Functions in C++

Problem-Solving Procedure With Modular Design:

Program development steps:

Analyze the problem

Develop a solution

Code the solution

Test/Debug the program

C ++ Function Definition:

A module or function in C++ performs a specific task.

-It is a set of C++ statements.

-It accepts inputs

-Processes it and provides **a single** output.

Advantages of using functions:

Modular way for program design makes it easy to:

- Maintain since we know which part of the program performs which task.

- Re-use, since we define the function once, but can use as many times as needed.

There are two types of functions:

Standard C++ functions and user defined functions.

Standard C++ functions:

These are the functions defined in the C++ standard library. They are not part of the language; their names are not *reserved words*.

There are several function libraries; the most common are the I/O library (for input /output operations and functions), the math library (for math operations and functions), string library, the time library…

Good programming means using library functions as much as possible and not re-inventing the wheel.

The Math Library functions:

Math library functions allow the programmer to perform certain common mathematical calculations.

Library functions are used in a program by writing the name of the function followed by the *argument* (or a comma separated list of arguments), which are included between parentheses.

For example, if we want to use the function that calculates the square root of 900.00 we might write

```
double a= sqrt(900.0);
```

When this statement is executed, the math library function sqrt is called to calculate the square root of the number contained in the parentheses (900.0).

The number 900.0 is called the argument of the sqrt function.

sqrt takes an argument of type double and returns a result of type double.

Almost all math functions take arguments of type double and return a result of type double.

In order to be able to use the math functions, we must include the math header file <math.h> at the beginning of our program.

A list of all available library functions is in the book.

User defined functions:

These are functions defined by the user and allow the program to be divided into modules each of which is specialized in performing a specific task.

Function definition:

```
return_type function_name (data_type var1, data_type va2, ..)
{
    function_body
    }
```

- ***function_name***

-The function name should be a valid C++ identifier**.**

-The name should be straightforward and describe what the function does.

-If you cannot find a suitable name for your function, then maybe its task is not well defined.

For example, a function that calculates power three of an integer will be called ***power_three.***

- ***(data_type var1, data_type va2, ..):***

-These are the arguments or parameters of the function. They are of the form:

datatype variable_name,

for example: int a, float b

- They describe the input data that the function uses to perform the task.

-Each argument should have its corresponding datatype.

If a datatype is omitted for an argument, the compiler assigns the ***integer type*** to that argument.

For example, the function power_three takes an integer as an input or argument and is defined as

power_three(int number)

-If the function does not take any arguments, we use empty parentheses **() or (void) as in main(void)**

int main(void){

….

return 0;}.

return_type

- This is the datatype of the result that the function returns as output.

For example, the function described above will process integers and the result will be an integer, so we say that it ***returns*** an integer.

We will declare the function power_three as:

int power_three(int number)

double averageThree(int number1, int number2, int number3)

With int indicating that the function returns an integer and int indicating that the function takes an integer argument.

-If a function ***does not return anything***, we use **void** as a return type.

```
void print_number(int number)

        { cout<<number

        }
```

-If the return type is omitted, then the C++ compiler assumes that the return value is int.

Therefore, fct(var) is equivalent to int fct1(int var).

{…..}

-A function is defined in a block.

-The beginning and ending of a block are noted with *braces*. An opening brace **{** for the beginning of the block, and a closing brace } for the end of the block.

function_body

A set of C++ declarations and instructions that perform the task.

-It contains the declaration for the variables that the function uses.

-It contains the statements that do the work.

The power_three function's body is:

```
int power_three(int number)
        { int cube = number*number*number;
                return cube;
        }/*end function */
```

Alternatively, it could be defined using a math library function:.

```
{               int y = pow(number,3);
                return y;
        }/*end function */
```

The complete definition of the function power_three is:

```
int power_three(int number)
{       int y = pow(number,3);
        return y;
        }/*end function */
```

Variables in functions:

Let's consider the statement:

```
int y = pow(number,3);
```

The variable y is declared *inside* the function body, it is a local variable.

