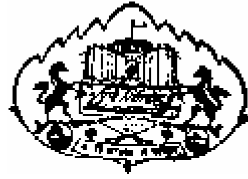


University of Pune

Object Oriented Concepts and Programming in C++ (CS-221)

S.Y.B.Sc.(Computer Science)

Semester II

Advisor:
Prof. A. D. Gangarde
(Chairman, Bord of Studies – Comp. Sc.)

Chairman:
Prof. S. S. Deshmukh
(Vice-Principal & Head, Modern College, Shivajinagar, Pune 5.)

Co-ordinator:
Mr. A. V. Sathe
(Modern College, Shivajinagar, Pune 5)

Authors:
Mrs. Manisha Suryavanshi (Modern College, Shivajinagar, Pune 5)
Mrs. Madhuri Ghanekar (Modern College, Shivajinagar, Pune 5)
Mr. S. G. Lakhdive (Prof. Ramkrishna More College, Akurdi, Pune)
Mr. Parag Tamhankar (Abasaheb Garware College, Pune.)
Mrs. Manisha Jagdale (Annasaheb Magar College, Hadapsar, Pune)

Board of Study (Computer Science) members:

1. Mr. M. N. Shelar
2. Mr. S. N. Shinde
3. Mr. U. S. Surve
4. Mr. V. R. Wani
5. Mr. Prashant Mule
6. Dr.Vilas Kharat
7. Mrs. Chitra Nagarkar

Table of Contents

About the work book……………………………………………………………4

1. Introduction to C++ Programming ……………………………………………6

2. Functions in C++…………………………………………………………..21

3.Classes and Objects………………………………………………………..27

4.Constructors and Destructors………………………………………………41

5.Operator Overloading……………………………………………………...48

6.Inheritance…………………………………………………………………...58

7. Formatted I/O…………………………………………………………………66

8.Exception Handling……………………………………………………………78

9.File I/O…………………………………………………………………………..84

10.Templates…………………………………………………………………… 89

About the Work Book

Objectives of this book

This workbook is intended to be used by S.Y.B.Sc(Computer Science) students for the two computer science laboratory courses.

The objectives of this book are
1. To define the scope of the course.
2. Bringing uniformity in the way course is conducted across different colleges.
3. Continuous assessment of the students.
4. Providing ready references for students while working in the lab.

How to use this book?

This book is mandatory for the completion of the laboratory course. It is a measure of the performance of the student in the laboratory for the entire duration of the course.

Instructions to the students

1) Students should carry this book during practical sessions of computer science.
2) Students should maintain separate journal for the source code, SQL queries/commands along with outputs.
3) Student should read the topics mentioned in Reading section of this book before coming for practical.
4) Students should solve only those exercises which are selected by practical in-charge as a part of journal activity. However, students are free to solve additional exercises to do more practice for their practical examination.

Exercise Set	Difficulty Level	Rule
SET A	Easy	All exercises are compulsory in this set.
SET B	Medium	At least one exercise is compulsory in this set.
SET C	Difficult	Not compulsory.

5) Students will be assessed for each exercise on a scale of 5

1. Note Done 0
2. Incomplete 1
3. Late Complete 2
4. Needs Improvement 3
5. Complete 4
6. Well Done 5

Instructions to the Practical In-charge

1) Explain the assignment and related concepts in around ten minutes using white board if required or by demonstrating the software.
2) Choose appropriate problems to be solved by student.
3) After a student completes a specific set, the instructor has to verify the outputs and sign in the provided space after the activity.
4) Ensure that the students use good programming practices.
5) You should evaluate each assignment carried out by a student on a scale of 5 as specified above ticking appropriate box.
6) The value should also be entered on assignment completion page of respected lab course.

SESSION 1
Introduction to C++ Programming

Start Date ____/____/______

Objectives

To learn about:

- Keywords, Data types and operators
- C++ program structure
- Type conversion in C++
- Reference variables
- Call by reference, return by reference

Reading

You should read following topics before starting this exercise:

1. Difference between C and C++.
2. Features of C++
3. Concept of Translators, Compilers
4. C++ functions

Ready References

1.1 C++ Keywords

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

1.2 Data Types

Basic Data Types

Type	Bit Width	Range
char	8	-128 to +127
wchar_t	16	0 to 65,535
int (16-bit environments)	16	-32,768 to 32,767
int (32-bit environments)	32	-2,147,483,648 to +2,147,483,647
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
bool	N/A	true/false
void	N/A	valueless

Data Types with Modifiers in Combination

Type	Bit Width	Range
unsigned char	8	0 to 255
signed char	8	-128 to +127
unsigned int	32	0 to 4,294,967,295
signed int	32	-2,147,483,648 to 2,147,483,647
short int	16	-32,768 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	-32,768 to 32,767
long int	32	same as int
unsigned long int	32	same as unsigned int
signed long int	32	same as signed int
long double	80	3.4E-4932 to 1.1E+4932

1.3 Operators In C++

Operator	Description	Example	Overloadable
Group 1 (no associativity)			
::	Scope resolution operator	Class::age = 2;	NO
Group 2			
()	Function call	isdigit('1')	YES
()	Member initalization	c_tor(int x, int y) : _x(x), _y(y*10){};	YES
[]	Array access	array[4] = 2;	YES
->	Member access from a pointer	ptr->age = 34;	YES
.	Member access from an object	obj.age = 34;	NO
++	Post-increment	for(int i = 0; i < 10; i++) cout << i;	YES
--	Post-decrement	for(int i = 10; i > 0; i--) cout << i;	YES
const_cast	Special cast	const_cast<type_to>(type_from);	NO
dynamic_cast	Special cast	dynamic_cast<type_to>(type_from);	NO
static_cast	Special cast	static_cast<type_to>(type_from);	NO
reinterpret_cast	Special cast	reinterpret_cast<type_to>(type_from);	NO
typeid	Runtime type information	cout « typeid(var).name(); cout « typeid(type).name();	NO
Group 3 (right-to-left associativity)			
!	Logical negation	if(!done) …	YES
not	Alternate spelling for !		
~	Bitwise complement	flags = ~flags;	YES
compl	Alternate spelling for ~		
++	Pre-increment	for(i = 0; i < 10; ++i) cout << i;	YES
--	Pre-decrement	for(i = 10; i > 0; --i) cout << i;	YES
-	Unary minus	int i = -1;	YES
+	Unary plus	int i = +1;	YES
*	Dereference	int data = *intPtr;	YES
&	Address of	int *intPtr = &data;	YES
new	Dynamic memory allocation	long *pVar = new long; MyClass *ptr = new MyClass(args);	YES
new []	Dynamic memory allocation of array	long *array = new long[n];	YES
delete	Deallocating the memory	delete pVar;	YES
delete []	Deallocating the memory of array	delete [] array;	YES
(type)	Cast to a given type	int i = (int) floatNum;	YES
sizeof	Return size of an object or type	int size = sizeof floatNum; int size = sizeof(float);	NO

Operator	Description	Example	Overloadable
Group 4			
->*	Member pointer selector	ptr->*var = 24;	YES
.*	Member object selector	obj.*var = 24;	NO
Group 5			
*	Multiplication	int i = 2 * 4;	YES
/	Division	float f = 10.0 / 3.0;	YES
%	Modulus	int rem = 4 % 3;	YES
Group 6			
+	Addition	int i = 2 + 3;	YES
-	Subtraction	int i = 5 - 1;	YES
Group 7			
<<	Bitwise shift left	int flags = 33 << 1;	YES
>>	Bitwise shift right	int flags = 33 >> 1;	YES
Group 8			
<	Comparison less-than	if(i < 42) …	YES
<=	Comparison less-than-or-equal-to	if(i <= 42) ...	YES
>	Comparison greater-than	if(i > 42) …	YES
>=	Comparison greater-than-or-equal-to	if(i >= 42) ...	YES
Group 9			
==	Comparison equal-to	if(i == 42) ...	YES
eq	Alternate spelling for ==		
!=	Comparison not-equal-to	if(i != 42) …	YES
not_eq	Alternate spelling for !=		
Group 10			
&	Bitwise AND	flags = flags & 42;	YES
bitand	Alternate spelling for &		
Group 11			
^	Bitwise exclusive OR (XOR)	flags = flags ^ 42;	YES
xor	Alternate spelling for ^		

<table>
<tr><td colspan="4">Group 12</td></tr>
<tr><td>|</td><td>Bitwise inclusive (normal) OR</td><td>flags = flags | 42;</td><td>YES</td></tr>
<tr><td>bitor</td><td colspan="3">Alternate spelling for |</td></tr>
<tr><td colspan="4">Group 13</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>if(conditionA && conditionB) ...</td><td>YES</td></tr>
<tr><td>and</td><td colspan="3">Alternate spelling for &&</td></tr>
<tr><td colspan="4">Group 14</td></tr>
<tr><td>||</td><td>Logical OR</td><td>if(conditionA || conditionB) ...</td><td>YES</td></tr>
<tr><td>or</td><td colspan="3">Alternate spelling for ||</td></tr>
<tr><td colspan="4">Group 15 (right-to-left associativity)</td></tr>
<tr><td>? :</td><td>Ternary conditional (if-then-else)</td><td>int i = (a > b) ? a : b;</td><td>NO</td></tr>
<tr><td colspan="4">Group 16 (right-to-left associativity)</td></tr>
<tr><td>=</td><td>Assignment operator</td><td>int a = b;</td><td>YES</td></tr>
<tr><td>+=</td><td>Increment and assign</td><td>a += 3;</td><td>YES</td></tr>
<tr><td>-=</td><td>Decrement and assign</td><td>b -= 4;</td><td>YES</td></tr>
<tr><td>*=</td><td>Multiply and assign</td><td>a *= 5;</td><td>YES</td></tr>
<tr><td>/=</td><td>Divide and assign</td><td>a /= 2;</td><td>YES</td></tr>
<tr><td>%=</td><td>Modulo and assign</td><td>a %= 3;</td><td>YES</td></tr>
<tr><td>&=</td><td>Bitwise AND and assign</td><td>flags &= new_flags;</td><td>YES</td></tr>
<tr><td>and_eq</td><td colspan="3">Alternate spelling for &=</td></tr>
<tr><td>^=</td><td>Bitwise exclusive or (XOR) and assign</td><td>flags ^= new_flags;</td><td>YES</td></tr>
<tr><td>xor_eq</td><td colspan="3">Alternate spelling for ^=</td></tr>
<tr><td>|=</td><td>Bitwise normal OR and assign</td><td>flags |= new_flags;</td><td>YES</td></tr>
<tr><td>or_eq</td><td colspan="3">Alternate spelling for |=</td></tr>
<tr><td><<=</td><td>Bitwise shift left and assign</td><td>flags <<= 2;</td><td>YES</td></tr>
<tr><td>>>=</td><td>Bitwise shift right and assign</td><td>flags >>= 2;</td><td>YES</td></tr>
<tr><td colspan="4">Group 17</td></tr>
<tr><td>throw</td><td>throw exception</td><td>throw EClass("Message");</td><td>NO</td></tr>
<tr><td colspan="4">Group 18</td></tr>
<tr><td>,</td><td>Sequential evaluation operator</td><td>for(i = 0, j = 0; i < 10; i++, j++) ...</td><td>YES</td></tr>
</table>

1.4 C++ Program Structure

Example 1: Simple Program in C++ to calculate weekly pay.

```
 1  #include <iostream>

 2  using namespace std;
 3  int main (void)
 4  {
 5      int workDays;
 6      float  workHours, payRate, weeklyPay;

 7      workDays = 5;
 8      workHours = 7.5;
 9      payRate = 38.55;
10      weeklyPay = workDays  * workHours * payRate;
11      cout << "Weekly Pay = ";
12      cout << weeklyPay;
13      cout << '\n';
14  }
```

1. The statement #include <iostream> is a preprocessor directive that includes the necessary definitions so that a program can do input and output using the iostream library.
2. The statement using namespace std
3. The main() function can have zero or more parameters. The function may have return type as int or void. All C++ programs must have exactly one main function. Program execution always begins from main.
4. The brace marks the beginning of the body of main.
5. The line defines an integer variable called as workdays.
6. The line defines three float variables – workHours, payRate, weeklyPay
7. The line is an assignment statement. It assigns value 5 to the variable workDays.
8. The line assigns 7.5 to the variable workHours.
9. The line assigns 38.55 to the variable payRate.
10. The line calculates weeklyPay as a product of workdays, workHours and payRate.

