {
      - entities;
      - }}
- **Name space Aliases:** shorter names can be assigned to namespaces as,
  - E.g., namespace oi = outer:: inner;

# Contd..

- Example2:

```
1  #include <iostream>
2  using namespace std;
3
4  // first name space
5  namespace first_space {
6      void func() {
7          cout << "Inside first_space" << endl;
8      }
9  }
10
11 // second name space
12 namespace second_space {
13     void func() {
14         cout << "Inside second_space" << endl;
15     }
16 }
17
18 int main () {
19     // Calls function from first name space.
20     first_space::func();
21
22     // Calls function from second name space.
23     second_space::func();
24
25     return 0;
26 }
```



```
Select C:\Users\hp\...
Inside first_space
Inside second_space

-----

Process exited after 1.872 seconds with return value 0
Press any key to continue . . .
```

# Input/ Output stream(cin and cout)

- C++ uses a **convenient abstraction** called **streams** (a flow of data) to perform input and output operations in sequential media such as the screen, the keyboard or a file.
- The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:
  - cout
  - cin

# Contd..

- The output with cout:

- On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is **cout**.
- For formatted output operations, cout is used together with the *insertion operator* (<<).
- The << operator inserts the data that follows it into the stream object that precedes it.
- **For example:** in the statement `cout<<"Welcome!"`; the << operator directs the string constant "Welcome" to cout, which sends it for the display to the monitor.

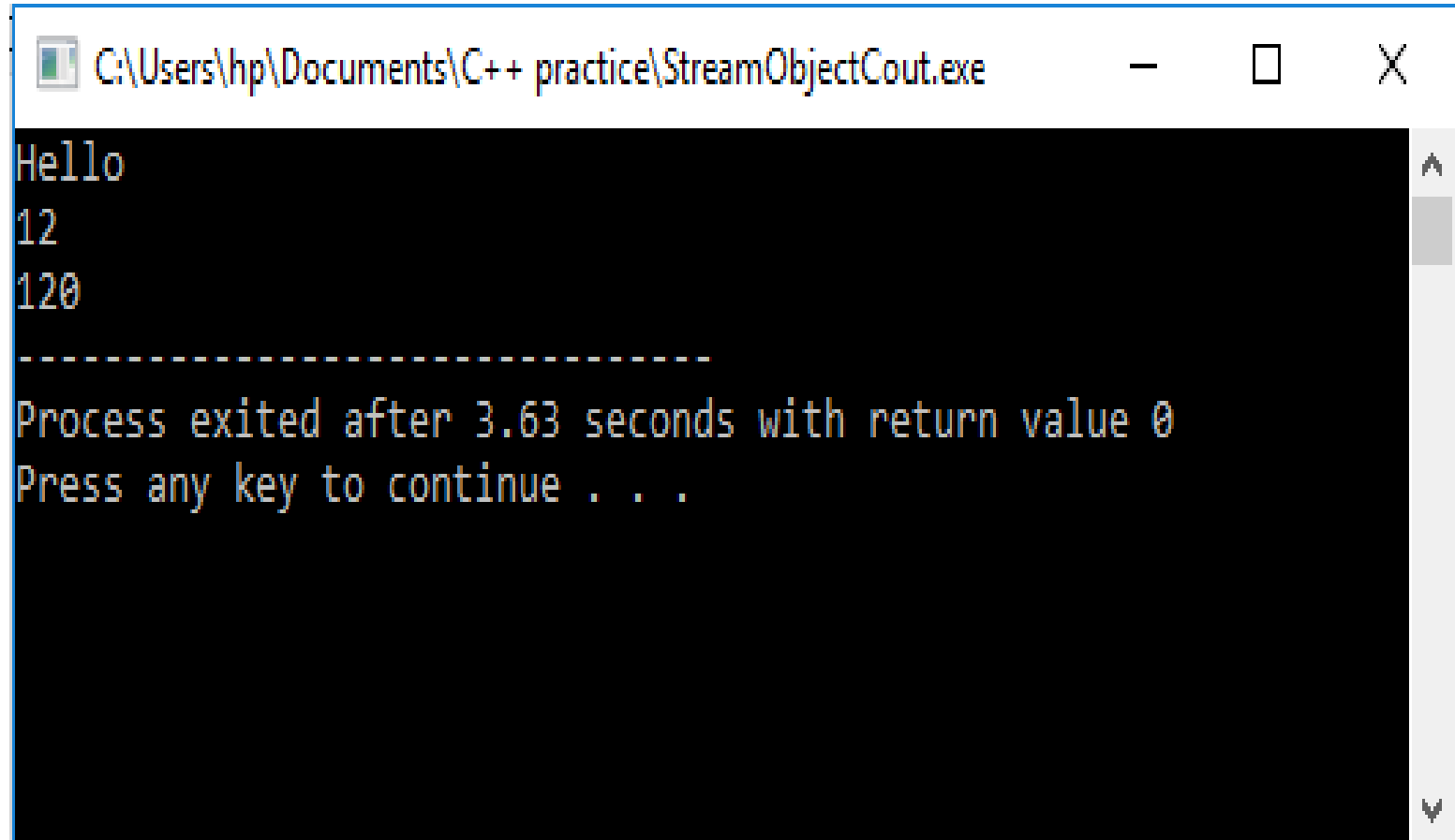
# Contd..

- E.g.,

```
1 // Program to illustrate the stream object cout
2 #include<iostream>
3 using namespace std;
4 int main()
5 {
6     int Hello= 12;
7
8     cout<<"Hello \n";    // Prints the string Hello on the screen
9
10    cout<<Hello <<"\n"; // Prints the content of variable Hello
11
12    cout<<120;           //prints number 120 on the screen
13
14    return 0;
15 }
```

# Contd...

- Output:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Users\hp\Documents\C++ practice\StreamObjectCout.exe" and includes standard minimize, maximize, and close buttons. The command prompt has a black background with white text. The output displayed is: "Hello", "12", "120", followed by a dashed line "-----". Below the dashed line, it says "Process exited after 3.63 seconds with return value 0" and "Press any key to continue . . .". A vertical scrollbar is visible on the right side of the window.

```
C:\Users\hp\Documents\C++ practice\StreamObjectCout.exe
Hello
12
120
-----
Process exited after 3.63 seconds with return value 0
Press any key to continue . . .
```

## Contd..

- **Cascading of insertion operator(<<):**
  - Multiple insertion operations (<<) can be chained in a single statement to direct a series of output stream to the cout object.
  - The streams are directed from left to right.
  - **For example:** In the statement `cout<<"sum ="<<x+y;` the string `"sum ="` will be directed first and then the value of `x+y`.

# Contd..

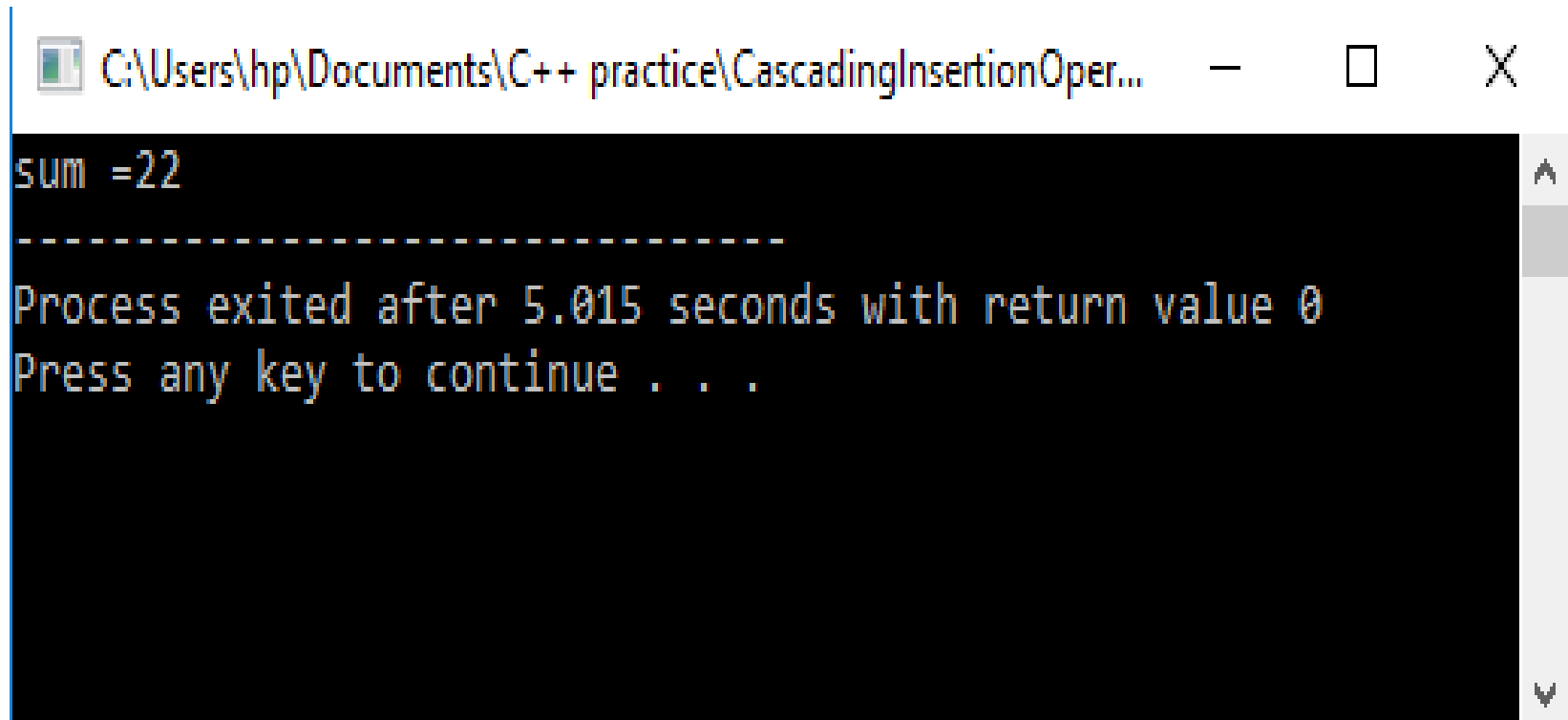
- E.g.,

```
1 // program to illustrate the cascading of << (insertion) operator
2 #include<iostream>
3
4 using namespace std;
5
6 int main()
7
8 {
9     int x = 12;
10
11     int y =10;
12
13     cout<<"sum ="<<x+y;
14
15     return 0;
16 }
```



# Contd..

- Output:



A screenshot of a Windows command prompt window. The title bar at the top shows the file path "C:\Users\hp\Documents\C++ practice\CascadingInsertionOper..." followed by standard window controls (minimize, maximize, close). The command prompt area has a black background with text in a monospaced font. The output displayed is: "sum =22", a dashed line of hyphens, "Process exited after 5.015 seconds with return value 0", and "Press any key to continue . . .".

```
C:\Users\hp\Documents\C++ practice\CascadingInsertionOper...  
sum =22  
-----  
Process exited after 5.015 seconds with return value 0  
Press any key to continue . . .
```

# Contd..

- **Input with cin:**

- In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.
- For formatted input operations, cin is used together with the extraction operator(>>).
- This operator is then followed by the variable where the extracted data is stored.
- The >> operator extracts the value from the stream object cin in its left and place to the variable on its right.
- **For example:** In the statement `cin>> a;` The >> operator extracts the value from cin object that is entered from the keyboard and assigns it to the variable a.

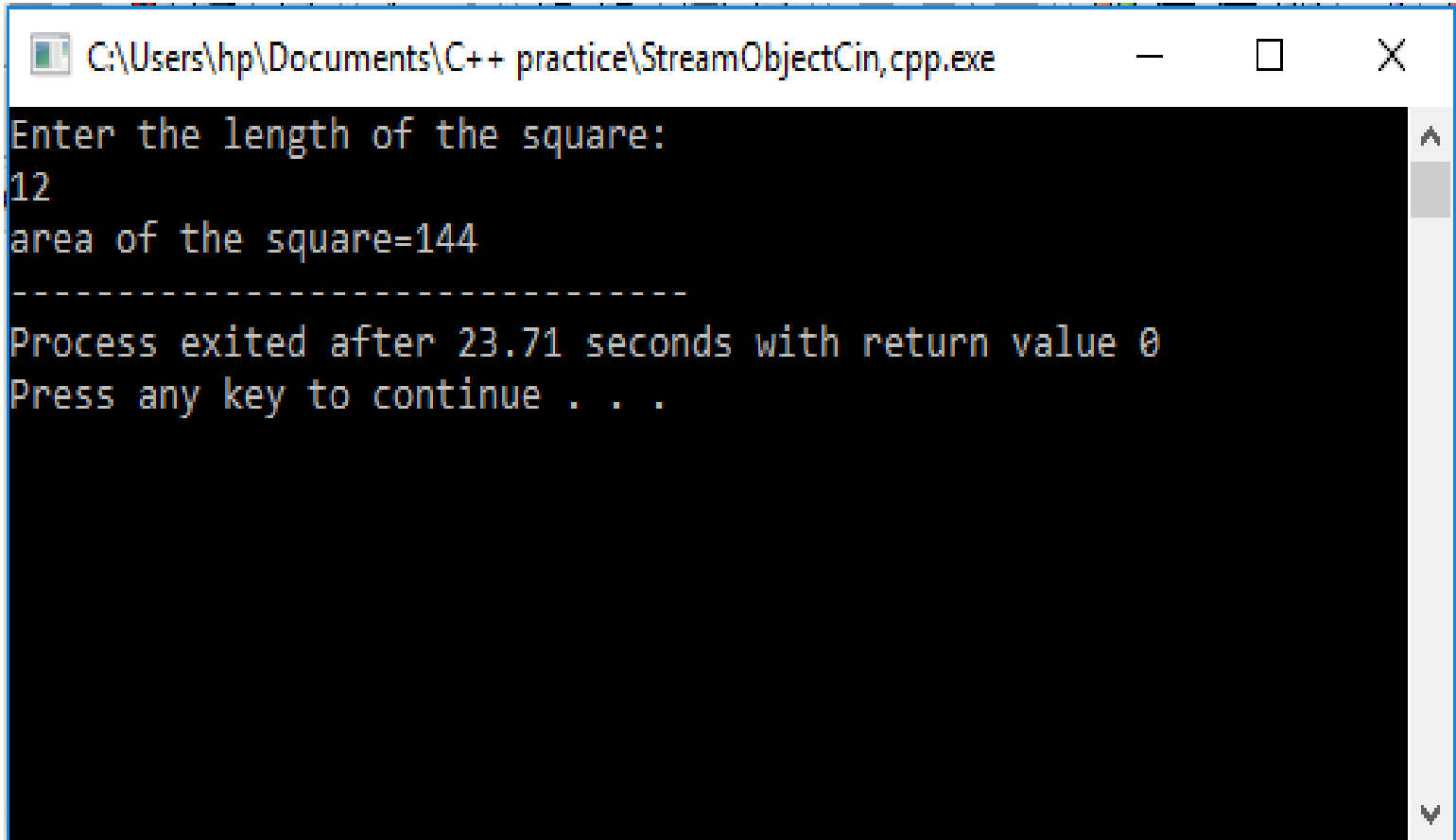
# Contd..

- E.g.,

```
1 //program to illustrate the Stream object cin
2 // find the area of the square
3 #include<iostream>
4 using namespace std;
5 int main()
6 {
7     int a, l;
8     cout<<"Enter the length of the square:\n";
9
10    cin>>l; // Extract(Takes) value from the cin object and assigns it to l
11    a = l*l;
12    cout<<"area of the square=";
13    cout<<a;
14    return 0;
15 }
```

# Contd..

- Output:



```
C:\Users\hp\Documents\C++ practice\StreamObjectCin,cpp.exe
Enter the length of the square:
12
area of the square=144
-----
Process exited after 23.71 seconds with return value 0
Press any key to continue . . .
```

# Contd..

- Cascading of extraction (>>) operator:
  - Multiple extraction operations (>>) can be chained in a single statement to extract a series of input stream from the cin object.
  - The streams are extracted and values are assigned from left to right.
  - **For example:** In the statement `cin>>l>>b;` , the first value entered will be assigned to the variable `l` and the second value entered will be assigned to variable `b`.

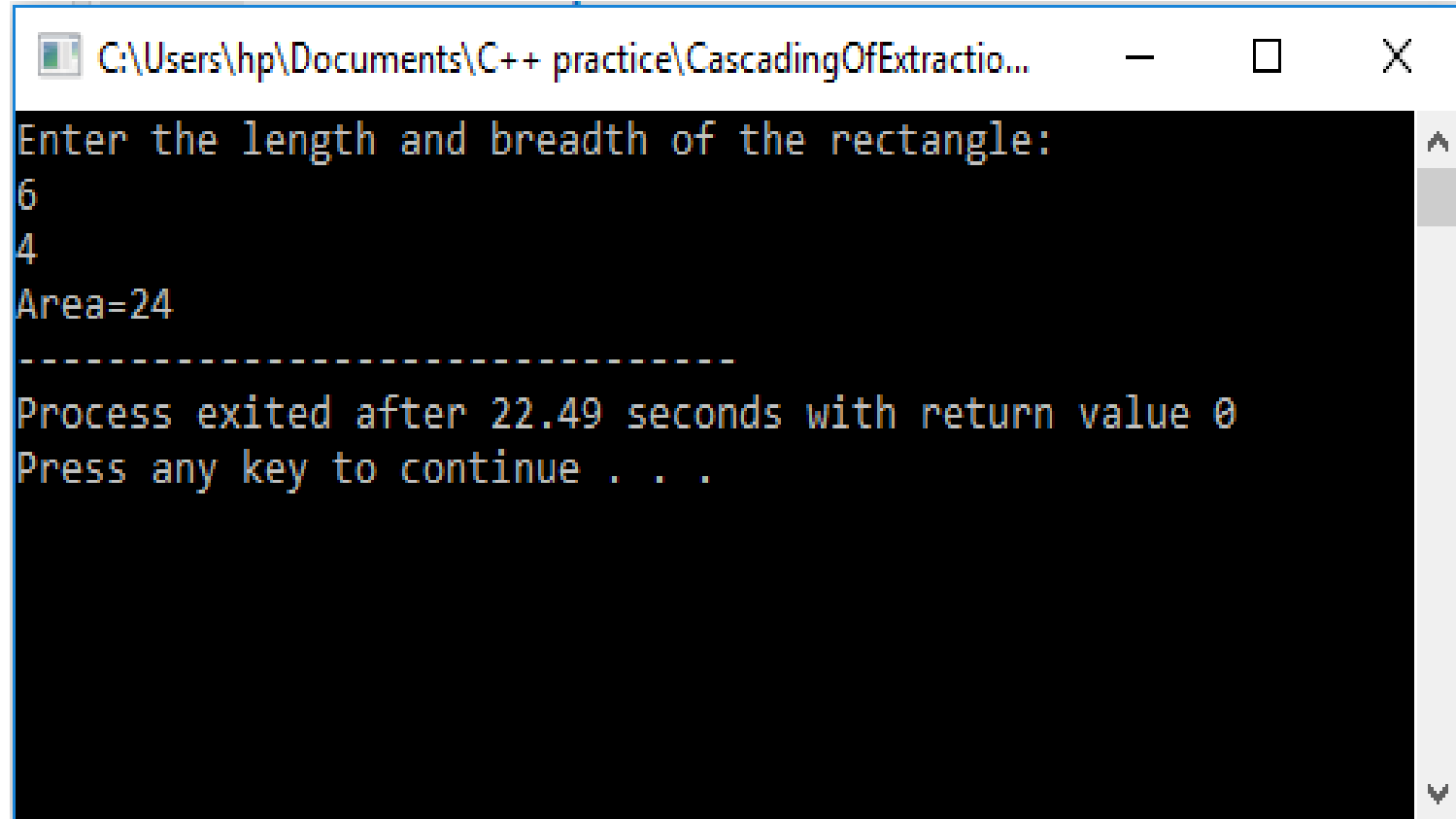
# Contd..

