Introduction to Programming (in C++)

Subprograms: procedures and functions

Jordi Cortadella, Ricard Gavaldà, Fernando Orejas
Dept. of Computer Science, UPC

Subprograms

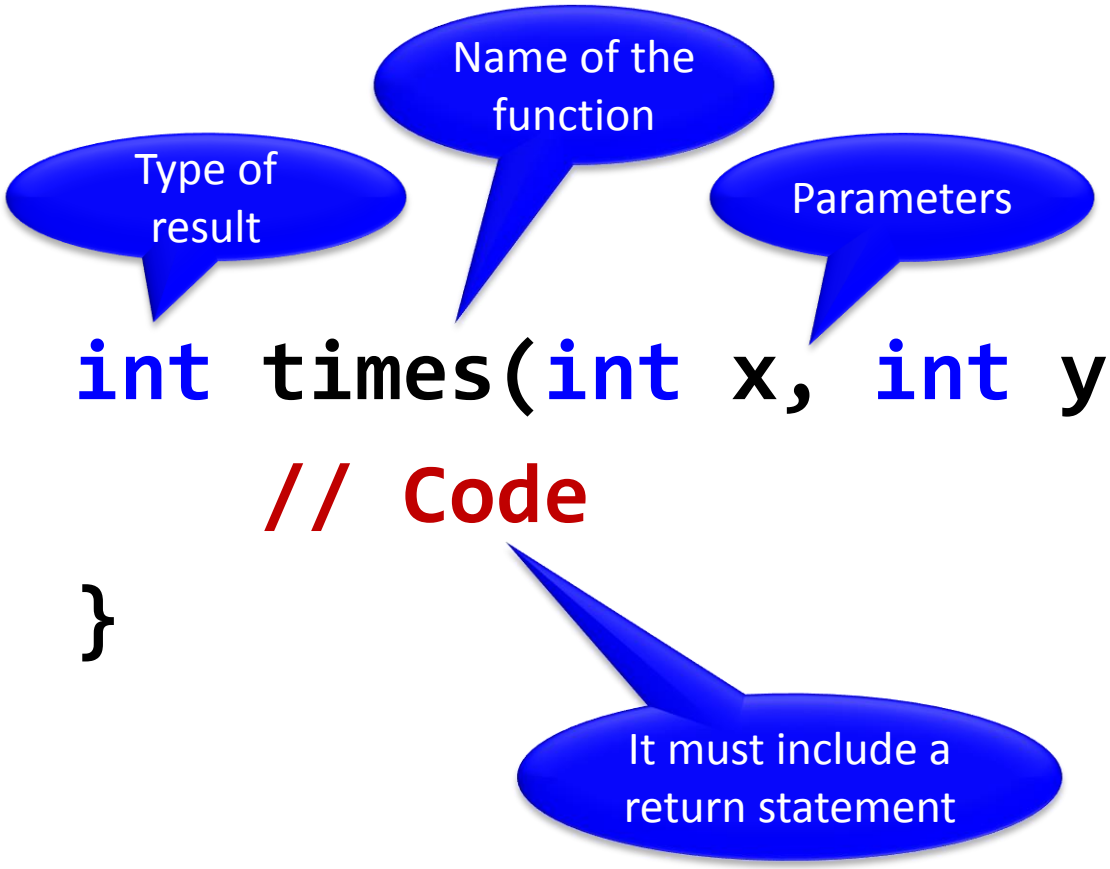
- Programming languages, in particular C++, not only provide a set of basic operations and statements, but also a means to define *our own* operations and statements.

- We call the operations and statements that we define *functions* and *procedures*, respectively.

- Procedures and functions (*subprograms*) may have parameters. These represent the objects from our program that are used in the subprogram.

Subprograms

- Functions are defined as follows:



Subprograms

```
int times(int x, int y) {
    int p = 0;
    while (y > 0) {
        if (y%2 == 0) {
            y = y/2;
            x = x*2;
        }
        else {
            p = p + x;
            y = y - 1;
        }
    }
    return p;
}
```

Subprograms

- Procedures are defined similarly, but without delivering any result:

No result

```
void factors(int x) {
    // Code
}
```

Subprograms

```
void factors(int x) {
    int f = 2;
    while (x != 1) {
        if (x%f == 0) {
            cout << f << endl;
            x = x/f;
        }
        else f = f + 1;
    }
}
```

Subprograms

- Subprogram definitions may appear before or after the main program.

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

int f() {
    // Code for f
}

int main() {
    // Code for the main program
}

void p(int a) {
    // Code for p
}
```

Subprograms

- A function can only be used if previously declared. A function can be declared and used before its code is defined.

```
double volume_sphere(double radius);

void some_geometry() {
    ...
    double V = volume_sphere(1.0);
    ...
}

double volume_sphere(double radius) {
    return 4*Pi*radius*radius*radius/3;
}
```

Subprograms

- Once a subprogram has been declared, it can be used.
  - Functions are used as operations within expressions.
  - Procedures are used as statements.