10-13. The symbol << is an output operator which takes an output stream as its left operand and an expression as its right operand.
These lines will produce following output:

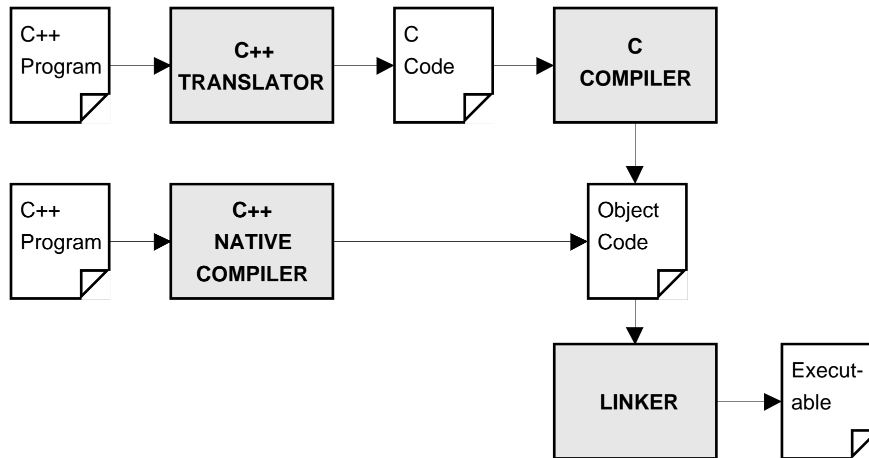
"Weekly Pay = 1445.625"

14. This brace marks the end of the body of main.

Compiling a C++ Program

```
1  $  g++ pay.cpp
2  $  ./a.out
3  Weekly Pay = 1445.625
4  $
```

1.4.1 C++ Compilation Process



Example 2: A simple C++ program to compute average velocity of car.

```
 1  // Program: Compute average velocity of car

 2  #include <iostream>

 3  using namespace std;

 4  int main()
 5  {

 6    cout << "Start Milepost?";
 7    int StartMilePost;
 8    cin >> StartMilePost;

 9    cout << "End Milepost?";
10    int EndMilePost;
11    cin >> EndMilePost;

12    cout << "End time (hours, minutes, seconds)?";
13    int EndHour, EndMinute, EndSecond;

14    cin >> EndHour>>EndMinute>>EndSecond;

15    float ElapsedTime = EndHour + (EndMinute/60.0)+(EndSecond / 3600.0);
16    int Distance =  EndMilePost - StartMilePost;
17    float Velocity = Distance / ElapsedTime;

18    cout << "\nCar traveled " << Distance << " miles in ";
19    cout << EndHour << " hrs " << EndMinute << " min "
20    << EndSecond << " sec\n";

21   cout << "Average velocity was " << Velocity << " mph" << endl;
22   return 0;
23  }
```

```
1  $g++ velocity.cpp
2  $./a.out
3  Start Milepost?100
4
5  End Milepost?1000

6  End time (hours, minutes, seconds)?12 20 30

7  Car traveled 900 miles in 12 hrs 20 min 30 sec
8  $
```

1.5 Type Conversions in C++

Expression Evaluation is the process of applying the operation to the operands. A set of conversions are applied to operands before binary operations are applied. These conversions are called as the unary binary conversions.

Result types for integer binary operations

Type of left operand	Type of right operand			
	char	char	int	long
char	int	int	int	long
short	int	int	int	long
int	int	int	int	long
long	long	long	long	long

Result types for binary floating point operations

Type of left operand	Type of right operand		
	float	double	long double
float	float	double	long double
double	double	double	long double
long double	long double	long double	long double

Result types for mixed mode arithmetic operations

Type of left operand	Type of right operand				
	int	long	float	double	long double
int	int	long	float	double	long double
long	long	long	float	double	long double
float	float	float	float	double	long double
double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double

1.6 Casts

It is possible to force an expression to be of a specific type by using a construct called cast. The general form of this cast is

(type)expression

For example, if you want to make sure the expression (x/2) is evaluated to type float, you can write

(float) x / 2

Example 3: Use of cast in C++.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int i;
6     for(i=1;i<100;i++)
7          cout << i << "  /  2 "  << (float) i / 2 << '\n' ;
8     return 0;
9 }
```

Compile above program using g++ and run.

NOTE:

- C++ defines five casting operators. The first is the traditional – style cast inherited from C. The remaining four casting operators are dynamic_cast, const_cast, reinterpret_cast, and static_cast.
- Another feature of C++ is Run- Time Type Identification (RTTI).

1.7 Reference Variables, Call by Reference and Return by Reference

1.7.1 Default method of passing arguments in C++

By default, C++ uses call-by-value method for passing arguments.

Example 4: Call-by-value method for passing arguments.

```
1   #include <iostream>

2   using namespace std;

3   int sqr_it(int x);

4   int main()
5   {
6     int t=10;
7     cout << sqr_it(t) << ' ' << t;
8     return 0;
9   }

10  int sqr_it(int x)
11  {
12    x = x*x;
13    return x;
14  }
```

*What will be the value of t after execution of sqr_it() function?*Does the value of t modified after the execution of sqr_it() function?
(Execute above program and analyze the output.)
Answer:

____

____

1.7.2 Use of pointer to create call-by-reference

Example 5: Call-by-reference method of passing arguments

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

3   // Declare swap() using pointers.

4    void swap(int *x, int *y);

5    int main()
6    {
7      int i, j;
```

```
8      i = 10;
9      j = 20;
10     cout << "Initial values of i and j: ";
11     cout << i << ' ' << j << '\n';
12     swap(&j, &i); // call swap() with addresses of i and j
13     cout << "Swapped values of i and j: ";
14     cout << i << ' ' << j << '\n';
15     return 0;
16   }

17  // Exchange arguments.

18   void swap(int *x, int *y)
19   {
20     int temp;
21     temp = *x; // save the value at address x
22     *x = *y;   // put y into x
23     *y = temp; // put x into y
24   }
```

What will be the values of j and i after execution of swap() function? Does the values of j and i are modified after execution of swap() function?
(Execute above program and analyze the output.)
Answer:

1.7.3 Reference Parameters

In C++, it is possible to tell the compiler to automatically use call-by-reference rather than call-by-value for one or more parameters of the particular function. You can accomplish this with a reference parameter.

Example 6: Use of reference parameters.

```
1    //Using a reference parameter.

2    #include <iostream>
3    using namespace std;

4    void f(int &i);

5    int main()
6    {
7      int val = 1;
8      cout << "Old value for val: " << val << '\n';
9      f(val); // pass address of val to f()

10     cout << "New value for val: " << val << '\n';
```

```
11      return 0;
12    }
13    void f(int &i)
14    {
15         i = 10; // this modifies calling argument
16    }
```

What will be the value of variable **val** *after execution of function f()?*
(Execute above program and analyze the output.)
Answer:

```
1  #include <iostream>
2  using namespace std;
3  // Declare swap() using reference parameters.
4  // void swap(int &x, int &y);
5  int main()
6  {
7   int i, j;
8   i = 10;
9   j = 20;
10  cout << "Initial values of i and j: ";
11  cout << i << ' ' << j << '\n';
12  swap(j, i);
13  cout << "Swapped values of i and j: ";
14  cout << i << ' ' << j << '\n';
15  return 0;
16 }
```

```
17 /* Here, swap() is defined as using call-by-reference,
18 not call-by-value. Thus, it can exchange the two
19 arguments it is called with.*/
20 void swap(int &x, int &y)
21 {
22  int temp;
23  temp = x; // save the value at address x
24  x = y;    // put y into x
25  y = temp; // put x into y
26 }
```

What will be the values of j and i after execution of swap() function? Does the values of j and i are modified after execution of swap() function?
(Execute above program and analyze the output.)
Answer:

1.7.4 Returning References

Example 7: Returning a reference

```
1  // Returning a reference.
2    #include <iostream>
3    using namespace std;
4    double &f(double&);

5    int main()
6    {
7       double val = 100.0;
8       double newval;
9       cout << f() << '\n'; // display val's value
```

10	`    newval = f(); // assign value of val to newval`
11	`    cout << newval << '\n'; // display newval's value`
12	`    f() = 99.1; // change val's value`
13	`    cout << f() << '\n'; // display val's new value`
14	`    return 0;`
15	`}`
16	`double &f(double &val)`
17	`{`
18	`    return val; // return reference to val`
19	`}`
	Does the value of variable **val** *is modified? How?* **(***Execute above program and analyze the output.)* *Answer:* ___

Programming Tips

- ✓ Remember to #include the headers for the facilities you use.
- ✓ Keep common and local names short, and keep uncommon and nonlocal names longer.
- ✓ Avoid similar looking names.
- ✓ Maintain a common naming style.
- ✓ Choose names to reflect meaning rather than implementations.
- ✓ Avoid complicated expressions.
- ✓ Don't declare a variable until you have a value to initialize it with.
- ✓ Don't return pointers or references to local variables.
- ✓ There are some restrictions that apply to reference variables:
  1. You cannot reference a reference variable
  2. You cannot create arrays of references.
  3. You cannot create a pointer to a reference. That is, you cannot apply the & operator to a reference.
  4. References are not allowed on bit fields.

Practical Assignment

SET A

1. Write a program that prompts for a person's age and heart rate per minute. The program computes and displays the number of heart beats since the person was born. You may assume that there are 365 days in a year.

2. Write a program that prints the sizes of the fundamental types, a few pointer types, and a few enumerations of your choice. Use the sizeof operator.

SET B

1. Write a function that counts the number of occurrences of a pair of letters in a string.
The prototype of the function is as follows:

```
int occurrences(char *str);
```

2. Write a function int my_atoi(const char *) that takes a string containing digits and returns the corresponding integer.

SET C

1. Write a function my_itoa(int i, char b[]) that creates a string representation of integer i in array b and returns array b.

2. Write a function named TimeToSeconds() that takes a single argument that represents a time. The time is a string with the format hh:mm:ss, where hh is hours, mm is minutes, and ss is seconds. Function TimeToSeconds() returns the equivalent time in seconds. It TimeToSeconds() receives an invalid time (e.g. 02:23:67), it should use assert() to report an error. (Please note assert() is a standard micro available in C++).

Assignment Evaluation

0: Not Done [ ]	1: Incomplete [ ]	2.Late Complete [ ]
3: Needs Improvement [ ]	4: Complete [ ]	5: Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 2
Functions In C++

Start Date ____/____/______

Objectives

To learn about:

11. Function overloading
12. default arguments
13. Inline function

Reading

You should read following topics before starting this exercise:

1. Concept of function overloading in C++
2. Default arguments in C++
3. Inline functions

Ready References

2.1 Function Overloading

In C++, two or more functions can share the same name, as long as their parameter declarations are different.

Example 1: Function overloading in C++.

```
1  // Overload a function three times.

2  #include <iostream>
3  using namespace std;

4  void f(int i);          // integer parameter
5  void f(int i, int j); // two integer parameters
6  void f(double k);       // one double parameter

7  int main()
8  {
9    f(10);      // call f(int)
10   f(10, 20); // call f(int, int)
11   f(12.23);   // call f(double)
12   return 0;
13 }

14 void f(int i)
15 {
16   cout << "In f(int), i is " << i << '\n';
17 }

18 void f(int i, int j)
19 {
20   cout << "In f(int, int), i is " << i;
21   cout << ", j is " << j << '\n';
22 }

23 void f(double k)
24 {
25   cout << "In f(double), k is " << k << '\n';
26 }
```

This program produces the following output:

In f(int), i is 10

In f(int, int), i is 10, j is 20

In f(double), k is 12.23

Example 2: Function overloading in C++.

```
1  // Create an overloaded version of abs() called myabs().

2  #include <iostream>
3  using namespace std;

4  // myabs() is overloaded three ways.
5  int myabs(int i);
6  double myabs(double d);
7  long myabs(long l);

8  int main()
9  {
10   cout << myabs(-10) << "\n";
11   cout << myabs(-11.0) << "\n";
12   cout << myabs(-9L) << "\n";
13   return 0;
14 }

15 int myabs(int i)
16 {
17   cout << "Using integer myabs(): ";
18   if(i<0) return -i;
19   else return i;
20 }

21 double myabs(double d)
22 {
23   cout << "Using double myabs(): ";
24   if(d<0.0) return -d;
25   else return d;
26 }

27 long myabs(long l)
28 {
29   cout << "Using long myabs(): ";
30   if(l<0) return -l;
31   else return l;
32 }
```

This program produces the following output:

Using integer myabs(): 10

Using double myabs(): 11

Using long myabs(): 9

2.2. Default Arguments

In C++, you can give a parameter a default value that is automatically used when no argument corresponding to that parameter is specified in a call to a function.

