Introduction to C++

Pt. 2

Emanuele Giona Department of Computer Science, Sapienza University of Rome

Internet of Things A.Y. 2022/23

Prof. Chiara Petrioli Department of Computer, Control and Management Engineering, Sapienza University of Rome
Dott. Michele Nati

Exercise 1: solution


```cpp
#include <iostream>

using namespace std;

int main(){
    int n1 = 0;
    float n2 = 0.f;
    float sum = 0.f;

    cout << "Please insert an integer value:" << endl;
    cin >> n1;
    cout << "Please insert a floating-point value:" << endl;
    cin >> n2;

    sum = n1 + n2;
    cout << "The sum of " << n1 << " and " << n2 << " equals to " << sum << endl;
    return 0;
}
```

Exercise 2: solution


```cpp
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

int main(){
    string nStr = "";
    int n = 0;
    string output = "";

    stringstream ss;

    cout << "Please insert an integer value:" << endl;
    cin >> nStr;

    ss << nStr;
    ss >> n;

    n = n + 5;

    ss.str("");
    ss.clear();

    ss << n;
    ss >> output;

    cout << "Summing 5 to " << nStr << " equals to " << output << endl;
    return 0;
}
```

1. Operators

Basic operators

- Assignment (=)
- Arithmetic operations

  Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulo (%)
- Compound assignment

  Arithmetic operation using the current value of a variable, assigning it the resulting value afterwards

  +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

Bitwise operators act on integer data at the bit level

Examples

```
y += 2 * x;     →   y = y + 2 * x;
a %= 3;         →   a = a % 3;
b /= 4;         →   b = b / 4;
```

```
int v = 10;
v <<= 1;        →   v = v << 1;
```

Base-2 representation of 10 is 1010. The shift-left operator (<<) is instructed to shift by 1 bit towards left, yielding a Base-2 value of 10100, and thus effectively multiplying by 2.

Increment and decrement operators

As seen in the name of C++ itself, there are specialized operators for incrementing (++) and decrementing (--) the value of a variable.

There are two ways to use such operators, yielding important consequences on a program:

- Prefix (++x, --x)
  The operation takes place before the evaluation of the variable contents

```
int a = 3;
int b = ++a;
```

→ a is incremented to 4 first and then b is assigned a's content (b = 4)

- Suffix (x++, x--)
  The operation takes place after the evaluation of the variable contents

```
int a = 3;
int b = a++;
```

→ b is assigned a's content first and then a is incremented to 4 (b = 3)

This might be a cause for silent failures in your program!

Relational, comparison, and logical operators

- Relational and comparison operators are used to compare two expressions and they evaluate to logical values (either `true` or `false`)
  Equality (==), not equality (!=), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=)
- Logical operators perform Boolean logic operations
  NOT (!), AND (&&), OR (||)

Examples

```
!true                    →   false
0 >= 4                   →   false
!(0 >= 4)                →   true
(5 == 5) && (3 > 6)      →   false
(5 == 5) || (3 > 6)      →   true
```

2. Program Flow Control

Statements

A C++ program consists of a series of statements that are executed in sequence, and there are several types of them:

- Labeled
- Expression
- Compound
- Selection (or conditional)
- Iteration
- Jump
- Declaration
- Try *blocks*
- Atomic / synchronization *blocks*

Also called block, it allows the sequential execution of multiple statements in place of a single statement
This can be done by simply enclosing them with curly brackets { }

These statements implement transactional memory, in the context of parallel execution (*i.e.* not allowing variables in such blocks to expose *thread-unsafe* states)

Each statement has a specific purpose and syntax, and *most* of them have to be succeeded by a semicolon (;).

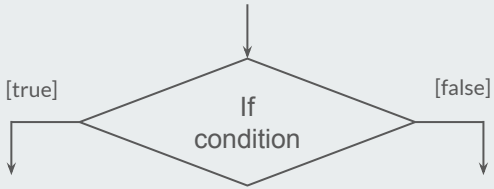
A special expression statement: the null statement is composed by just a semicolon.

