CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test 1 – March 2017

Sub:	Programming in C++					Code:	10TE661/10EC665
Date:	30/03/17	Duration:	90 mins	Max Marks:	50	Sem: VI	Branch: TE- A & B EC- C & D

Note: Answer any 5 questions. All questions carry equal marks. Total marks: 50

		Marks	OBE CO	OBE RBT
1(a)	Explain difference in C & C++.	[05]	CO1	L2
(b)	Define the terms class & object.	[05]	CO2	L1
2(a)	Explain preprocessor directives with suitable example.	[08]	CO2	L2

(b)	Write output of following code:
	`#include <iostream.h>`
	`#define max(a,b) a>b?a:b`
	`main( )`
	`{`
	`int x=2, y=3;`
	`int z = 10+max(x,y);`
	`cout << z;`
	`return 0;`
	`}`
		[02]	CO2	L3

3.	Explain all looping control statements with examples.	[10]	CO2	L2
4.	Define i) typedef ii) enum iii) constant qualifier iv) volatile qualifier with example.	[10]	CO2	L2
5.	Write a C++ program i) to reverse a integer whole number. ii) to reverse string without using strrev( ) built-in function.	[10]	CO3	L3
6.	Explain switch statement. Write a program to perform simple calculator using switch case statement.	[8+2]	CO3	L3
7.	Define array. Write a program to find largest element in set of elements.	[10]	CO3	L3
8.	Explain pointer variable. What do you mean by dynamic memory allocation? How it is implemented in C++.	[4+3+3]	CO3	L3

Mapping COs(Course Outcome) to Program Outcome(Pos): Substantial - 3, Moderate – 2, Low – 1, No – 0/-

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1:	Achieve Knowledge of design the solution of the problem in C++	2	2	3	2	1	-	-	-	1	1	1	2
CO2:	Understand the data type, operators, I/O statements & program constructs of C++ language	2	2	3	2	1	-	-	-	1	1	1	2
CO3:	Implement array & pointer to objects by dynamic memory allocation	2	2	3	2	1	-	-	-	1	1	1	2
CO4:	Develop Object Oriented Programming skills in C++ language using classes	2	2	3	2	1	-	-	-	1	1	1	2
CO5:	Design & develop effective utilization inheritance & polymorphism	2	2	3	2	1	-	-	-	1	1	1	2
CO6:	Implement compile time & run time polymorphism in C++	2	2	3	2	1	-	-	-	1	1	1	2

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*

Question #	Description	Marks Distribution		Max Marks
1 a	Comparison min 4 points	5M	5M	5M
1 b	Object Class example	1.5M 1.5M 1M	5M	5M
2 a	preprocessor directives with suitable example	8M	8M	8M
2 b	For correct output	2M	2M	2M
3	Looping statement: while , do..while, for example	7M 1M each	10M	10M
4 a	Enumeration, typedef, constant qualifier, volatile qualifier Example	2M each 0.5 M each	10M	10M
5 a	Program to reverse a integer whole number	5M	5M	5M
5 b	Program to reverse string without using strrev( ) built-in function.	5M	5M	5M
6	Switch statement with syntax Program simple calculator	2M 8M	10M	10M
7	Array definition program to find largest element in set of elements.	4M 6M	10M	10M
7	Pointer variable DMA New and delete operator	4M 3M 3M	10M	10M

1a)

C **programming**

C **programming** uses a list of instructions to tell the computer what to do step-by-step. Procedural programming relies on - you guessed it - procedures, also known as routines or subroutines. A procedure contains a series of computational steps to be carried out. Procedural programming is also referred to as imperative programming. Procedural programming languages are also known as top-down languages.

Procedural programming is intuitive in the sense that it is very similar to how you would expect a program to work. If you want a computer to do something, you should provide step-by-step instructions on how to do it. It is, therefore, no surprise that most of the early programming languages are all procedural. Examples of procedural languages include Fortran, COBOL and C, which have been around since the 1960s and 70s.