Example 3: Default arguments

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

3  void clrscr(int size=25);
4  int main()
5  {
6    int i;

7    for(i=0; i<30; i++ ) cout << i << '\n';
8    clrscr(); // clears 25 lines
9    for(i=0; i<30; i++ ) cout << i << '\n';
10   clrscr(10); // clears 10 lines
11   return 0;
12 }

13 void clrscr(int size)
14 {
15    for(; size; size--) cout << '\n';
16 }
```

As this program illustrates, when the default value is appropriate to the situation, no argument need be specified when calling clrscr( ). However, it is still possible to override the default and give size a different value.

2.3 Inline Functions

An inline function is a function whose code is expanded in line at the point at which it is invoked, rather than being called.
There are two ways to create an inline function.
The first is to use the inline modifier.

```
inline int f()
{
        ..........
}
```

There is another way to create an inline function. This is accomplished by defining the code to a member function inside a class declaration.

Example 4: Use of Inline Function.

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

3  inline int max(int a, int b)
4  {
5    return a>b ? a : b;
6  }

7  int main()
8  {
9    cout << max(10, 20);
10   cout << " " << max(99, 88);
11   return 0;
12 }
```

```
As far as the compiler is concerned, the preceding program is
equivalent to
#include <iostream>

using namespace std;

int main()

{

  cout << (10>20 ? 10 : 20);

  cout << " " << (99>88 ? 99 : 88);

  return 0;

}
```

Programming Tips

- ✓ Use overloading when functions perform conceptually the same task on different types.
- ✓ When overloading on integers, provide enough functions to eliminate common ambiguities.
- ✓ Default arguments may be provided for trailing arguments only.
- ✓ The small, frequently used functions must be defined as inline functions.

Practical Assignment

SET A

1. Write a C++ program to accept records of n employees and store it in array. Use array of structures. The structure of employee should be as follows:

```
struct  Employee
{
  int EmpNo;
 char Name[20];
 char Gender;
```

```
 long                                                    Salary;
};
```

Implement following search functions:
int search(struct Employee arr[], int EmpNo);
int search(struct Employee arr[], char Name[]);
int search(struct Employee arr[], char Gender);
int search(struct Employee arr[], long Salary);

Also implement a function to return an employee who is drawing highest salary by using following function:
Inline struct Employee compare(long a, long b);

2. Write a C++ program to sort n elements in ascending order. Use any sorting technique. Function overloading is expected in this program.

SET B

1. Write a C++ program to encrypt data entered by user and display. Also provide a mechanism to decrypt the message. If user tries to re-encrypt the encoded text with the same key, then it will produce original text. (Use bitwise XOR and one character key)

2. Write a C++ program to convert integer number to string and string to integer using function overloading. (It is similar to atoi() and itoa() functions in C.)

SET C

1. Write a C++ program to invert a two-dimenstional array.

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 3
Classes and Objects

Start Date ____/____/______

Objectives

To learn about:
- Class Declaration
- Defining Data Members And Member Functions
- Creating Objects
- Array of Objects
- Access Specifiers – public, private, protected
- Static Class Members
- Friend Functions
- const Member Functions

Reading

You should read following topics before starting this exercise:
- Concept of Class in C++
- Types of Member Functions in Class
- Creating Objects
- Array of Objects
- Access Specifiers
- Static Class Member Functions
- Friend Class and Friend Functions
- const Member Functions

Ready References

3.1 Classes in C++

A class is a way to bind the data and its associated functions together. It allows the data(and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built in type. Generally , a class specifications has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.
The general form of a class declaration is:

```
class class_name
{
        private:
                variable declaration;
                function declaration;
        public:
```

```
                variable declaration;
                function declaration;
};
```

Example:

```
class item
{
        int number;             // variables declaration
        float cost;             // private by default
    public:
        void getdata(int a, float b);   // functions declaration
        void putdata(void);             // using prototype
};
```

Some Characteristics:

1. The class declaration is similar to a struct declaration.
2. The keyword class specifies that what follows is an abstract data of type class_name. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions.
3. These functions and variables are collectively called class members.
4. These are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords public and private are known as visibility labels.

3.2 Objects in C++

You should remember that the declaration of item as shown above does not define any objects of item but only specifies what they will contain. Once a class has been declared, we can create variables of that type by using the class name(like any other built-in type variable). For example,

```
item x;         // memory for x is created
```

creates a variable x of type item. In C++, the class variables are known as Objects. Therefore, x is called an object of type item. We may also declare more then one object in one statement.
Example:

```
item x,y,z;
```

Some Characteristics:

- The declaration of object is similar to that of a variable of any basic type.
- The necessary memory space is allocated to an object at this stage. Note : that class specification, like a structure, provides only a template and does not create any memory space for the objects.

3.3 Defining Data Members and Member Functions

The variables declared inside the class are known as data members and the functions known as member functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed form outside the class. The binding of data and functions together into a single class-type variable is referred to as Encapsulation.

Example:

```
class employee
{
        int age;
        char name[20];
    public:
        void getdata();
        void putdata(void);
};
```

Some Characteristics:

We usually give a class some meaningful name, such as employee. The name now becomes new type identifier that can be used to declare instances of that class type. The class Employee contains two data members and two function members. The data members are private by default while both the functions are public by declaration.

The function getdata() can be used to assign values for member variables age, name, and putdata() for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of class employee.

3.4 Array of Objects

We know that an array can be of any data type including struct. Similarly, we can also have arrays of variables that are of the type class. Such variables are called arrays of objects. Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
    public:
        void getdata(void);
        void putdata(void);
};
```

The identifier employee is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];        // array of manager
employee foreman[15];       // array of foreman
employee worker[75];        // array of worker
```

The array manager contains three objects(managers), namely, manager[0], manager[1] and manager[2], of type employee class. Similarly, the foreman array contains 15 objects(foreman) and the worker array contains 75 objects(workers).
Since an array of objects behaves like any other array, we can use the usual array accessing methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

will display the data of the ith element of the array manager. That is, this statement requests the object manager[i] to invoke the member function putdata().

Some Characteristics:

1. An array of objects is stored inside the memory in the same way as a multi-dimensional array.
2. Note that only the space for data items of the objects is created. Member functions are stored separately an will be used by all the objects.

3.5 Constant Member Functions

There are only two kinds of class member functions- those that make modifications to the state of an object , and those that merely report on the state. The latter kind of function is called an accessor function. In other words, these functions merely access the data, as opposed to doing any modification.

The C++ compiler has no inherent way to distinguish a mutator function from an accessor function, but you can make this difference explicit. To make a member function into accessor function, append the keyword const after the formal argument list in both the definition and declaration (if present).

Example 1:

```
#include<iostream>
class integer
{
      private:
            int number;
      public:
            void store(int);
            int get() const;
};
inline void integer::store(int n)
{
      number = n;
}
inline int integer:: get() const
{
      return number;
}
```

```
int main()
{
      // Define an instance of the class
      integer in;
      // Store a number
      in.store(5);
      // Retrieve the number and print it
      printf("%d\n", in.get());
      return 0;
}
```

```
The output is:
     5
```

Now let's modify the function integer::get() so that one is added to the data member number before it is returned.

```
inline int integer::get() const
{
      // Cannot modify number
      return ++number;
}
```

The compiler error message is:

Cannot modify a const object in function integer::get() const

Some Characteristics:

1. The use of the const keyword is valid only for nonstatic class member functions. In other words, you cannot append the const keyword after a global function or a static member functions.

3.6 Access Specifiers – Private, Public and Protected

The principle of data hiding and the preceding scenarios, are all very nice, except that inherently it means nothing to the compiler. In other words, you are responsible for explicitly telling the compiler which class members obey the principle of data hiding, and which do not. This is done by using access specifiers within the class definition.

There are three different access specifiers:

- ✓ Private
- ✓ Public
- ✓ Protected

The second access specifier is called public. Unlike a class, this is the default for a structure definition in order to be compatible with a C program that is compiled using C++. The rule of public class members is simple: they may be accessed by both member and non-member

functions. This is the means by which you can communicate with a class – by sending messages to the public member functions.
To write an access specifier, within the class definition use the appropriate keyword followed by a colon before the class members to which the specifier applies. For Example, we can expand the definition of the integer class to include the access specifiers:

```
class integer
{
        private:            // optional
        // All private members here
        public:
        // All public members here
};
```

3.7 Static Members – Variables and Functions

Static Variables

You already know that each object has its own set of member variables. However, in some situations, it is preferable to have one or more variables that are common for all objects. C++ provides static variable for this purpose.

As its name suggests, static variables retain their value even after the function to which it belongs has been executed. This means that a static variable retains its value throughout the program.

NOTE:
The static variables have to be initialized explicitly either inside the class declaration or outside the class.

Static variables are initialized outside the member functions or class definition. The following example illustrates the declaration an initialization of a static variable.

EXAMPLE:

```
class staticExample
{
        int data;
        static int staticVar;  // Static variable declared
        public:
          // Defination of member function
};
int staticExample :: staticVar=0;  //Static variable initialized to 0
```

in the above example , static variable is initialized outside the class definition. Creating more then one object will not re-initialize the static variable. Unlike member variables, only one copy of static variable exists in memory for all objects of that class. Thus, all objects share one copy of the static variable in memory. Consider a situation where you are required to keep track of the number of objects created for a given class. If you declare a non-static member variable called counter, every instance of that class will create a separate variable, and there

would be no single variable having a count of the total number of object created. To avoid this, you use static member variables because it is this single static variable that gets incremented by all objects.

You can also initialize the static variable inside the class definition as shown in the following example.

Example 2:

```
#include <iostream>

class myclass
{
  void increment()
  {
    static int i=5;
    cout<<i<<endl;
    i++;
  }

  public:
  void display()
  {
    cout<<"calling increment for first time"<<endl;
    increment();
    cout<<"calling increment for second time"<<endl;
    increment();
  }
};

int main()
{
   myclass m1;
   m1.display();
   return 0;
}
```

Compile and run this program to see the output.

Static Functions

Like static variables, you can also declare a function to be static. Static functions can only access static variables and not non-static variables. You use the class name to access the static function, as shown below:

ClassName :: StaticFunctionName;

The following code samples illustrates the use of static functions:

Example 3:

```
class staticExample
{
        int data;
        static int staticVar;  // Static variable declared
      public:
            static void display()
            {
                cout<<"staticVar= "<< staticVar";
                cout<<"data= "<<data;  // Error! Static functions
                                       //  cannot access non-
                                       //  static variables.

            }
};
Int staticExample :: staticVar=0;  // Static variable initialized to 0
 void main()
 {
   staticExample:: display();  // Without creating an object of a class, the
                               // static function can be accessed
 }
```

Compile and run this program to see the output.

In a situation where you want to check whether or not any object of a class has been created, you make use of static functions because they come into existence even before the object is created.

Example 4: You have defined a Dealer class as part of devoluping the billing system software for Diaz Telecommunications, Inc. The class, which you've defined, is as follows:

```
class Dealer
{
       private:
                  char mobileNo[11];
                  char dealerName[25];
                  char dealerAddress[51];
                  char  dealerCity[25];
                  char phoneNo[11];
       public:
                  static int CompanyID;
                  static void showed()
                  {
                          cout<<"The dealer name is"<<dealername;
                          cout<<"The company ID is "<<CompanyID;
                  }
                  void get()
                  {
                           // Code to accept the dealer details
```

```
                    }
                    void print()
                    {
                            // Code to print the dealer deatails
                    }
};
int CompanyID=6519;
```

The Dealer class definition shown above is incorrect. Identify the errors in the Dealer class and write a correct definition for it.

3.8 Friend Functions and Classes

Friend functions

If a member of a particular class is private or protected, functions outside the class cannot access the non-public members of the class. This is the primary objective of encapsulation. However, at times, it becomes a problem for the programmer. In order to access the non-public members of a class, C++ provides the friend keyword.

Any non-member function may be declared a friend by a class, in which case the function may directly access the private member attributes and methods of the class objects.
For example:

```
class z
{
        friend void fn();
        private:
                int a;
};
void fn()
{
        z one;
        one.a=5;    // Okay, as fn() is a friend
}
void fn1()
{
        z one;
        one.a=5;    // Wrong as fn1() is not friend of 'z'
}
```

Note that 'fn()' is a global (non-member) function that has been declared a friend. Member functions of other classes may also be declared as friends by declaring the class as a friend class.

Some Characteristics:

1) friend functions are used when the design of a class requires that another class or non-member function should access the private elements of the class. In such a case, friend function declarations may be used to enable the class to access those variables directly.
2) Friend functions may also be inline functions. That is, if the friend function is defined within the scope of class definition, then the inline request to the compiler is automatically made. If the friend function is defined outside the class definition, then you must precede the return type with the keyword inline in order to make the request.
3) Function operating on objects of two different classes. This is the ideal situation where the friend function can be used to bridge two classes.
4) Friend functions can be used to increase the versatility of overloaded operators.
5) Sometimes, a friend allows a more obvious syntax for calling a function, rather then what a member function can do.