A local variable is a variable that has been declared inside a function.

Local variables are only known in the function in which they are defined.

A function's arguments are **local variables**.

Local variables are **destroyed** once the function finishes executing.

Scope is a programming term for a part of a program in which a variable exists and can be used.

The scope of a local variable is the function_body in which it was created.

The **return** statement in a function:

There are 3 ways to return control to the calling program:

-if the return type is void, then control is returned when the end of block is reached "**}**". The return statement can also be used**.**

-If the return type is not void, we use:

return *expression.*

The data type of *expression* should be the same as the *return_type* declared in the function definition.

The value of *expression* is the value returned from the function

-the **return** statement can be used in several places in the function.

For example:

```
int find_max (int a, int b)
{       if ( a > b)
                return a;
        else return b;
}
int main(void) {
int y;
y=find_max(3,5);
cout<<y;
return 0;}
```

Calling a function:

Once the function has been defined, how do we use it?

In order to use a function, we ***invoke it*** or ***call it.***

To call a function, it is only necessary to write its name and replace its arguments with variables that have a value.

For example, if we want to use the previously defined function find_max (int a, int b), we would call it by using statements such as:

y =find_max(1,2)

The code associated with the function name is executed at that point in the program.

When the function terminates, the program execution continues with the statement, which follows the function call.

```
int main(void) {

    int x= 23;

    int y= power_three(x);

    cout<< x <<" to the power 3 is " << y<<endl;

    return 0;}
```

Before calling a function, we have to make sure that the arguments used to call a function are compatible with the type of its parameters.

If the types are incompatibles, the compiler does not issue an error message, but unexpected results occur.

One way to avoid this kind of error or mismatch in arguments is to define a *function prototype*.

Function prototype:

A function prototype provides the compiler with the following information:

- The type of data returned by a function,
- The number of parameters the function expects to receive,
- The data type of these parameters.
- The order in which these parameters are expected.

A function prototype has the form:

return_type function_name(data_type1var1, data_type2 var2,…)

The function prototype of power_three is:

int power_three (int);

-Function prototypes are listed at the beginning of the program.

-Often, they might be placed in a users .h (header) file.

-A function call that does not match a function prototype causes a syntax error.

-If the function prototype and the function definition disagree, this causes a syntax error.

For example, if we had defined the prototype as:

void power_three(int);

it would have caused a syntax error.

Therefore, the whole program for our power_three is:

```
#include <iostream>

#include <math.h>

using namespace std;

int power_three(int);/*prototype */

int main(void) {

int x= 23;

int y= power_three(x);/*fct call */

cout<<x <<”  to the power 3 is  “<< y;

return 0;}

int power_three (int number) {

        int y = pow(number,3);

        return y; }/*end function
```

Call by Value, Call by reference:

In general, arguments are passed to functions in one of two ways:

Call by value:

This method copies the value of an argument and the copied value is passed to the calling function in the formal parameter list.

Changes made to the value of the parameter list have no effect on the values of the variables with which it has been called.

For example:

```
int f (int x) { x= x+ 1;
                return x;}
int main(void) {
    int a = 2;
    int b= f (a); }
```

2

At the call f(a) copy of the value of a is made and used in the function call.

	Value of a	**Value of copy of a**
Before function call	2	2
After function call	2	3

Changes are made to the copy and do not effect the value of the caller variable.

Call by value should be used whenever the function does not need to modify the value of the caller's original variables.

By convention, all C++ function calls are call by value.

Call by reference:

In a call by reference, the user allows the value of the calling variable to be modified by the called function.

C++ simulates a call be reference by passing an address operator which points to the address of the calling arguments.

More on Data types and math functions:

The six data types that we have seen so far are:
char, int, float, double, bool, and void.