- E.g.,:

```
1 // cascading of << and >> operator
2 // program to find the area of the rectangle
3 #include<iostream>
4 using namespace std;
5 int main()
6 {
7     int a, l, b;
8     cout<<"Enter the length and breadth of the rectangle:\n";
9
10    cin>>l>>b;           // cascading of extraction operator
11
12    a = l*b;
13    |
14    cout<<"Area="<<a; // cascading of insertion operator
15    return 0;
16 }
```

# Contd..

- Output:



```
C:\Users\hp\Documents\C++ practice\CascadingOfExtractio...  
Enter the length and breadth of the rectangle:  
6  
4  
Area=24  
-----  
Process exited after 22.49 seconds with return value 0  
Press any key to continue . . .
```

# Contd..

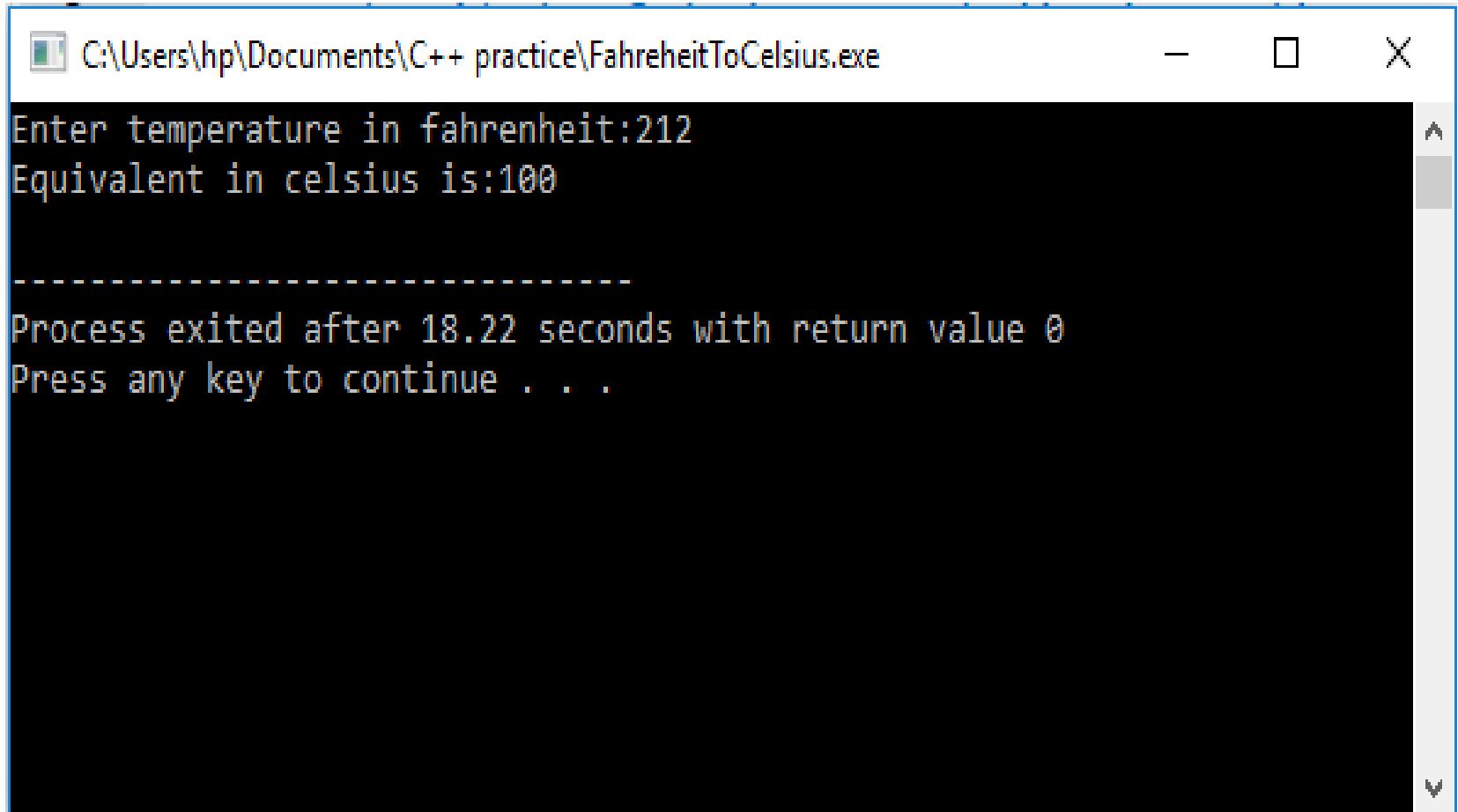
- **Example Program:** to convert temperature in Fahrenheit to

```
1  /*program asks the user for a temperature in degrees Fahrenheit,  
2  converts it to Celsius, and displays the result.*/  
3  #include<iostream>  
4  using namespace std;  
5  int main()  
6  {  
7      int ftemp;                //for temperature in fahrenheit  
8      cout<< "Enter temperature in fahrenheit:";  
9  
10     cin >> ftemp;  
11  
12     int ctemp = (ftemp-32) * 5 / 9;  
13  
14     cout << "Equivalent in celsius is:" << ctemp << '\n';  
15     return 0;  
16 }
```



# Contd..

- Output:



```
C:\Users\hp\Documents\C++ practice\FahreheitToCelsius.exe
Enter temperature in fahrenheit:212
Equivalent in celsius is:100

-----
Process exited after 18.22 seconds with return value 0
Press any key to continue . . .
```

# Manipulators

- Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.
- There are numerous manipulators available in C++. Some of the more commonly used manipulators are:
  - endl
  - setw
- To be able to use manipulators in our program, we must include `<iomanip>` header file in our source program.
- But here is an exception – the `endl` manipulator can be used **without** including the `<iomanip>` file.

# Contd..

- The endl manipulator:

- It has same effect as using the `newline` character `'\n'`.

- but additionally clears the output buffer also.

- **For example:** The statement

```
cout<<"First value="<< first<<endl<< "second value= "<<second;
```

Will cause two lines of output.

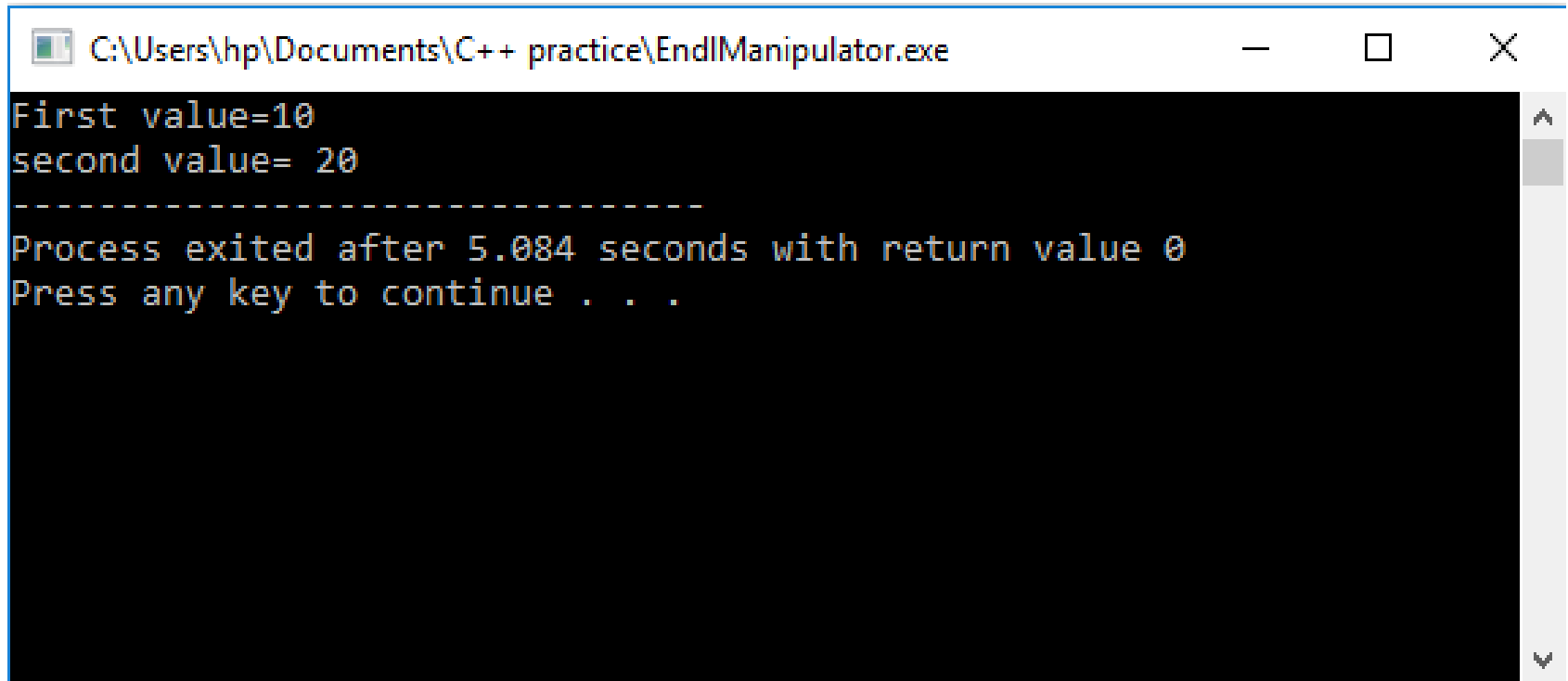
# Contd..

- **Program:**

```
1 // program to illustrate the endl manipulator
2 #include<iostream>
3 using namespace std;
4 int main()
5
6 {
7     int first = 10, second=20;
8     cout<<"First value="<< first<<endl<< "second value= "<<second;
9
10 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Users\hp\Documents\C++ practice\EndlManipulator.exe" and includes standard minimize, maximize, and close buttons. The command prompt area has a black background with white text. The output displayed is: "First value=10", "second value= 20", a dashed line "-----", "Process exited after 5.084 seconds with return value 0", and "Press any key to continue . . .". A vertical scrollbar is visible on the right side of the command prompt window.

```
C:\Users\hp\Documents\C++ practice\EndlManipulator.exe
First value=10
second value= 20
-----
Process exited after 5.084 seconds with return value 0
Press any key to continue . . .
```

# Contd..

- **The `setw` manipulator:**

- This manipulator causes the output stream that follows it to be printed within a field of `n` characters wide, where `n` is the argument to `setw`.
- The output is right justified within the field.
- If we do not use `setw`, the output is left justified by default and it occupies the space in the monitor equal to number of characters in it.

# Control statements

- Not many programs execute all their statements in strict order from beginning to end.
- Most programs decide what to do in response to changing circumstances.
- The flow of control jumps from one part of the program to another, depending on calculations performed in the program.
- Program statements that cause such jumps are called control statements.
- There are three major categories:
  - decisions/selection/branching statements
  - loops/ iterations/repetitions statements and
  - Jumps statements

# Decisions(Selection structures)

- This structures makes one-time decision, causing a one-time jump to a different part of the program, depending on the value of an expression.
- In C++ program decisions can be made in two ways:
  - **By using If statements** : chooses between two alternatives.
  - **By using switch statements**: creates branches for multiple alternative sections of code, depending on the value of a single variable.



# Contd..

- **The if statements:**

- Syntax:

```
if (testExpression)
```

```
{
```

```
    // statements
```

```
}
```

# Contd..

- How if statement works?

Test expression is true

```
int test = 5;  
  
if (test < 10)  
{  
    // codes  
}  
  
// codes after if
```

Test expression is false

```
int test = 5;  
  
if (test > 10)  
{  
    // codes  
}  
  
// codes after if
```

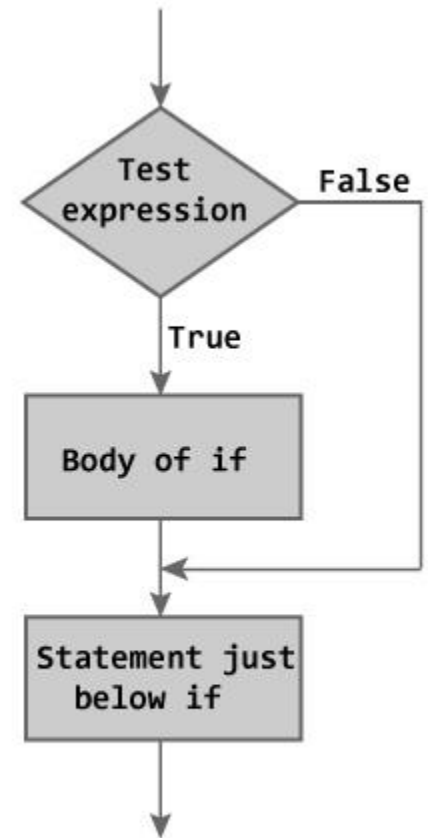


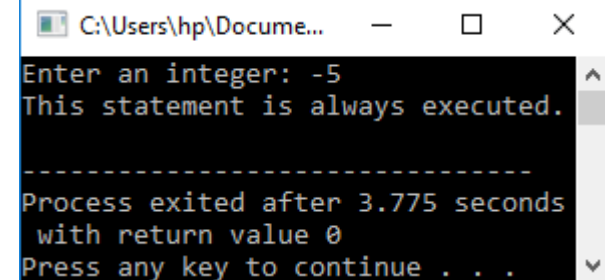
Figure: Flowchart of if Statement

# Contd..

- Example Program:**

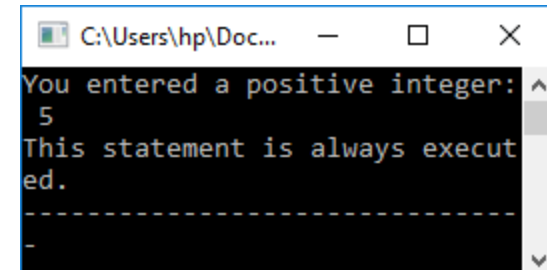
```
1 // Program to print positive number entered by the user
2 // If user enters negative number, it is skipped
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int number;
10    cout << "Enter an integer: ";
11    cin >> number;
12
13    // checks if the number is positive
14    if ( number > 0)
15    {
16        cout << "You entered a positive integer: " << number << endl;
17    }
18
19    cout << "This statement is always executed.";
20    return 0;
21
22 }
```

output1



```
C:\Users\hp\Docume...
Enter an integer: -5
This statement is always executed.
-----
Process exited after 3.775 seconds
with return value 0
Press any key to continue . . .
```

output2



```
C:\Users\hp\Doc...
You entered a positive integer:
5
This statement is always execut
ed.
-----
-
```

# Contd..

- **The if...else statements:**

- **Syntax:**

- If(testExpression)  
  
    {  
  
        //statements  
  
    }  
  
else  
  
    {  
  
        //statements  
  
    }

# Contd..

- How if...else statement works?

Test expression is true

```
int test = 5;

if (test < 10)
{
    // codes
}
else
{
    // codes
}
// codes after if...else
```

Test expression is false

```
int test = 5;

if (test > 10)
{
    // codes
}
else
{
    // codes
}
// codes after if...else
```

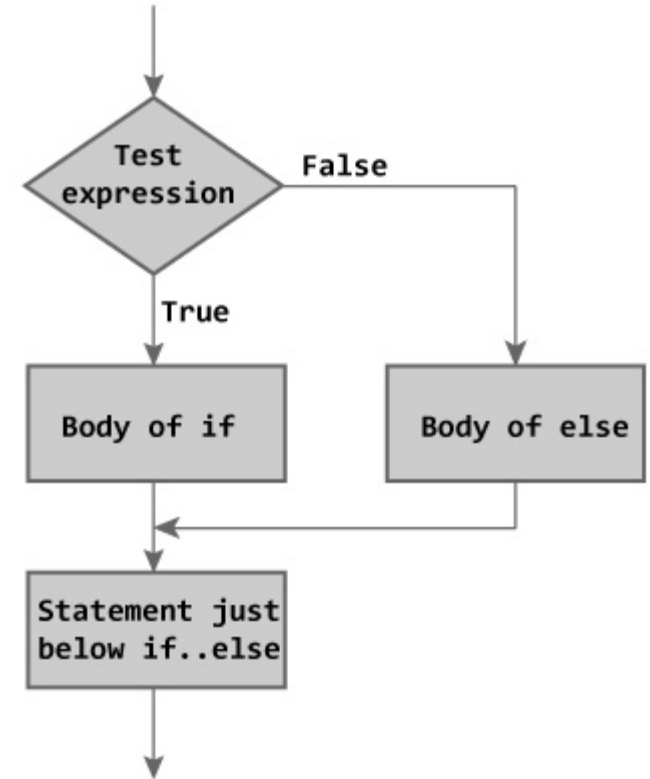
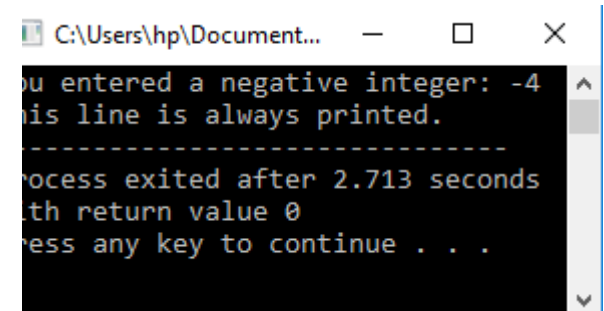


Figure: Flowchart of if...else Statement

# Contd..

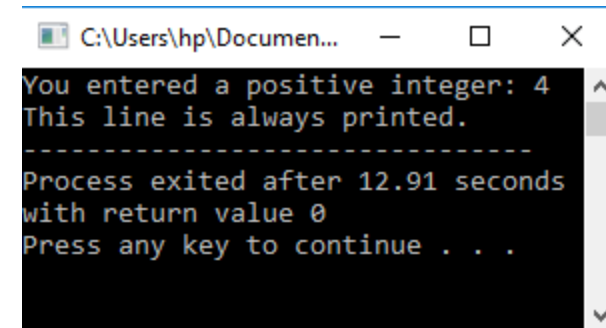
- **Example program:**