C++

Object-oriented programming, or **OOP**, is an approach to problem-solving where all computations are carried out using objects. An **object** is a component of a program that knows how to perform certain actions and how to interact with other elements of the program. Objects are the basic units of object-oriented programming. A simple example of an object would be a person. Logically, you would expect a person to have a name. This would be considered a property of the person. You would also expect a person to be able to do something, such as walking. This would be considered a method of the person.
A method in object-oriented programming is like a procedure in procedural programming. The key difference here is that the method is part of an object. In object-oriented programming, you organize your code by creating objects, and then you can give those objects properties and you can make them do certain things.

A key aspect of object-oriented programming is the use of classes. A class is a blueprint of an object. You can think of a class as a concept, and the object as the embodiment of that concept. So let's say you want to use a person in your program. You want to be able to describe the person and have the person do something. A class called 'person' would provide a blueprint for what a person looks like and what a person can do. Examples of object-oriented languages include C#, Java, Perl and Python.

1b)

Class	Object
Definition	Class is mechanism of binding data members and associated methods in a single unit.	Instance of class or variable of class.
Existence	It is logical existence	It is physical existence
Memory Allocation	Memory space is not allocated , when it is created.	Memory space is allocated, when it is created.
Declaration /definition	Definition is created once.	it is created many time as you require.

2a) Explain all preprocessor directives with suitable example. (8)

Line that begin with # are called preprocessing directives.

Use of #include

Let us consider very common preprocessing directive as below:

#include <iostream.h>

Here, "iostream.h" is a header file and the preprocessor replace the above line with the contents of header file.

Use of #define

Preprocessing directive #define has two forms. The first form is:

#define identifier token_string

token_string part is optional but, are used almost every time in program.

Example of #define

#define c 299792458 /*speed of light in m/s */

The token string in above line 2299792458 is replaced in every occurance of symbolic constant *c*.

Complier control Directives

1. **#if, #elif, #else, #endif**

These preprocessing directives create conditional compiling parameters that control the compiling of the source code. They must begin on a separate line.

Syntax:

#if *constant_expression*
#else
#endif

or

#if *constant_expression*
#elif *constant_expression*
#endif

The compiler only compiles the code after the **#if** expression if the *constant_expression* evaluates to a non-zero value (true). If the value is 0 (false), then the compiler skips the lines until the next **#else**,**#elif**, or **#endif**. If there is a matching **#else**, and the *constant_expression* evaluated to 0 (false), then the lines between the **#else** and the **#endif** are compiled. If there is a matching **#elif**, and the preceding**#if** evaluated to false, then the *constant_expression* after that is evaluated and the code between the **#elif** and the **#endif** is compiled only if this expression evaluates to a non-zero value (true).

Examples:

```
int main(void)
 {
  #if 1
    printf("Yabba Dabba Do!\n");
  #else
    printf("Zip-Bang!\n");
```

```
 #endif
 return 0;
}
```

Only "Yabba Dabba Do!" is printed.

```
int main(void)
 {
 #if 1
   printf("Checkpoint1\n");
 #elif 1
   printf("Checkpoint2\n");
 #endif
 return 0;
}
```

Only "Checkpoint1" is printed. Note that if the first line is #if 0, then only "Checkpoint2" would be printed.

```
#if OS==1
 printf("Version 1.0");
#elif OS==2
 printf("Version 2.0");
#else
 printf("Version unknown");
#endif
```

Prints according to the setting of OS which is defined with a #define.

2. #define, #undef, #ifdef, #ifndef

The preprocessing directives **#define** and **#undef** allow the definition of identifiers which hold a certain value. These identifiers can simply be constants or a macro function. The directives **#ifdef** and **#ifndef**allow conditional compiling of certain lines of code based on whether or not an identifier has been defined.