Except ***void and bool,*** the basic data types may have various modifiers preceding them.
These modifiers may alter the way the base type is represented.
This is done to fit various situations more precisely.
The modifiers are:

- signed
- unsigned
- long
- short

We can apply the modifiers signed, short, long and unsigned to characters and integers.
You may apply long to double.

However, you may not apply long to float because long has the same meaning as double.

-Applying the unsigned modifier to floating-point types (float, double) is not recommended.

Type	*Size in bytes(bits)*	*Range*
char	1 (8)	-127 to 127
unsigned char	1 (8)	0 - 255
signed char	1 (8)	-127 to 127
int	2 (16)	-32,767 to 32,767
unsigned int	2 (16)	0 - 65,535
signed int	2 (16)	Same as int
short int	2(16)	Same as int
unsigned short int	2 (16)	0 – 65,535
signed short integer	2 (16)	Same as int
long int	4 (32)	-2,147,483,647 to 2,147,483,647
signed long int	4 (32)	Same as long int
unsigned long int	4 (32)	0 to 4,294,967,295
float	4 (32)	Six digits of precision
double	8 (64)	Ten digits of precision
long double	16 (128)	Ten digits of precision

Type conversion

In library function calls:
If a library function is called with an argument of a different type than the one it has been defined with, it converts the arguments into the data type of its definition, this is called ***coercion of arguments.***

Note that:
Converting from a higher precision to a lower precision results in a loss of precision.

Converting from lower precision to a higher precision does not result is more accuracy rather it is a different representation.

Original type	***Target type***	***Loss***
int	char	High order 8-bits
long int	char	High order 24 bits
long int	int	High order 16 bits
float	int	Fractional part or more
double	float	Precision, result rounded
long double	double	Precision, result rounded

Symbolic Constants:

In C++, constants refer to fixed values that the program may not alter.

Each computer system has symbolic constants defined, which represent the maximum value of variables of a given type.

Random Number generator:

In many programs in which we want to simulate game playing or the element of chance a function called rand() is used.
rand() is a standard function that returns a pseudo-random number between 0 and a large constant named RAND_MAX (inclusive).
the value of RAND-MAX must be at least 32767.
rand() and RAND_MAX are defined in stdlib.h.

```
#include <iostream>

#include <stdlib.h>
using namespace std;
int main(void) {
        cout<< RAND_MAX;
        return 0;
    }
```

```
Program output is:
32767.
```

The range of values produced by rand() is usually different from the range of values needed for a specific application.

For example if we want to simulate coin tossing, we will only need two values, 0 for “heads” and 1 for “tails”, so the range is 0 or 1.

How can we go from a range of 0- 32767 to 0-1?

We have to perform what is called *scaling.*

For this we use the modulus operator %, which gives the rest of an integer division.

Since, what we want to generate a number that is either 0 or 1, we can use the rest of a division by 2, since the rest is always either a 0 or a 1.

Therefore, we select our scaling as the operation:

rand() %2.

The results are interpreted as:

- if the result is 0, it means that we got heads.
- if it is 1, it means we got tails.

Now we want to simulate the rolling of a die.

When we roll a die, we can have any of the six faces of the die with equal probability.

The values that we could get are 1, 2, 3, 4, 5, and 6, so the range is 1 - 6

We would like to get from a range of 0-32767 to a range of 1 to 6.

How do we do that ? To get the same type of scaling we use the same operator %

This time we divide rand() by 6, the range of results is [0..5], and we add 1 to it , so the range becomes [1-6]

```
#include <iostream>

#include <stdlib.h>

using namespace std;

int main(void) {

    int i;

    for (i=1; i<=20; i++) {

        cout<<"\t"<<1+ (rand() % 6)<<"\t";

        if (i%5 == 0)

        coutn<<endl;

        } /*end for */

    return 0;

    }/*end main */
```

Sample output:

```
5   3   5   5   2
4   2   5   5   5
3   2   2   1   5
1   4   6   4   6
```