```
1 // Program to check whether an integer is positive or negative
2 // This program considers 0 as positive number
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int number;
10    cout << "Enter an integer: ";
11    cin >> number;
12
13    if ( number >= 0)
14    {
15        cout << "You entered a positive integer: " << number << endl;
16    }
17
18    else
19    {
20        cout << "You entered a negative integer: " << number << endl;
21    }
22
23    cout << "This line is always printed.";
24    return 0;
25 }
```



C:\Users\hp\Document... — □ ×

```
You entered a negative integer: -4
This line is always printed.
-----
Process exited after 2.713 seconds
with return value 0
Press any key to continue . . .
```



C:\Users\hp\Documen... — □ ×

```
You entered a positive integer: 4
This line is always printed.
-----
Process exited after 12.91 seconds
with return value 0
Press any key to continue . . .
```

# Contd..

- **The Nested if...else statements:**

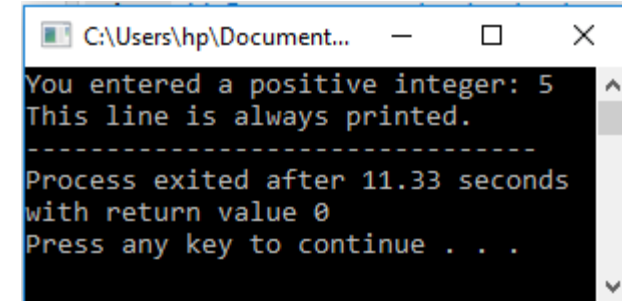
- **Syntax:**

```
if (testExpression1)
{
    // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
    // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression 3)
{
    // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
}
..
else
{
    // statements to be executed if all test expressions are false
}
```

# Contd..

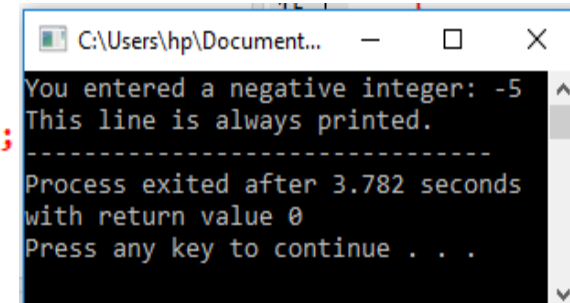
- **Example program:**

```
1 // Program to check whether an integer is positive, negative or zero
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int number;
9     cout << "Enter an integer: ";
10    cin >> number;
11
12    if ( number > 0)
13    {
14        cout << "You entered a positive integer: " << number << endl;
15    }
16    else if (number < 0)
17    {
18        cout<<"You entered a negative integer: " << number << endl;
19    }
20    else
21    {
22        cout << "You entered 0." << endl;
23    }
24
25    cout << "This line is always printed.";
26    return 0;
27 }
```



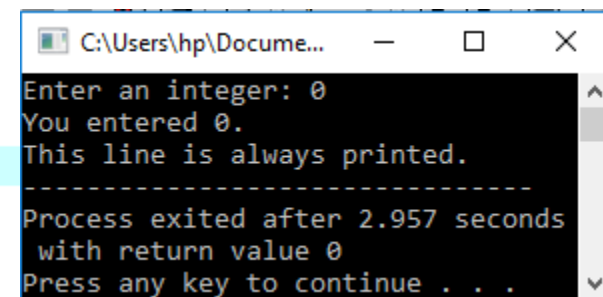
C:\Users\hp\Document... — □ ×

You entered a positive integer: 5  
This line is always printed.  
-----  
Process exited after 11.33 seconds  
with return value 0  
Press any key to continue . . .



C:\Users\hp\Document... — □ ×

You entered a negative integer: -5  
This line is always printed.  
-----  
Process exited after 3.782 seconds  
with return value 0  
Press any key to continue . . .



C:\Users\hp\Docume... — □ ×

Enter an integer: 0  
You entered 0.  
This line is always printed.  
-----  
Process exited after 2.957 seconds  
with return value 0  
Press any key to continue . . .



## Contd..

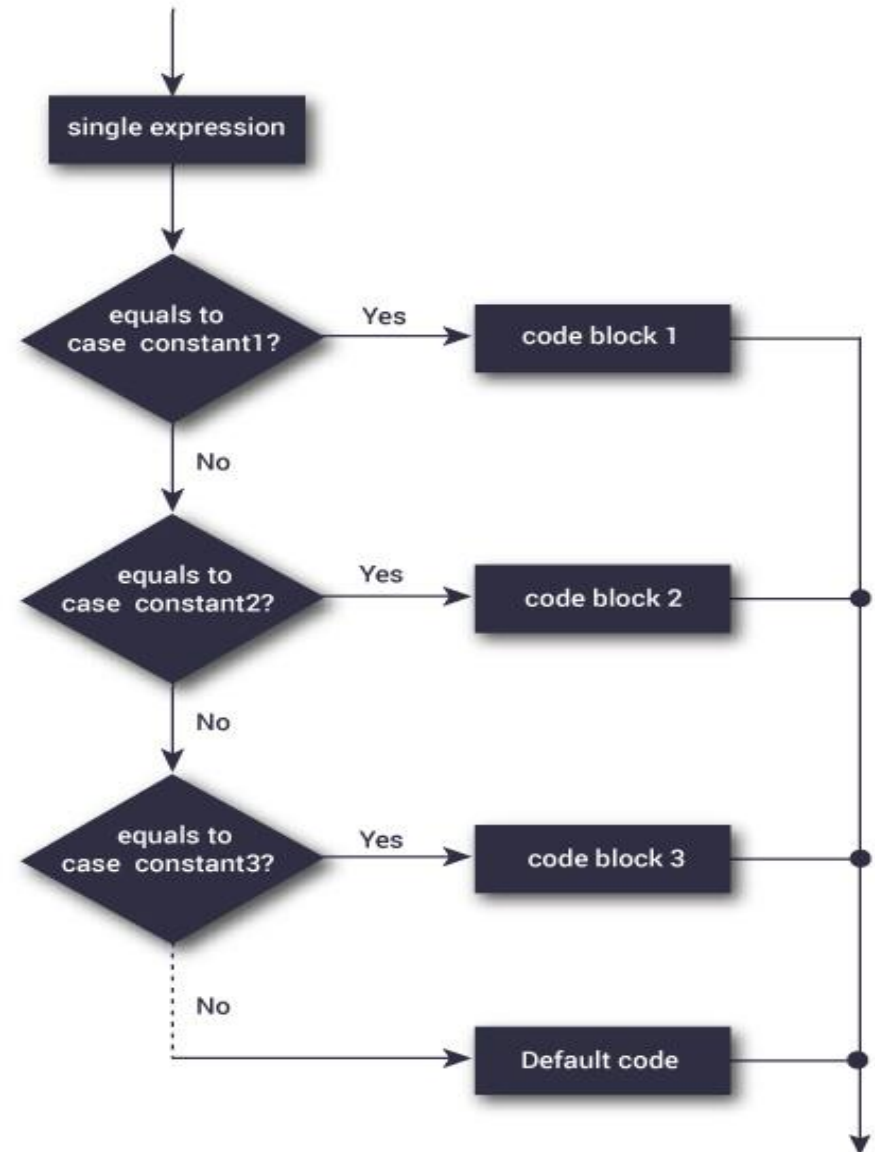
- **The switch statements:** When a case constant is found that matches the switch expression, control of the program passes to the block of code associated with that case.

- **Syntax:**

```
switch (n)
{
    case constant1:
        // code to be executed if n is equal to constant1;
        break;
    case constant2:
        // code to be executed if n is equal to constant2;
        break;
    . . .
    default:
        // code to be executed if n doesn't match any constant
}
```

# Contd..

- How switch statements works?



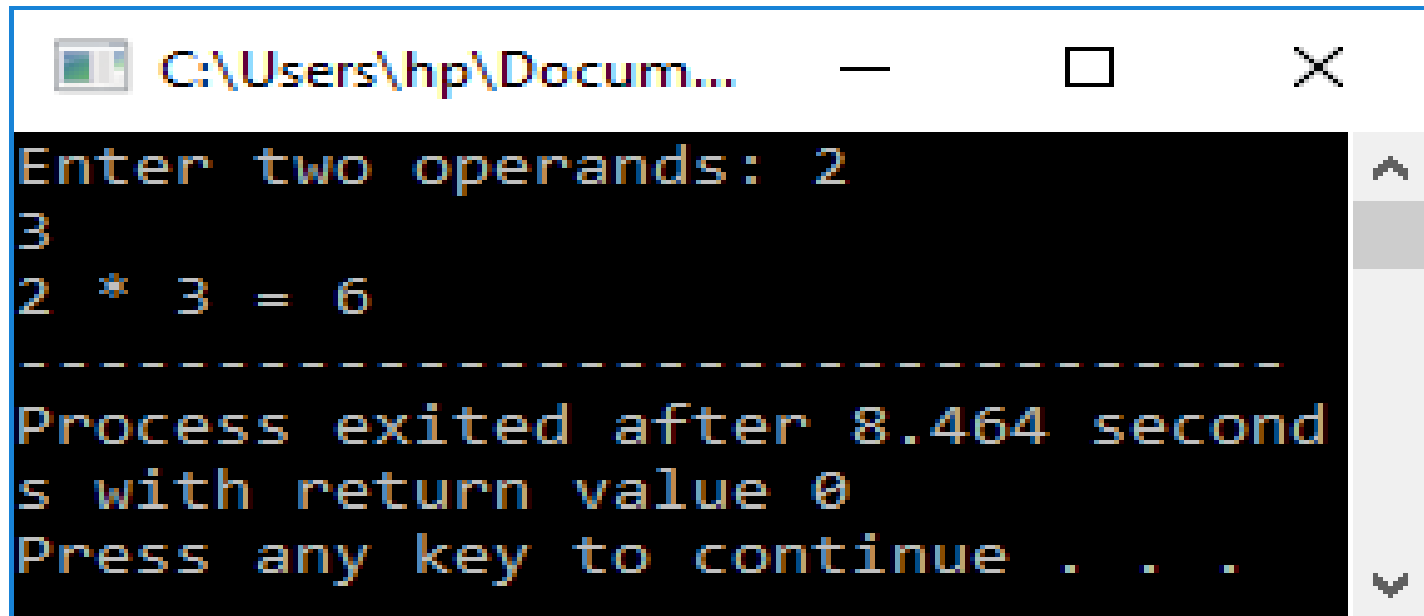
# Contd..

- Example:

```
1 // Program to built a simple calculator using switch Statement
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char o;
9     float num1, num2;
10
11     cout << "Enter an operator (+, -, *, /): ";
12     cin >> o;
13
14     cout << "Enter two operands: ";
15     cin >> num1 >> num2;
16
17     switch (o)
18     {
19         case '+':
20             cout << num1 << " + " << num2 << " = " << num1+num2;
21             break;
22         case '-':
23             cout << num1 << " - " << num2 << " = " << num1-num2;
24             break;
25         case '*':
26             cout << num1 << " * " << num2 << " = " << num1*num2;
27             break;
28         case '/':
29             cout << num1 << " / " << num2 << " = " << num1/num2;
30             break;
31         default:
32             // operator is doesn't match any case constant (+, -, *, /)
33             cout << "Error! operator is not correct";
34             break;
35     }
36
37     return 0;
38 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Users\hp\Docum...". The command prompt has a black background with white text. The text displayed is: "Enter two operands: 2", "3", "2 \* 3 = 6", a dashed line "-----", "Process exited after 8.464 second", "s with return value 0", and "Press any key to continue . . .".

```
Enter two operands: 2
3
2 * 3 = 6
-----
Process exited after 8.464 second
s with return value 0
Press any key to continue . . .
```

# Loops(Iteration statements)

- Loops cause a section of your program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop.
- There are three kinds of loops in C++:
  - For loop
  - While loop and
  - Do...while loop

# Contd..

- **The For loops:**

- **Syntax:**

```
for(initializationStatement;    testExpression;    updateStatement)
{
    // codes
}
```

**Note:** where, only testExpression is mandatory.

# Contd..

- How for loops works?

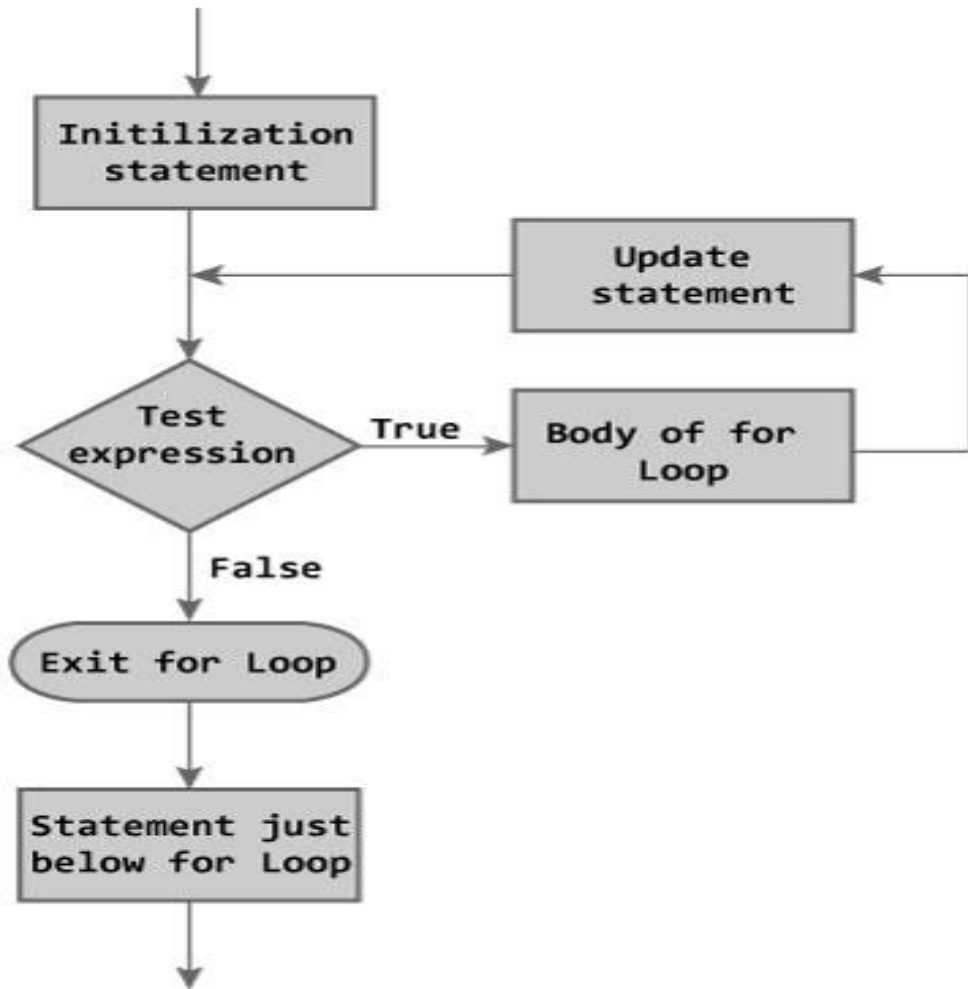
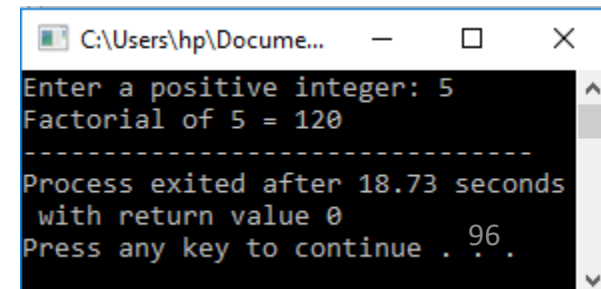


Figure: Flowchart of for Loop

# Contd..

- **Example program:**

```
1 // C++ Program to find factorial of a number
2 // Factorial on  $n = 1*2*3*...*n$ 
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int i, n, factorial = 1;
10
11     cout << "Enter a positive integer: ";
12     cin >> n;
13
14     for (i = 1; i <= n; i++) {
15         factorial *= i; // factorial = factorial * i;
16     }
17
18     cout << "Factorial of " << n << " = " << factorial;
19     return 0;
20 }
```



```
C:\Users\hp\Docume...
Enter a positive integer: 5
Factorial of 5 = 120
-----
Process exited after 18.73 seconds
with return value 0
Press any key to continue . 96.
```



# Contd..

- **The while loop:**

- **Syntax:**

```
while (testExpression)
{
    // codes
}
```

where, testExpression is checked on each entry of the while loop.