Syntax:

#define *identifier replacement-code*

#undef *identifier*

#ifdef *identifier*
#else or **#elif**
#endif

#ifndef *identifier*
#else or **#elif**
#endif

#ifdef *identifier* is the same is **#if defined(** *identifier***)**.
#ifndef *identifier* is the same as **#if !defined(***identifier***)**.
An identifier defined with **#define** is available anywhere in the source code until a **#undef** is reached.
A function macro can be defined with **#define** in the following manner:

#define *identifier***(***parameter-list***) (***replacement-text***)**

The values in the *parameter-list* are replaced in the *replacement-text*.

Examples:

```
#define PI 3.141

printf("%f",PI);


#define DEBUG

#ifdef DEBUG

  printf("This is a debug message.");

#endif


#define QUICK(x) printf("%s\n",x);

QUICK("Hi!")


#define ADD(x, y) x + y

z=3 * ADD(5,6)
```

This evaluates to 21 due to the fact that multiplication takes precedence over addition.

```
#define ADD(x,y) (x + y)

z=3 * ADD(5,6)
```

This evaluates to 33 due to the fact that the summation is encapsulated in parenthesis which takes precedence over multiplication.

3. #include

The **#include** directive allows external header files to be processed by the compiler.

Syntax:

#include <*header-file***>**

or

#include "*source-file***"**

When enclosing the file with < and >, then the implementation searches the known header directories for the file (which is implementation-defined) and processes it. When enclosed with double quotation marks, then the entire contents of the source-file is replaced at this point. The searching manner for the file is implementation-specific.

Examples:

#include <stdio.h>
#include "my_header.h"

Reference : http://www.acm.uiuc.edu/webmonkeys/book/c_guide/1.7.html

2b)

```
#include <iostream>
using namespace std;
#define max(x,y) x>y?x:y
int main ()
{
  int m=2,n=3;
  int z=10+max(m,n);   10+2>3?2:3
  cout<<z;
   return 0;
}
```

Output: 2

3) A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax

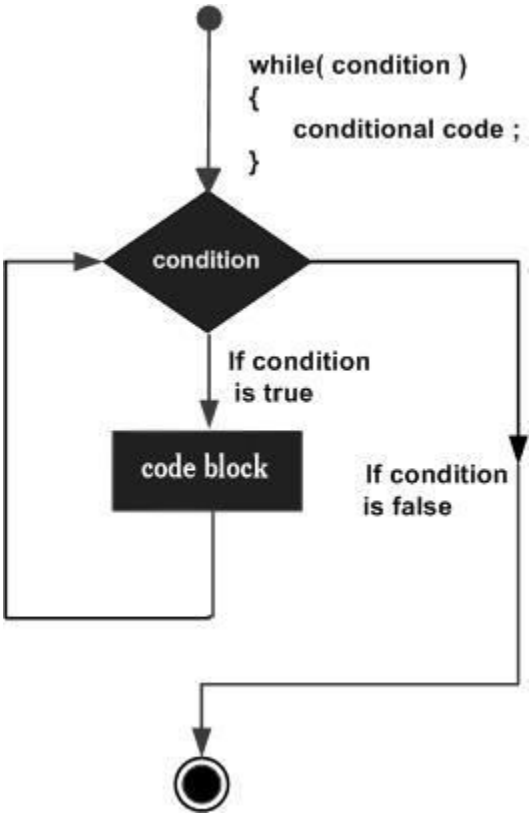
The syntax of a while loop in C++ is:

```
while(condition){
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

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

int main () {
  // Local variable declaration:
  int a = 10;

  // while loop execution
  while( a < 20 ) {
    cout << "value of a: " << a << endl;
    a++;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
```

```
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

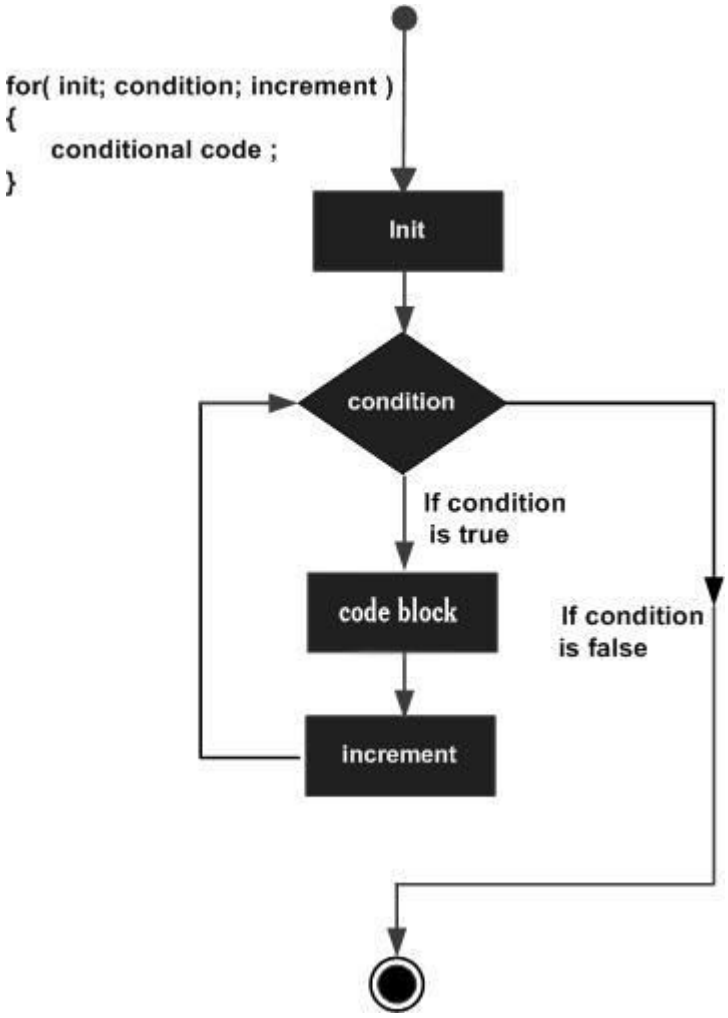
The syntax of a for loop in C++ is:

```
for ( init; condition; increment ) {
   statement(s);
}
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram



Example

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

int main () {
  // for loop execution
  for( int a = 10; a < 20; a = a + 1 ) {
    cout << "value of a: " << a << endl;
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

```
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax

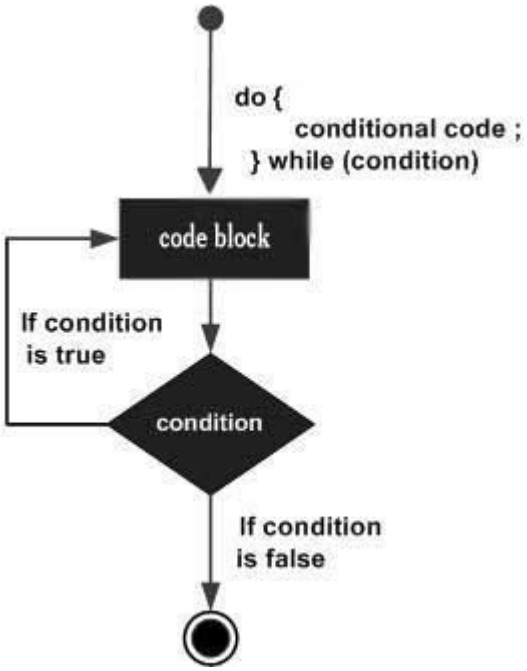
The syntax of a do...while loop in C++ is:

```
do {
  statement(s);
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram



Example

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

int main () {
  // Local variable declaration:
  int a = 10;

  // do loop execution
```

```
   do {
      cout << "value of a: " << a << endl;
      a = a + 1;
   }while( a < 20 );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

4)

Constant and Volatile Qualifiers

const

- const is used with a datatype declaration or definition to specify an unchanging value

  Examples:

  const int five = 5;

  const double pi = 3.141593;

- const objects may not be changed

  The following are illegal:

  const int five = 5;

  const double pi = 3.141593;

  pi = 3.2;

  five = 6;

volatile

- volatile specifies a variable whose value may be changed by processes outside the current program