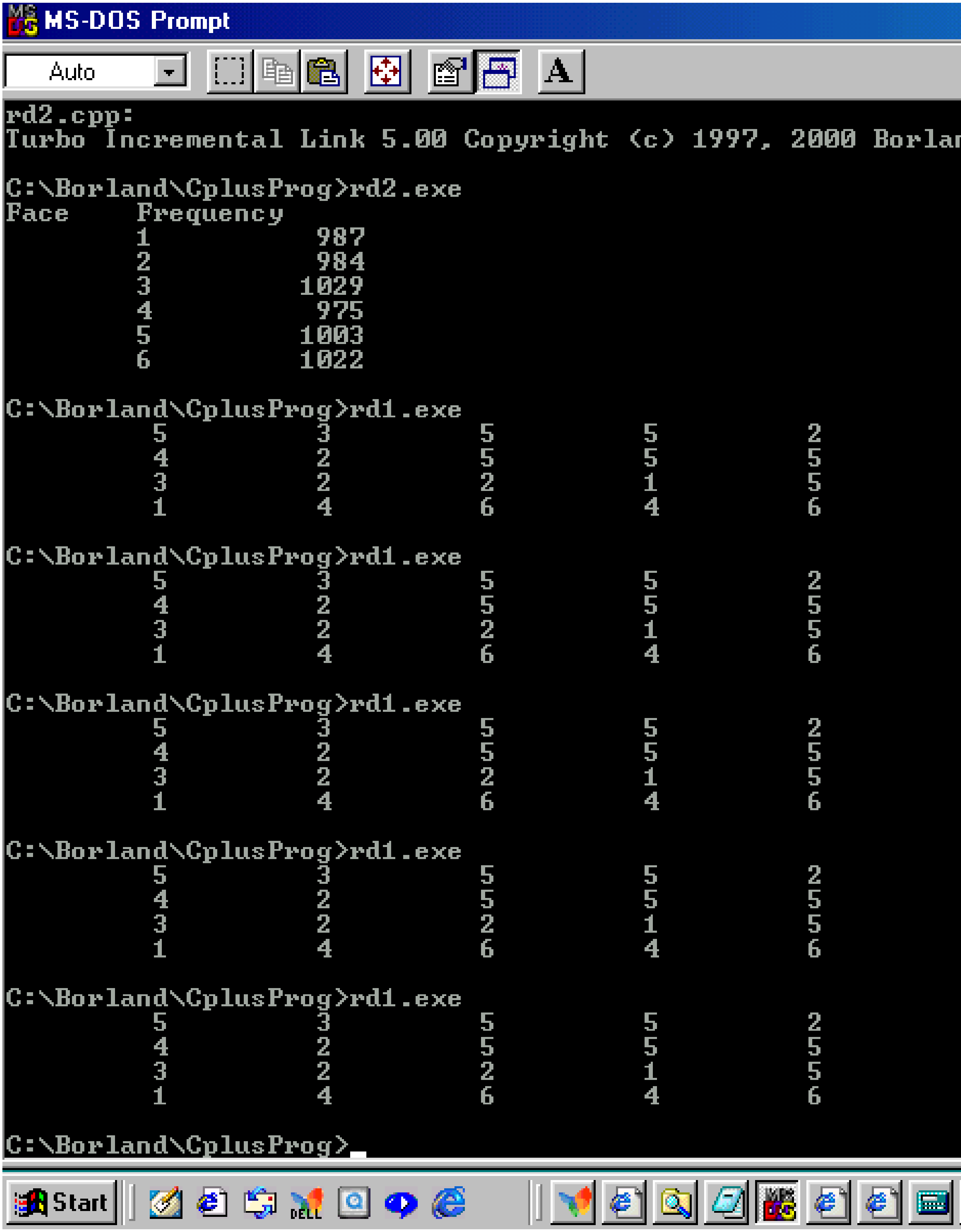
```
#include <iostream>
#include <stdlib.h>
using namespace std;
main() {
int face,roll, frequency1=0, frequency2=0,
frequency3=0, frequency4=0,
frequency5=0, frequency6=0;
for (roll=1; roll<=6000; roll++) {
    face = 1+ rand() % 6;
    switch (face) {
        case 1: ++frequency1;break;
        case 2:++frequency2;break;
        case 3:++frequency3; break;
```

```
        case 4:++frequency4; break;
        case 5:++frequency5; break;
        case 6:++frequency6; break;
        }/*end switch */
    } /*end for */
    cout<<  "Face \t \t Frequency \n”;
    cout<< “  1 ”<< frequency1<<endl;
    cout<< “  2 " << frequency2<<endl;
    cout<< "  3 ” << frequency3 <<endl;
    cout<< "  4 " << frequency4 <<endl;
    cout << " 5 "<< frequency5 << endl;
    cout << " 6 " << frequency6<< endl;
    return 0;
}/*end main */
```