# Contd..

- **How while loop works?**

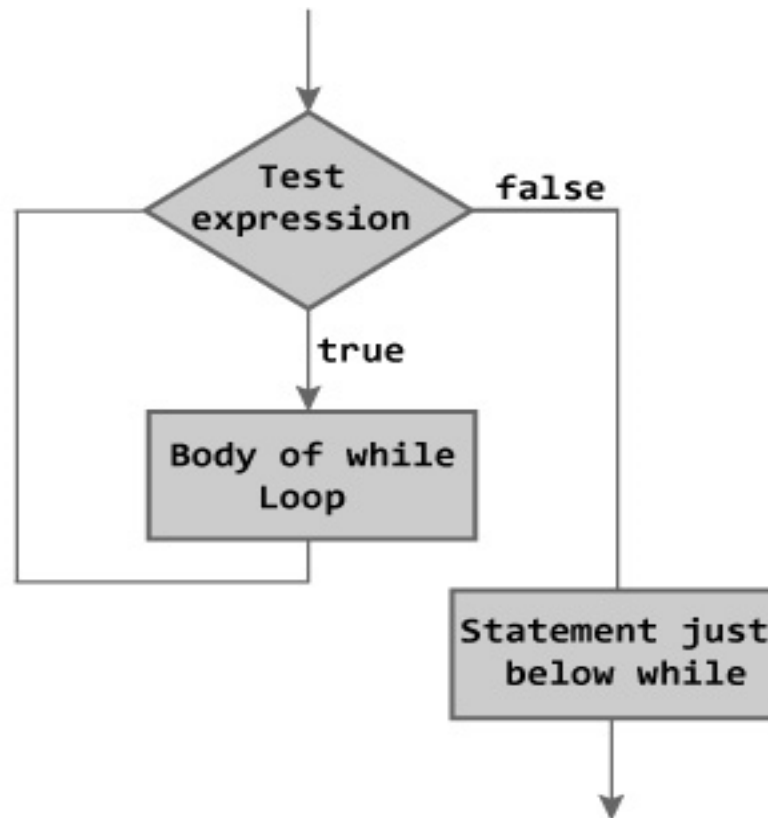
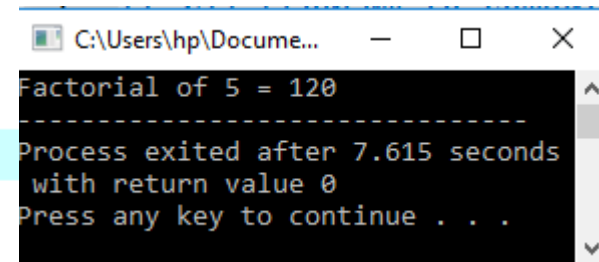


Figure: Flowchart of while Loop

# Contd..

- **Example program:**

```
1 // C++ Program to compute factorial of a number
2 // Factorial of n = 1*2*3...*n
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     int n, i = 1, factorial = 1;
10
11     cout << "Enter a positive integer: ";
12     cin >> n;
13
14     while ( i <= n) {
15         factorial *= i;    //factorial = factorial * i;
16         i++;
17     }
18
19     cout<<"Factorial of "<< n | <<" = "<< factorial;
20     return 0;
21 }
```



C:\Users\hp\Docume... — □ ×

```
Factorial of 5 = 120
-----
Process exited after 7.615 seconds
with return value 0
Press any key to continue . . .
```

# Contd..

- **The do...while loop:**

- The do...while loop is a variant of the while loop with one important difference. The body of do...while loop is executed once before the test expression is checked.
- The syntax of do..while loop is:

```
do
```

```
{
```

```
    // codes;
```

```
} while (testExpression);
```

# Contd..

- **How do...while loop works?**

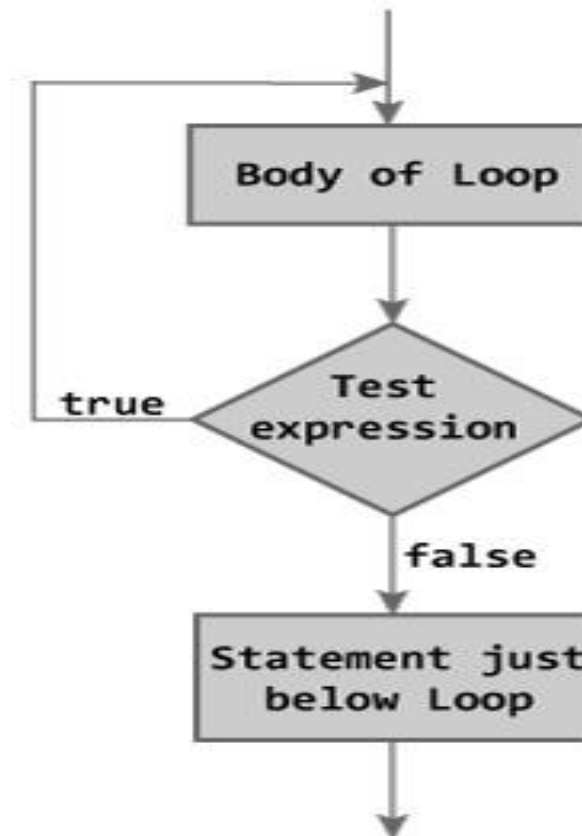


Figure: Flowchart of do...while Loop

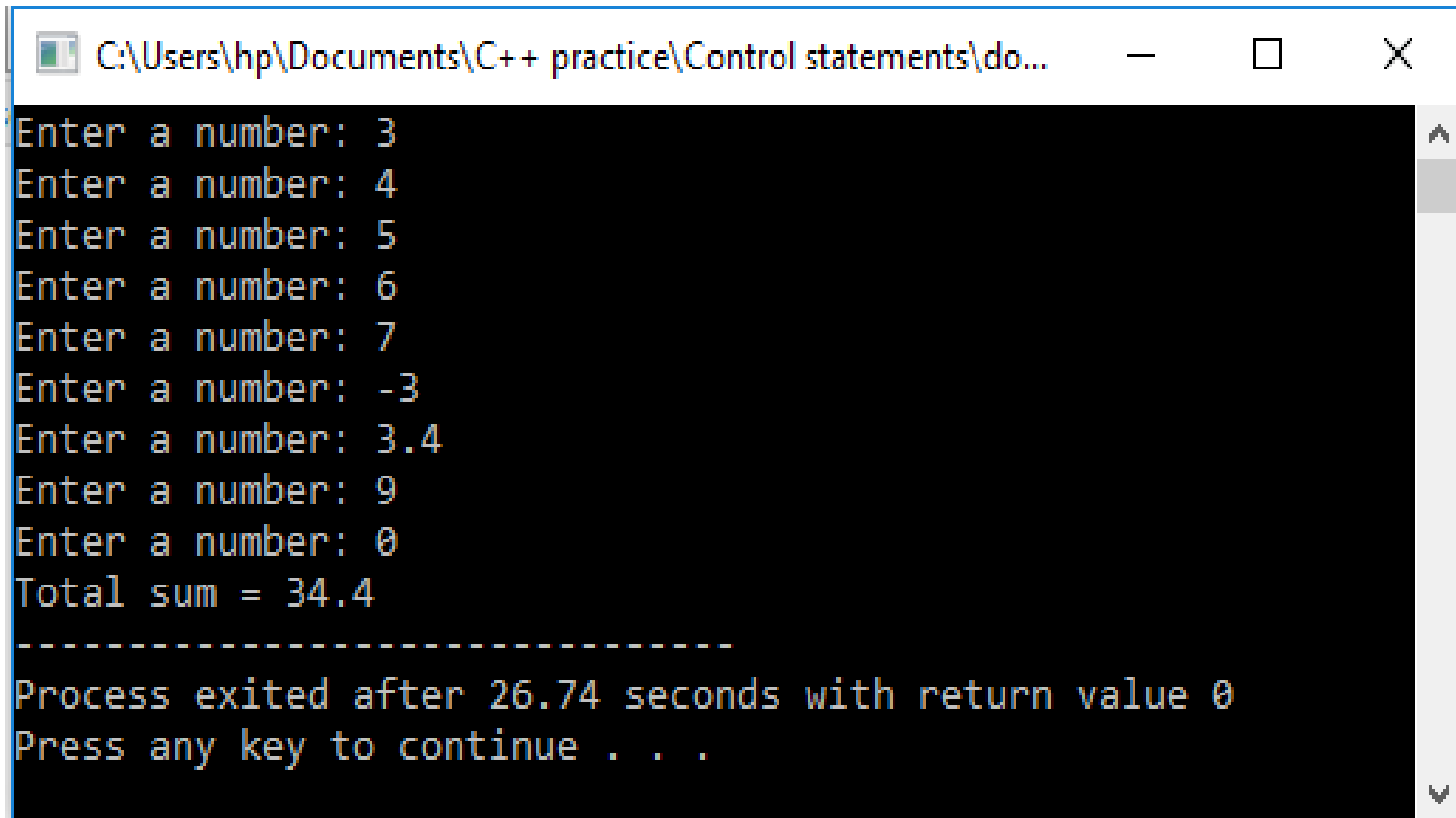
# Contd..

- **Example program:**

```
1  // C++ program to add numbers until user enters 0
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      float number, sum = 0.0;
9
10     do {
11         cout<<"Enter a number: ";
12         cin>>number;
13         sum += number;
14     }
15     while(number != 0.0);
16
17     cout<<"Total sum = "<<sum;
18
19     return 0;
20 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ practice\Control statements\do...  
Enter a number: 3  
Enter a number: 4  
Enter a number: 5  
Enter a number: 6  
Enter a number: 7  
Enter a number: -3  
Enter a number: 3.4  
Enter a number: 9  
Enter a number: 0  
Total sum = 34.4  
-----  
Process exited after 26.74 seconds with return value 0  
Press any key to continue . . .
```

# Jump statements

- Jump statements are used to transfer control of a program execution from one part of the program to another.
- C++ supports the following jump statements:
  - The break statement
  - The continue statement
  - The goto statement
  - The return statement



# Contd..

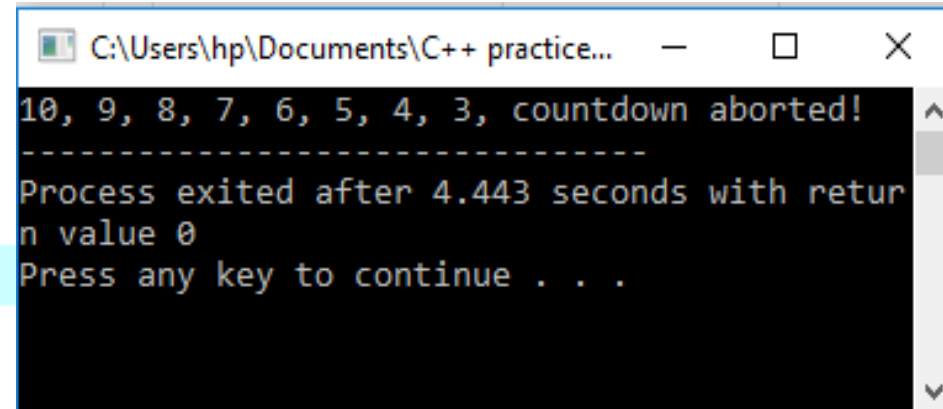
- **The break statement:**

- The use of break statement causes the immediate termination of the switch statement and the loop from the point of break statement.
- The control then passes to the statements following the switch statement and the loop.

# Contd..

- **E.g.,:**

```
1 // break loop example
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7     for (int n=10; n>0; n--)
8     {
9         cout << n << ", ";
10        if (n==3)
11        {
12            cout << "countdown aborted!";
13            break;
14        }
15    }
16 }
```



C:\Users\hp\Documents\C++ practice...

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
-----
Process exited after 4.443 seconds with return
value 0
Press any key to continue . . .
```

# Contd..

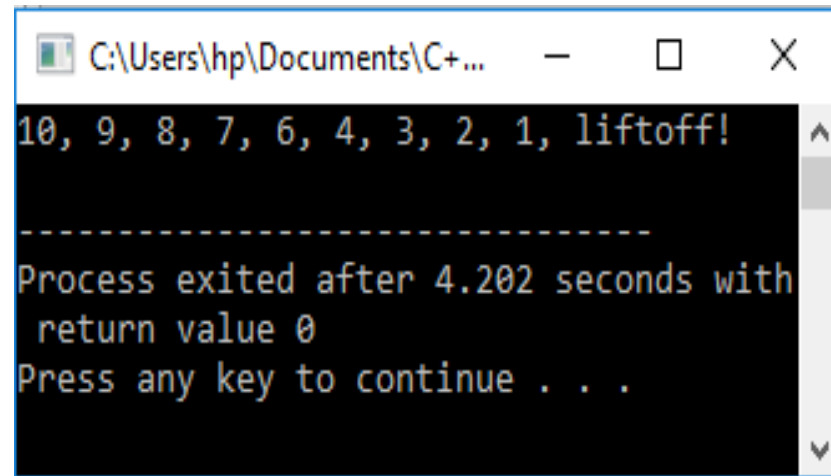
- **The Continue statements:**

- The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.

# Contd..

- **E.g.,**

```
1 // continue loop example skip number 5 in our countdown:
2
3 #include <iostream>
4
5 using namespace std;
6
7 int main ()
8 {
9     for (int n=10; n>0; n--)
10     {
11         if (n==5) continue;
12         cout << n << ", ";
13     }
14     cout << "liftoff!\n";
15 }
```



```
C:\Users\hp\Documents\C+...  -  □  ✕
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
-----
Process exited after 4.202 seconds with
return value 0
Press any key to continue . . .
```

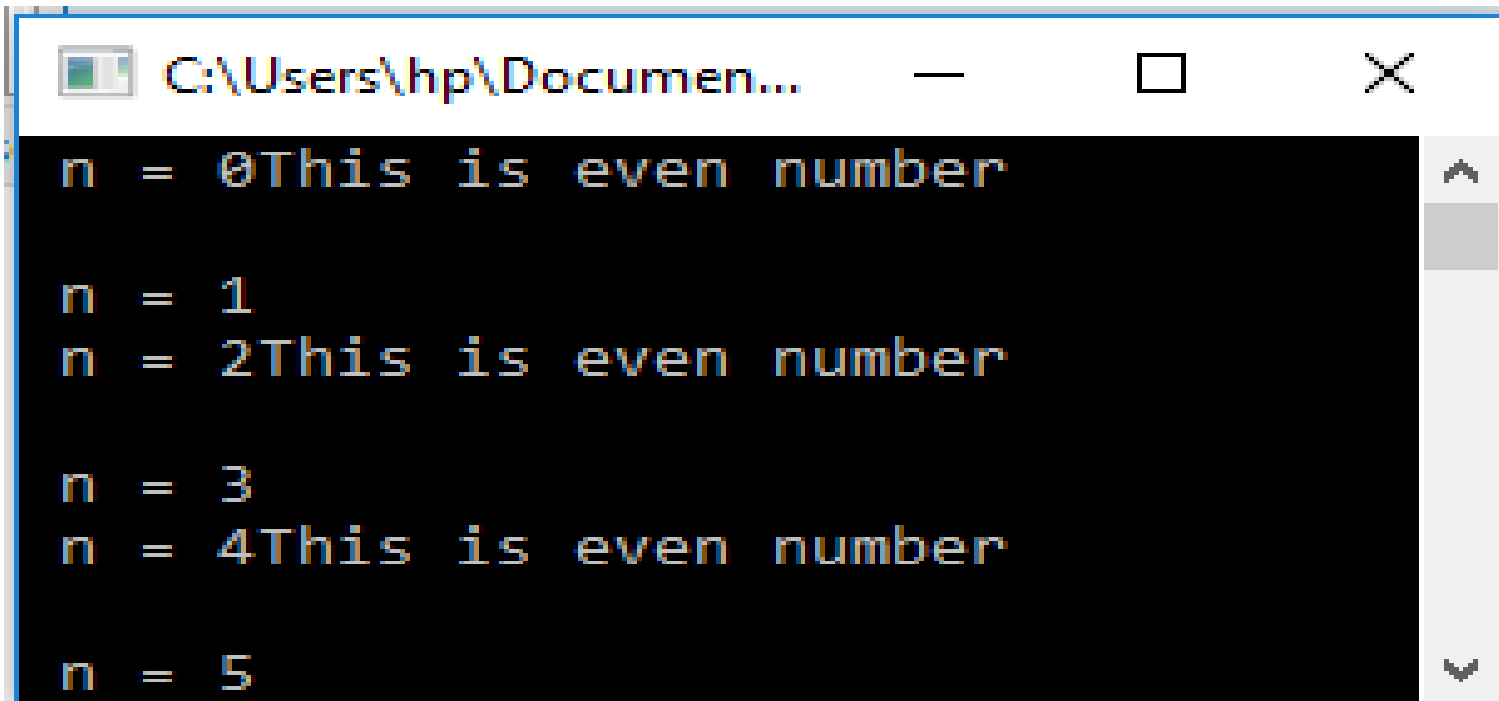
# Contd..

- **Example2:**

```
1  //continue statement example2
2  #include<iostream>
3
4  using namespace std;
5  int main()
6  {
7      for(int n =0; n<6; n++)
8      {
9          cout<<"\n| n = "<<n ;
10         if(n%2==1) continue;
11         {
12             cout<<"This is even number \n";
13         }
14     }
15
16     return 0;
17
18 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documen...  
n = 0This is even number  
  
n = 1  
n = 2This is even number  
  
n = 3  
n = 4This is even number  
  
n = 5
```

# Contd..


- **The goto statement:**

- It allows making an absolute jump to another point in the program.
- The destination point is identified by a label, which is then used as an argument for the goto instruction.
- The label is made of a valid identifier followed by a colon(:).

# Contd..

- **Example:**

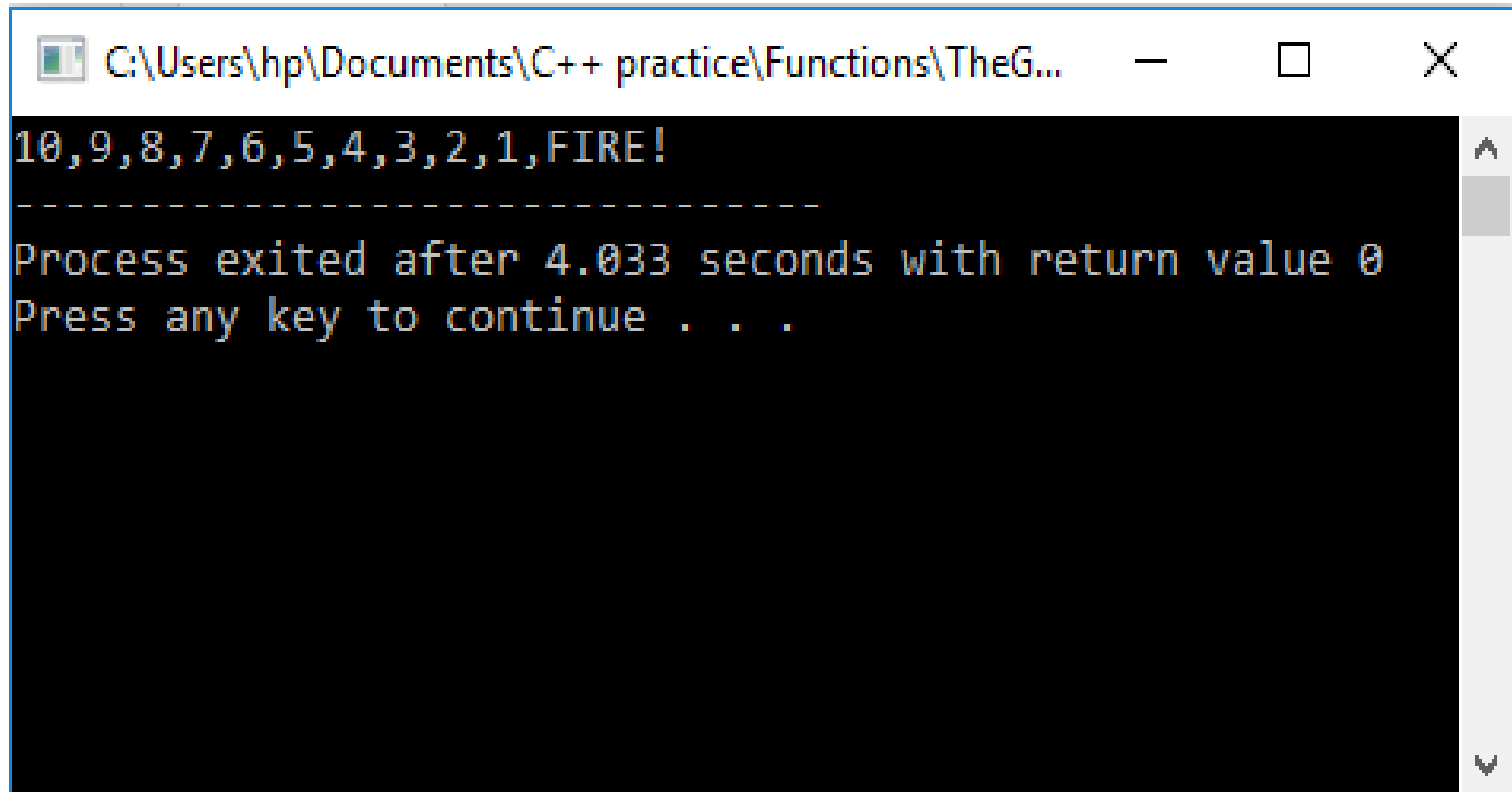
```
1  // The goto statement
2  #include<iostream>
3  using namespace std;
4  int main()
5  {
6      int n =10;
7      loop:
8      cout<<n<<', ';<
9      n--;
10     if(n>0)
11         goto loop;
12     cout<<"FIRE!";
13     return 0;
14 }
```





# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\hp\Documents\C++ practice\Functions\TheG... The window has standard minimize, maximize, and close buttons. The command prompt area has a black background with white text. The output displayed is: 10,9,8,7,6,5,4,3,2,1,FIRE! followed by a dashed line, then 'Process exited after 4.033 seconds with return value 0', and finally 'Press any key to continue . . .'. A vertical scrollbar is visible on the right side of the text area.

