مدرس المقرر: د.عيسى الغنام 2023-2024

Lecture No.7

Example: exp(x) using a 21-term Taylor series

```
#include <iostream>
using namespace std;
int main() {
        double x;
        cout << "Enter a real number: ";
        cin >> x;

        // Approximate exp(x) using a 21-term Taylor series:
        //
        //          1   2   1   3          1   20
        // 1 + x + -- x  + -- x  + ... + --- x
        //          2!      3!          20!

        double result = 1.0;
        double term = 1.0;
        int n = 1;
        while ( n <= 20 ) {
                term = term * x / static_cast<double>( n );
                result = result + term;
                n = n + 1;
        }
        cout << "The approximation of exp(" << x << ") is " << result << endl;
        return 0;
}
/* n=1 =>   term=5/1=5            res=1+5                                    n=2
n=2 =>   term=5*(5/2)          res=1+5+ (5*5)/2                           n=3
n=3 =>   term=((5*5)/2)*(5/3)    res=1+5+ (5*5)/2+ (5*5*5)/(3*2)                n=4
n=4 =>   term=((5^3)/6)*(5/4)    res=1+5+ (5*5)/2+ (5*5*5)/(3*2) + (5*5*5*5)/(4*3*2)    n=5 */
```

Example 2: insert an element to array

```
# include <iostream>
using namespace std;
const int size = 50;
int main()
{       int a[size];
        int num, k, n = 0, i, j;
        char ch;
        cout << "enter a array and 0 to end :";
        for (i = 0; i<size; i++){
                cin >> a[i];
                if (a[i] == 0)break;
                n++;}
        cout << "a array is :";
        for (i = 0; i<n; i++)
                cout << a[i] << "  ";
        cout << endl;
        while (n<size){
                cout << "enter the element to insert it:";
                cin >> num;
               do{cout << "enter the index:";
                cin >> k;}
                while(k<0 || k> n-1);
                for (j = n - 1; j >= k; j--)
                        a[j + 1] = a[j];
                a[k] = num;
                n++;
                cout << "the new array is:";
                for (i = 0; i<n; i++)cout << a[i] << " ";
                cout << endl;
                cout << "press anykey to insert another element and n to finish :";
                cin >> ch;
```

```
            if (ch == 'n')break;}
return 0;}
```

Part1 C++ Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Create a Function

C++ provides some pre-defined functions, such as main(), which is used to execute code. But you can also create your own functions to perform certain actions.

To create (often referred to as *declare*) a function, specify the name of the function, followed by parentheses **()**:

Syntax

```
void myFunction() {
 // code to be executed
}
```

Example Explained

myFunction() is the name of the function

void means that the function does not have a return value. You will learn more about return values later in the next chapter

inside the function (the body), add code that defines what the function should do

Call a Function

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are called. To call a function, write the function's name followed by two parentheses () and a semicolon ; In the following example, myFunction() is used to print a text (the action), when it is called:

EXAMPLE: Inside main, call myFunction():

```
// Create a function
void myFunction() {
 cout << "I just got executed!";
}

int main() {
 myFunction(); // call the function
 return 0;
}
// Outputs "I just got executed!"
```

A function can be called multiple times: Example:

```
void myFunction() {
 cout << "I just got executed!\n";
}
int main() {
 myFunction();
 myFunction();
 myFunction();
 return 0;
}
// I just got executed!
// I just got executed!
// I just got executed!
```

Function Declaration and Definition

A C++ function consist of two parts:

- **Declaration:** the return type, the name of the function, and parameters (if any)
- **Definition:** the body of the function (code to be executed)

```
void myFunction() { // declaration
 // the body of the function (definition)
}
```

Note: If a user-defined function, such as myFunction() is declared after the main() function, **an error will occur**:

Example

```
int main() {
 myFunction();
 return 0;}

void myFunction() {
 cout << "I just got executed!";
}
// Error
```

However, it is possible to separate the declaration and the definition of the function - for code optimization.

You will often see C++ programs that have function declaration above main(), and function definition below main(). This will make the code better organized and easier to read: **Example:**

```
// Function declaration
void myFunction();
// The main method
int main() {
myFunction();  // call the function
 return 0;}
```

```
// Function definition
void myFunction() {
 cout << "I just got executed!";}
```

Parameters and Arguments

Information can be passed to functions as a parameter. Parameters act as variables inside the function.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma:

Syntax

```
void functionName(parameter1, parameter2, parameter3) {
 // code to be executed
}
```

The following example has a function that takes a string called fname as parameter. When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
void myFunction(string fname) {
 cout << fname << " Refsnes\n";
}

int main() {
 myFunction("Liam");
 myFunction("Jenny");
 myFunction("Anja");
 return 0;
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

When a **parameter** is passed to the function, it is called an **argument**. So, from the example above: fname is a **parameter**, while Liam, Jenny and Anja are **arguments**.