Program output:

```
Face   Frequency
1      987
2      984
3      1029
```

4	975
5	1003
6	1022



```
rd2.cpp:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borla

C:\Borland\CplusProg>rd2.exe
Face    Frequency
        1          987
        2          984
        3         1029
        4          975
        5         1003
        6         1022

C:\Borland\CplusProg>rd1.exe
         5         3         5         5         2
         4         2         5         5         5
         3         2         2         1         5
         1         4         6         4         6

C:\Borland\CplusProg>rd1.exe
         5         3         5         5         2
         4         2         5         5         5
         3         2         2         1         5
         1         4         6         4         6

C:\Borland\CplusProg>rd1.exe
         5         3         5         5         2
         4         2         5         5         5
         3         2         2         1         5
         1         4         6         4         6

C:\Borland\CplusProg>rd1.exe
         5         3         5         5         2
         4         2         5         5         5
         3         2         2         1         5
         1         4         6         4         6

C:\Borland\CplusProg>rd1.exe
         5         3         5         5         2
         4         2         5         5         5
         3         2         2         1         5
         1         4         6         4         6

C:\Borland\CplusProg>_
```

Program Output at 1st call:

5	3	5	5	2
4	2	5	5	5
3	2	2	1	5
1	4	6	4	6

Program Output at 2sd call:

5	3	5	5	2
4	2	5	5	5
3	2	2	1	5
1	4	6	4	6

Program Output at 3rd call:

5	3	5	5	2
4	2	5	5	5
3	2	2	1	5
1	4	6	4	6

…….

On repeated runs of the program, we get random numbers, but we always get the same sequence of random numbers.

If the function ***rand()*** and examined the results, eventually one would detect a circular pattern in the sequence of numbers; thus, it is not purely random.

The return value of rand() is called a "**pseudo-**random number" because it is impossible on a digital computer to compute purely random numbers.

Instead, ***rand()*** produces a sequence of numbers that for most practical purposes appears random.

This is good while testing your program so that you can compare the results from two different runs of you program more easily.

The function srand()

That’s where srand() becomes important.

srand() randomizes or produces a different sequence of random numbers each time the program is run.

The srand() function is also defined in stdlib.h.

srand() is passed a single parameter of type “unsigned” called the “seed”.

The “seed” is a value that determines which sequence of random numbers will be generated from rand().

In order, to get a different sequence of random numbers every time the program is run a different seed must be given to srand().

-One way to do this is to ask the user to enter the seed:

```
/*Randomizing die-rolling program */
#include <stdlib.h>
#include <iostream>
using namespace std;
main() {
        int i;
        unsigned seed;
        cout<< “Enter seed”;
        cin>> seed;
        srand(seed);
        for (i=1; i <=10; i++)
         { cout<< “\t”<< 1+ (rand() %6));
                if (i % 5 ==0)
                cout<< endl;
        }/*end for */
        return 0;
}/*end main */
```