```
C:\Users\hp\Documents\C++ practice\Functions\TheG...  
10,9,8,7,6,5,4,3,2,1,FIRE!  
-----  
Process exited after 4.033 seconds with return value 0  
Press any key to continue . . .
```

## Contd..

- **The return statement:**
  - It is used to transfer the program control back to the caller of the method.

# Functions

- In programming, function refers to a segment that groups code to perform a specific task.
- Depending on whether a function is predefined or created by programmer; there are two types of function:
  - Library Function
  - User-defined Function

# User-defined Function

- C++ allows programmer to define their own function.
- A user-defined function groups code to perform a specific task and that group of code is given a name(identifier).
- When the function is invoked from any part of program, it all executes the codes defined in the body of function.

# Contd..

- **How C++ functions works?**

```
#include <iostream>

void function_name() {
    ... ..
    ... ..
}

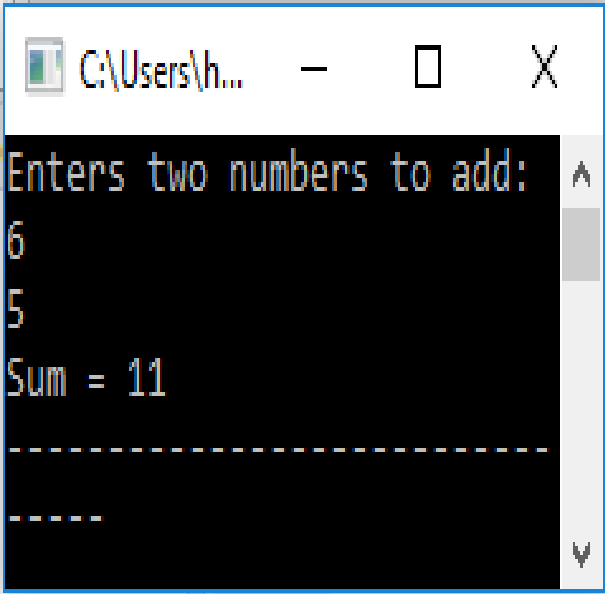
int main() {
    ... ..
    function_name();
    ... ..
}
```

The diagram illustrates the execution flow of a C++ program. It shows two function definitions: `function_name()` and `main()`. An arrow originates from the `function_name();` line within the `main()` function and points to the opening curly brace of the `function_name()` function definition. Another arrow originates from the closing curly brace of the `function_name()` function definition and points back to the line immediately following the function call in the `main()` function, indicating the return of control to the caller.

# Contd..

- For Example:

```
1 //C++ program to add two integers.
2 //Make a function add() to add integers and
3 // display sum in main() function.
4 #include <iostream>
5 using namespace std;
6 // Function prototype (declaration)
7 int add(int, int);
8 int main()
9 {
10     int num1, num2, sum;
11     cout<<"Enters two numbers to add: ";
12     cin >> num1 >> num2;
13
14     // Function call
15     sum = add(num1, num2);
16     cout << "Sum = " << sum;
17     return 0;
18 }
19 // Function definition
20 int add(int a, int b)
21 {
22     int add;
23     add = a + b;
24     // Return statement
25     return add;
26 }
```



```
C:\Users\h...
Enters two numbers to add:
6
5
Sum = 11
-----
-----
```

# Default Argument

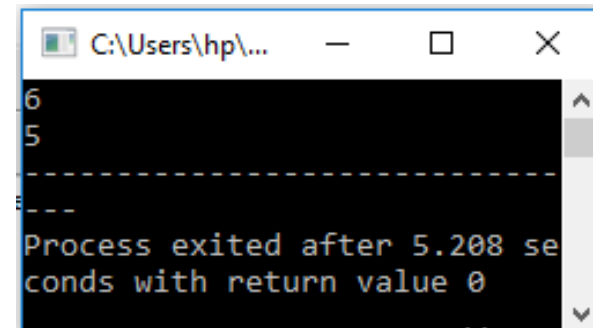
- When declaring a function we can specify a default value for each of the last parameters which are called default arguments.
- If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

# Contd..

- **For Example:**

```
1 // program to illustrate the concept of default arguments
2 #include<iostream>
3 using namespace std;
4 int divide(int a, int b= 2 )
5 {
6     int result;
7     result = a/b;
8     return(result);
9 }
10 int main()
11 {
12     cout<< divide(12)<< endl; // uses the default value of b
13
14     cout<<divide(20,4); // ignores the default value of b
15     return 0;
16
17 }
```

Output:



```
C:\Users\hp\...
6
5
-----
Process exited after 5.208 seconds with return value 0
```



# Inline Function

- We make a function if the same operation is to be repeated at different location of the program.
- Functions save the memory space as all the calls to the function cause the same code to be executed.
- But, calling a function increases the execution time overhead.
- One possible way to reduce such overhead by copying a code at the point of call.
- Such copying of a code can be achieved by making a function inline.

# Contd..

- If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- The compiler can ignore the inline qualifier in case defined function is more complex(e.g., contains loop).
- To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function.
- **Syntax of Inline Function**

```
inline return_type function_name (argument list)
```

```
{
```

```
    // body of function
```

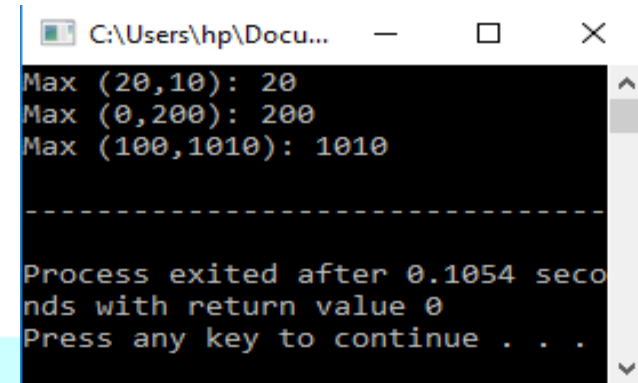
```
}
```

# Contd..

- For example 1:

```
1 // program to illustrate the concept of the inline function
2 // inline function returns maximum of two number
3
4 #include <iostream>
5
6 using namespace std;
7
8 //inline function
9 inline int Max(int x, int y)
10 {
11     return (x > y)? x : y;
12 }
13
14 // Main function for the program
15 int main() {
16     cout << "Max (20,10): " << Max(20,10) << endl;
17     cout << "Max (0,200): " << Max(0,200) << endl;
18     cout << "Max (100,1010): " << Max(100,1010) << endl;
19
20     return 0;
21 }
```

Output:

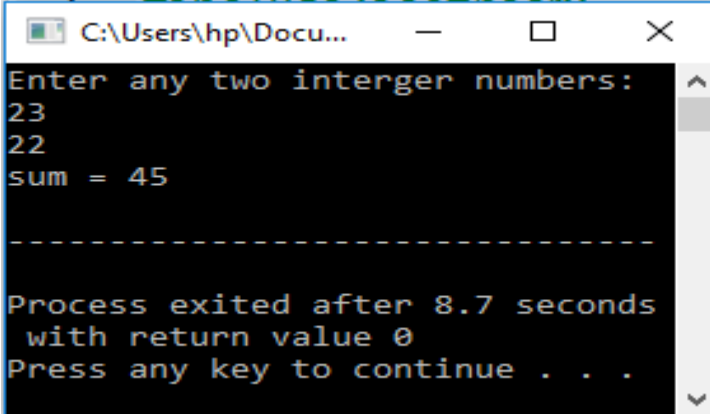


```
C:\Users\hp\Docu...
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
-----
Process exited after 0.1054 seconds with return value 0
Press any key to continue . . .
```

# Contd..

- **Example 2: Inline function**

```
1 // inline function to add two number
2 #include<iostream>
3 using namespace std;
4 inline void sum( int a, int b)
5 {
6     int s;
7     s= a+b;
8     cout<<"sum = "<<s<<endl;
9 }
10 int main()
11 {
12     int x, y;
13     cout<<"Enter any two interger numbers:"<<endl;
14     cin>>x>>y;
15     sum(x,y);
16     return 0;
17 }
```



```
C:\Users\hp\Docu...
Enter any two interger numbers:
23
22
sum = 45

-----

Process exited after 8.7 seconds
with return value 0
Press any key to continue . . .
```

## Contd..

- **Advantages of inline function:**

- No function call overhead, therefore execution of the program becomes faster than using normal functions.
- Better error checking is performed as compared to macros.
- Can access members of a class which can not be done with macros.

# Contd..

- **Disadvantages of inline function:**

- If the function call is made repeatedly, object code size increases and thus requires more memory at the time of execution.
- The compiler can not perform inlining if the function is too complicated.

# Function overloading

- Function Overloading is a mechanism that allows a single function name to be used for different functions in a program.
- Two or more functions having same name but different arguments (i.e., different signature) are known as overloaded functions.
  - E.g.,:
    - `display(int)`
    - `display(float)`

# Contd..

- Functions can be overloaded in two ways:
  - Function overloading with different types of arguments
  - Function overloading with different number of arguments



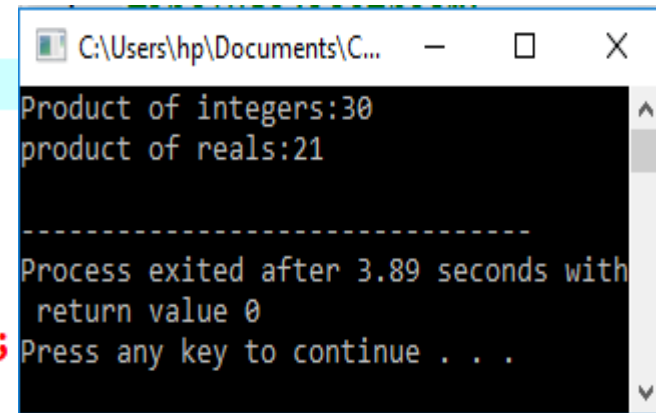
# Contd..

- **Function overloading with different types of arguments:**
  - When we need to have same operation on different types of variables, we overload the functions with different type of arguments.
  - The following example illustrates this concept:

# Contd..

- For Example:

```
1 //function overloading with differnt type of arguments
2 #include<iostream>
3 using namespace std;
4 // overloaded functions
5 int mul(int a, int b)
6 {
7     return(a*b);
8 }
9 float mul(float a, float b)
10 {
11     return(a*b);
12 }
13 // main function
14 int main()
15 {
16     int x = 5, y= 6;
17     float m = 7.0, n = 3.0;
18     cout<<"Product of integers:"<< mul(x,y)<<endl;
19     cout<<"product of reals:"<<mul(m,n)<<endl;
20     return 0;
21 }
```



```
C:\Users\hp\Documents\C...
Product of integers:30
product of reals:21

-----
Process exited after 3.89 seconds with
return value 0
Press any key to continue . . .
```

# Contd..

- **Function overloading with different number of arguments**
  - As in type of argument, overloading can be done with the number of arguments.
  - As the name is same the compiler differentiates the function with the number of arguments.
  - Following example illustrates this concept:

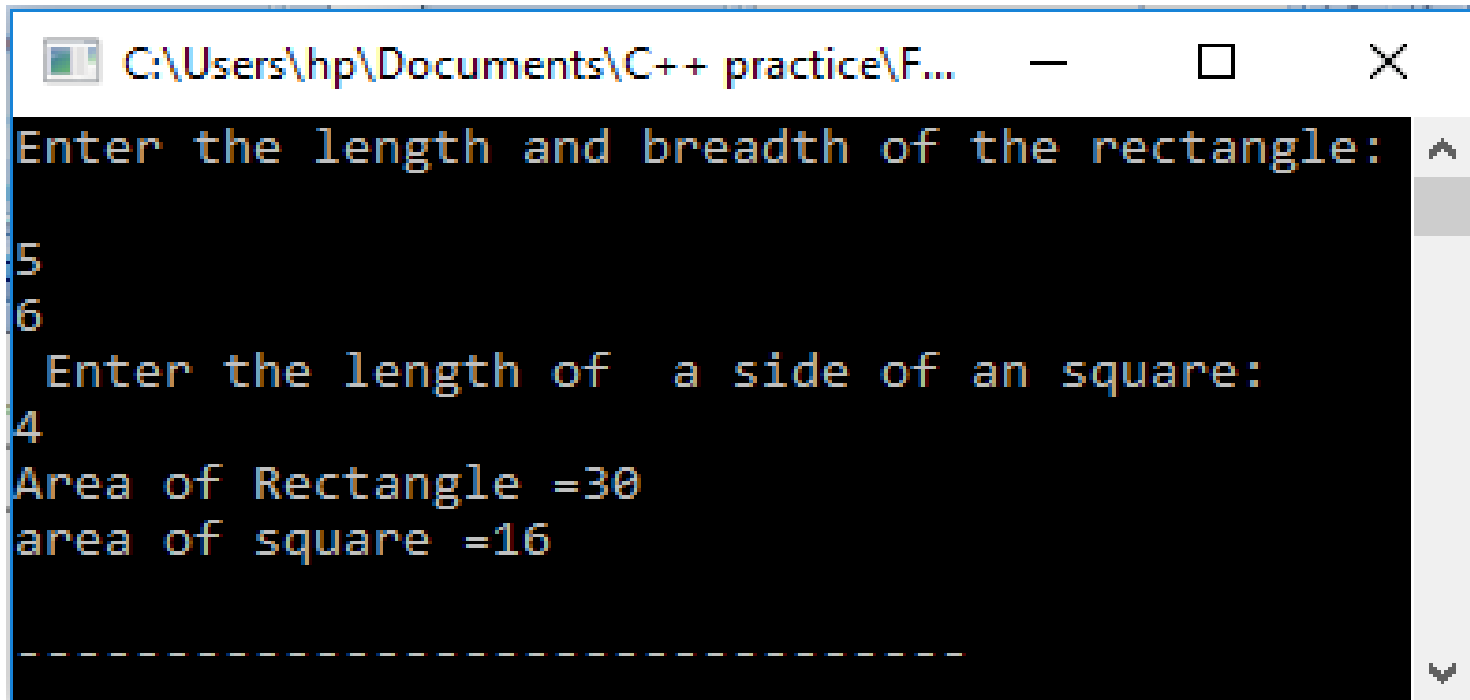
# Contd..

- For example:

```
1  //Function overloading with different number of arguments
2  #include<iostream>
3  using namespace std;
4  int area(int l, int b)
5  {
6      return(l*b);
7  }
8  int area(int l)
9  {
10     return(l*l);
11 }
12 int main()
13 {
14     int sArea, rArea, l, b;
15     cout<<"Enter the length and breadth of the rectangle:"<<endl;
16     cin>>l>>b;
17     rArea = area(l, b);
18     cout<<" Enter the length of a side of an square:"<<endl;
19     cin>>l;
20     sArea = area(l);
21     cout<<"Area of Rectangle ="<<rArea<<endl;
22     cout<<"area of square ="<<sArea<<endl;
23     return 0;
24 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ practice\F...
Enter the length and breadth of the rectangle:
5
6
Enter the length of a side of an square:
4
Area of Rectangle =30
area of square =16
-----
```

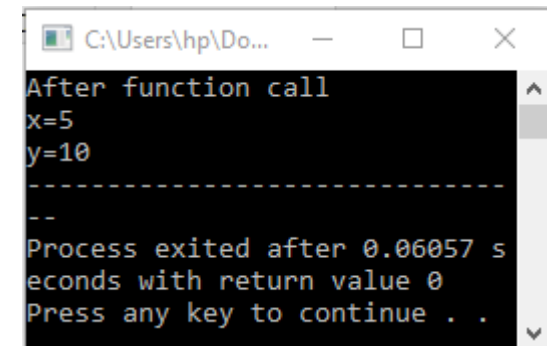
# Passing arguments to the function

- Three ways to pass arguments to the function (Types of function call):
  - Pass by value
  - Pass by reference
  - Pass by pointer

# Contd..

- **Pass by value:** Copies of the arguments are passed of the function not the variables themselves.
- **Example:**

```
1 //pass by value
2 #include<iostream>
3 using namespace std;
4 int exchange(int a, int b)
5 {
6     int temp;
7     temp = a;
8     a = b;
9     b = temp;
10
11 }
12 int main()
13 {
14     int x = 5, y = 10;
15     exchange(x,y);
16     cout<<"After function call"<<endl;
17     cout<<"x="<<x<<endl<<"y="<<y;
18     return 0;
19 }
```

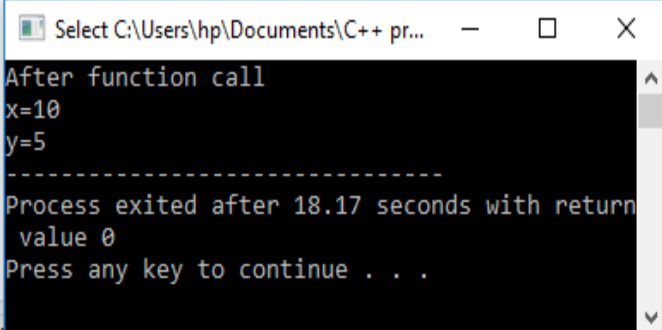


```
C:\Users\hp\Do...
After function call
x=5
y=10
-----
--
Process exited after 0.06057 s
econds with return value 0
Press any key to continue . .
```

## Contd..

- **Pass By reference:** In case of pass by reference, address of the variable(i.e., variable itself) not copies of the arguments are passed to the function.
- **Example:**

```
1 //pass by reference
2 #include<iostream>
3 using namespace std;
4 int exchange(int &a, int &b)
5 {
6     int temp;
7     temp = a;
8     a = b;
9     b = temp;
10
11 }
12 int main()
13 {
14     int x = 5, y = 10;
15     exchange(x,y);
16     cout<<"After function call"<<endl;
17     cout<<"x="<<x<<endl<<"y="<<y;
18     return 0;
19 }
```



Select C:\Users\hp\Documents\C++ pr... - □ ×

After function call  
x=10  
y=5  
-----  
Process exited after 18.17 seconds with return value 0  
Press any key to continue . . .



## Contd..

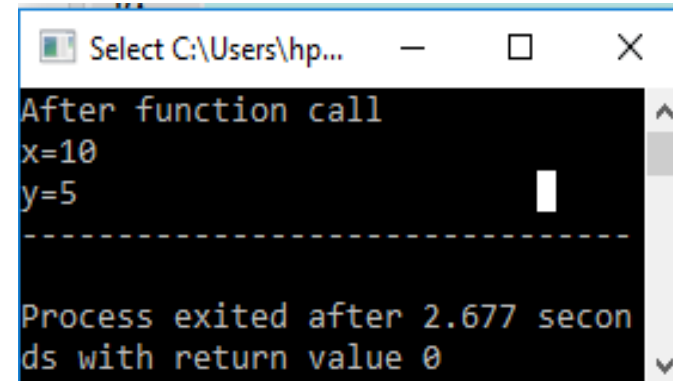
- **Pass by pointer:**

- Working principle of pass by pointer is same as the pass by reference.
- Only the difference is that instead of using the reference variable we use the pointer variable in function definition and pass address of the variable from the function call statement.

# Contd..

- **Example:**

```
1  // pass by pointer
2  #include<iostream>
3  using namespace std;
4  void exchange(int *a, int *b)
5  {
6      int temp;
7      temp = *a;
8      *a = *b;
9      *b = temp;
10
11 }
12 int main()
13 {
14     int x = 5, y = 10;
15     exchange(&x, &y);
16     cout<<"After function call"<<endl;
17     cout<<"x="<<x<<endl<<"y="<<y;
18     return 0;
19 }
```



```
Select C:\Users\hp...
After function call
x=10
y=5
-----
Process exited after 2.677 seconds with return value 0
```

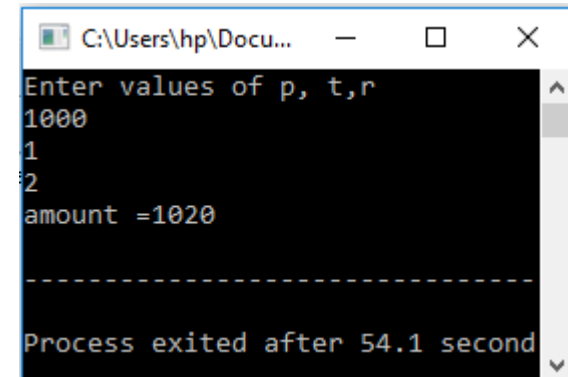
# Returning form Function

- Three way to return value form one function to another function.
  - Return by value
  - Return by reference
  - Return by pointer

# Contd..

- Return By value:** When a value is returned a copy of that value is returned to the caller from the called function.