`if` and `switch` statements

Selection statements, used to implement a choice among multiple control flows within a program

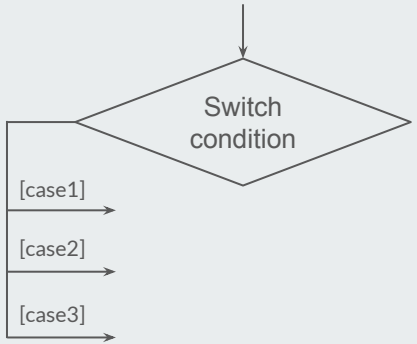
`if` statement

- Determines the execution of a statement depending on a condition
- Suitable for unstructured conditions
- Usually employed in cases of relatively small number of alternative paths



`switch` statement

- Determines the execution of one among several statements, transferring control to it as well
- Suitable for structured conditions
- Also employed in cases of high number of alternative paths



`if` syntax

```
if ( condition ) statement[1] else statement[1]
```

The `else` clause is optional within the syntax of the `if` statement

Expression evaluating to a `bool` value

The first statement is executed if the condition evaluates to `true`, otherwise it is skipped. In the presence of an `else` clause instead, the second statement is the one executed.

The statement used in the `else` clause might itself be an `if` statement, obtaining a nested `if` statement.

`switch` syntax

```
switch ( condition ) statement[1]
```

Expression evaluating to an integral value, such as: `char`, `signed char`, `unsigned char`, `short int`, `signed short int`, `unsigned short int`, `int`, `signed int`, and `unsigned int`

The statement is often a compound statement consisting of special statements:

- `case` statements (one or more)
- `default` statement (at most one)

1: Both single and compound statements are allowed

`if` examples

```
1 if (n % 2 == 0){
2   cout << "The number is even" << endl;
3 }
```

```
1 if (n % 2 == 0){
2   cout << "The number is even" << endl;
3 }
4 else {
5   cout << "The number is odd" << endl;
6 }
```

```
1 if (n % 2 == 0){
2   cout << "Divisible by 2" << endl;
3 }
4 else if (n % 3 == 0){
5   cout << "Divisible by 3" << endl;
6 }
7 else {
8   cout << "Not divisible by 2 or 3" << endl;
9 }
```

`switch` examples


```
1 switch (n % 2) {
2   case 0:
3     cout << "The number is even" << endl;
4     break;
5 }
```

```
1 switch (n % 2) {
2   case 0:
3     cout << "The number is even" << endl;
4     break;
5   default:
6     cout << "The number is odd" << endl;
7     break;
8 }
```

Last `if` example shows an unstructured condition, hence the switch statement is not appropriate.

More on the `switch` statement


```
1 switch (n % 2) {
2   case 0:
3     cout << "The number is even" << endl;
4     break;
5   default:
6     cout << "The number is odd" << endl;
7     break;
8 }
```

Both `case` and `default` statements can use compound statements after the column (`:`). These statements are used to select a flow based on a specific value (`case`), or execute a flow for all values which were not previously considered by `case` statements (`default`).

Moreover, the `break` statement is required to stop execution fallthrough: in the `switch` statement, whenever the program flow executes a specific `case` block the execution continues to the next defined block. In order to prevent this, each `case` block can control the flow by ending the execution of successive blocks within a `switch` statement by inserting a `break` statement.

Iterative statements (or loops)

These statements are used to repeat the execution of a block, subject to the evaluation of a condition. There are three types of loops provided by C++:

- `while ( condition ) statement`
  Statement execution after evaluation of `condition`, repeating it until the condition evaluates to `true`
- `do statement while ( condition );`
  Statement execution before evaluation of `condition`, if it evaluates to `true`, it will be repeated until it does
- `for ( initializer; condition; expression statement`[1]`) statement`
  `initializer` is executed first and only once at the start, then `condition` is evaluated; if it evaluates to `true`, the statement is executed, after which the `expression statement` is executed. The repetition occurs until `condition` evaluates to `true` after the first run

Loops support some jump statements to provide more granular flow control: `break` and `continue` statements can be included within the statement block that should be repeated.

1: One or more expression statements are supported; for multiple statements, each statement must be succeeded by a comma (,) instead of a semicolon (;).

Printing first 5 integers

while

```
1 int n = 0;
2 while(n < 5){
3   cout << n << endl;
4   n++;
5 }
```

- `condition` must be `true` in order to run for the first time
- Repetition occurs until `condition` evaluates to `true` after statement execution
- Statement must handle updating `condition` variables (if any)

do while

```
1 int n = 0;
2 do {
3   cout << n << endl;
4   n++;
5 } while (n < 5);
```

- First run is executed before `condition` evaluation
- Repetition occurs until `condition` evaluates to `true` after statement execution
- Statement must handle updating `condition` variables (if any)

for

```
1 for(int n=0; n<5; n++) {
2   cout << n << endl;
3 }
```

- `condition` must be `true` in order to run for the first time
- Repetition occurs until `condition` evaluates to `true` after expression statement execution
- `condition` variables (if any) are updated directly by expression statement

Jump statements

These statements unconditionally transfer the control flow of the program.