MS-DOS Prompt

Auto

```
C:\Borland\CplusProg>bcc32 seed.cpp
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 20
seed.cpp:
Turbo Incremental Link 5.00 Copyright (c) 1997, 20

C:\Borland\CplusProg>seed.exe
Enter seed    :45
          4          1          6          4          6
          1          6          4          6          2

C:\Borland\CplusProg>seed.exe
Enter seed    :5
          3          5          6          2          4
          4          3          3          1          6

C:\Borland\CplusProg>seed.exe
Enter seed    :67
          6          5          1          4          5
          6          3          1          2          1

C:\Borland\CplusProg>seed.exe
Enter seed    :0
          5          5          3          5          5
          2          4          2          5          5

C:\Borland\CplusProg>seed.exe
Enter seed    :1
          5          3          5          5          2
          4          2          5          5          5

C:\Borland\CplusProg>seed.exe
Enter seed    :7
          3          4          5          6          4
          6          4          2          4          2

C:\Borland\CplusProg>seed.exe
Enter seed    :8
          4          2          3          6          1
          2          4          3          4          4

C:\Borland\CplusProg>
C:\Borland\CplusProg>_
```

However, it is not a good idea to have the program wait for user input.

The best way to do this is to have the program get the computer's internal clock value as the seed.

Using this method, the only way to get identical output from the rand() would be to run the program at exactly the same time on different days.

The function time()

C has a library function found in "time.h" called time() that reads the internal clock.

Call time() with no parameters, and it will return a value of type "time_t".

The type, time_t, is also defined in time.h and may be different on different operating systems; we do not need to know how it is defined in order to solve the problem of initializing the random number generator.

We will take the result of time(), type cast it to unsigned, and pass it to srand(). The resulting statement becomes:

```
srand( (unsigned) time(NULL) );
```

This causes the computer to read its internal clock. The time function returns the current time of the day in seconds. This value is converted to an unsigned integer and used as the seed to the random number generator.

This statement should be put in the initialization section of the program so that it is called only once.

It will initialize the random number seed, and the output from calls to rand() will be a sequence of random numbers that is different virtually every time the program is run.

Game of craps:

- A player rolls two dice.
- After the dice have come to rest, the sum of the two upward faces is calculated.
- If the sum is 7 or 11, the player wins.
- If the sum is 2, 3, or 12, the player loses.
- If the sum is any other number, this becomes the player's "point".
- To win, the player must keep rolling until they make their points, that is score the same point again in a roll of dice.
- If the player rolls a 7 before making the point, they loose.

Input: Roll of the two dice.

Partial Output: Value of face on die1,
Value of face on die2.
Sum of values.

Output:

Game won, game lost, or play again:
-we need a variable that will take 3 different values depending on the game status.
Game_status =1 if game is won,
Game_status = 2 if game is lost
Game_status = 0 if game is still on play again

We need a variable that will store the sum of points at each throw

We need a variable that will store the player score in case the player needs to play again to match this score.

Pseudocode:
Simulate dice throw
Calculate sum
If sum = 7 or sum =11
Then Game_status = 1 (Game is won)
Else If sum = 2 or sum =3, or sum =12
  Then Game_status= 2, (Game is lost),

```
        Else  Player_points = sum.
                    Game_status =0 (Play again)
While Game_status =0 {
        Simulate dice throw
        Calculate sum
         if  sum = player_points
        then Game_status =1 (Game won)
        else
              if sum = 7
              then Game_status =2 (Game lost)
              }
if Game_status =1
then print "Player wins"
else print "Player loses"
*end *
```

Function Overloading:

Function Signature:

Each function has a signature. The function signature is the name of the function and the data types of the function's parameters in the proper order.

Function overloading is the practice of declaring the same function with different signatures. The same function name will be used with different number of parameters and parameters of different type. But overloading of functions with different return types are not allowed.

The function returntype is not part of the signature.

let us assume an AddAndDisplay function with different types of parameters.