One example of a volatile object might be a buffer used to exchange data with an external device:

int

check_iobuf(void)

{

volatile int iobuf;

```
        int val;

        while (iobuf == 0) {
        }
        val = iobuf;
        iobuf = 0;
        return(val);
    }
```

if iobuf had not been declared volatile, the compiler would notice that nothing happens inside the loop and thus eliminate the loop

- const and volatile can be used together
    - An input-only buffer for an external device could be declared as const volatile (or volatile const, order is not important) to make sure the compiler knows that the variable should not be changed (because it is input-only) and that its value may be altered by processes other than the current program

Enum:

An enumeration provides context to describe a range of values which are represented as named constants and are also called enumerators. In the original C and C++ enum types, the unqualified enumerators are visible throughout the scope in which the enum is declared. In scoped enums, the enumerator name must be qualified by the enum type name. The following example demonstrates this basic difference between the two kinds of enums:

C++

```
namespace CardGame_Scoped
{
   enum class Suit { Diamonds, Hearts, Clubs, Spades };

   void PlayCard(Suit suit)
   {
      if (suit == Suit::Clubs) // Enumerator must be qualified by enum type
      { /*...*/}
   }
}

namespace CardGame_NonScoped
{
   enum Suit { Diamonds, Hearts, Clubs, Spades };

   void PlayCard(Suit suit)
   {
      if (suit == Clubs) // Enumerator is visible without qualification
      { /*...*/
      }
   }
}
```

Every name in an enumeration is assigned an integral value that corresponds to its place in the order of the values in the enumeration. By default, the first value is assigned 0, the next one is assigned 1, and so on, but you can explicitly set the value of an enumerator, as shown here:

C++

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

The enumerator Diamonds is assigned the value 1. Subsequent enumerators, if they are not given an explicit value, receive the value of the previous enumerator plus one. In the previous example, Hearts would have the value 2, Clubs would have 3, and so on.

Every enumerator is treated as a constant and must have a unique name within the scope where the enum is defined (for unscoped enums) or within the enum itself (for scoped enums). The values given to the names do not have to be unique. For example, if the declaration of a unscoped enum Suit is this:

C++

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

Then the values of Diamonds, Hearts, Clubs, and Spades are 5, 6, 4, and 5, respectively. Notice that 5 is used more than once; this is allowed even though it may not be intended. These rules are the same for scoped enums.

Casting rules

Unscoped enum constants can be implicitly converted to int, but an int is never implicitly convertible to an enum value. The following example shows what happens if you try to assign hand a value that is not a Suit:

C++

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
```

A cast is required to convert an int to a scoped or unscoped enumerator. However, you can promote a unscoped enumerator to an integer value without a cast.

Typedef:

The C programming language provides a keyword called **typedef**, which you can use to give a type, a new name. Following is an example to define a term **BYTE** for one-byte numbers −

```
typedef unsigned char BYTE;
```

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

```
typedef unsigned char byte;
```

You can use **typedef** to give a name to your user defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

```
#include <stdio.h>
#include <string.h>

typedef struct Books {
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} Book;

int main( ) {

   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences −

- **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.
- **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

The following example shows how to use #define in a program −

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( ) {
   printf( "Value of TRUE : %d\n", TRUE);
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of TRUE : 1
Value of FALSE : 0
```

5) Write a C++ program i) to reverse a integer whole number. ii) to reverse string without using strrev( ) built-in function.

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

int main()
{
   int n, reversedNumber = 0, remainder;

   cout << "Enter an integer: ";
   cin >> n;

   while(n != 0)
```

```
    {
        remainder = n%10;
        reversedNumber = reversedNumber*10 + remainder;
        n /= 10;
    }

    cout << "Reversed Number = " << reversedNumber;

    return 0;
}
```

6) Explain switch statement. Write a program to perform simple calculator using switch case statement.