- `break`
  Exits a loop, regardless of the condition evaluation
- `continue`
  Skips the rest of the statements within the current iteration, starting the following one
- `return`
  Terminates the function currently in execution, returning control to the caller of this function
- `goto`
  Transfers control to an arbitrary location identified by a labeled statement; it should be used carefully

`break` and `continue` statements are often executed as a result of an if statement, for early loop termination.

`return` statement has two forms:

- `return;` → for functions without return type
- `return` *`expression statement`*`;` → for functions returning a specific type

Printing odds only within first 5 integers

while

```
1 int n=0;
2 while(n<5){
3   if(n % 2 == 0)
4     continue;
5   cout << n << endl;
6   n++;
7 }
```

Infinite loop!

do while

```
1 int n=0;
2 do {
3   if(n % 2 == 0)
4     continue;
5   cout << n << endl;
6   n++;
7 } while (n<5);
```

Infinite loop!

for

```
1 for(int n=0; n<5; n++) {
2   if(n % 2 == 0)
3     continue;
4   cout << n << endl;
5 }
```

Correctly prints values 1 and 3

Although the `continue` statements are all placed at the same point within the three loops, `while` and `do while` loops will result in an infinite execution: their condition is based on variable n, which is not updated if the `continue` statement is executed. Instead, the for loop updates n via its expression statement, which is still executed after the `continue` statement and before running the new iteration.

`goto` and labeled statements

Labeled statements

```
identifier: statement
```

Within the `switch` statement, `case` and `default` statements are examples of labeled statements, using reserved identifiers. However, outside of a switch statement users can define arbitrary labels for statements, subject to the rules of general identifiers.

Control flow via `goto`

goto should be avoided in general, given the many choices of control flow management in C++. The usage of `goto` is straightforward:

```
goto label;
```

`label` being the identifier of a labeled statement

Labels must be defined within the same function the `goto` statement is being used.

3. Functions

Function basics

A function is a group of statements with a given identifier, which can be called from some point of the program. The most common definition of a function can resemble the following:

```
type name ( parameter1, parameter2, ...) { statement }
```

- `type`: Type of the value returned by the function
- `name`: Identifier used to call the function
- `( parameter1, parameter2, ...)`: List of parameters (zero or more). Each parameter consists of type and identifier. Parameters are separated by commas
- `{ statement }`: Body of the function, consisting of zero or more statements