Friend Classes

Objectives: Concept
Need.

Reading:

Just as a function can be made a friend of a class, a class can also be made a friend of another class.

Example:

```
class distance
{
        ………..
        friend class length;
        ………..
};
```

In this example, the length class is the friend of the class distance. All the members of the class distance can now be accessed from the length class. The program below illustrates an entire class declared as a friend of another class.

Example 5:

```
# include<iostream>
class class1
{
   private:
            int i;
   public:
            friend class class2;
};
class class2
{
   public:
        void func()
        {
          class1 A1;
```

```
          // The function can access the private data of class1

          cout<<"The value of i in class class1 is  "<<A1.i<<endl;
        }
};
int main()
{
      class2 B1;
      B1.func();
      return 0;
}
```

In the preceding program, the class2 class is a friend of the class1 class. Now, all the member functions of class2 can access the non-public data of class1.

Compile and run this program to see the output.

Some characteristics:

1) Friendship is not mutual by default. That is, once class B is declared as a friend of class A, this does not give class A the right to access the private members of the class B.
2) Friendship, when applied to program design, is an escape mechanism which creates exceptions to the rule of data hiding. Usage of friend classes should, therefore, be limited to those cases where it is absolutely essential.

3.9 const Member Functions

There are only two kinds of class member functions- those that make modifications to the state of an object , and those that merely report on the state. The latter kind of function is called an accessor function. In other words, these functions merely access the data, as opposed to doing any modification.
The C++ compiler has no inherent way to distinguish a mutator function from an accessor function, but you can make this difference explicit. To make a member function into accessor function, append the keyword const after the formal argument list in both the definition and declaration (if present).

Example 6:

```
#include<iostream>
      class integer
      {
            private:
                  int number;
            public:
                  void store(int);
                  int get() const;
      };
      inline void integer::store(int n)
      {
            number = n;
      }
```

```
      inline int integer:: get() const
      {
            return number;
      }
      int main()
      {
            // Define an instance of the class
            integer in;
            // Store a number
            in.store(5);
                              // Retrieve the number and print it
            printf("%d\n", in.get());
            return 0;
      }
The output is:
      5
```

Now let's modify the function integer::get() so that one is added to the data member number before it is returned.

```
inline int integer::get() const
{
      // Cannot modify number
      return ++number;
}
```

The compiler error message is:

Cannot modify a const object in function integer::get() const

Some Characteristics:

1. The use of the const keyword is valid only for nonstatic class member functions. In other words, you cannot append the const keyword after a global function or a static member functions.

Programming Tips

- ✓ Make a function member only if it requires direct access to the representation of a class.
- ✓ Make a member function that doesn't modify the value of its object a const member function.
- ✓ Make a function that needs access to the representation of a class but need not be called for a specific object a static member function.
- ✓ If a class has a pointer member, it needs copy operations (copy constructor and copy assignment).
- ✓ Use enumerators when you need to define integer constants in class declarations.
- ✓ Remember temporary objects are destroyed at the end of the full expression in which they are created.

Practical Assignment

SET A

1. Define a class to represent a bank account. Include the following members:

   Data members
   1) Name of the depositor
   2) Account number
   3) Type of account
   4) Balance amount in the account.

   Member functions:
   1) To assign initial values
   2) To deposit an amount
   3) To withdraw an amount after checking the balance
   4) To display name and balance.

   Write a main program to test the program.

2. Define class Date which will provide operations like adding day, month or year to existing date. Also provide function to check for the leap year.

SET B

1. Write a class to represent a vector. Include member functions to perform the following tasks:
   1) To create the vector
   2) To modify the value of given element.
   3) To multiply by a scalar value
   4) To display the vector in the form(10,20,30,…)

2. Create a Hen class. Inside this, nest a Nest class. Inside Nest, place an Egg class. Each class should have a display( ) member function. In main( ), create an instance of Hen class and call the display( ) function for each one.

SET C

1. Define a class named Sequence for storing sorted strings. Following are the member functions for Sequence class:
   - Insert which inserts a new string into its sort position.
   - Delete which deletes an existing string.
   - Find which searches the sequence for a given string and returns true if it finds it, and false otherwise.
   - Print which prints the sequence strings.

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion

____/____/______

SESSION 4
Constructors and Destructors

Start Date ____/____/______

Objectives

To learn about:

- Concept of constructor
- Need of constructors
- Types of constructors
- Concept of destructor
- Need of destructors
- Use of new and delete operator

Reading

You should read following topics before starting this exercise:

- What is constructor?
- Role of constructor in class
- Types of constructor
- Role of destructor and how it is different from constructor?

Ready References

4.1Constructors

A constructor is special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name.

Some characteristics

1. They should be declared in public section.
2. They are invoked automatically when the objects are created.
3. They do not have return types, not even the void and therefore, they cannot return values.
4. We cannot refer to their addresses.

Example 1:

```
class test
{
        int m, n;
        public:
        test(void);             //constructor
        {
                m=0;
                n=0;
        }
                        --------
                        --------
};
```

When we create object for above class test, it will get initialized automatically.
Ex. Test obj;
It will create object and initializes data members m and n to zero.

4.2 Types of constructors

a. Default Constructor

It does not accept any parameters. Purpose of this, is only to create object and initialize it to the values given.

b. Parameterized Constructor

It accepts any number of formal parameters and uses these parameters to initialize the objects.

Passing of parameters can be done in two ways:

a. Call a constructor explicitly

    test obj = new test(1,100);

b. Call a constructor implicitly

    Test obj(1,100);

c. Overloaded Constructor

If we want to declare two objects, with one object we pass the arguments and second does not have any arguments. Once we define a constructor we must define the "do-nothing" constructor.

d. Constructor with default arguments

It is possible to define constructors with default arguments. When we specify actual parameters, they override the default value.

Example 2:

```
class test
{
   int x,y;
   public:
         void test()
         {
            cout<< "You are in default constructor…..\n";
         }
         void test(int m, int n)
         {
             x=m;
             y=n;
         }
         void test(float m, float n=5.3)
         {
               x=m;
               y=n;
               cout<<x<<y;
         }
};
void main()
{
   test obj1;
   test obj2(2,3);
   test obj3(4.4, 5.4);
}
```

Compile and run this program to see the output.

e. Copy Constructor

It takes a reference to an object of the same class as itself as argument. It copies data from one object to other. It would not create an object as it is just an assignment statement.

Example 3:

```
class test
{
      private:
            int k, m;
      public:
            test()
            {
                  k=15; m=18;
            }
            test(test & s)
            {
                  k=s.k;     m=s.m;
            }
            void display()
            {
```

```
                        Cout<<k<<m;
                }
        };
        void  main()
        {
                test obj1;
                test obj2(obj1);
                test obj3=obj1;
                obj1.display();
                obj2.display();
                obj3.display();
        }
```

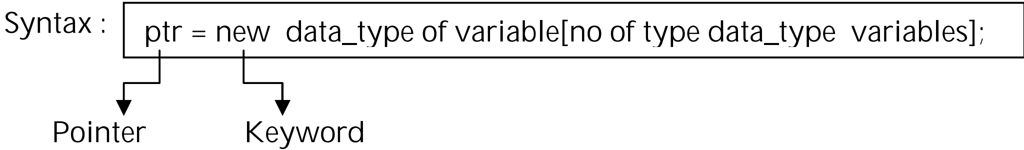
```
        Output:

                15   18          // for obj1
                15   18          // for obj2
                15   18          // for obj3
```

f. Dynamic Constructor

This type of constructor can also be used to allocate memory while creating objects. Two operators new and delete are used in dynamic memory allocation.

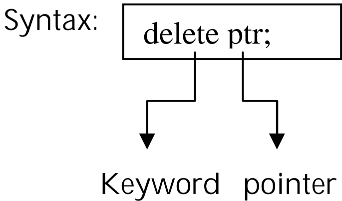
Operator 'new'

Syntax :



Operator 'delete'

Syntax:



Example 4:

```
class  test
{
      private:
            int x;
      public:
            void getdata()
            {
                  cout<<"Enter value:\n";
                  cin>>x;
            }
            void display()
            {
                  cout<<"x="<<x;
            }

};
void main()
{
            test *ptr;
            ptr=new test;
            ptr->getdata(); ptr->display();
            delete ptr;
}
```

Compile and run this program to see the output.

4.3 Destructors

Destructor is a special member function which is called automatically whenever a class object is destroyed.

Whenever any object goes out of scope of its existence, destructor will be called automatically and it takes out the allocated memory.

Some characteristics

- A destructor name must be same as the class name in which it is declared, prefixed or preceded by a symbol tilde(~).
- Does not have any return type.
- Cannot be declared static, constant.
- Declared as a public member function.

```
class test
{
      int m, n;
      public:
            test(void);           //constructor
      {
            m=0;
            n=0;
      }
                  --------
                  --------

      ~test();
};
```

Programming Tips

- ✓ Use a constructor to establish invariant for a class.
- ✓ If a constructor requires a resource, its class needs a destructor to release the resource.
- ✓ If a class has a pointer member or reference member, it needs copy operations (copy constructor, assignment operator).
- ✓ If a class needs copy operation of a destructor, it probably needs a constructor, a destructor, a copy assignment(=) and a copy constructor.
- ✓ When writing a copy constructor, be careful to copy every element that needs to be copied (deep copying). Beware of default initializer.

Practical Assignment

SET A

1. Write a c++ program to read a set of numbers upto n (accepted from the user) and print the contents of the array in the reverse order using dynamic memory allocation operators new and delete.

2. Define a class named Complex for representing complex numbers. A complex number has the general form a + ib, where a is the real part and b is the imaginary part (i stands for imaginary). Complex arithmetic rules are as follows:

   (a + ib) + (c + id) = (a + c) + i(b + d)
   (a + ib) – (c + id) = (a + c) – i(b + d)
   (a + ib) * (c + id) = (ac – bd) + i(bc + ad)

   Define these operations as member functions of Complex.

SET B

1. Write a c++ program to read the information like plant_name, plant_code, plant_type, price.Construct the database with suitable member functions for initializing and for destroying the data, viz. constructor, copy constructor, destructor.

SET C

1. Define class named `BinTree` for storing sorted strings as a binary tree with all necessary functions for total leaves, total nodes, traversing operations.

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 5 Start Date ____/____/______

Operator Overloading

Objectives

To learn about:

- What is operator overloading
- Overloading Unary operators as member function and friend function
- Overloading Binary operators as member function. and friend function
- Usage of this pointer
- Overloading >> and << operator

Reading

You should read following topics before starting the exercise:

- Function Overloading and polymorphism in C++
- Operators in C++ and functions associated with each operator
- Rules for function overloading
- insertion (>>) and extraction(<<) operators
- concept of this pointer in C+

Ready References

5.1 What is Operator Overloading

Operator overloading is used to give special meaning to the commonly used operators (such as +, -, * etc.) with respect to a class. By overloading operators, we can control or define how an operator should operate on data with respect to a class.

5.1.1 Overloadable Operators (refer to first session in the workbook)

<table>
<tr><th>Overloadable operators</th></tr>
<tr><td>+ - * / = < > += -= *= /= << >>
<<= >>= == != <= >= ++ -- % & ^ ! |
~ &= ^= |= && || %= [] () , ->* -> new
delete new[] delete[]</td></tr>
</table>

5.1.2 Operators which cannot be overloaded

Operators which cannot be Overloaded
. .* :: Sizeof ?:

5.1.3 How to Overload Operators

Operators are overloaded in c++ by creating operator functions either as a member or as non member functions which are Friend Function of a class.

Operator functions are declared using the following general form:

```
ret-type operator#(arg-list);
```

and then defining it as a normal member function.

Here, ret-type is commonly the name of the class itself as the operations would commonly return data (object) of that class type.

is replaced by any valid operator such as +, -, *, /, ++, -- etc.

5.1.4 Overloading Unary Operators

You overload a unary operator with either a member function that has no parameters, or a nonmember function (declared as friend function within the class) that has one parameter. Suppose a unary operator # is called with the statement #t, where t is an object of type T.

A member function that overloads this operator would have the following form:

```
return_type operator#()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator#(T)
```

eg. Sample program 1

5.1.5 Overloading Binary Operators

You overload a binary unary operator with either a member function that has one parameter, or a nonmember function(declared as friend function within the class) that has two parameters. Suppose a binary operator # is called with the statement t # u, where t is an object of type T, and u is an object of type U.

A member function that overloads this operator would have the following form:

```
return_type operator#(T)
```

A nonmember function that overloads the same operator would have the following form:

return_type operator#(T, U)

An overloaded binary operator may return any type.

Eg. Sample program 1

5.2 The keyword this

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.