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

The syntax for a **switch** statement in C++ is as follows:

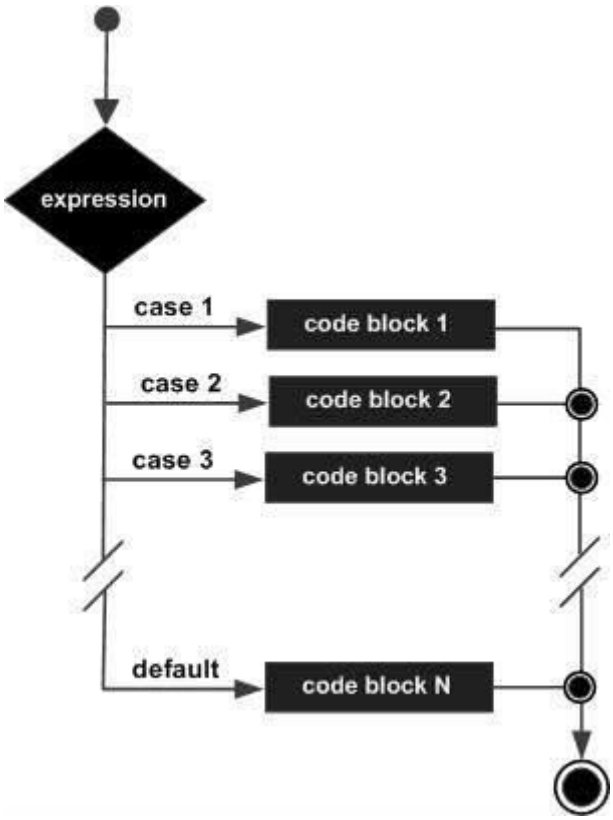
```
switch(expression){
  case constant-expression  :
     statement(s);
     break; //optional
  case constant-expression  :
     statement(s);
     break; //optional

  // you can have any number of case statements.
  default : //Optional
     statement(s);
}
```

The following rules apply to a switch statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram:



Example

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

int main () {
  // local variable declaration:
  char grade = 'D';

  switch(grade) {
    case 'A' :
```

```
      cout << "Excellent!" << endl;
      break;
    case 'B' :
    case 'C' :
      cout << "Well done" << endl;
      break;
    case 'D' :
      cout << "You passed" << endl;
      break;
    case 'F' :
      cout << "Better try again" << endl;
      break;
    default :
     cout << "Invalid grade" << endl;
  }

  cout << "Your grade is " << grade << endl;

  return 0;
}
```

Program to perform simple calculator:

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

int main()
{
   char op;
   float num1, num2;

   cout << "Enter operator either + or - or * or /: ";
   cin >> op;

   cout << "Enter two operands: ";
   cin >> num1 >> num2;
```

```
    switch(op)
    {
        case '+':
            cout << num1+num2;
            break;

        case '-':
            cout << num1-num2;
            break;

        case '*':
            cout << num1*num2;
            break;

        case '/':
            cout << num1/num2;
            break;

        default:
            // If the operator is other than +, -, * or /, error message is shown
            cout << "Error! operator is not correct";
            break;
    }

    return 0;
}
```

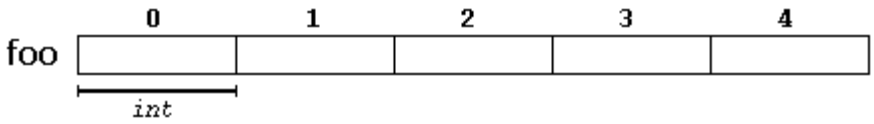
7. Define array. Write a program to find largest element in set of elements.