- Sample code for function overloading

```
void AddAndDisplay(int x, int y)
{
    cout<<" C++ Tutorial - Integer result: "<< (x+y);
}

void AddAndDisplay(double x, double y)
{
    cout<< " C++ Tutorial - Double result: "<<(x+y);
}

void AddAndDisplay(float x, float y)
{
    cout<< " C++ Tutorial - float result: "<<(x+y);

}
```

Some times when these overloaded functions are called, they might cause ambiguity errors. This is because the compiler may not be able to decide what signature function should be called.

If the data is type cast properly, then these errors will be resolved easily. Typically, function overloading is used wherever a different type of data is to be dealt with.

For example this can be used for a function which converts farenheit to celsius and vice versa. One of the functions can deal with the integer data, other can deal float for precision etc.,

C++ Storage Classes

A storage class is the memory area in which a variable is created and stored. The memory area will determine the scope of the variable (which parts of the program as a whole can access it) and how long it lives. Types of storage class variables - Automatic, External and Static explained with examples.

Storage classes:

In the context of scope of variables in functions exists the important concept of storage class.

What is Storage Class?

Storage class defined for a variable determines the accessibility and longevity of the variable. The accessibility of the variable relates to the portion of the program that has access to the variable. The longevity of the variable refers to the length of time the variable exists within the program.

Types of Storage Class Variables in C++:

- Automatic
- External
- Static

Automatic:

Variables defined within the function body are called automatic variables. Auto is the keyword used to declare automatic variables. By default and without the use of a keyword, the variables defined inside a function are automatic variables.

For instance:

```
void exforsys( )
{
auto int x;
auto float y;
.........
............
}
```

is same as

```
void exforsys( )
{
int x;
float y;            //Automatic Variables
.........
............
}
```

In the above function, the variable x and y are created only when the function exforsys( ) is called. An automatic variable is created only when the function is called. When the function exforsys( ) is called, the variable x and y is allocated memory automatically. When the function exforsys( ) is finished and exits the control transfers to the calling program, the memory allocated for x and y is automatically destroyed. The term automatic variable is used to define the process of memory being allocated and automatically destroyed when a function is called and returns. The scope of the automatic variables is only within the function block within which it is defined. Automatic variable are also called local variables.

External:

External variables are also called global variables. External variables are defined outside any function, memory is set aside once it has been declared and remains until the end of the program. These variables are accessible by any function. This is mainly utilized when a programmer wants to make use of a variable and access the variable among different function calls.

Static:

The static automatic variables, as with local variables, are accessible only within the function in which it is defined. Static automatic variables exist until the program ends in the same manner as external variables. A static variable is initialized only once, when the function in which it has been defined is called for the first time, it maintains its value between function calls.

For example:

```
#include <iostream.h>
int example(int);
void main( )
{
int in,out;
in =1;
while(in!=0)
{
cout<<”Enter input value:”;
cin>>in;
out=example(in);
cout<”\nResult:”<<out;
}
cout<”\n End of Program”<<out;
}
```

```
int example(int x)
{
static int a=0;
static int b=0;
a++;
b=b+x;
return(b/a);
}
```

In the above program, the static variables a and b are initialized only once in the beginning of the program. Then the value of the variables is maintained between function calls.

When the program begins, the value of static variable a and b is initialized to zero. The value of the input in is 5 (which is not equal to zero) and is then passed to the function in variable x. The variable a is incremented thus making a as equal to 1. Variable b becomes equal to 5 and thus, the return of value from function exforsys( ) for the first time is 5, which is printed in the called function.

The second time the value of the input in is 7 (which is not equal to zero) and is passed to the function in variable x. The variable a (which is declared as static) has the previous value of 1. This is incremented and the value of a is equal to 2. The value of b is maintained from the previous statement as 5 and new value of b now is b=5+7 = 12 and thus, the return value

from the function is 12/2=6 which is printed in the called function.

Thus the output of the above program is:

```
Enter input value:5
Result:5
Enter input value:7
Result:6
Enter input value:0
End of Program
```