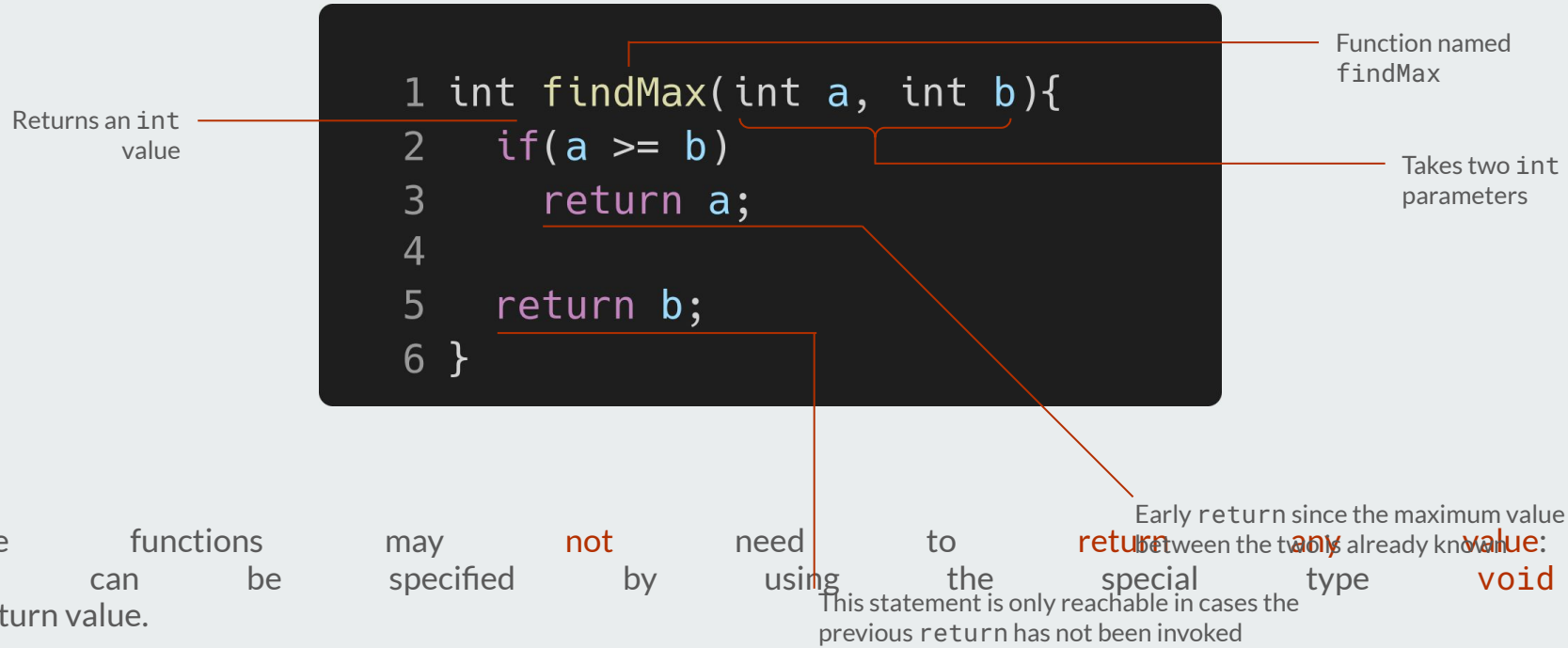
Example

```
int main() { statements }
```

- `int`: `main` returns a value of type `int`, i.e. the execution status
- `main`: Executing a program calls `main()`
- `()`: No parameters
- `{ statements }`: All the programs you implemented so far

Function basics

Return the maximum value between two integers.




```
1 int findMax(int a, int b){
2   if(a >= b)
3     return a;
4
5   return b;
6 }
```

Some functions may not need to return any value: this can be specified by using the special type `void` as return value.

Function basics

`findMax` declaration

findMax definition

```
1 #include <iostream>
2 using namespace std;
3
4 int findMax(int a, int b);
5
6 int main(){
7   cout << findMax(42, 89) << endl;
8   return 0;
9 }
10
11 int findMax(int a, int b){
12   if(a >= b)
13     return a;
14   return b;
15 }
```

Function must be declared or entirely defined before the `main` function

If a function has only been declared, a definition must be provided

Passing arguments to functions

C++ allows two methods for passing arguments to functions:

- By value
  Parameters in a function will receive copies of the the variable contents upon invocation
- By reference
  Parameters in a function will receive the variable itself upon invocation

Pass-by-value is the default method, but requiring a pass-by-reference can be specified at function declaration time for a particular parameter by inserting an ampersand (&) after the parameter's type:

```
void mulByTwo(int &n) { n *= 2; };
```

This function multiplies the value by 2 in place, i.e. directly modifying the contents of the variable that has been passed

Why use pass-by-reference?

Whenever a function is expected to modify the variables with which it has been invoked. Sometimes it can be done for efficiency purposes as well.

4. Exercises

Exercises

1. Write a program that prompts the user to insert an integer value and sums its digits, printing them at the end
   (*e.g.* for a provided integer value of 76, the result should be 13)
2. Write a program that includes a function that swaps the contents of two `float` variables, printing the two variables before and after function invocation
   (*hint:* function should return `void`)
3. Implement the `for` loop, without using `while` or `do while` loops, only using `if` and `goto` statements

References

- https://www.cplusplus.com/doc/tutorial/operators/
- https://en.cppreference.com/w/cpp/language/statements
- https://en.cppreference.com/w/cpp/language/if
- https://en.cppreference.com/w/cpp/language/switch
- https://en.cppreference.com/w/cpp/language/while
- https://en.cppreference.com/w/cpp/language/do
- https://en.cppreference.com/w/cpp/language/for
- https://en.cppreference.com/w/cpp/language/range-for
- https://en.cppreference.com/w/cpp/language/goto
- https://www.cplusplus.com/doc/tutorial/functions/