```
1 // Return by value
2 #include<iostream>
3 using namespace std;
4 float calculate_amount(float p, float t, float r)
5 {
6     float si;
7     si = p*t*r/100;
8     return(p+si);
9 }
10 int main()
11 {
12     float p, t, r, a;
13     cout<< "Enter values of p, t,r"<< endl;
14     cin>>p>>t>>r;
15     a = calculate_amount(p,t,r);
16     cout<<"amount ="<<a<<endl;
17     return 0;
18 }
```



A screenshot of a Windows command prompt window titled "C:\Users\hp\Docu...". The window shows the execution of a C++ program. The prompt "Enter values of p, t,r" is followed by the user input "1000", "1", and "2" on separate lines. The program then outputs "amount =1020". At the bottom, it says "Process exited after 54.1 second".

```
C:\Users\hp\Docu...
Enter values of p, t,r
1000
1
2
amount =1020
-----
Process exited after 54.1 second
```

# Contd..

- **Return by reference:** when a variable is returned by reference, a reference to the variable is passed back to the caller.
- **Example:**

```
1 // Return by reference
2 #include<iostream>
3 using namespace std;
4 int &min(int &x, int &y)
5 {
6     if (x<y)
7         return x;
8     else
9         return y;
10 }
11 int main()
12 {
13     int a, b;
14     cout<<"Enter two numbers:"<<endl;
15     cin>>a>>b;
16     min(a,b) = 0; // smaller between a and b is determined and set to 0
17     cout<<"a ="<<a<<endl<<"b ="<<b;
18 }
```

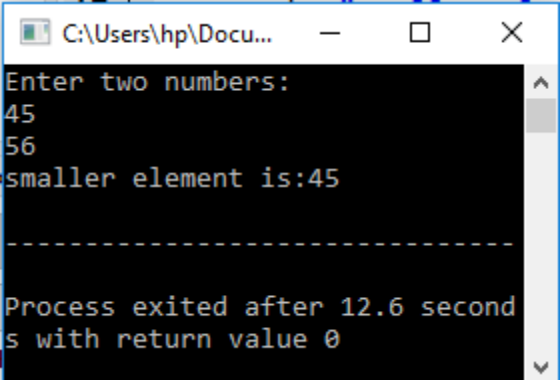
```
Enter two numbers:
9
7
a =9
b =0
```

```
Enter two numbers:
2
3
a =0
b =3
```

# Contd..

- **Return By pointer:** Returns the address of the variable.
- **Example:**

```
1  // Return by reference
2  #include<iostream>
3  using namespace std;
4  int *min(int *x, int *y)
5  {
6      if(*x < *y)
7          return x;
8      else
9          return y;
10 }
11 int main()
12 {
13     int a, b;
14     cout<<"Enter two numbers:"<<endl;
15     cin>>a>>b;
16     int *m = min(&a, &b);
17     cout<<"smaller element is:"<<*m<<endl;
18     return 0;
19 }
```



```
C:\Users\hp\Docu...
Enter two numbers:
45
56
smaller element is:45
-----
Process exited after 12.6 second
s with return value 0
```

# Scope/Visibility and Storage class

- **Scope (Visibility):**

- scope of a variable determines which part of the program can access it.
- There are three types of variable scope:
  - Local scope
  - Global scope(file scope)
  - Class scope

## Contd..

- **Local Scope:** A variable declared within a block of code enclosed by braces({ }) is accessible only within that block, and only after the point of declaration. Outside that they are unavailable.
- **Global scope:** Any variable declared outside all blocks has global scope. It is accessible anywhere in the file after its declaration.
- **Class scope:** In general members of classes have class scope.(we will discuss in detail later).



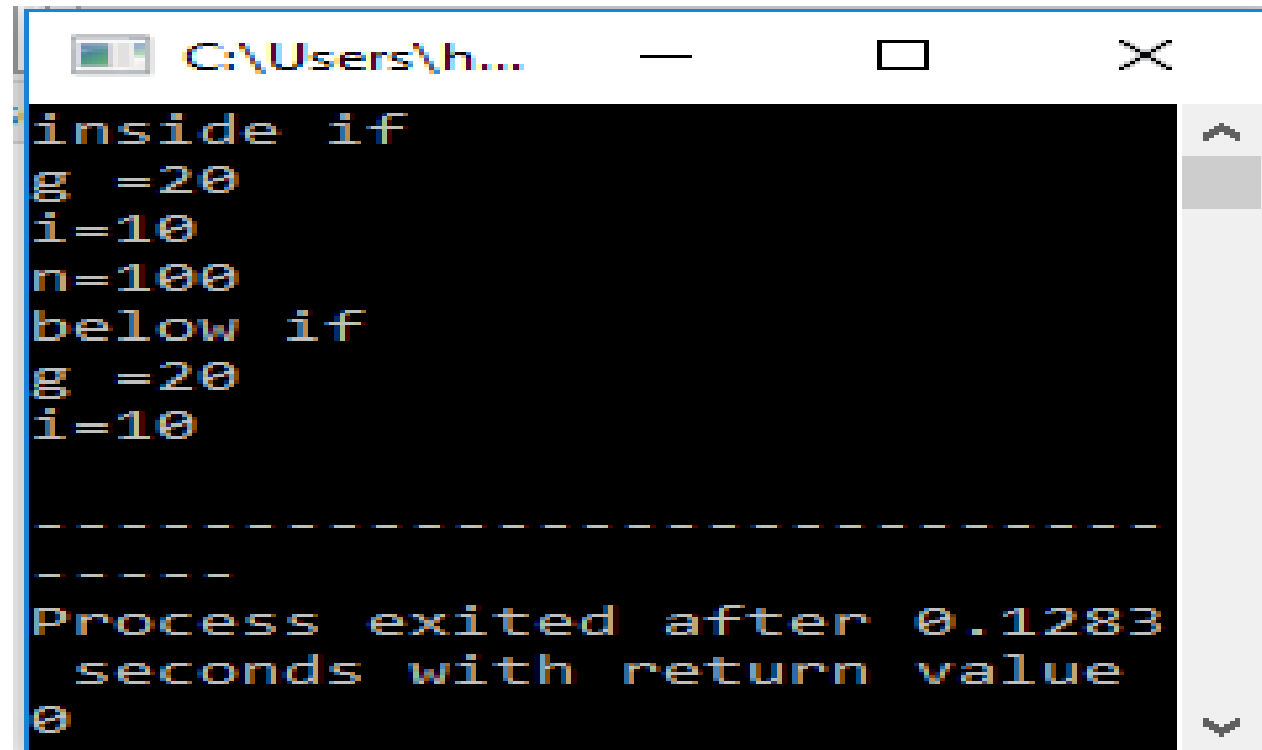
# Contd..

- **Example:**

```
1 // Program to illustrate the scope of variable
2 #include<iostream>
3 using namespace std;
4 int g = 20; // global variable
5 int main()
6 {
7     int i = 10; // local variable
8     if(i<20)
9     { // if condition scope starts
10         int n = 100; // local variable
11         cout<<"inside if"<< endl;
12         cout<<"g ="<<g<<endl;
13         cout<<"i ="<<i<<endl;
14         cout<<"n ="<<n<<endl;
15     } // if condition scope ends
16
17     cout<< "below if"<< endl;
18     cout<<"g ="<<g<<endl;
19     cout<<"i ="<<i<<endl;
20     //cout<<"n ="<<n<<endl; -- error can not be accessed here
21
22 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\h...' and standard window controls. The command prompt displays the following text: 'inside if', 'g =20', 'i=10', 'n=100', 'below if', 'g =20', 'i=10', followed by a dashed line '-----', another dashed line '-----', and finally 'Process exited after 0.1283 seconds with return value 0'.

```
C:\Users\h...  
inside if  
g =20  
i=10  
n=100  
below if  
g =20  
i=10  
-----  
-----  
Process exited after 0.1283  
seconds with return value  
0
```

# Contd..

- **Storage Class:**

- Every variable has two features: type and storage class.
- Type specifies the type of data that can be stored in a variable.  
For example: int, float, char etc.
- And, storage class controls two different properties of a variable: **lifetime** (determines how long a variable can exist) and **scope** (determines which part of the program can access it).

## Contd..

- There are five types of storage classes of C++. They are:
  - Automatic
  - External
  - Static
  - Register
  - Mutable

# Contd..

- **Automatic Storage Class:**

- The keyword `auto` is used to declare automatic variables(or local variables).
- **Syntax:** `auto int var;`
- However, if a variable is declared without any keyword inside a function, it is automatic by default.
- The scope of automatic variable is only limited to the function where it is defined
- The life of a automatic variable ends (It is destroyed) when the function exits.







# Contd..

- **Example:**

```
1 // socpe and visibility (Local variable(automatic variable))
2 #include <iostream>
3 using namespace std;
4
5 void test();
6
7 int main()
8 {
9     // local variable to main()
10    int var = 5;
11
12    test();
13
14    // illegal: var1 not declared inside main()
15    var1 = 9;
16 }
17
18 void test()
19 {
20    // local variable to test()
21    int var1;
22    var1 = 6;
23
24    // illegal: var not declared inside test()
25    cout << var;
26 }
```

# Contd..

- **Output:**

 Compiler (4)  Resources  Compile Log  Debug  Find Results  Close			
Line	Col	File	Message
		C:\Users\hp\Documents\C++ practice\Control statem...	In function 'int main()':
15	5	C:\Users\hp\Documents\C++ practice\Control stateme...	[Error] 'var1' was not declared in this scope
		C:\Users\hp\Documents\C++ practice\Control statem...	In function 'void test()':
25	13	C:\Users\hp\Documents\C++ practice\Control stateme...	[Error] 'var' was not declared in this scope

# Contd..

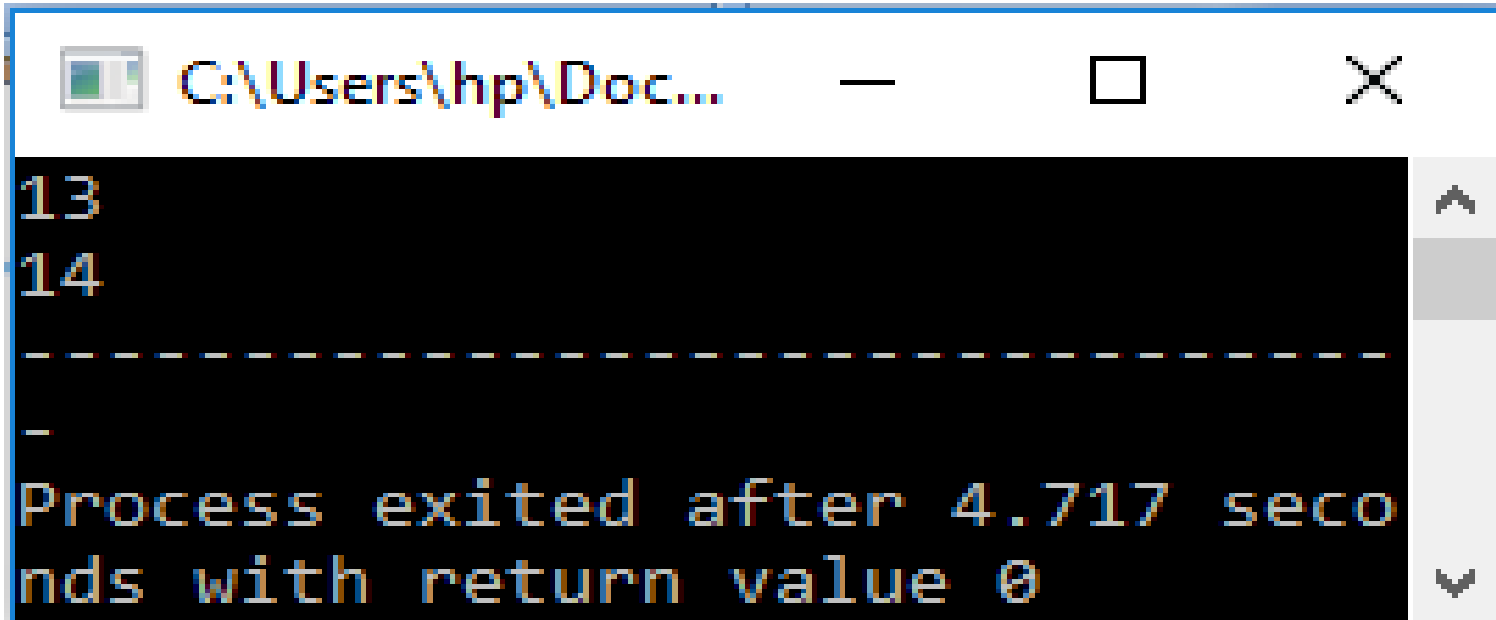
- **Example:** scope and life of global variable is entire program

```
1 // illustration of global variable
2 #include <iostream>
3 using namespace std;
4
5 // Global variable declaration
6 int c = 12;
7
8 void test();
9
10 int main()
11 {
12     c++;
13
14     // Outputs 13
15     cout << c << endl;
16     test();
17
18     return 0;
19 }
20
21 void test()
22 {
23     c++;
24
25     // Outputs 14
26     cout << c;
27 }
28
```



## Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\hp\Doc...' and standard window controls. The command prompt has a black background with white text. It displays the numbers '13' and '14' on separate lines. Below these is a dashed line, followed by a single hyphen '-' on a new line. The final line of output reads 'Process exited after 4.717 seconds with return value 0'. A vertical scrollbar is visible on the right side of the text area.

```
13
14
-----
-
Process exited after 4.717 seconds with return value 0
```

# Contd..

- **External Storage class:**

- The keyword **extern** is used to declare external variable.
- The **extern** modifier is most commonly used when there are two or more files sharing the same global variables.
- When you have multiple files and you define a global variable , which will be used in other files also, then *extern* will be used in another file to give reference of defined variable .

## Contd..

- **Example: Mul.cpp** file to illustrate external storage class

```
1 // program to demonstrate external variable
2
3 int test = 100; // assigning value to test
4 |
5 void multiply(int n)
6 {
7     test = test* n;
8
9 }
```

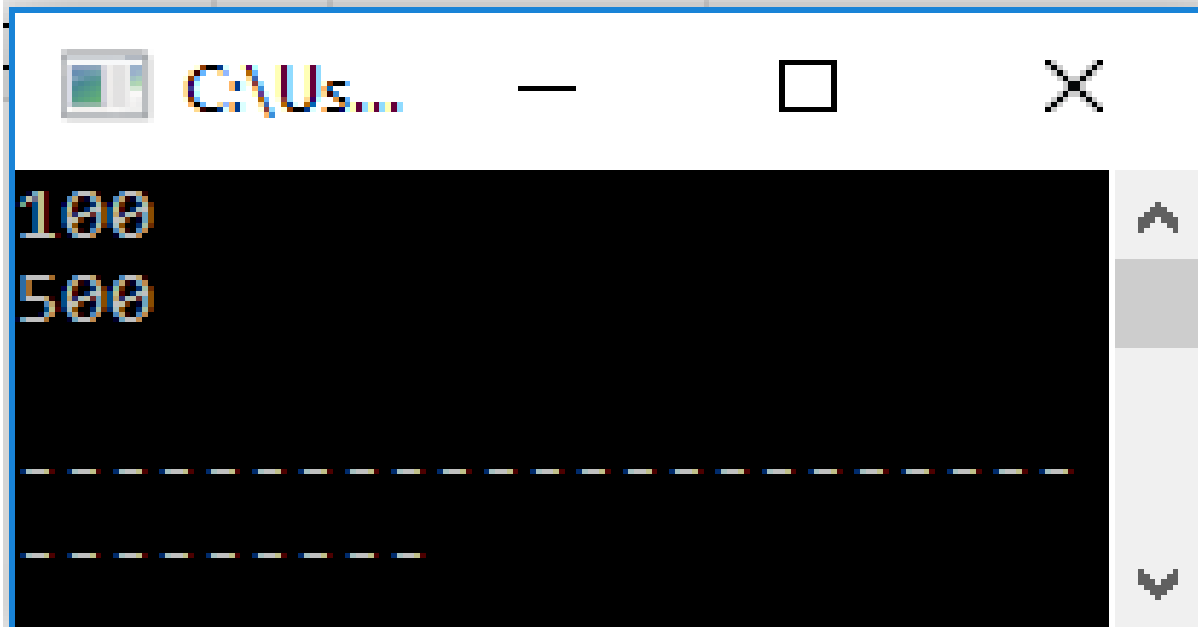
# Contd..

- **Example: Mul.cpp** file to illustrate external storage class

```
1  // program to demonstrate external variable
2
3  #include<iostream>
4
5  #include"mul.cpp" // include the content of mul.cpp
6
7  using namespace std;
8
9  extern int test; // declaring test
10
11 int main()
12 {
13     cout<< test<< endl;
14
15     multiply(5);
16     |
17     cout<<test<<endl;
18 }
```

# Contd..

- **Output:**



```
C:\Us...  
100  
500  
-----  
-----
```

# Contd..

- **Static storage class:**

- Keyword **static** is used for specifying a static variable.
- **For example:**

```
int main()
{
    static float a;
    ... ..
}
```

- A static local variable exists only inside a function where it is declared (similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.
- The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

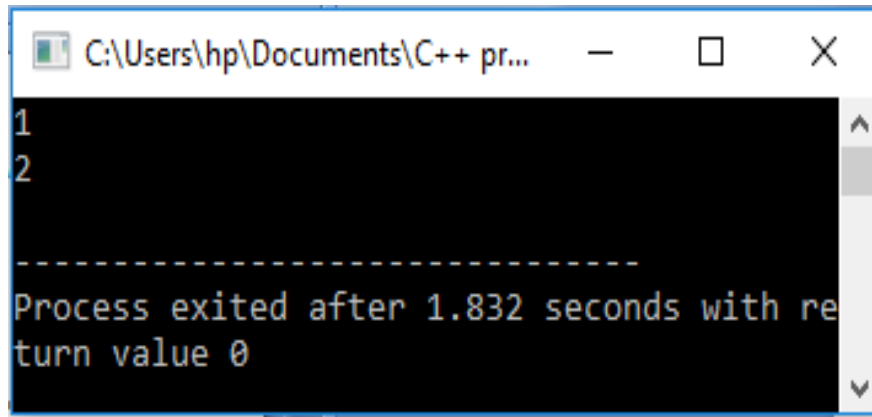
# Contd..