One of its uses can be to check if a parameter passed to a member function is the object itself.

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects).

Example 1:

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

class MyClass {
  int x, y;

public:
  MyClass() {
     x=0;
     y=0;
  }

  MyClass(int i, int j) {
     x=i;
     y=j;
  }


  //unary operators as friend function
  friend MyClass operator--(MyClass &ob);              // prefix
  friend MyClass operator--(MyClass &ob, int notused); // postfix
  //binary operators as friend function
  MyClass operator++();


  //binary operators
  MyClass operator+(MyClass op2);
  MyClass operator=(MyClass op2);
  MyClass operator*(MyClass object2);
  MyClass operator/(MyClass object2);


  //overloading >> and >>
```

```
  friend ostream &operator<<(ostream &stream, MyClass ob);
  friend istream &operator>>(istream &stream, MyClass &ob);



  void show() ;



};

// Overload -- (prefix) for MyClass class using a friend.
MyClass operator--(MyClass &ob)
{
  ob.x--;
  ob.y--;
  return ob;
}


// Overload -- (postfix) for MyClass class using a friend.
MyClass operator--(MyClass &ob, int notused)
{
  ob.x--;
  ob.y--;
  return ob;
}

// Overload + relative to MyClass class.
MyClass MyClass::operator+(MyClass op2)
{
  MyClass temp;

  temp.x = x + op2.x;
  temp.y = y + op2.y;

  return temp;
}

// Overload = relative to MyClass class.
MyClass MyClass::operator=(MyClass op2)
{
  x = op2.x; // These are integer assignments
  y = op2.y; // and the = retains its original
            // meaning relative to them.
  return *this;
}

// Overload ++ relative to MyClass class.
MyClass MyClass::operator++()
{
  x++; // increment x and y
  y++;
    return *this;
}
```

```
// Overload * relative to MyClass class.
MyClass MyClass::operator*(MyClass object2)
{
  MyClass temp;

  temp.x = x * object2.x;
  temp.y = y * object2.y;

  return temp;
}

// Overload / relative to MyClass class.
MyClass MyClass::operator/(MyClass object2)
{
  MyClass temp;

  temp.x = x / object2.x;
  temp.y = y / object2.y;

  return temp;
}
// Overload << relative to MyClass class.

ostream &operator<<(ostream &stream, MyClass ob)
{
  stream << ob.x << ' ' << ob.y << '\n';

  return stream;
}
// Overload >> relative to MyClass class.

istream &operator>>(istream &stream, MyClass &ob)
{
  stream >> ob.x >> ob.y;

  return stream;
}


void MyClass::show()
{
  cout << x << ", ";
  cout << y << ", ";

}



int main()
{
  MyClass object1(10, 10);
  int x, y;

  --object1; // decrement object1 an object
```

```
  cout << "(--object1) : " << endl;
  object1.show;

  object1--; // decrement object1 an object
  cout << "(object1--) : " << endl;
  object1.show;


  MyClass a(1, 2), b(10, 10), c;
  a.show();
  b.show();

  c = a + b;
  c.show();

  c = a + b + c;
  c.show();

  c = b = a;
  c.show();
  b.show();

  ++c;
  c.show();



  MyClass object1(10, 10), object2(5, 3), object3;

  object3 = object1 * object2;
  cout << "(object1*object2) : " << object3<< endl;


  object3 = object1 / object2;
  cout << "(object1/object2) : " << object3<<endl;



 return 0;

  }
```

Compile and Run the program to see the output.

5.3 Overloading Insertion(<<) and Extraction (>>) operator

Overloading <<

For classes that have multiple member variables, printing each of the individual variables on the screen can get tiresome fast.

It would be much easier if you could simply type:

```
Point cPoint(5.0, 6.0, 7.0);
cout << cPoint;
```

Consider the expression cout << cPoint.

If the operator is <<, what are the operands?

The left operand is the cout object, and the right operand is your Point class object. cout is actually an object of type ostream. Therefore, our overloaded function will look like this:

```
friend ostream& operator<< (ostream &out, Point &cPoint);
```

Overloading <<

It is also possible to overload the input operator. cin is an object of type istream.

Therefore, our overloaded function will look like this:

```
friend istream& operator>> (istream &in, Point &cPoint)
```

Sample program Example 2 using both the overloaded operator<< and operator>>:

Example 2

```
// A Simple Example of a overloaded operator << and operator>>

#include <iostream>
using namespace std;

class Point
{
private:
   double m_dX, m_dY, m_dZ;

public:
   Point(double dX=0.0, double dY=0.0, double dZ=0.0)
   {
   m_dX = dX;
   m_dY = dY;
```

```
    m_dZ = dZ;
    }

    friend ostream& operator<< (ostream &out, Point &cPoint);
    friend istream& operator>> (istream &in, Point &cPoint);

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can access
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}

istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}

int main()
{
    using namespace std;
    cout << "Enter a point: " << endl;

    Point cPoint;
    cin >> cPoint;

    cout << "You entered: " << cPoint << endl;

    return 0;
}
```

Compile and run the program to see the output.

Programming Tips

- ✓ You cannot define new operators, such as **.
- ✓ You cannot redefine the meaning of operators when applied to built-in data types.
- ✓ Overloaded operators must either be a nonstatic class member function or a global function. A global function that needs access to private or protected class members must be declared as a friend of that class. A global function must take at least one argument that is of class or enumerated type or that is a reference to a class or enumerated type.
- ✓ Operators obey the precedence, grouping, and number of operands dictated by their typical use with built-in types.
- ✓ Unary operators declared as member functions take no arguments; if declared as global functions, they take one argument.
- ✓ Binary operators declared as member functions take one argument; if declared as global functions, they take two arguments.
- ✓ If an operator can be used as either a unary or a binary operator (&, *, +, and -), you can overload each use separately.
- ✓ Overloaded operators cannot have default arguments.

Practical Assignment

SET A

1. Create a class Float that contains one float data member . Overload the 4 arithmetic operators and unary +,- on the objects of Float.Also overload insertion and extraction operators.

2. Create a class Rational to represent a Rational number. Perform the Basic Arithematic operation : Addition, Subtraction, Multiplication and Division for Two Rational Numbers.

SET B

1. Create a class String to represent a string. Overload the + operator to concatenate 2 strings, <=, == , >= to compare 2 strings.

2. Create a class Fraction to represent a fraction of the type 2/5. Overload the 4 arithmetic operators and <,>,!=,++,-- on the objects of Fraction.

3. Complete the implementation of the following `String` class. Note that two versions of the constructor and = are required, one for initializing/assigning to a `String` using a `char*`, and one for memberwise initialization/assignment. Operator `[]` should index a string character using its position. Operator + should concatenate two strings.

```
class String {
public:
                String      (const char*);
                String      (const String&);
                String      (const short);
                ~String     (void);

    String&     operator =  (const char*);
    String&     operator =  (const String&);
    char&       operator [] (const short);
    int         Length      (void)      {return(len);}
friend String    operator + (const String&, const String&);
friend ostream& operator <<(ostream&, String&);

private:
    char        *chars;     // string characters
    short       len;    // length of string
};
```

SET C

1. Create a class Matrix of size m*n to represent a matrix. Define all possible matrix operations for Matrix objects eg +,-,*,scaling.

2. Write a program to overload the new and delete operator

3. Sparse matrices are used in a number of numerical methods (e.g., finite element analysis). A sparse matrix is one which has the great majority of its elements set to zero. In practice, sparse matrices of sizes up to 500 × 500 are not uncommon. On a machine which uses a 64-bit representation for reals, storing such a matrix as an array would require 2 megabytes of storage. A more economic representation would record only nonzero elements together with their positions in the matrix. Define a SparseMatrix class which uses a linked-list to record only nonzero elements, and overload the +, -, and * operators for it. Also define an appropriate memberwise initialization constructor and memberwise assignment operator for the class.

4. A bit vector is a vector with binary elements, that is, each element is either a 0 or a 1. Small bit vectors are conveniently represented by unsigned integers. For example, an `unsigned char` can represent a bit vector of 8 elements. Larger bit vectors can be defined as arrays of such smaller bit vectors. Complete the implementation of the `Bitvec` class, as defined below. It should allow bit vectors of any size to be created and manipulated using the associated operators.

```
enum Bool {false, true};
typedef unsigned char uchar;

class BitVec {
public:
            BitVec       (const short dim);
            BitVec       (const char* bits);
            BitVec       (const BitVec&);
            ~BitVec      (void)       { delete vec; }
    BitVec& operator =      (const BitVec&);
    BitVec& operator &=  (const BitVec&);
    BitVec& operator |=  (const BitVec&);
    BitVec& operator ^=  (const BitVec&);
```

```
    BitVec& operator <<= (const short);
    BitVec& operator >>= (const short);
    int     operator []   (const short idx);
    void    Set           (const short idx);
    void    Reset         (const short idx);

    BitVec  operator ~        (void);
    BitVec  operator &        (const BitVec&);
    BitVec  operator |        (const BitVec&);
    BitVec  operator ^        (const BitVec&);
    BitVec  operator <<   (const short n);
    BitVec  operator >>   (const short n);
    Bool    operator ==   (const BitVec&);
    Bool    operator !=       (const BitVec&);
friendostream& operator << (ostream&, BitVec&);

private:
    uchar *vec;          // vector of 8*bytes bits
    short  bytes;        // bytes in the vector
};
```

Assignment Evaluation

0: Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]

3:Needs Improvement [ ]   4:Complete [ ]   5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 6
Inheritance

Start Date ____/____/______

Objectives

To learn about:

- To know about inheritance.
- Types of inheritance.
- To know about virtual base class.
- To know about abstract class.
- To know about virtual and pure virtual functions.

Reading

You should read following topics before starting this exercise:

- concept of inheritance
- types of inheritance
- polymorphism – run time and compile time
- virtual inheritance
- Concept of Overriding - virtual functions and pure virtual functions

Ready References

6.1 Inheritance

Inheritance is the process of creating new classes from an existing class. The existing class is known as base class and the newly created class called as a derived class.

Defining the derived class:

```
class derived_class_name : visibility_mode  base_class_name
{
        --------
        --------
};
```

Example:

```
class ABC
{
        ----------
        ---------
};
class  XYZ : private ABC
{
        ---------
        ----------
};
```

- Here class ABC is base class.
- Here class XYZ is derived class.
- Following table shows how the visibility of base class members undergoes modifications in all three types of derivation.

Base class Visibility	Derived class visibility		
	public derivation	Private Derivation	Protected Derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

6.2 Types of Inheritance

1. Single inheritance

If the derived class inherits from a single parent, the inheritance is said to be single inheritance.
Example: same as above.

2. Multiple inheritance

Multiple inheritance is the process of creating a new class from more than one base classes.
Example:

```
class ABC {
----------
---------
};
class  XYZ  {
---------
----------
};
class test : public XYZ, public ABC {
----------
---------
};
```

3. Multilevel inheritance

One class derived from a base class, is further used as a base class of some other class. Therefore, the class which provides link between two classes is known as intermediate base class.

```
class ABC
{
        ----------
        ---------        };
class  XYZ : private ABC
{
        ---------
        ---------        };
class  LMN : public XYZ
{
        ------------
        ----------              };
```

4. Hierarchical Inheritance

This is the process where many derived classes share the properties of base class.
Example:
A student shows a hierarchy as Arts, medical or engineering students. The medical is again has a hierarchy as medicine, surgery, radiology, gynaec, etc.

5. Hybrid Inheritance

It is referred as multiple base class of a derived class having the same base class.
Example: case of multilevel inheritance where student, test and result are three classes. But suppose some one or say authority wants that some weightage should be given for sports to finalize the result.

6.3 Virtual Base Classes

In this, child can inherit the properties via two direct base classes or it may inherit directly. The duplication of inherited members due to these multiple paths can be avoided by making common base class as virtual base class while declaring the direct or intermediate base classes.
Example:
Consider, the 'child' has two direct base classes parent1 and parent2 which themselves have a common base class 'grandparent'.