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:



where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

type name [elements];

where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

```
int foo [5];
```

NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

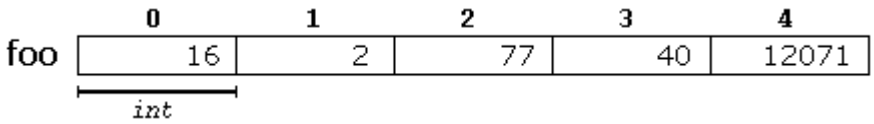
Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

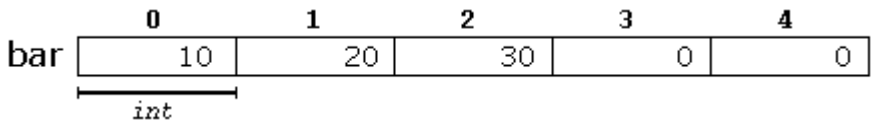
This statement declares an array that can be represented like this:



The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

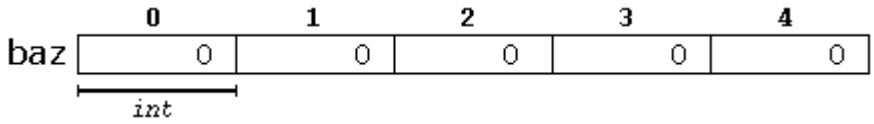
Will create an array like this:



The initializer can even have no values, just the braces:

```
int baz [5] = { };
```

This creates an array of five int values, each initialized with a value of zero:



When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty []. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

```
int foo[] = { 10, 20, 30 };
int foo[] { 10, 20, 30 };
```

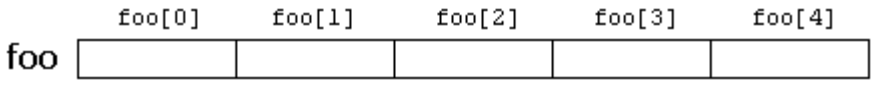
Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name[index]

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:



For example, the following statement stores the value 75 in the third element of foo:

```
foo [2] = 75;
```

and, for example, the following copies the value of the third element of foo to a variable called x:

```
x = foo[2];
```

```
int main()
{
   int i, n;
   float arr[100];
```

```
  cout << "Enter total number of elements(1 to 100): ";
  cin >> n;
  cout << endl;

  // Store number entered by the user
  for(i = 0; i < n; ++i)
  {
    cout << "Enter Number " << i + 1 << " : ";
    cin >> arr[i];
  }

  // Loop to store largest number to arr[0]
  for(i = 1;i < n; ++i)
  {
    // Change < to > if you want to find the smallest element
    if(arr[0] < arr[i])
       arr[0] = arr[i];
  }
  cout << "Largest element = " << arr[0];

  return 0;
        }
```

8) Explain pointer variable. What do you mean by dynamic memory allocation? How it is implemented in C++.

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined:

```
#include <iostream>

using namespace std;

int main () {
```

```
   int  var1;
   char var2[10];

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;

   cout << "Address of var2 variable: ";
   cout << &var2 << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

What Are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++:

There are few important operations, which we will do with the pointers very frequently. **(a)** we define a pointer variables **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <iostream>


using namespace std;
```

```
int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Dynamic memory allocation and Implementation :

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

The new and delete operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

```
new data-type;
```

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);

}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the delete operator as follows:

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how new and delete work:

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;
```

```
   delete pvalue;         // free up the memory.

   return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Value of pvalue : 29495
```

Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;   // Pointer initialized with null
pvalue  = new char[20]; // Request memory for the variable
```

To remove the array that we have just created the statement would look like this:

```
delete [] pvalue;        // Delete array pointed to by pvalue
```

Following is the syntax of new operator for a multi-dimensional array as follows:

```
int ROW = 2;
int COL = 3;
double **pvalue  = new double* [ROW]; // Allocate memory for rows

// Now allocate memory for columns
for(int i = 0; i < COL; i++) {
    pvalue[i] = new double[COL];
}
```

The syntax to release the memory for multi-dimensional will be as follows:

```
for(int i = 0; i < ROW; i++) {
   delete[] pvalue[i];
}
delete [] pvalue;
```

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

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

```
class Box {
  public:
    Box() {
      cout << "Constructor called!" <<endl;
    }

    ~Box() {
      cout << "Destructor called!" <<endl;
    }
};

int main( ) {
  Box* myBoxArray = new Box[4];

  delete [] myBoxArray; // Delete array

  return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```