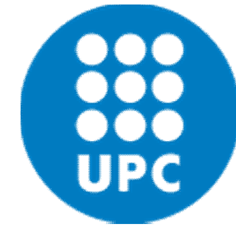


7. The C++ language, 1

Programming and Algorithms II

Degree in Bioinformatics

Fall 2018

Hello world

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

int main() {
   cout << "Hello, world!" << endl;
}
```

Line breaks are irrelevant. Can also be written as:

```
#include <iostream>
using namespace std;int main(){cout<<"Hello, world!"<<endl;}
```

Hello world

- #include is similar to import <file> means it's a system file. If you write #include "iostream"

it means it's a program you wrote, not a system module

- using namespace std. We'll explain later.
- Every c++ program must have one and exactly one function called main(). That's where the execution will start. "int" means it returns an integer. It'll be 0 if no errors occurred, something else (related to the type of error) otherwise.
- "cout << stuff" is like "print(stuff)" to standard output (terminal). It is defined in iostream, that's why we do the include
- several "<< stuff" admitted; << endl jumps to new line
- There is a "cin >> variable" to read a value from standard input (keyboard) into variable
- Simple statements are terminated with ;
- Composite statements go between { … }
- No indentation is mandatory, as {…} indicates what goes in a block. But highly recommended for readability

Prime numbers

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

// assuming n >= 2, returns true iff n is prime
bool is_prime(int n) {
   int m = 2;
   while (m*m <= n) {
      if (n % m == 0) return false;
      ++m;    // or also m = m + 1
   }
   return true;
}

int main() {
   while (true) {
     int n;
     string s;
     cout << "number? "; cin >> n;
     if (is_prime(n)) s = "prime!";
     else s = "nonprime!";
     cout << n << " is " << s << endl;
  }
}
```

Variables

Big differences with Python! C is a *strongly typed language*

- Every variable is *declared before it can be used.*
- Initializing it at the momento of declaration is optional
- It is declared with a type and it can only hold variables of that type
- It only exists within the {…} where it was declared;
  - good practice: declare as inside as possible

- Multiple declarations in a row posible (not necessarily most readable)

```
int i, j, k = 0;
```

- Initializes k (only) to 0

Comments

```
// one line comments

/* multiline comments
   can be written like this */

/*
 * sometimes
 * also written
 * like this for readability
 */
```

Functions

- Functions need to be declared with return type
- Function parameters also need to be declared with type

```
string substring(string s, int i, int j);
```