```
class grand
{       -------------
        -------------     };
                class p1 : virtual public grand
                {       ---------------     -----------------   };
                class p2 : virtual public grand
                {       ---------------     -----------------   };
class child : public p1, public p2
{       ---------
        ----------   // only one copy of grand will be inherited.
};
```

6.4 Abstract Classes

- It is not used to create objects.
- As long as no base class constructor takes any arguments, the derived class need not have a constructor function.

6.5 Constructors in derived classes

General form of defining a derived constructor

```
derived-constructor(arglist1, arglist2,.......arglistN, arglistD)

base1(arglist1),
base2(arglist2),
.
.

baseN(arglistN)
{

                    Body of derived constructor
}
```

Execution of base class constructor

Method Of Inheritance	Order Of Execution
class B : public A { };	A() : base constructor B(): derived constructor
class A : public B, public C { };	B(); base(first) C() : base(second) A(): derived
class A : public B, virtual public C { };	C(): virtual base B() : ordinary base A(): derived

6.6 Destructors in derived classes

Destructors in class hierarchy are invoked in the reverse order of the constructors invocation. The constructor of that class whose constructor was executed last, will be executed first whenever the object goes out of the scope.
If destructors are missing in any class in the hierarchy of classes; that class's destructor is not invoked.

6.7 Virtual Function

It is a member function that is declared within a base class and redefined by a derived class. They cannot be static members. Prototype of base class version of virtual function and all the derived class version must be identical.

They are accesses by using object pointers.

We cannot use a pointer to a derived class to access an object of the base type.

Example 1:

```
class base
      {
            public:
            void disp()
            {   cout<<"base";  }
            virtual void show()
            {    cout<< "showbase";    }
      };
      class der : public base
      {
            public:
            void disp()
            {   cout<<"derived";  }

            void show()
            {     cout<<"show derived";      }
      };
      void main()
      {
            base ob1;  der ob2;
            cout<<"bptr points to base";
            bptr=&ob1;
            bptr->disp();    // call base version
            bptr->show();    // call base version
            cout<<"bptr points to derived";
            bptr=&ob2;
            bptr->disp();    // call to base
            bptr->show();    // call to derived
      }
```

```
Output:
bptr points to base
base
showbase
bptr points to derived
base
derived
```

6.8 Pure Virtual Function

A pure virtual function is a virtual function that has no definition within the base class.

```
virtual type func_name(parameter-list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition.
A class containing pure virtual function is an abstract class.

Programming Tips

- ✓ Use pointers and references to avoid slicing.
- ✓ Use abstract classes to focus design on the provision of clean interfaces.
- ✓ Use abstract classes to minimize interfaces.
- ✓ Use abstract classes to keep implementation details out of interfaces.
- ✓ Use virtual functions to allow new implementation o be added without affecting user code.
- ✓ A class with a virtual function should have virtual destructor.

Practical Assignment

SET A

1. Write a program in c++ to read the following information from the keyboard in which the base class consists of employee name, code and designation. The derived class Manager which contains the data members namely, year of experience and salary. Display the whole information on the screen.

2. Create base class called shape. Use this class to store two double type values that could be used to compute the area of figures. Derive three specific classes called triangle, circle and rectangle from the base shape. Add to the base class, a member function get_data() to initialize base class data members and display_area() to compute and display area. Make display_area() as a virtual function.(Hint: **use default value for one parameter while accepting values for shape circle.)

SET B

1. Write a program in C++ to define class PriorityQueue. Derive two sub classes from it AscendingPriorityQueue and DescendingPriorityQueue.

2. Write a class with one virtual function and one nonvirtual function. Inherit a new class, make an object of this class, and upcast to a pointer of the base-class type. Use the clock( ) function found in <ctime> (you'll need to look this up in your local C library guide) to measure the difference between a virtual call and non-virtual call.You'll need to make multiple calls to each function inside

your timing loop in order to see the difference.

3. Write a C++ program to maintain information about Instructor who is the student and employee of same college. Use multiple inheritance. (Derive Instructor class from Employee and Student.)

SET C

1. An **abstract class** is a class which is never used directly but provides a skeleton for other classes to be derived from it. Typically, all the member functions of an abstract are virtual and have dummy implementations. The following is a simple example of an abstract class:

```
class Database {
public:
    virtual voidInsert  (Key, Data)=0
    virtual voidDelete  (Key)   =0
    virtual DataSearch  (Key)          {return 0;}
};
```

It provides a skeleton for a database-like classes. Examples of the kind of classes which could be derived from database include: linked-list, binary tree, and B-tree. Derive a B-tree class from Database:

```
class BTree : public Database { /*...*/ };
```

Assignment Evaluation

0: Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]

3:Needs Improvement [ ]   4:Complete [ ]   5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 7
Formatted I/O

Start Date ____/____/______

Objectives

To learn about:

- Managing console I/O
- C++ stream classes
- Formatted and Unformatted console I/O
- Usage of manipulators

Reading

You should read following topics before starting this exercise:

- Concept of streams in C++ (text and binary)
- C++ Stream Classes
- Concept of manipulators

Ready References

7.1 The C++ Stream Classes

C++ provides support for its I/O system in <iostream> header.

Template Class	Characterbased Class	Wide-Characterbased Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

basic_streambuf
This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. It is required only in advanced I/O programming.

basic_ios
This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for basic_ios is called ios_base, which defines several nontemplate traits used by basic_ios.)
basic_ios is used as a base for several derived classes, including basic_istream, basic_ostream, and basic_iostream. These classes are used to create streams capable of input, output, and input/output, respectively.

ios
The ios class contains many member functions and variables that control or monitor the fundamental operation of a stream.

7.2 Predefined Streams In C++

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

Stream	Meaning	Default Device
cin	Standard Input	Keyboard
cout	Standard Output	Screen
cerr	Standard Error	Screen
clog	Buffered Version of cerr	Screen

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

By default, the standard streams are used to communicate with the console.However, in environments that support I/O redirection (such as DOS, Unix, OS/2, and Windows), the standard streams can be redirected to other devices or files. For the sake of simplicity, the examples in this chapter assume that no I/O redirection has occurred.

Standard C++ also defines these four additional streams: win, wout, werr, and wlog. These are wide-character versions of the standard streams. Wide characters are of type wchar_t and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

7.3 Formatted I/O

7.3.1 Formatting using the ios members

The ios class declares a bitmask enumeration called fmtflags in which the following values are defined. (Technically, these values are defined within ios_base, which, as explained earlier, is a base class for ios.)

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

Setting the format flags using setf() function

To set a flag, use the setf() function. This function is a member of ios. Its most common form is shown here:

```
fmtflags setf(fmtflags flags);
```

This function returns the previous settings of the format flags and turns on those flags specified by flags. For example, to turn on the showpos flag, you can use this statement:

stream.setf(ios::showpos);

Here, stream is the stream you wish to affect. Notice the use of ios:: to qualify showpos. Since showpos is an enumerated constant defined by the ios class, it must be qualified by ios when it is used.

Example:

```
#include <iostream>
using namespace std;
int main()
{
        cout.setf(ios::showpoint | ios::showpos);
        cout << 100.0; // displays +100.0
        return 0;
}
```

7.3.2 Clearing Format Flags

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

void unsetf(fmtflags flags);

Example 1:

```
#include <iostream>
using namespace std;
int main()
{
      cout.setf(ios::uppercase | ios::scientific);
      cout << 100.12; // displays 1.0012E+02
      cout.unsetf(ios::uppercase); // clear uppercase
      cout << " \n" << 100.12; // displays 1.0012e+02
      return 0;
}
```

Compile and run this program to see the output.

7.3.3 An Overloaded Form of setf()

There is an overloaded form of setf() that takes this general form:

fmtflags setf(fmtflags flags1, fmtflags flags2);

In this version, only the flags specified by flags2 are affected. They are first cleared and then set according to the flags specified by flags1. Note that even if flags1 contains other flags, only those specified by flags2 will be affected. The previous flags setting is returned.

Example 2:

```
#include <iostream>
```

```
using namespace std;
int main( )
{
      cout.setf(ios::showpoint | ios::showpos, ios::showpoint);
      cout << 100.0; // displays 100.0, not +100.0
      return 0;
}
```

Compile and run this program to see the output.

The most common use of the two-parameter form of setf() is when setting the number base, justification, and format flags.

Example 3:

```
#include <iostream>
using namespace std;
int main()
{
      cout.setf(ios::hex, ios::basefield);
      cout << 100; // this displays 64
      return 0;
}
```

Compile and run this program.

Here, the basefield flags (i.,e., dec, oct, and hex) are first cleared and then the hex flag is set.

Remember, only the flags specified in flags2 can be affected by flags specified by flags1. For example, in this program, the first attempt to set the showpos flag fails.

Example 4:

```
// This program will not work.
#include <iostream>
using namespace std;
int main()
{
      cout.setf(ios::showpos, ios::hex); // error, showpos not set
      cout << 100 << '\n'; // displays 100, not +100
      cout.setf(ios::showpos, ios::showpos); // this is correct
      cout << 100; // now displays +100
      return 0;
}
```

Compile and run this program.

7.3.4 Examining the Formatting Flags

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

fmtflags flags( );

Example 5:

```
#include <iostream>
using namespace std;
void showflags() ;
int main()
{
      // show default condition of format flags
      showflags();
      cout.setf(ios::right | ios::showpoint | ios::fixed);
      showflags();
      return 0;
}
// This function displays the status of the format flags.
void showflags()
{
      ios::fmtflags f;
      long i;
      f = (long) cout.flags(); // get flag settings
      // check each flag
      for(i=0x4000; i; i = i >> 1)
            if(i & f) cout << "1 ";
            else cout << "0 ";
      cout << " \n";
}
```

```
The output from the program is shown here:
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 1 0 0 0 1 0 1 0 0 1 0 0 0 1
```

7.3.5 Setting All Flags

The flags() function has a second form that allows you to set all format flags associated with a stream. The prototype for this version of flags() is shown here:

fmtflags flags(fmtflags f);

Example 6:

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

```
void showflags();
int main()
{
	// show default condition of format flags
	showflags();
	// showpos, showbase, oct, right are on, others off
	long f = ios::showpos | ios::showbase | ios::oct | ios::right;
	cout.flags(f); // set all flags
	showflags();
	return 0;
}
```

Compile and run this program.

7.3.6. Member Functions of ios : width(), precision(), and fill()

width() Function:

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the width() function. Its prototype is shown here:

```
streamsize width(streamsize w);
```

Here, w becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The streamsize type is defined as some form of integer by the compiler.

precision() Function:

When outputting floating-point values, you can determine the number of digits to be displayed after the decimal point by using the precision() function. Its prototype is shown here:

```
streamsize precision(streamsize p);
```

Here, the precision is set to p, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

fill() Function:

When a field needs to be filled, it is filled with spaces. You can specify the fill character by using the fill() function. Its prototype is

```
char fill(char ch);
```

After a call to fill() , ch becomes the new fill character, and the old one is returned.

Example 7:

```
#include <iostream>
using namespace std;
int main()
{
      cout.precision(4) ;
      cout.width(10);
      cout << 10.12345 << "\n"; // displays 10.12
      cout.fill('*');
      cout.width(10);
      cout << 10.12345 << "\n"; // displays *****10.12
      // field width applies to strings, too
      cout.width(10);
      cout << "Hi!" << "\n"; // displays *******Hi!
      cout.width(10);
      cout.setf(ios::left); // left justify
      cout << 10.12345; // displays 10.12*****
      return 0;
}
```

```
This program's output is shown here:
10.12
*****10.12
*******Hi!
10.12*****
```

There are overloaded forms of width() , precision() , and fill() that obtain but do not change the current setting. These forms are shown here:

char fill( );
streamsize width( );
streamsize precision( );

7.4 Manipulators to Format I/O

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

Manipulator	Purpose	Input/Output
boolalpha	Turns on boolapha flag.	Input/Output
dec	Turns on dec flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on fixed flag.	Output
flush	Flush a stream.	Output
hex	Turns on hex flag.	Input/Output
internal	Turns on internal flag.	Output

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

Example 8:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
      cout << hex << 100 << endl;
      cout << setfill('?') << setw(10) << 2343.0;
      return 0;
}
```

```
This displays
64
??????2343
```

Example 8:

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

```
        cout << setiosflags(ios::showpos);
        cout << setiosflags(ios::showbase);
        cout << 123 << " " << hex << 123;
        return 0;
}
```

Compile and run this program.

Example 9:

```
#include <iostream>
using namespace std;
int main()
{
      bool b;
      b = true;
      cout << b << " " << boolalpha << b << endl;
      cout << "Enter a Boolean value: ";
      cin >> boolalpha >> b;
      cout << "Here is what you entered: " << b;
      return 0;
}
```

```
The output is:
1 true
Enter a Boolean value: false
Here is what you entered: false
```

7.5 Creating Your Own Manipulators

Parameterless manipulator output functions have this skeleton:

```
ostream &manip-name(ostream &stream)
{
// your code here
return stream;
}
```

As a simple first example, the following program creates a manipulator called sethex() , which turns on the showbase flag and sets output to hexadecimal.

Example 10:

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

// A simple output manipulator.
ostream &sethex(ostream &stream)
{
      stream.setf(ios::showbase);
```

```
      stream.setf(ios::hex, ios::basefield);
      return stream;
}

int main()
{
      cout << 256 << " " << sethex << 256;
      return 0;
}
```

```
Compile and run this program.
```

Custom manipulators need not be complex to be useful. For example, the simple manipulators la() and ra() display a left and right arrow for emphasis, as shown here:

Example 11:

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

// Right Arrow
ostream &ra(ostream &stream)
{
      stream << "-------> ";
      return stream;
}

// Left Arrow
ostream &la(ostream &stream)
{
      stream << " <-------";
      return stream;
}

int main()
{
      cout << "High balance " << ra << 1233.23 << "\n";
      cout << "Over draft " << ra << 567.66 << la;
      return 0;
}
```

```
This program displays:

High balance -------> 1233.23
Over draft -------> 567.66 <-------
```

parameterless input manipulator functions have this skeleton:

istream &manip-name(istream &stream)
{

// your code here
return stream;
}

The following program creates the getpass() input manipulator, which rings the bell and then prompts for a password:

Example 12:

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

// A simple input manipulator.
istream &getpass(istream &stream)
{
      cout << '\a'; // sound bell
      cout << "Enter password: ";
      return stream;
}

int main()
{
char pw[80];
do {
      cin >> getpass >> pw;
} while (strcmp(pw, "password"));
cout << "Logon complete\n";
return 0;
}
```

Compile and run this program to see the output.