Default Parameter Value

You can also use a default parameter value, by using the equals sign (=). If we call the function without an argument, it uses the default value ("Norway"):

Example

```
void myFunction(string country = "Norway") {
 cout << country << "\n";
}

int main() {
 myFunction("Sweden");
 myFunction("India");
 myFunction();
 myFunction("USA");
 return 0;
}

// Sweden
// India
// Norway
// USA
```

A parameter with a default value, is often known as an "**optional parameter**". From the example above, country is an optional parameter and "Norway" is the default value.

Multiple Parameters

Inside the function, you can add as many parameters as you want:

Example

```
void myFunction(string fname, int age) {
  cout << fname << " Refsnes. " << age << " years old. \n";
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Liam Refsnes. 3 years old.
// Jenny Refsnes. 14 years old.
// Anja Refsnes. 30 years old.
```

Note that when you are working with multiple parameters, the function call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Return Values

The void keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int, string, etc.) instead of void, and use the return keyword inside the function:

Example

```
int myFunction(int x) {
  return 5 + x;
}
```

```
int main() {
 cout << myFunction(3);
 return 0;
}

// Outputs 8 (5 + 3)
```

This example returns the sum of a function with **two parameters**:

Example

```
int myFunction(int x, int y) {
 return x + y;
}

int main() {
 cout << myFunction(5, 3);
 return 0;
}

// Outputs 8 (5 + 3)
```

You can also store the result in a variable:

Example

```
int myFunction(int x, int y) {
 return x + y;
}
int main() {
 int z = myFunction(5, 3);
 cout << z;
 return 0;
}

// Outputs 8 (5 + 3)
```

C++ Pass Array to a Function

Pass Arrays as Function Parameters

You can also pass arrays to a function:

Example

```
void myFunction(int myNumbers[5]) {
  for (int i = 0; i < 5; i++) {
    cout << myNumbers[i] << "\n"; }
}
int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  myFunction(myNumbers);
  return 0;}
```

Example Explained

The function (myFunction) takes an array as its parameter (int myNumbers[5]), and loops through the array elements with the for loop.

When the function is called inside main(), we pass along the myNumbers array, which outputs the array elements.

Note that when you call the function, you only need to use the name of the array when passing it as an argument myFunction(myNumbers). However, the full declaration of the array is needed in the function parameter (int myNumbers[5]).

C++ Function Overloading

With **function overloading**, multiple functions can have the same name with different parameters:

Example

int myFunction(int x)

float myFunction(float x)

double myFunction(double x, double y)

Consider the following example, which have two functions that add numbers of different type:

Example

```
int plusFuncInt(int x, int y) {
 return x + y;
}
double plusFuncDouble(double x, double y) {
 return x + y;
}
int main() {
 int myNum1 = plusFuncInt(8, 5);
 double myNum2 = plusFuncDouble(4.3, 6.26);
 cout << "Int: " << myNum1 << "\n";
 cout << "Double: " << myNum2;
 return 0;}
```

Instead of defining two functions that should do the same thing, it is better to overload one. In the example below, we overload the plusFunc function to work for both int and double:

Example

```
int plusFunc(int x, int y) {
 return x + y;
}

double plusFunc(double x, double y) {
 return x + y;
}
int main() {
 int myNum1 = plusFunc(8, 5);
 double myNum2 = plusFunc(4.3, 6.26);
 cout << "Int: " << myNum1 << "\n";
 cout << "Double: " << myNum2;
 return 0;}
```

Note: Multiple functions can have the same name as long as the number and/or type of parameters are different.

C++ Recursion

Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example

```
int sum(int k) {
 if (k > 0) {
  return k + sum(k - 1);
 } else {
  return 0;
 }
}

int main() {
 int result = sum(10);
 cout << result;
 return 0;
}
```

Example Explained

When the sum() function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

10 + sum(9)

10 + (9 + sum(8))

10 + (9 + (8 + sum(7)))

...

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0

Since the function does not call itself when k is 0, the program stops there and returns the result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

انتهت المحاضرة