- **Example:**

```
1  //static storage class
2  #include <iostream>
3  using namespace std;
4
5  void test()
6  {
7      // var is a static variable
8      static int var = 0;
9      var++;
10
11      cout << var << endl;
12  }
13
14  int main()
15  {
16      test();
17      test();
18
19      return 0;
20  }
```

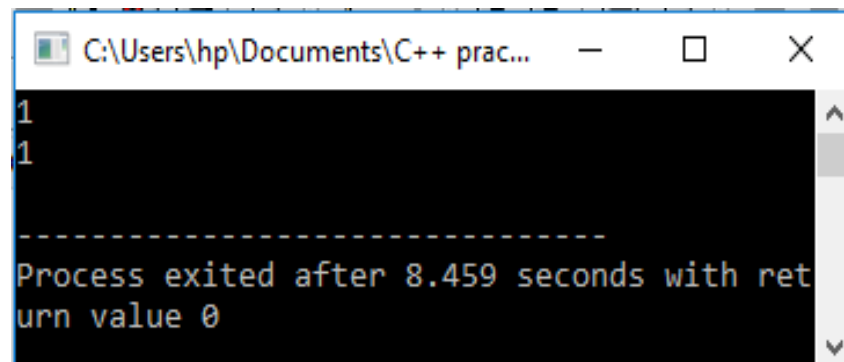
# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ pr...  
1  
2  
-----  
Process exited after 1.832 seconds with re  
turn value 0
```

- **Output of above program if var was not specified as static variable:**



```
C:\Users\hp\Documents\C++ prac...  
1  
1  
-----  
Process exited after 8.459 seconds with ret  
urn value 0
```



# Contd..

- **Register storage class:**

- Keyword **register** is used for specifying register variables.
  - **Syntax:** register datatype var\_name1 [= value];
- Register variables are similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables.
- If a program encounters a register variable, it stores the variable in processor's register rather than memory if available. This makes it faster than the local variables.
- However, this keyword was deprecated in C++11 and should not be used.

## Contd..

- **Mutable storage class:**

- In C++, a class object can be kept constant using keyword **const**. This doesn't allow the data members of the class object to be modified during program execution. But, there are cases when some data members of this constant object must be changed.
- In those cases, we can make these variables modifiable using a **mutable** storage class.
- **Syntax:** mutable datatype var\_name1;

# Pointers

- A **pointer** is a variable whose value is the address of another variable.
- Pointers are used in C++ program to access the memory and manipulate the address.
- Pointers solve two common software problems:
  - Pointers allow different sections of code to share information easily.
  - Pointers enable to build complex linked data structures like linked lists trees , etc.

# Contd..

- **Pointer variables declaration:**

- Like any variable, you must declare a pointer before you can work with it.
- The general form of a pointer variable declaration is :
  - **Syntax:** `type *var-name; // pointer variable is declared by placing * before the variable name.`
- Where,
  - **type** is the data type pointed to by the pointer. This type is not the type of the pointer itself, but the type of the data the pointer points to.
  - **Var\_name** = name of pointer variable (should be valid identifier)
  - **asterisk (\*)** used when declaring a pointer only means that it is a pointer

# Contd..

- **For Example:**

- `int *ip; // pointer to an integer`
- `double *dp; // pointer to a double`
- `float *fp; // pointer to a float`
- `char *ch // pointer to character`
- The **actual data type of the value of all pointers**, whether integer, float, character, or otherwise, **is the same**, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## Contd..

- When declaring multiple pointer variable of same type on the same line then \* must be preceded in each variable as:
  - `Int * p1, *p2;`
- `Int * p1, p2;` // declares p1 as pointer variable and p2 as a normal variable

# Contd..

- **Pointer initialization:**

- Pointers can be initialized either to the address of a variable , or to the value of another pointer(or array):

- **Pointer initialization to the address of a variable:**

- For example:

```
int myvar;
```

```
int * myptr = &myvar;
```

- **Pointer initialization to the value of another pointer:**

- For example:

```
int myvar;
```

```
int *f = &myvar;
```

```
int *bar = f;
```

–

# Contd..

- **Operators in Pointers:**
  - Reference operator (&)/address-of and
  - Deference operator (\*)/ value at the address



## Contd..

- **Reference operator (&)/address-of and:**

- The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as address-of operator.

- **For example:** `int a = 10;`

`int *p = &a; //pointer declaration and initialization`

`int b = a; // value of a is assigned to b so, b = 10`

- This would assign the address of variable **a** to pointer variable **p**, we are no longer assigning the content of the variable itself to **p**, but its address..

# Contd..

- **Dereference operator (\*):**

- An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator (\*). The operator itself can be read as "value pointed to by".
- For example: `int x = *p;`
- This could read as **x equal to value pointed by p**, and the statement would actually assign the value of 10 to x

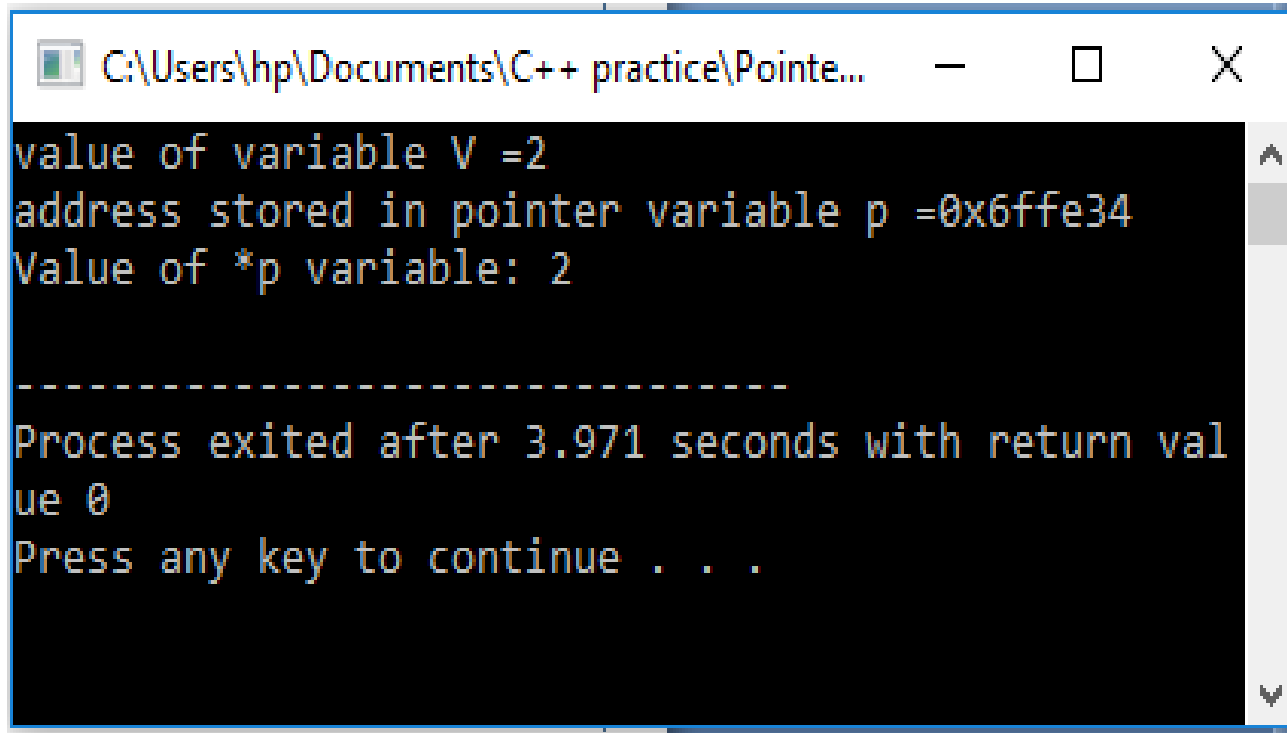
# Contd..

- **Example1:**

```
1 // program to illustrate reference(&)and Deference(*) operator
2 #include<iostream>
3 using namespace std;
4 int main()
5 {
6     int v = 2; // actual variable declaration and initialization
7     int * p; // declares p as a pointer variable
8     // initialization of pointer variable p by the address of normal variable v
9     p= &v;
10    cout<<"value of variable V ="<<v<<endl;
11    cout<<"address stored in pointer variable p ="<<p<<endl;
12    cout << "Value of *p variable: "<< *p << endl;
13    return 0;
14 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ practice\Pointe...  
value of variable V =2  
address stored in pointer variable p =0x6ffe34  
Value of *p variable: 2  
  
-----  
Process exited after 3.971 seconds with return val  
ue 0  
Press any key to continue . . .
```

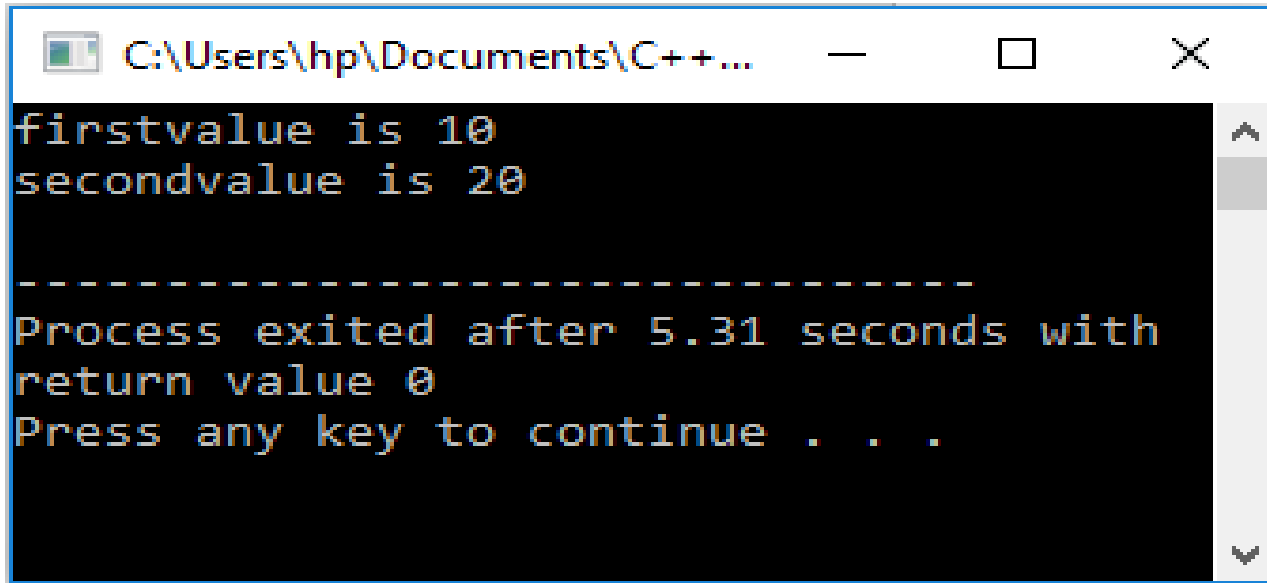
# Contd..

- **Example2:**

```
1  // program2 to illustrate & and * operator in pointer
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7      int firstvalue, secondvalue;
8      int * mypointer;
9
10     mypointer = &firstvalue;
11     *mypointer = 10;
12     mypointer = &secondvalue;
13     *mypointer = 20;
14     cout << "firstvalue is " << firstvalue << '\n';
15     cout << "secondvalue is " << secondvalue << '\n';
16     return 0;
17 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\hp\Documents\C++...' and standard window controls. The black console area displays the following text in a monospaced font: 'firstvalue is 10', 'secondvalue is 20', a dashed line separator, 'Process exited after 5.31 seconds with return value 0', and 'Press any key to continue . . .'. A vertical scrollbar is visible on the right side of the console area.

```
C:\Users\hp\Documents\C++...  
firstvalue is 10  
secondvalue is 20  
-----  
Process exited after 5.31 seconds with  
return value 0  
Press any key to continue . . .
```

# Contd..

- **Pointers and Arrays:**

- The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type.
- For example, consider these two declarations:
  - `int x[20];`
  - `int* p;`
- Now the following assignment operation is valid:
  - `p = x;`

## Contd..

- After that, **p** and **x** would be equivalent and would have very similar properties.
- The main difference being that **p** can be assigned a different address, whereas **x** can never be assigned anything, and will always represent the same block of 20 elements of type int.
- Therefore, the following assignment would not be valid:
  - `x = p;`



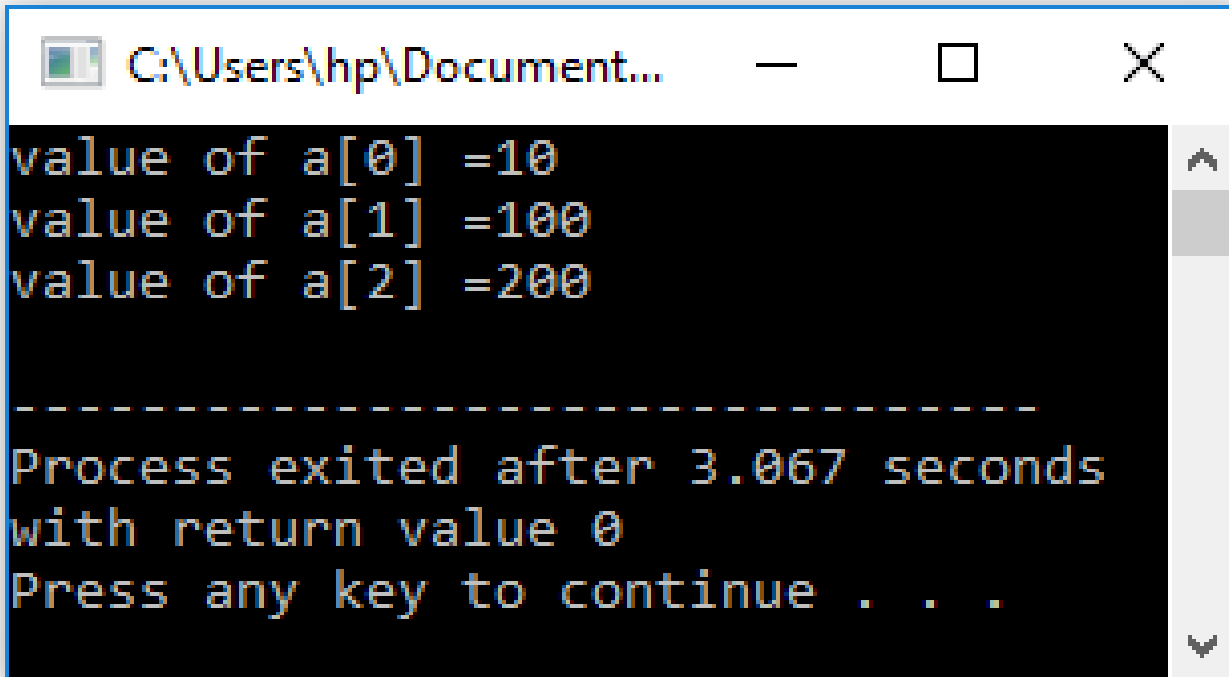
# Contd..

- **Example:**

```
1  // pointer and array
2  #include<iostream>
3  using namespace std;
4  const int MAX = 3;
5  int main()
6  {
7      int a[MAX] = { 10, 100, 200};
8      int * p;
9      p = a;
10     for(int i =0; i<MAX; i++)
11     {
12         cout<<"value of a["<<i<<" ] ="<<*p<< endl;
13         p++; // points to the next location of array
14     }
15     return 0;
16 }
```

# Contd..

- **Output:**

A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\hp\Document...'. The window has standard minimize, maximize, and close buttons. The command prompt displays the following text: 'value of a[0] =10', 'value of a[1] =100', 'value of a[2] =200', followed by a dashed line separator, 'Process exited after 3.067 seconds', 'with return value 0', and 'Press any key to continue . . .'.

```
C:\Users\hp\Document...  
value of a[0] =10  
value of a[1] =100  
value of a[2] =200  
-----  
Process exited after 3.067 seconds  
with return value 0  
Press any key to continue . . .
```

- **Note:** Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.

# Contd..

- **Passing Array to a Function in C++:**
  - C++ does not allow to pass an entire array as an argument to a function.
  - However, You can pass a pointer to an array by specifying the array's name without an index.
    - E.g., `display(marks);`
  - To pass a array as an argument in a function declare function formal parameter in one of following three ways:

## Contd..

- **Way-1 :Formal parameters as a pointer as follows –**

```
void myFunction(int *param)
```

```
{
```

```
.....
```

```
.....
```

```
.....
```

```
}
```

# Contd..

- **Way-2: Formal parameters as a sized array as follows :**

```
void myFunction(int param[10])
```

```
{
```

```
    ...
```

```
    ....
```

```
    .....
```

```
}
```

## Contd..

- **Way-3: Formal parameters as an unsized array as follows :**

```
void myFunction(int param[])
```

```
{
```

```
...
```

```
....
```

```
....
```

```
}
```

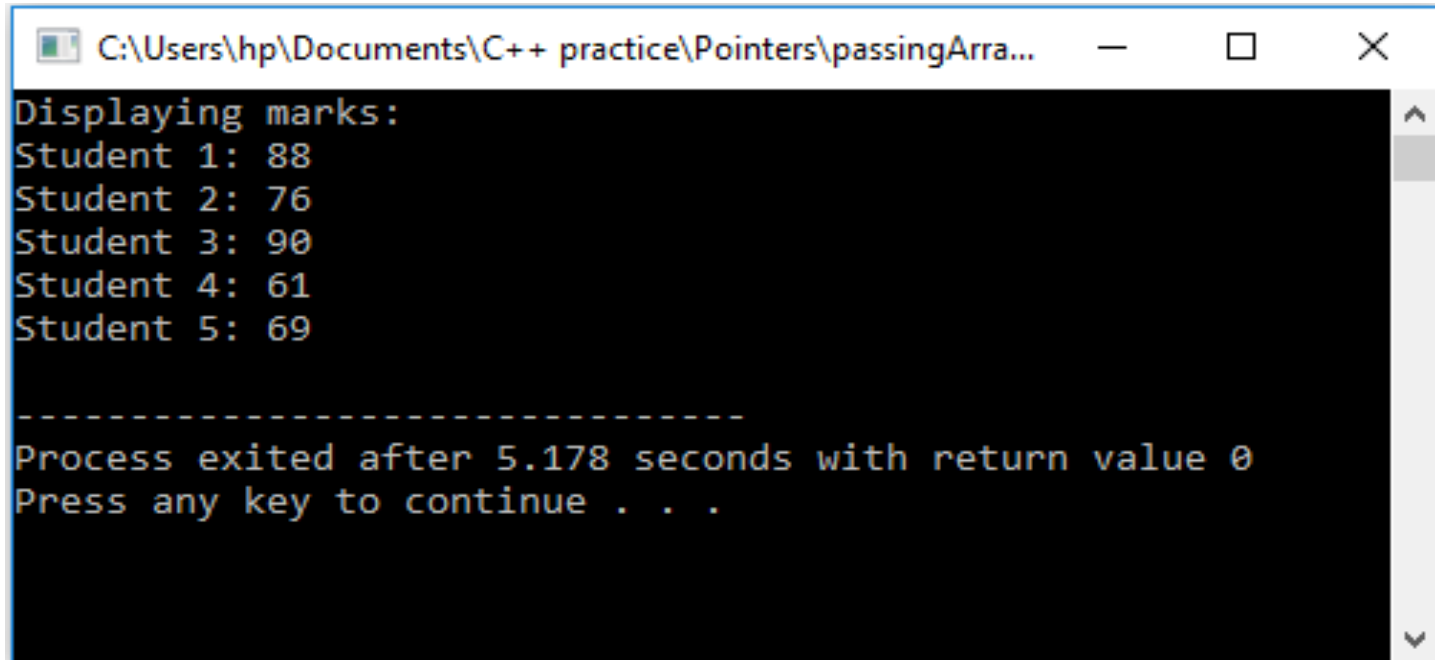
# Contd..