Programming Tips

- ✓ Because the format flags are defined within the ios class, you must access their values by using ios and the scope resolution operator. For example, showbase by itself will not be recognized. You must specify ios::showbase.
- ✓ Remember that it is crucial that your manipulator return stream. If it does not, your manipulator cannot be used in a series of input or output operations.
- ✓ Use lower level input functions such as get() and read() only when runtime efficiency is at premium.
- ✓ Be careful with the termination criteria when using get(), getline() and read().
- ✓ Remember that width specifications are applicable to following I/O operations only.
- ✓ Remember that precision specifications are applicable to following floating – point output operations only.
- ✓ Prefer manipulators to state flags for controlling I/O.

Practical Assignment

SET A

1. Write a program to accept student details and print marksheet on screen using appropriate manipulators.

SET B

1. Implement an encrypt(k) manipulator that ensures that output on its ostream is encrypted using the key k. Provide the similar decrypt(k) manipulator for an istream. Provide the means for turning the encryption off for a stream so that further I/O is cleartext.

NOTE: this session contains one exercise in SET A and one in SET B. SET C is not provided for this session.

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 8
Exception Handling

Start Date ____/____/______

Objectives

To learn about :

- C++ Exception Handling
- Three Keywords : try, catch , throw
- Uncaught Exceptions
- C++ standard exception class
- std::exception class

Reading

You should read following topics before starting this exercise:
- What is exception?
- What is the purpose of exception handling?
- How C++ provides exception handling?

Ready References

8.1 Exception Handling

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called handlers.

Good program is stable and fault tolerant.

In a good program, exceptional ("error") situations must be handled.

- Exceptions are unusual events (erroneous or not)
- They are detectable by either hardware or software
- They may require special processing
- Exception handler is part of the code that processes an exception

Some examples for exceptions:

- EOF is reached while trying to read from a file
- Division by 0 is attempted (result is meaningless)
- Array subscript is out of range , Bad input

8.2 Three Keywords

try : identifies a code block where exception can occur
throw : causes an exception to be raised (thrown)
catch : identifies a code block where the exception will be handled (caught)

General Form

```
try
 {
        /*try block*/
  }
catch (type1 arg1)
 {
      /*catch block*/
  }
catch (type2 arg2)
  {
      /*catch block*/
  }
...
 catch (typeN argN)
{
      /*catch block*/
}
```

8.2.1. The try Block

- Bounded by curly braces {}
  - – The region of code to be monitored
  - – Entire program can be monitored, if necessary
- Monitoring stays in effect even during function calls

8.2.2.The catch Block

- Each catch block catches a particular type
  - – One catch block catches all ints thrown, another catches all floats
- The thrown value is received in the catch block parameter
- The first catch block must immediately follow the try block with which it is associated

8.2.3 The throw Statement

- Exceptions can be thrown only from within a try block
- Any value of any type can be thrown
- Syntax : throw exception ;
- When an exception is thrown, control immediately passes to the first appropriate catch block
- Unwinds the stack

Note:

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a try block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

Example 1

```
// C++ program using try , catch , throw

#include <iostream>
using namespace std;
void main ()
{
      cout << "Inside Main" << endl;
      try
      {
            cout << "Inside try block " << endl;
            throw 10;
      }
      catch (int i)
      {
            cout << "Exception caught: " << i << endl;
      }
      cout << "Ending ….." << endl;
}
```

Output for Example :

```
Inside Main
Inside try block
Exception caught: 10
Ending….
```

Note:

Uncaught Exceptions

- A thrown exception which is not caught will cause the terminate () function to be invoked
- By default, this simply calls abort ()

8.3 C++ standard exception class – "exception

- A base class specifically designed to declare objects to be thrown as exceptions.
- It is defined in the <exception> header file under the namespace std.
- It has
  - default and copy constructors,
  - operators and
  - destructors,
  - a virtual member function

Example 2

```
// C++ program for checking divide by zero
#include<iostream.h>
#include <string.h>
int Quotient( int, int );
class DivByZero {}; // Exception class

int main()
{
      int numer;    // Numerator
      int denom;    // Denominator
      //read in numerator  and denominator
      while(cin)
      {
           try
           {
                 cout << "Their quotient: "  << Quotient(numer,denom)
<<endl;
           }
           catch ( DivByZero )//exception handler
           {
                 cout<<"Denominator can't be 0"<< endl;
           }
           // read in numerator and denominator
      }
      return 0;
}
```

```
Compile and Run this program.
```

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class. These are:

Exception	Description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid

ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try
{
            int * myarray= new int[1000];
}
catch (bad_alloc&)
{
            cout << "Error allocating memory." << endl;
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

Example 3

```
// bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;

int main ()
{
       try
       {
               int* myarray= new int[1000];
       }
       catch (exception& e)
       {
              cout << "Standard exception: " << e.what() << endl;
       }
       return 0;
}
```

Compile and run this program.

Programming Tips

- ✓ Don't use exceptions where more local control structures will suffice.
- ✓ Avoid throwing exception from copy constructor and destructor.
- ✓ Have main() catch and report all exceptions.
- ✓ Keep ordinary code and error handling code separate.

- ✓ Don't assume that every exception is derived from class exception.

Practical Assignment

SET A

1) Write a C++ program that prompts the user for a positive no, If negative no. is entered, then throw an exception as no. not positive

2) Write a C++ program to print successive integers 1, 2, 3, ... StopNum - 1. Throws an integer exception when StopNum is reached. StopNum : The stopping number to be used.

SET B

1.Write a C++ program for calculator which has operators as "+","-","*","/". Display appropriate exceptions mesg. like divide by zero etc. and display the calculated results.

2. Write a C++ program that prompts the user to enter a sequence of numbers (floats) and then prints them in reverse order. Use stack. Give appropriate exception message as "Push not done - out of space."

SET C:

1. Define appropriate exceptions for the Matrix class and its functions so that they throw exceptions when errors occur, including the following:

- When the sizes of the operands of + and - are not identical.
- When the number of the columns of the first operand of * does not match the number of rows of its second operand.
- When the row or column specified for () is outside its range.
- When heap storage is exhausted?

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 9
File I/O

Start Date ____/____/______

Objectives

To learn about:
- File Operations
- File Pointers and Their Manipulations
- File Updations and Random Access

Reading

You should read following topics before starting this exercise:
- Types of files supported in C++ : Text and Binary
- Difference between binary and text files
- Concept of file pointers and their manipulations in C++

Ready References

To perform file I/O, you must include the header <fstream> in your program. It defines several classes, including ifstream, ofstream, and fstream.

9.1 Opening and Closing a File

We have to connect the output stream to the file.

Decide the following things for a file:
1. Name of the file
2. Data type and the structure
3. Purpose
4. Opening method

Creating files with constructor function:

```
1.  ofstream outfile("results");     // open file for output, creates outfile and
                                     //  connects "results" to it
     ........
     ........
     outfile.close();                // disconnect "results" from outfile

2.  ifstream  infile ("data");       // open file for input, creates infile and
                                     // connects  "data"  to it
     ..........
     ..........
    infile.close();                  // disconnect "data"  from infile
```

Creating the file using the member function open():

Syntax 1:

```
File-stream-class stream-object;
Stream-object .open  ("filename");
```

Example:

```
ofstream  outfile;        //  create stream for output
outfile.open("DATA");   //  connect stream to DATA
……….
……….
outfile.close();
```

Syntax 2:

```
File-stream-class stream-object;
Stream-object .open  ("filename",mode);
```

File mode parameters:

Parameter	Meaning
ios::in	Open file for reading only
ios::out	Open file for writing only
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Open as binary file
ios::nocreate	Does not create a new file, opens the existing one
ios::noreplace	Open file if the file already exists
ios::trunc	Delete the contents of the file if it exists

Example:

```
ofstream  outfile;
outfile.open("DATA", ios :: app | ios :: nocreate);
```

9.2 File pointers and their manipulations:

There are two file pointers

– get pointer – which points towards the read position in the file and

- put pointer – which points towards the write position in the file.

Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

- seekg()   Moves get pointer to a specified position
- seekp()   Moves put pointer to a specified position.
- tellg()   Gives the current position of the get pointer.

- tellp() Gives the current position of the put pointer.

Syntax:

- seekg(offset, refposition);
- seekp(offset, refposition);

where offset specifies the number of bytes the file pointer is to be moved from the reference position. The refposition can be any one of the following :

- ios::beg indicates beginning of the file
- ios::cur indicates current position of the pointer
- ios::end indicates the end of the file

  - Offset can be 0 or positive w.r.t. ios::beg
  - Offset can be positive or negative w.r.t. ios::cur
  - Offset can be 0 or negative w.r.t. ios::end

The file size can be obtained using the function tellg() or tellp() when the file pointer is located at the end of the file.

9.3 Input and output operations

- put () and get () functions – put() writes a single character to the associated stream and get() reads a single character from the associated stream.
- read () and write () functions- These functions handle the data in the binary form and read or write blocks of binary data

Syntax :

- file.put(ch);
- file.get(ch)

where file is declared as -- fstream file; // input and output stream and ch is declared as of type char ;

- file.read ((char *) & v, sizeof (v));
- file.write ((char *) & v, sizeof(v));

where v is some variable. First parameter to the above two functions is address of the variable v, cast to type char * and second parameter is length of that variable in bytes.

Programming Tips

- ✓ Use lower-level input functions such as get() and read() only when run-time efficiency is at a premium.
- ✓ Be careful with the termination criteria when using get(), getline(), and read().

Practical Assignment

SET A

1. Write a program that reads a text file and creates another file that is identical except that every sequence of consecutive blank spaces is replaced by a single space.

2. Write a program to count the number of characters, number of words and number of lines in a file.

3. Write a program which copies a user-specified file to another user-specified file. Your program should be able to copy text as well as binary files.

SET B:

1. A file contains a list of telephone numbers in the following form:

```
Ajay     12345
Vijay    98765
-------  --------
-------  --------
```

The names contain only one word and the names and telephone numbers are separated by white spaces. Write a program to read the file and output the list in two columns. The names should be left-justified and the numbers right-justified.

2. Write a program which reads a C++ source file and checks that all instances of brackets are balanced, that is, each '(' has a matching ')', and similarly for [] and {}, except for when they appear inside comments or strings. A line which contains an unbalanced bracket should be reported by a message such as the following sent to standard output:

```
'{' on line 15 has no matching '}'
```

SET C:

1. Write an interactive, menu-driven program that will access the file created in the above program, and implement the following tasks.
   a. Determine the telephone number of the specified person.
   b. Determine the name if a telephone number is known.
   c. Update the telephone number, whenever there is a change.

2. Write a program to accept file name and string. Search string in file and display entire file with markers to indicate occurrence of given string. Also show the frequency of that string.

For example, if the contents of the file XYZ.DAT are as follows:

Because C++ and C code are often intermixed, C++ streams I/O is sometimes mixed with the C printf() family of I/O functions. The C-style I/O functions are presented by <cstdio> and <stdio.h>. Also, because C functions can be called from C++ some programmers may prefer to

use the more familiar C I/O functions.

The command line argument for the above program could be file name (XYZ.DAT) and string ("fun").
The output will be as follows:

Because C++ and C code are often intermixed, C++ streams I/O is sometimes mixed with the C printf() family of I/O <fun>ctions. The C-style I/O <fun>ctions are presented by <cstdio> and <stdio.h>. Also, because C <fun>ctions can be called from C++ some programmers may prefer to use the more familiar C I/O <fun>ctions.