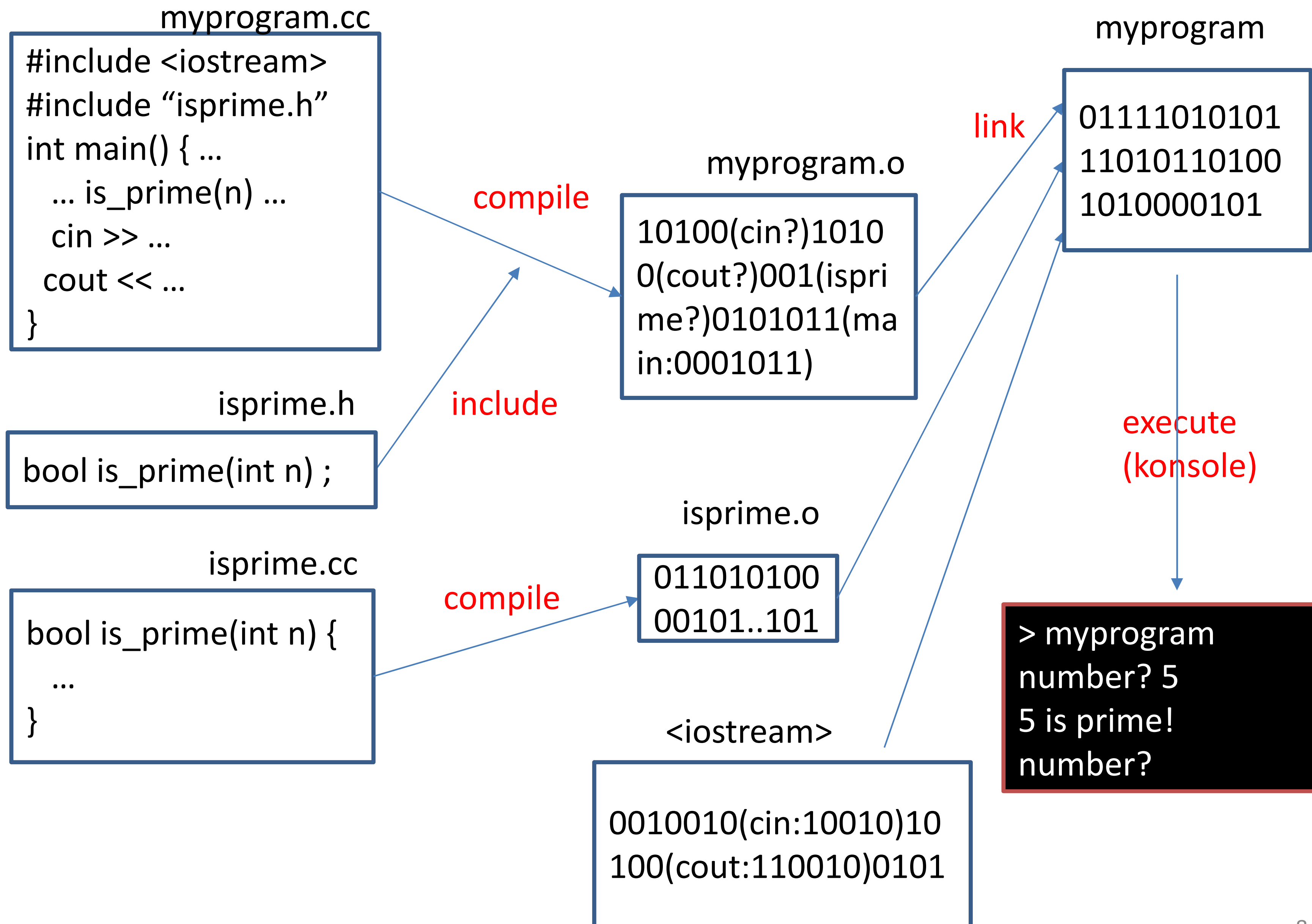
See: no {…} above. This is declaring the function.

It can be defined somewhere else

Functions with type "void" do not return anything (but no Python "None")

```
void print_results(parameters to be printed) { … cout … }
// fill a, b, c with results, then:
print_results(a,b,c);   // ok if a,b,c, match types
x = print_results(a,b,c);   // error
```

Compiling vs interpreting



Compiling vs interpreting

There is no “console” for C++. No interpreter. It is a compiled language:

- A **compiler** program translates C++ files to **object** files
- Several **object** files are linked to form an **executable** file
- Every name declared must appear exactly once in linked object files
- Including name “main()”
- The executable file contains instructions in the machine language. Directly executed by the CPU

Compiling vs interpreting

- An executable file may be specific for one machine / operating system
- If taken to another system / machine, it may need recompiling

C++ is faster: the type of each variable is known at compile time

Work that Python needs to do in runtime can be done in compile time

Three kinds of files involved:

- .cc or .cpp files: contain C++ declarations and code
- .h header files: also C++, but usually not intended to generate object files. More, declarations to be #included in .cc or .cpp files
- library files: object files with standard libraries

Types

int. Finite range (e.g., 32 bits… -2^{31} to $2^{31}-1$)

unsigned int (or unsigned) (e.g., 32 bits… 0 to $2^{32}-1$)

float and double

all of the above subject to OVERFLOW

char 'a' ; 'abc' is wrong

bool: true or false

int, float, double, char, bool are NOT objects. Similar to unmutable types in Python

string "a", "abc" ; comparing "a"=='a' gives an error; s[i] ok

#include <string>

Expressions

Much like Python

== , !=

numeric types: + - * / % < > <= >=

no //: if both arguments are int or unsigned, it is like //

x = f(…) : return type of f must be same (or subtype) of x's type

incorrect if f is void (returns nothing)

f(arg1, arg2, arg3)

calling function f with arguments arg1, arg2, arg3

their types must be the same (or subtype) of those in the definition of f

string substring(string s, int i, int j) { … }

string s1 = substring(s2,0,len(s)/2); // ok

int m = substring(s2,0,len(s)/2); // not ok

string s3 = substring(s2,true,"abc") // not ok

Statements

Statements can be simple or composite

- variable = expression;
- f(…); // where f returns void… or not, and return is ignored
- return expression

- if (bool expression) Statement else Statement
- while (bool expression) Statement
- repeat Statement while (bool expression)
- for (int i = initialization of i; condition(i); change i) Statement

other instructions but less common

No list comprehensions; you need to write loops!!

x = sum(lst), x = [i for i in range(0:10) if (condition(i))]

Parameter passing

By value:

int free_rooms(hospital h) { … }

when we all free_rooms(my_hosp), a copy of my_hosp is assigned to h.