- **Example1:**

```
1  //passing array to a function Way 1
2  //C++ Program to display marks of 5 students
3  #include <iostream>
4  using namespace std;
5
6  void display(int marks[5]);
7
8  int main()
9  {
10     int marks[5] = {88, 76, 90, 61, 69};
11     display(marks);
12     return 0;
13 }
14
15 void display(int *p)
16 {
17     cout << "Displaying marks: " << endl;
18
19     for (int i = 0; i < 5; ++i)
20     {
21         cout << "Student " << i + 1 << ": " << *p << endl;
22     }
23 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ practice\Pointers\passingArra...
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69

-----
Process exited after 5.178 seconds with return value 0
Press any key to continue . . .
```



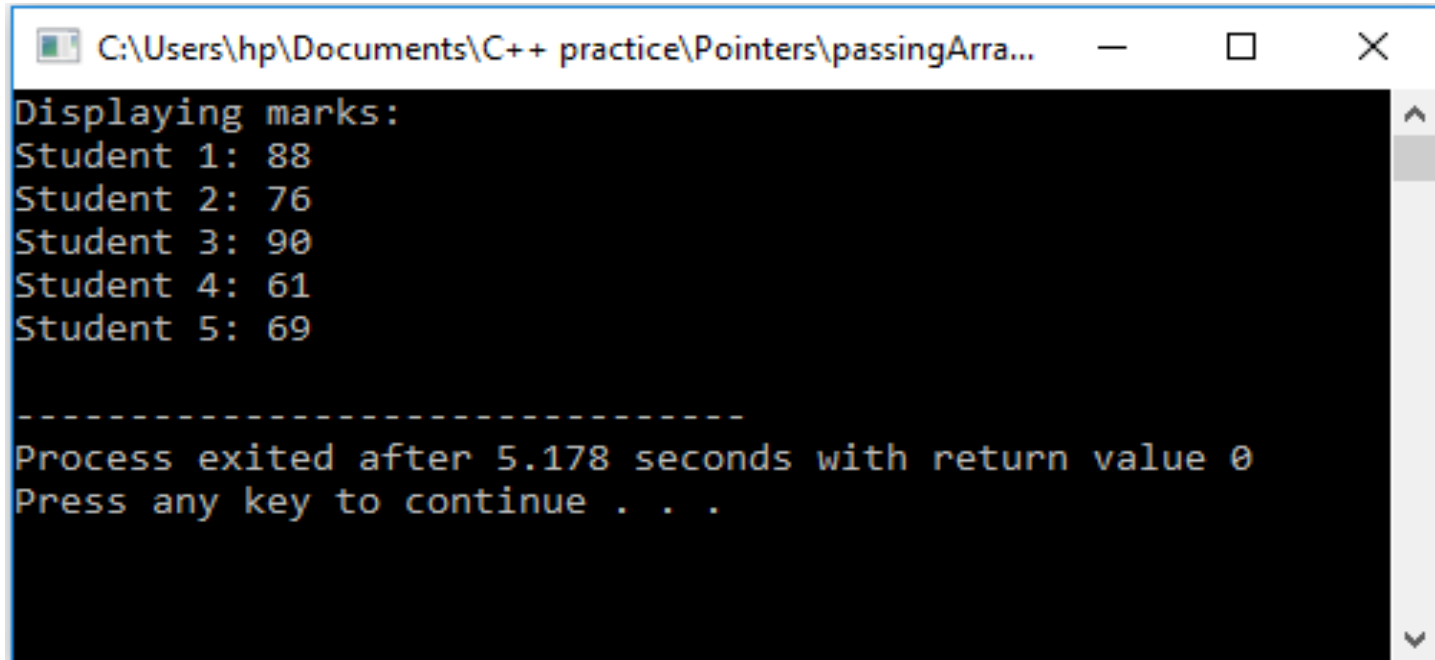
# Contd..

- **Example2:**

```
1  //passing array to a function Way 2
2  //C++ Program to display marks of 5 students
3  #include <iostream>
4  using namespace std;
5
6  void display(int marks[5]);
7
8  int main()
9  {
10     int marks[5] = {88, 76, 90, 61, 69};
11     display(marks);
12     return 0;
13 }
14
15 void display(int m[5])
16 {
17     cout << "Displaying marks: " << endl;
18
19     for (int i = 0; i < 5; ++i)
20     {
21         cout << "Student " << i + 1 << ": " << m[i] << endl;
22     }
23 }
```

# Contd..

- **Output:**



```
C:\Users\hp\Documents\C++ practice\Pointers\passingArra...
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69

-----
Process exited after 5.178 seconds with return value 0
Press any key to continue . . .
```

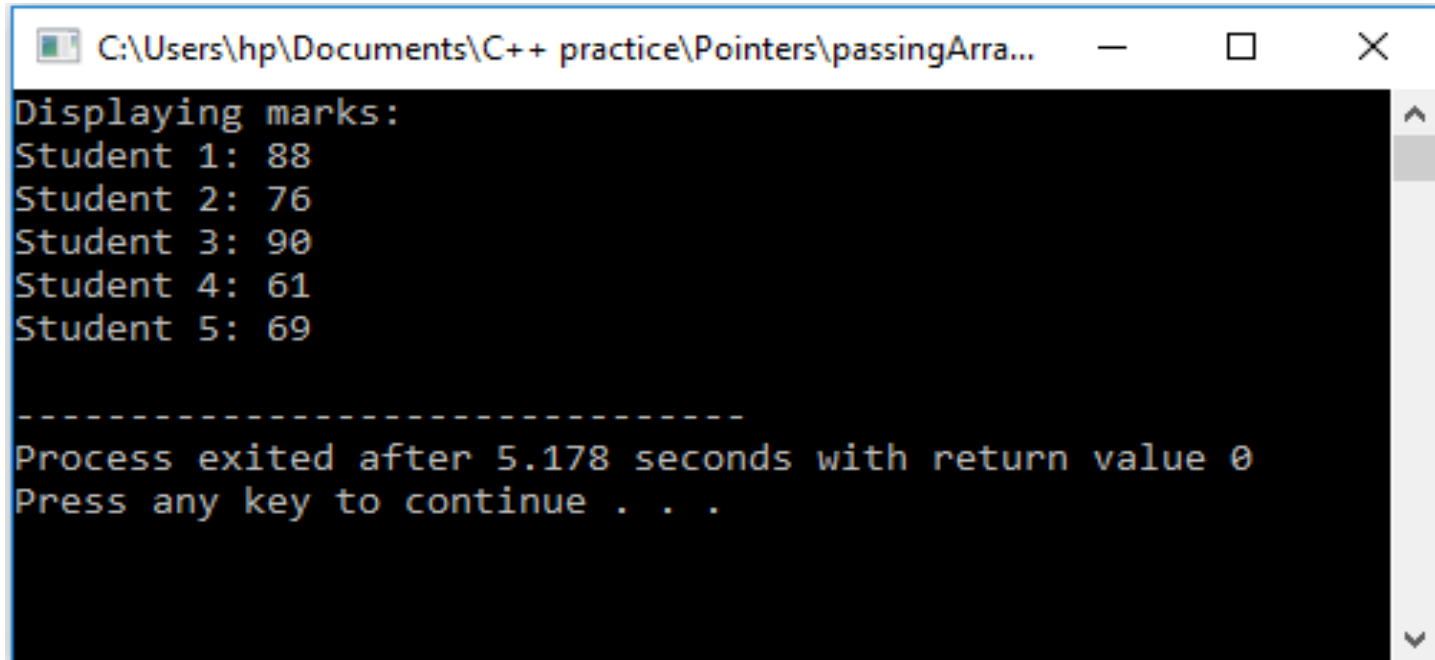
# Contd..

- **Example3:**

```
1  //passing array to a function Way 3
2  //C++ Program to display marks of 5 students
3  #include <iostream>
4  using namespace std;
5
6  void display(int marks[5]);
7
8  int main()
9  {
10     int marks[5] = {88, 76, 90, 61, 69};
11     display(marks);
12     return 0;
13 }
14
15 void display(int m[])
16 {
17     cout << "Displaying marks: " << endl;
18
19     for (int i = 0; i < 5; ++i)
20     {
21         cout << "Student " << i + 1 << ": " << m[i] << endl;
22     }
23 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\hp\Documents\C++ practice\Pointers\passingArra... The window has standard minimize, maximize, and close buttons. The output text is as follows:

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69

-----
Process exited after 5.178 seconds with return value 0
Press any key to continue . . .
```

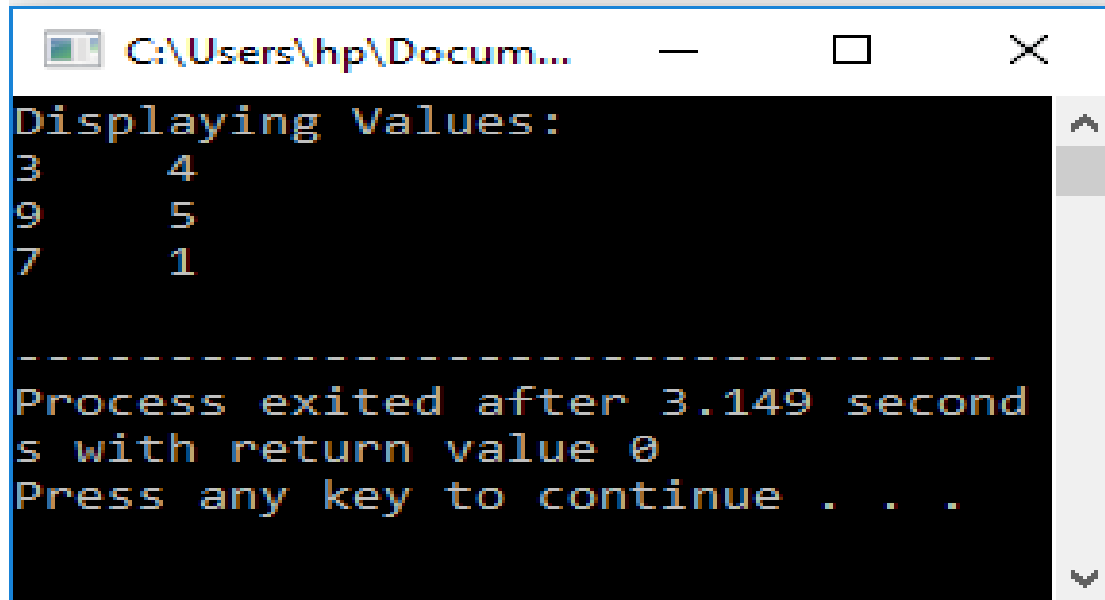
# Contd..

- Example: passing 2-D Array to function

```
1 //passing two dimensional array to a function
2 #include <iostream>
3 using namespace std;
4
5 void display(int n[3][2]);
6
7 int main()
8 {
9     int num[3][2] = {
10         {3, 4},
11         {9, 5},
12         {7, 1}};
13     display(num);
14     return 0;
15 }
16
17 void display(int n[3][2])
18 {
19
20     cout << "Displaying Values: " << endl;
21     for(int i = 0; i < 3; ++i)
22     {
23         for(int j = 0; j < 2; ++j)
24         {
25             cout << n[i][j] << "    ";
26         }
27         cout<<endl;
28     }
29 }
```

# Contd..

- **Output:**



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\hp\Docum...' and standard window controls. The command prompt displays the following text:

```
Displaying Values:  
3      4  
9      5  
7      1  
  
-----  
Process exited after 3.149 second  
s with return value 0  
Press any key to continue . . .
```

# Dynamic Memory Allocation with **new** and **delete**

- Arrays can be used to store multiple homogenous data but there are serious drawbacks of using arrays.
- You should allocate the memory of an array when you declare it but most of the time, the exact memory needed cannot be determined until runtime.
- The best thing to do in this situation is to declare an array with maximum possible memory required (declare array with maximum possible size expected).
- The downside to this is unused memory is wasted and cannot be used by any other programs.
- To avoid wastage of memory, you can dynamically allocate memory required during runtime using new and delete operator in C++.

## Contd..

- **Operators new and new[]**

- Dynamic memory is allocated using operator new.
- new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated



## Contd..

- **Its syntax is:**

`pointer = new type`

`pointer = new type [number_of_elements]`

The first expression is used to allocate memory to contain one single element of type type.

- The second one is used to allocate a block (an array) of elements of type type, where number\_of\_elements is an integer value representing the amount of these.

## Contd..

- For example:

```
int * p;
```

```
p = new int [5];
```

- In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to p (a pointer).

Therefore, p now points to a valid block of memory with space for five elements of type int.

# Contd..

- **Operators delete and delete[]**
- In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose **syntax is:**  
  
    delete pointer;  
  
    delete[] pointer;
- The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

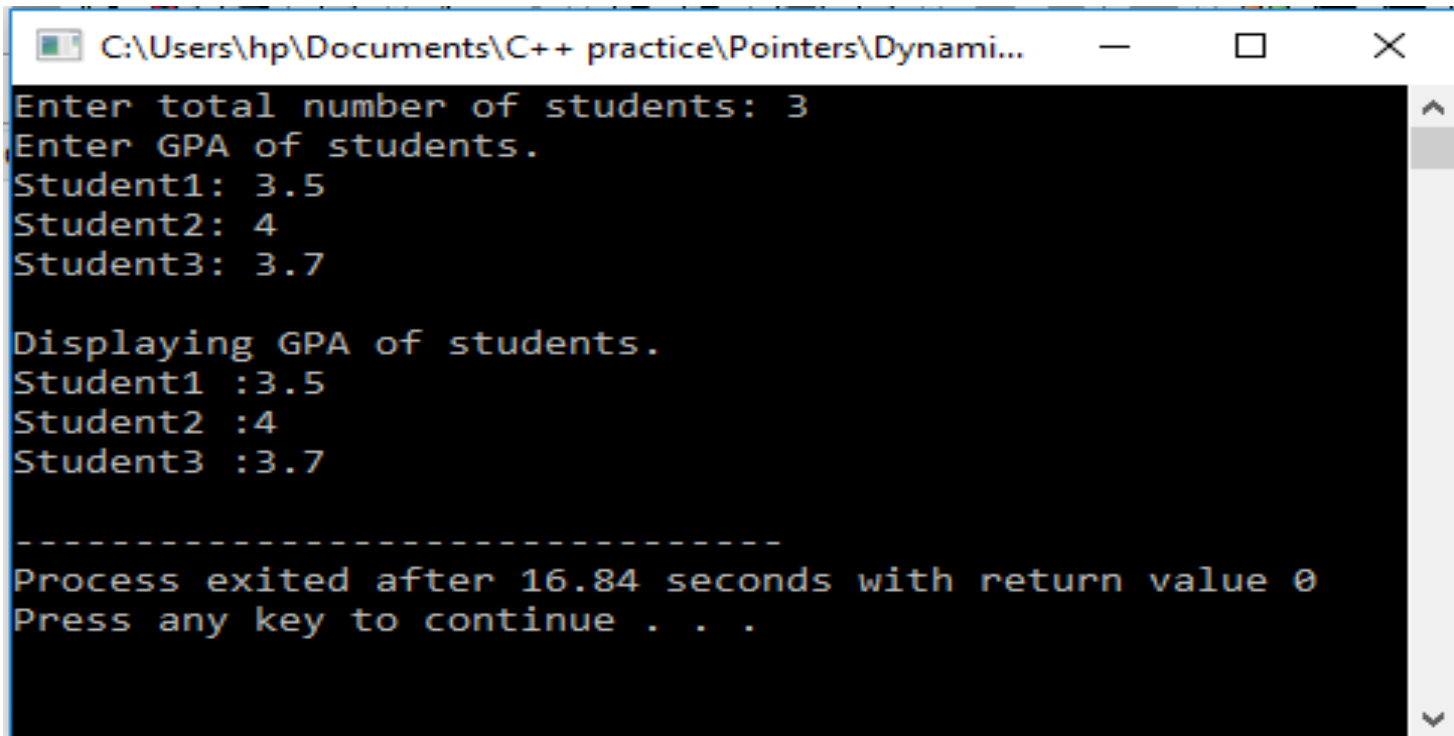
# Contd..

- **Example:**

```
1  #include <iostream>
2  // Dynamic memory allocation with new and delete
3  //C++ Program to store GPA of n number of students and display it
4  //where n is the number of students entered by user.
5  #include <cstring>
6  using namespace std;
7
8  int main()
9  {
10     int num;
11     cout << "Enter total number of students: ";
12     cin >> num;
13     float* ptr;
14
15     // memory allocation of num number of floats
16     ptr = new float[num];
17
18     cout << "Enter GPA of students." << endl;
19     for (int i = 0; i < num; ++i)
20     {
21         cout << "Student" << i + 1 << ": ";
22         cin >> *(ptr + i);
23     }
24
25     cout << "\nDisplaying GPA of students." << endl;
26     for (int i = 0; i < num; ++i) {
27         cout << "Student" << i + 1 << " : " << *(ptr + i) << endl;
28     }
29
30     // ptr memory is released
31     delete [] ptr;
32
33     return 0;
34 }
```

# Contd..

- **Output**



```
C:\Users\hp\Documents\C++ practice\Pointers\Dynami...
Enter total number of students: 3
Enter GPA of students.
Student1: 3.5
Student2: 4
Student3: 3.7

Displaying GPA of students.
Student1 :3.5
Student2 :4
Student3 :3.7

-----
Process exited after 16.84 seconds with return value 0
Press any key to continue . . .
```

# Thank You !

