Programmazione e Calcolo Scientifico

Matteo Cicuttin

Politecnico di Torino

March 12, 2024

Declarations

In a program, we introduce entities by **declaring** them.

```
char c;     int n;     double d;
```

With a declaration we introduce an entity of a certain type:

- Type: set of possible values and possible operations on an object
- Object: some memory that holds a value of a certain type
- Value: set of bits interpreted according to a type
- Variable: a named object

Hint: Avoid declaring a variable without initializing it.

NOT `int x;`   BUT `int x = 42;`

C++ basic types

On **64 bit machines** the C++ types **usually** have the following sizes:

- `bool`: true/false, usually 8 bits
- `char`: 8 bits signed
- `short`: 16 bits signed
- `int`: 32 bits signed
- `long`: 64 bits signed
- `float`: IEEE 754 single precision
- `double`: IEEE 754 double precision
- `long double`: extended precision, possibly not IEEE 754

Integer types can have the `unsigned` qualifier to specify that you want an unsigned integer. Be sure to read `https://en.cppreference.com/w/cpp/language/types`.

- The size of a type in C++ can be obtained with the `sizeof()` operator.
- If you need portable integers, `#include <cstdint>` and use `int8_t`, `uint8_t`, `int16_t`, `uint16_t` and so on.
- It it makes sense, get used to use the `const` qualifier as in `const int answer = 42;`

Literals

```
int x = 10;               // 10 is a signed integer literal
unsigned long y = 26U;    // 26 is an unsigned integer literal
long z = 42L;             // 42 is a signed long literal
float flt = 75.0f;        // 75.0 is a single precision literal
double dbl = 88.1;        // 88.1 is a double precision literal
double dbl2 = 8.5e-2;     // 8.5e-2 is a double precision literal

double pi = 3,14;         // Happy debugging.
```

Strings will be covered in the lab class, but:

```
#include <string>
std::string mystr = "hello";
```

Everything about literals:

- https://en.cppreference.com/w/cpp/language/integer_literal
- https://en.cppreference.com/w/cpp/language/floating_literal

Scoping rules

In C++ all objects have a **lifetime**. An object declared into a block is visible only on that block and in its sub-blocks. Functions can't be nested.

```cpp
int myfun(int x) {
    int y = 5;
    if (x != y) {
        int z = x+y;
        return z;
    }
    // z is not visible here
    return x;
}
```

```cpp
using namespace std;

int myotherfun(int x) {
    for (int i = 0; i < x; i++) {
        cout << "it " << i << "\n";
    }
    // i is not visible here
    return x + 42;
}
```

Beware of shadowing

If in an inner block you declare a variable that has the same name of a variable in an outer block, the outer variable gets **shadowed**:

```cpp
int f(int x, int y) {
    int z = x+y;
    if (z > 0) {
        double x = 42.0;
        std::cout << x << "\n"; // valid: 'double x' shadows 'int x'
    }
    std::cout << x << "\n"; // this is again 'int x'
    return z;
}
```

Arithmetic and comparison

Arithmetic operators

- `x+y`: plus
- `+x`: unary plus
- `x-y`: minus
- `-x`: unary minus
- `x*y`: multiplication
- `x/y`: division
- `x%y`: remainder (integer)

Comparison operators

- `x == y`: equal
- `x != y`: not equal
- `x < y`: less than
- `x > y`: greater than
- `x <= y`: less or equal than
- `x >= y`: greater or equal than

Logical operators

- `x & y`: bitwise and
- `x | y`: bitwise or
- `x ^ y`: bitwise xor
- `~ x`: bitwise negation
- `x && y`: logical and
- `x || y`: logical or
- `!x`: logical negation

In addition: pre-increment (decrement) `++i` (`--i`), post-increment (decrement) `i++` (`i--`), combined operation and assignment (for example `+=`).

Pitfalls of arithmetic operators

```
double x_bad = 1/2;             // x_bad is zero: integer division
double x_ok = 1./2.;            // x_ok is 0.5

int coffee = 0xCA00 | 0x00FE;   // | is bitwise: coffee = 0xCAFE;
char c = 'H' | 0x20;            // | is bitwise: c = 'h';

int bad_coffee = 0xCA00 && 0x00FE; // & is logical: coffee is nonzero
```

Input/output in C++

Input/output in C++ is based on an abstraction called **stream**.