Frequency of Occurrence of word "fun" is 4.

Assignment Evaluation

0: Not Done [ ]   1:Incomplete [ ]   2.Late Complete [ ]

3:Needs Improvement [ ]   4:Complete [ ]   5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______

SESSION 10
Templates

Start Date ____/____/______

Objectives

To learn about:
- Concept of Templates
- Function Templates and Class Templates
- Overloading of Function Templates
- Function Templates With Multiple Parameters
- Class Templates With Multiple Parameters

Reading

You should read following topics before starting this exercise:
- Generic Programming in C++.
- Compile Time and Run Time Polymorphism and Templates.
- Concept of Templates in C++

Ready References

10.1 What is Template?

Templates provide direct support for generic programming, that is , programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a class or a function.
Templates are introduced with the primary focus on techniques needed for the design, implementation, and use of the standard library. The standard library requires greater degree of generality, flexibility and efficiency.

10.2 Function Templates

Function templates provide a functional behavior that can be called for different types. In other words, a function template represents a family of functions. The representation looks a lot like an ordinary function, except that some elements of the function are left undetermined: These elements are parameterized.

10.2.1 Defininig the Template

The following is a function template that returns the maximum of two values:

```
template <class T>
inline T const& max (T const& a, T const& b)
{
    // if a < b then use b else use a
    return a<b?b:a;
}
```

This template definition specifies a family of functions that returns the maximum of two values, which are passed as function parameters a and b. The type of these parameters is left open as template parameter T. As seen in this example, template parameters must be announced with syntax of the following form:

template < comma-separated-list-of-parameters >

10.2.2. Using the Template

The following program shows how to use the max() function template:

```
#include <iostream>
#include <string>

int main()
{
    int i = 42;
    std::cout << "max(7,i): " << ::max(7,i) << std::endl;

    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2): " << ::max(f1,f2) << std::endl;

    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2): " << ::max(s1,s2) << std::endl;
}
```

Inside the program, max() is called three times: once for two ints, once for two doubles, and once for two std::strings. Each time, the maximum is computed. As a result, the program has the following output:

```
max(7,i):   42
max(f1,f2): 3.4
max(s1,s2): mathematics
```

Templates aren't compiled into single entities that can handle any type. Instead, different entities are generated from the template for every type for which the template is used. Thus, max() is compiled for each of three types.

An attempt to instantiate a template for a type that doesn't support all the operations used within it will result in a compile-time error

Templates are compiled twice:

1. Without instantiation, the template code itself is checked for correct syntax. Syntax errors are discovered, such as missing semicolons.
2. At the time of instantiation, the template code is checked to ensure that all calls are valid. Invalid calls are discovered, such as unsupported function calls.

When we call a function template such as max() for some arguments, the template parameters are determined by the arguments we pass. If we pass two ints to the parameter types T const& , the C++ compiler must conclude that T must be int. Note that no automatic type conversion is allowed here. Each T must match exactly. For example:

```
template <class  T>
inline T const& max (T const& a, T const& b);
…
max(4,7)     // OK: T is int for both arguments
max(4,4.2)   // ERROR: first T is int, second T is double
```

There are three ways to handle such an error:

1. Cast the arguments so that they both match:

   ```
   max(static_cast<double>(4),4.2)    // OK
   ```

2. Specify (or qualify) explicitly the type of T:

   ```
   max<double>(4,4.2)              // OK
   ```

3. Specify that the parameters may have different types.

10.2.3 Template Parameters

Function templates have two kinds of parameters:

1. Template parameters, which are declared in angle brackets before the function template name:

   ```
   template <class T>           // T is template parameter
   ```

2. Call parameters, which are declared in parentheses after the function template name:

   ```
   …  max (T const& a, T const& b)   // a and b are call parameters
   ```

You may have as many template parameters as you like.

10.2.4 Overloading Function Templates

Like ordinary functions, function templates can be overloaded. That is, you can have different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call.

The following short program illustrates overloading a function template:

```
// maximum of two int values
inline int const& max (int const& a, int const& b)
{
```

```
    return a<b?b:a;
}

// maximum of two values of any type
template <class  T>
inline T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

// maximum of three values of any type
template <class T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);     // calls the template for three arguments
    ::max(7.0, 42.0);     // calls max<double> (by argument deduction)
    ::max('a', 'b');      // calls max<char> (by argument deduction)
    ::max(7, 42);         // calls the nontemplate for two ints
    ::max<>(7, 42);       // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
    ::max('a', 42.7);     // calls the nontemplate for two ints
}
```

Compile and run this program to see the output.

A more useful example would be to overload the maximum template for pointers and ordinary C-strings:

```
#include <iostream>
#include <cstring>
#include <string>

// maximum of two values of any type
template <class T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// maximum of two pointers
template <class T>
inline T* const& max (T* const& a, T* const& b)
{
    return *a < *b ? b : a;
}
```

```
// maximum of two C-strings
inline char const* const& max (char const* const& a,
                               char const* const& b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}

int main ()
{
    int a=7;
    int b=42;
    ::max(a,b);       // max() for two values of type int

    std::string s="hey";
    std::string t="you";
    ::max(s,t);       // max() for two values of type std::string

    int* p1 = &b;
    int* p2 = &a;
    ::max(p1,p2);     // max() for two pointers

    char const* s1 = "David";
    char const* s2 = "Nico";
    ::max(s1,s2);     // max() for two C-strings
}
```

```
Compile and run this program.
```

Note that in all overloaded implementations, we pass all arguments by reference. In general, it is a good idea not to change more than necessary when overloading function templates.

10.3 Class Templates

Similar to functions, classes can also be parameterized with one or more types. Container classes, which are used to manage elements of a certain type, are a typical example of this feature. By using class templates, you can implement such container classes while the element type is still open. In this chapter we use a stack as an example of a class template.

10.3.1 Defining a Class Template

To indicate we're defining a template rather than a straightforward class definition, we insert the keyword template and the type parameter, T, between angled brackets, just before the keyword class and the class name, Samples.
Inside the class template, T can be used just like any other type to declare members and member functions.

```
template <class T>
class Samples
{
```

```
   private:
      T values[100];        // Array to store samples
      int free;             // Index of free location in values

   public:
      // Constructor definition to accept an array of samples
      Samples(T vals[], int count)
      {
         free = count<100? count:100; // Don't exceed the array
         for(int i=0; i<free; i++)
            values[i] = vals[i]; // Store count number of samples
      }

      // Constructor to accept a single sample
      Samples(T val)
      {
         values[0] = val;      // Store the sample
         free = 1;             // Next is free
      }

      // Default constructor
      Samples()
      {
         free = 0;             // Nothing stored, so first is free
      }

      // Function to add a sample
      bool Add(T& val)
      {
         bool OK = free<100; // Indicates there is a free place
         if(OK)
            values[free++] = val; // OK true, so store the value
         return OK;
      }

      // Function to obtain maximum sample
      T Max()
      {
         T theMax = values[0];      // Set first sample as maximum
         for(int i=1; i<free; i++)  // Check all the samples
         if(values[i]>theMax)
            theMax = values[i];        // Store any larger sample
         return theMax;
      }
};
```

10.3.2 Template Member Function

The function declaration appears in the class template definition in the normal way. For instance:

```
template <class T>
class Samples
{
```

```
  // Rest of the template definition...
  T Max();// Function to obtain maximum sample
  // Rest of the template definition...
}
```

This declares the Max() function within the class. You now need to create a separate function template for the definition of the member function. You must use the template class name plus the parameters in angled brackets to identify the class to which the function belongs:

```
template<class T>
T Samples<T>::Max()
{
   T theMax = values[0];        // Set first sample as maximum
   for(int i=1; i<free; i++)    // Check all the samples
      if(values[i]>theMax)
         theMax = values[i];    // Store any larger sample
   return theMax;
}
```

Defining a constructor or a destructor outside of the class template definition is very similar. We could write the definition of the constructor accepting an array as:

```
template<class T>
Samples<T>::Samples(T vals[], int count)
{
   free = count<100? count:100;   // Don't exceed the array
   for(int i=0; i<m_Free; i++)
      values[i] = vals[i];       // Store count number of samples
}
```

The class to which the constructor belongs is specified in the template in the same way as for an ordinary member function. Note that the constructor name doesn't require the parameter specification. You only use the parameter with the class name preceding the scope resolution operator.

10.3.3 Creating Objects from a Class Template

When we used a function defined by a function template, the compiler was able to generate the function from the types of the arguments used. The type parameter for the function template was implicitly defined by the specific use of a particular function. Class templates are a little different. To create an object based on a class template, you must always specify the type parameter following the class name in the declaration.

For example, to declare a Samples object to handle samples of type double, you could write the declaration as:

Samples<double> MyData(10.0);

This defines a Samples object that can store samples of type double, and the object is created with one sample stored with the value 10.0.

```
// Trying out a class template
#include <iostream>
using namespace std;

// Put the Box class definition here

// Put the Samples class template definition here

int main()
{
   Box Boxes[] = {                    // Create an array of boxes
                   Box(8.0, 5.0, 2.0), // Initialize the boxes...
                   Box(5.0, 4.0, 6.0),
                   Box(4.0, 3.0, 3.0)
                  };

   // Create the Samples object to hold Box objects
   Samples<Box> MyBoxes(Boxes, sizeof Boxes/sizeof Box);
   Box MaxBox = MyBoxes.Max();          // Get the biggest box
   cout << endl                         // and output its volume
        << "The biggest box has a volume of "
        << MaxBox.Volume()

        << endl;
   return 0;
}
```

```
Add appropriate class definitions as mentioned in above program.
Compile and run it to see the output.
```

10.3.4 Class Templates with Multiple Parameters

Using multiple type parameters in a class template is a straightforward extension of the example using a single parameter, which we have just seen. You can use each of the type parameters wherever you want in the template definition. For example, you could define a class template with two type parameters:

```
template<class T1, class T2>
class ExampleClass
{
  // Class data members
  private:
    T1 value1;
    T2 value2;
    // Rest of the template definition...
};
```

The types of the two class data members shown will be determined by the types you supply for the parameters when you instantiate an object.

The parameters in a class template aren't limited to types. You can also use parameters that require constants or constant expressions to be substituted in the class definition. In our Samples template, we arbitrarily defined the values array with 100 elements. We could, however, let the user of the template choose the size of the array when the object is instantiated, by defining the template as:

```
template <class T, int Size> class Samples
{
   private:
      T values[Size];        // Array to store samples
      int free;              // Index of free location in values

   public:
      // Constructor definition to accept an array of samples
      Samples(T vals[], int count)
      {
         free = count<Size? count:Size; // Don't exceed the array
         for(int i=0; i<m_Free; i++)
            values[i] = vals[i]; // Store count number of samples
      }

      // Constructor to accept a single sample
      Samples(T val)
      {
         values[0] = val;    // Store the sample
         free = 1;           // Next is free
      }

      // Default constructor
      Samples()
      {
         free = 0;           // Nothing stored, so first is free
      }

      // Function to add a sample
      bool Add(T& val)
      {
         bool OK = free<Size; // Indicates there is a free place
         if(OK)
            values[free++] = val; // OK true, so store the value
         return OK;
      }

      // Function to obtain maximum sample
      T Max()
      {
         T theMax = values[0];      // Set first sample as maximum
         for(int i=1; i<free; i++) // Check all the samples
            if(values[i]>theMax)
               theMax = values[i]; // Store any larger sample
         return theMax;
      }
};
```

The value supplied for Size when you create an object will replace the appearance of the parameter throughout the template definition. Now we can declare the Samples object from the previous example as:

```
Samples<Box, 3> MyBoxes(Boxes, sizeof Boxes/sizeof Box);
```

Programming Tips

- ✓ Use templates to express containers.
- ✓ Prefer templates over derived classes when run-time efficiency is at a premium.
- ✓ Prefer derived classes over a template if adding new variants without recompilation is important.
- ✓ Prefer template over derived classes when no common base can be defined.

Practical Assignment

SET A

1. Implement a template function for quick sort.

2. Define a `Swap` function template for swapping two objects of the same type.

3. Define a class template stack which can be instantiated as follows:

```
stack <int, 20> stk; //size of stack is 20.
```

SET B

1. Define template class for linked list with all necessary operations.

2. Using the Standard C++ Library vector as an underlying implementation, create a Set template class that accepts only one of each type of object that you put into it.

SET C

1. Using Standard Template Library(STL), write a program to obtain the list of files present in current directory and display these files in sorted order of name, type and size.

Assignment Evaluation

0: Not Done [ ]	1:Incomplete [ ]	2.Late Complete [ ]
3:Needs Improvement [ ]	4:Complete [ ]	5:Well Done [ ]

Signature of the Instructor   Date of Completion ____/____/______