Functions الدوال

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

type name (parameter1, parameter2, ...) { statements }
Where:
- **type** , is the type of the value returned by the function.
- **name**, is the identifier by which the function can be called.
- **parameters** (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.

Use of function استخدام الدالة

عند استخدام أي داله يجب القيام بالاتي:

1- Function Declaration الاعلان عن الدالة

1- يكون الاعلان عن الدالة قبل الدالة الرئيسية main() وينتهي الإعلان بفاصلة منقوطة ;

```
#include <iostream>
int addition(int, int);
int main ()
{
  int x,y,sum;
  cout<<"enter values  x and y :\n";
  cin>> x >> y;
  sum = addition (x,y);
  cout << "The result is: " << sum;

}

int addition (int a, int b)
{
  int r =0;
  r =a+b;
  return r;
}
```

اعلان الدالة

كتابة تفاصيل الدالة

2- Function details تفاصيل الدالة

يتم كتابة تفاصيل الدالة أسفل الدالة الرئيسية main()

```
int addition (int a, int b)
{
int r=0 ;
r =a+b;
return  r ;
}
```

3-Functions with no type (void)

```
#include <iostream>

void printmessage()
{
  cout << "I'm a function!";
}

void main ()
{
  printmessage ();
}
```

4 - Function Calling استدعاء الدالة

يتم استدعاء الدالة داخل البرنامج الرئيسي main( ) عن طريق اسمها مع مراعاة نوع وعدد المعلمات التي تستقبلها. وبذلك فان الدالة addition التي كتبت تفاصيلها اعلاه يمكن استدعائها بالصيغ التالية:

a- Arguments passed by clear value:

addition (5,7); استدعاء بقيم محددة

parameters in the function declaration have a clear correspondence to the arguments passed in the function call.
At the point which the function is called from within main, the control is passed to function addition: here, execution of main is stopped, and will only resume once the addition function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables int a and int b within the function.

b- Arguments passed by value:

```
int x=5, y=3, z;
z = addition ( x, y );
```

In this case, function addition is passed 5 and 3, which are copies of the values of x and y, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the

function's definition, but any modification of these variables within the function has no effect on the values of the variables x and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.

ملاحظة:

1- يمكن استدعاء الدالة عن طريق اسمها فقط دون اسناد الناتج الى المتغير z :

addition (x , y);

2– هناك طريقة اخرى يمكن بواسطتها الاستغناء عن اعلان الدالة قبل البرنامج الرئيسي , وهي كتابة الدالة كاملة قبل البرنامج الرئيسي main() .

c-Arguments passed by reference:

In certain cases, though, it may be useful to access an external variable from within a function. To do that, arguments can be passed *by reference*, instead of *by value*. For example, the function duplicate in this code duplicates the value of its three arguments, causing the variables used as arguments to actually be modified by the call:

```
#include <iostream>

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=2, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

To gain access to its arguments, the function declares its parameters as *references*. In C++, references are indicated with an ampersand (&) following the parameter type, as in the parameters taken by duplicate in the example above.

In fact, a, b, and c become aliases of the arguments passed on the function call (x, y, and z) and any change on a within the function is actually modifying variable x outside the function. Any change on b modifies y, and any change on c modifies z. That is why when, in the example, function duplicate modifies the values of variables a, b, and c, the values of x, y, and z are affected.

Example: Write C++ program to find the maximum number between tow numbers by using functions?

```
#include <iostream>
int  max(int   num1, int   num2)
 {
    int   result;
    if (num1> num2)
      result=num1;
    else
      result=num2;
 }

int main()
{
   int   a=100,  b=50;
   int   res;
   res=max(a,b);
   cout<<" MAXIMUM  NUMBER IS :"<< res <<endl;
   return 0;
}
```

Recursivity الدوال ذاتية الاستدعاء

Recursivity is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1
More concretely, 5! (factorial of 5) would be: 5! = 5 * 4 * 3 * 2 * 1 = 120

And a recursive function to calculate the factorial in C++ could be:

```
#include <iostream>

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main (){

  long number = 5;
  cout<< number << "! = " << factorial (number);
  return 0;}
```

H.W:

1- اكتب برنامج لتوليد N من حدود سلسلة فيبوناتشي باستخدام recursive function :

1,1,2,3,5,8,13 ,……N

2-Write c++ program to calculate the GCD (القاسم المشترك الاعظم) by using recursive function .

Functions Overloading التحميل الزائد للدوال

Two or more functions having same name but different argument(s) are known as overloaded functions.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different. These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int  test( )   {   }

int  test(int  a)  {   }

float test(double  a)   {  }

int  test(int   a , double    b)   {  }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
//Error  code

Int  test(int   a)  {   }

Double   test(int  b)  {  }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example:

```
#include <iostream>

void display(int);
void display(float);
void display(int, float);

int main() {

    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);

    return 0;
}

void display(int var) {
    cout << "Integer number: " << var << endl;
}

void display(float var) {
    cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Function templates:

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class mytype>
mytype GetMax (mytype a, mytype b) {
 return (a>b?a:b);
}
```

To use this function template we use the following format for the function call:

```
<type> function_name (parameters);
```

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
(int) GetMax (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

```
// function template
#include <iostream>

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}

int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=(int) GetMax(i,j);
  n=(long) GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = (int,long) GetMin (j,l);
```

or simply:

```
i = GetMin (j,l);
```