If we change h inside free_roms, my_hosp is not modified

By reference: (the usual one in Python)

int free_roms(hospital& h) { … } (or hospital &h)

a reference to my_hosp is assigned to h

if we modify h in free_rooms, my_hosp is modified

good for efficiency; potentially unwanted side-effects

By constant reference:

int free_roms(const hospital& h) { … } (or hospital &h)

passes a constant reference to my_room; think of h as a constant

compiler will not allow any change to h (e.g. assignement)

efficient alternative to passing by value for complex types

Arrays and vectors

Implementation similar to Python lists: Consecutive memory positions

Native arrays:

int a[10];   // 10 integers, called a[0], … , a[9]

Cannot be moved, copies (automatically), …

If you access a[20], no error is (necessarily) generated. Big problem.

a does not even remember its length. No len(a)

Name “a” is synonymous with
  “the memory position where compiler placed the first of these 10 integers”
(More next course, when you see C++ pointers)

Arrays and vectors

```
#include <iostream>
using namespace std;
// reads a series of digits and says which is the most common one
int main() {
   int a[10];
   for (int i = 0; i <= 9; ++i) a[i] = 0;

   int n;
   while (cin >> n) {
       if (n < 0 or n > 9)
          cout << "digits only, please!" << endl;
       else
          ++a[n];
   }

   int m = 0;
   for (int i = 1; i <= 9; ++i)
       if (a[m] < a[i]) m = i;

   cout << "the most common digit in input is " << m << endl;
}
```

Arrays and vectors

Class vector: in the standard C library

```
#include <vector>
vector<int> v(10);
vector<char> r(10,’a’);
v[5] = 100;
r[3] = ‘b’;
```

= in vectors equals Deep Copy:

```
vector<string> s1(100,”abc”)
s2 = s1;      // s1, s2 do not interfere
```

Arrays and vectors

Matrices are vectors of vectors

```
// define a constant; note the syntax
#define n 20
// matrix nxn with 0's
vector<vector<float>> mat(n,vector<float>(n,0));
// make it the identity
for (int i = 0; i < n; ++i) mat[i][i] = 1;

// compute C = A x B
// note: C needs to be already created!
for (int i = 0; i < n; ++i)
     for (int j = 0; j < n; ++j) {
            float s = 0;
            for (k = 0; k < n; ++k) s += A[i][k]*B[k][j];
            C[i][j]  = s;
     }
```

Arrays and vectors

Cost of operations:

- Creating: O(size) for vectors; O(1) for arrays
- Accessing one component (a[i]) : O(1)
- Searching sequentially for some value: O(size)
- Binary search (needs to be sorted): O(log(size))

The below for vectors:

- Copying (=): O(size)
- Appending to end, removing from end: O(1) amortized

  (v.push_back(x), v.pop_back())
- Inserting in the middle, deleting in the middle: O(size)

(not predefined in arrays: you program your loops)

Tuples and type definitions

Tuples are called structs. Not objects. Perhaps you want to use classes instead

Often used with “typedef” construct, which defined a new type:

```
typedef struct {
       string name;
       int age;
       bool alive;
}  Person;

Person p1, p2, p3;

p1.name = “Ricard”; p1.age = 25; p1.alive = true;

typedef vector<vector<float>> Matrix;
Matrix m;
```

Summary

- Compiled, not interpreted language
- Everything must be declared before use
- Definition includes the type.
- So types are known at compile time, not at runtime

- Less flexibility; longer programs
- More speed, more control of what is really going on

- Basic types, structs and native arrays are not objects
- Vectors, maps (dictionaries), sets, and lists are objects
- Python lists ≈ c++ vectors
- c++ lists are another beast (next day)