Let's say you have some object s with two operations:

- "put into" denoted with <<
- "take from" denoted with >>

Let's focus on "put into":

```
int x = 10;
s << x;      // put x into s
s << " ";    // put a space into s
s << 1.234; // put a floating point constant into s
s << "\n";   // put a newline into s
```

This results in

```
10 1.234
```

being printed **somewhere**;

Output in C++

The operator “put into” denoted with << takes a stream on the left and some object on the right.

It is somehow a function $<< : \mathbb{S} \times \mathsf{T} \to \mathbb{S}$, where $\mathbb{S}$ is the space of the streams and T is some type.

Our previous program therefore can be written in the shorter form

```
int x = 10;
s << x << " " << 1.234 << "\n"; // groups left-to-right
```

Another example. The program

```
std::string name = "Matteo";
int age = 39;
s << "My name is " << name << " and I am " << age << " years old.\n";
```

prints **somewhere** the string

```
My name is Matteo and I am 39 years old.
```

The different types of stream

But where is **“somewhere”**? C++ provides different types of output streams:

- `std::ostream`: generic output stream, declared in `<iostream>`
- `std::ofstream`: specialization for output on files, declared in `<fstream>`
- `std::ostringstream`: specialization for output on files, declared in `<sstream>`

To print on the terminal you use the two global objects of type `std::ostream`:

- `std::cout`, for printing on the *standard output*
- `std::cerr`, for printing on the *standard error*

The difference between the two is related to the way the Unix operating system is implemented.

File output streams

To do I/O on files, you use the file streams.

Remember to #include <fstream>.

```
int main(void) {
    std::string filename = "testfile.txt";
    std::ofstream ofs(filename);
    if ( ofs.is_open() ) { // Check if file successfully opened
        ofs << "Hello, world\n";
        ofs.close(); // Optional: ofstream is RAII
    }
    return 0;
}
```

RAII: “Resource Acquisition Is Initialization” is a common programming idiom in C++. When you create the ofstream, you acquire the file resource. When the ofstream gets out of scope, resource is automatically closed and released.

String streams

String streams are used to build strings.

Remember to `#include <sstream>`. In addition, you usually don't care if a stringstream is for input and output, you just use `std::stringstream`.

```
using namespace std;

int main(void) {
    string name = "Matteo";
    int age = 39;
    stringstream ss;
    ss << "My name is " << name << " and I am " << age << " years old.";
    string fullstr = ss.str();
    cout << fullstr << "\n";
    return 0;
}
```

Input in C++

Almost all what we said until now holds in the “opposite direction” with the operator `>>`.

- Input from terminal: use the global `std::cin` object (of type `std::istream`)
- Input from files: create and use objects of type `std::ifstream`
- Input from strings: `std::stringstream`

Input from terminal

```cpp
int main(void) {
    std::cout << "Enter your name: \n";
    std::string name;
    std::cin >> name;
    std::cout << "Enter your age: \n";
    int age;
    std::cin >> age;
    s << "Your name is " << name << " and you are " << age << " years old.\n";
    return 0;
}
```

File input streams

```
using namespace std;

int main(void) {
    string filename = "meteo.txt";
    ifstream ifs(filename);
    if ( ifs.is_open() ) { // Check if file successfully opened
        while( !ifs.eof() ) {
            string location;
            double temp;
            ifs >> location >> temp; // also >> : 𝕊 × T → 𝕊
            cout << "Temperature at " << location << " is " << temp << "\n";
        }
    }
    return 0;
}
```

Input/output on string streams

```
int main(void) {
    std::string mystr = "1.234";
    std::stringstream ss;
    ss << mystr;
    double x;
    ss >> x;
    std::cout << x << std::endl;
    return 0;
}
```

Input streams: dealing with errors

It can happen that (for example) you want an integer but on the file there's a string. How to detect stream extraction errors?

```
int x;
s >> x;
if( s.fail() ) {
    std::cerr << "error extracting from stream\n";
}
```

Or, another version:

```
int x;
if( ! (s >> x) ) {
    std::cerr << "error extracting from stream\n";
}
```