```
i = times(3, i + 2) + 1; //
function
...
factors(i);                // procedure
...
```

Subprograms

- Appropriate use of subprograms:
  - Increases *readability*: programs are better structured and easier to understand.
  - Enables the use of *abstraction* in the program design.
  - Facilitates *code reuse*.

Subprograms

- Evaluating the expression

```
times(3, i + 2) + 1
```

means executing the code of **`times`** over the arguments **`3`** and **`i+2`** and then adding **`1`** to the result returned by the function.

Subprograms

- Evaluating the statement

```
factors(i);
```

means executing the code of **`factors`** over the argument **`i`**.

Subprograms: parameter passing

- When a subprogram is called, the arguments are *passed* to the subprogram, so that its code can be executed:

```
times(3, i + 2) + ...

int times(int x, int y) { … }
```

- Each argument must have the *same type* as its corresponding parameter.

Subprograms: parameter passing

- In general, any expression can be the argument of a subprogram:

```
double maximum(double a, double b);
...
z = maximum(x, y);
...
r = maximum(3, gcd(s - 4, i) + alpha);
...
m = maximum(x, maximum(y + 3, 2*Pi*radius));
```

Subprograms: parameter passing

- An object (a variable) is associated with a *value* and a *memory location*. In C++, there are two methods for parameter passing:
  - Passing the value (*call-by-value*). This is denoted by just declaring the type and the name of the parameter.
  - Passing the memory location (*call-by-reference*). This is denoted by adding the symbol **&** next to the parameter type.

```
void p(int x, int& y) { ... }
```

Call-by-value

Call-by-reference

Subprograms: parameter passing

- Call-by-value makes a copy of the argument at the beginning of the subprogram. It is equivalent to having, a statement that assigns the value of each argument to the corresponding parameter:

```
times(3, i + 2)
```

is equivalent to:

```
int times(int x, int y) {
    x = 3; y = i + 2;
    int p = 0;
    ...
}
```

Subprograms: parameter passing

- The effect of call-by-reference is that the parameter becomes the same object (variable) as the argument, i.e., the parameter becomes an *alias* of the argument.

- Example: procedure to swap the value of two variables

```
void exchange(int& x, int& y) {
    int z = x;
    x = y;
    y = z;
}
```

Subprograms: parameter passing

```
exchange(a, b)
```

Is equivalent to having:

```
void exchange(int& x, int& y) {
    int z = a;
    a = b;
    b = z;
}
```

Subprograms: parameter passing

```
// Pre: n >= 1
// Post: returns whether n is prime.
//   If it is not prime, d is a divisor.
bool is_prime(int n, int& d) {
    d = 2;
    bool prime = (n != 1);
    while (prime and d < n) {
        if (n%d == 0) prime = false;
        else d = d + 1;
    }
    return prime;
}
```

```
int x, divisor;
bool p;
```

x: 6

divisor: 3

p: false

```
...

cin >> x;
p = false;

...
```

reference

copy

n: 9

d:

prime: false

Warning: we do not recommend the use of non-void functions with reference parameters in this course.

Subprograms: parameter passing

- Use *call-by-value* to pass parameters that must not be modified by the subprogram.

- Use *call-by-reference* when the changes made by the subprogram must affect the variable to which the parameter is bound.

- In some cases, call-by-reference is used to avoid copies of large objects, even though the parameter is not modified.

Subprograms: parameter passing

- To define a subprogram that, given two integers x and y, returns their quotient and remainder, we can write:

```
void div(int x, int y, int& q, int& r) {
    q = x/y;
    r = x%y;
}
```

Subprograms: parameter passing

- For instance, if the parameters would be passed by reference in the function **`times`**, after the execution of the statements:

```
int a = 4;
int b = 2;
int c = times(a, b);
```

the value of **`a`** would be **`0`** and the value of **`b`** would be **`8`** (and the value of **`c`** would be **`8`**).

Subprograms: parameter passing

- For instance, after the definition:

```
void exchange(int x, int y) {
    int z = x;
    x = y;
    y = z;
}
```

the statement **`exchange(a,b)`** would not have any effect on **`a`** and **`b`**.

Subprograms: parameter passing

- A call-by-value parameter can receive any expression as an argument.

- A call-by-reference parameter can only be bound to variables.

```
void exchange (int& a, int& b);
...

exchange(a, b + 4);
```

Incorrect parameter passing.

The Least Common Multiple (LCM)

- Design a function that calculates the LCM of two numbers. Assume that we can use a function **gcd(a,b)** that calculates the greatest common divisor.

```
// Pre:  a>0, b>0
// Post: returns the LCM of a and b
int lcm(int a, int b) {
    return (a/gcd(a,b))*b;
}
```