C++ Programming

M1 Math
Michail Lampis
michail.lampis@dauphine.fr

Administration

- Class website

  http://www.lamsade.dauphine.fr/~mlampis/Cpp/

- Cours: Mondays 8:30-10:00
- TDs: Mondays 10:15-11:45 or Wednesdays 17:15-18:45
- Check schedule and announcements on web page
- Final exam: in May/June
- No mid-term exam! (partiel)
- TDs will include programming exercises
- Grade = 70% Final Exam + 30% TDs

C++ overview

- Who cares?
  - One of the most popular programming languages
  - Widely available, multi-platform, standardized
  - Powerful, lots of features, great for “low-level” system development
- Why not C++?
  - Less “programmer-friendly” → bugs
  - Too complicated??

C++ vs C

- History: C++ == C with added features
  - Objects (!!!)
  - References
  - Templates
  - Exceptions
  - …
- Lots of C++ code is basically C code with some “extra sauce”
- Important: memory management, pointers, etc.

C++ vs Java

- Java was designed **after** C++
  - Java == (C++) - -
  - Meaning: Java tries to keep the “good parts” of C++
  - Basic syntax is same
  - “Confusing” parts are simplified
  - **Important:** Garbage collection in Java vs C++

My first C++ program

```
#include <iostream>
using namespace std;
int main()
{
        cout << "Hello world!" << endl;
        return 0;
}
```

How to run this

In a Unix/Linux terminal

```
$ g++ hello.cpp -o hello
$ ./hello
```

- Generally:
  - Programs are compiled from source files (.cpp) to executables. g++ is the compiler.
  - A program may be broken into multiple .cpp files, which are compiled separately and **linked** (more later)
  - Part of a program may be in a **header** file (.h) (more later)

My first C++ program explained

```
#include <iostream>          //Preprocessor directive for cout
using namespace std;         //std::cout, std::endl otherwise
int main()                   //Always the main function in C++
{
        cout << "Hello world!" << endl;
        return 0;            //Tell the OS that all is OK
}
```

Program Structure (in C)

- A program is a collection of **functions**
- A function is a sequence of **statements**, terminated by semi-colons (;)

```
int main()
{
    stmt1; stmt2; stmt3; ...
}
```

Program Structure (cont'd)

- Whitespace is ignored
- Statements can be grouped into **blocks** denoted with curly braces { }

```
int main()
{
    {stmt1;} {stmt2;} {stmt3;} ...
}
```

Variables

- Data in C/C++ is stored into **variables**
- Variables must be declared, along with type

```
int x = 5; //integer
double y; //floating point number, e.g. y=0.8
char c; //character, e.g. c='a'
bool b; //true or false
```

Variable Scope/Memory

- Variables must be declared **before** use
- Their **scope** is the block in which they are declared (usually a function)
- They can be **global** (bad idea)
- Declaring a variable → The computer allocated some space in memory to store the appropriate value (depends on type)

Dealing with simple variable types

- The = operator means “assign” the right-hand-side to the left-hand-side (which should be usually a variable)

```
int x,y;

x = 5; //The value 5 is stored in x

y = x; //The value 5 is also stored in y

x = 3; //Now 3 is stored in x but 5 is still in y
```

Other operators

- +,-,/,*,% have standard meaning (but int/int returns int, rounded)
- ==, != → checks if x==y, returns true or false
  - (differs from x=y)
- >, <, >=, <= as usual
- ++, - - increase/decrease a variable by 1
- ||, &&, ! → logical OR, AND, NOT
  - All non-zero variables are considered TRUE
- <<, >> → bit-wise shift operators
- A?B:C

Control Flow

- C++ (and C and Java) offers statements to control code flow
  - if(<condition>) stmt1; else stmt2;

    <condition> is evaluated (as boolean) and if true, stmt1 is executed, otherwise stmt2;
  - x = 5;

    if(x) cout << “Yes”; else cout << “No”;
  - if(x=5) vs if(x==5)!!!

Dangling else

- What is the result of the following piece of code?

```
int x = 4;

if (x > 5)

if (x < 8) cout << “Case 1”;

else cout << “Case 2”;
```

Dangling else

- What is the result of the following piece of code?

```
int x = 4;
if (x > 5)
if (x < 8) cout << “Case 1”;
else cout << “Case 2”;
```

- else is always “attached” to closest if
- Use {} to make code clear!!

For loops

```
for( i=0; i<5; i++){
    cout << “Iteration “ << i << endl;
}
```

- Three expressions given
  - Initialize, check, repeat
  - Each can be empty
    - for(; ;) ; //infinite loop!!
- Use , expression to make complicated inits
  - for(i=0, j=5; i+j<7; i+=2, j--)

While loops

while(<condition>) stmt;

- Equivalent to

  for(; <condition> ;) stmt;

- Vice-versa

  for(expr1 ; expr2 ; expr3) stmt;

  //same as

  expr1; while(expr2) { stmt; expr3; }

Do while loops

- This is actually slightly different

```
do{
    stmt;
}while(expr);
//same as
stmt; while(expr) stmt;
```

Other flow commands

- Break
- Continue
- Switch – Case
- Goto (please don't!)

Functions

- A function is an independent piece of code meant to achieve a certain task
- Functions are given **parameters** as input
- They **return** a value of a certain type (unless the type is **void**)

Function example

```
int max( int x, int y) //define return, param types
{
    if( x>y) return x;
    return y;
    cout << “This will never happen!”;
}
// later, in another function.
cout << max(3,5); // prints 5
```

Function declarations

- It is possible to just declare a function that will be defined later.

```
void f1(int);

void f2(){

//bla bla

f1(5); //OK because f1 declared

}

//later...

void f1(int x){ stmt; stmt; ... }
```

Function exercise

- Write a function isSqRt(x,y) which is given two integers and returns true if x is the rounded-down value of the square root of y.
- What is the function **prototype**?
- isSqrt(5,25) = ?
- isSqrt(5,26) = ?
- isSqrt(5,24) = ?

Solution

```
bool isSqRt(int x, int y)
{
        if (x*x <= y && (x+1)*(x+1)>y)
                return true;
        return false;
} //Can you make this code shorter?
```

Call-by-value

- A parameter is passed to a function **by value**
  - A copy of the value is made and given to the function
  - The function does not “touch” the original value

```
void f(int x) { x++; }

int main() { int y=5; f(y); cout << y; }

//Output?
```

Call-by-Value

- Call-by-Value allows us to pass complicated values to a function (not lvalues)

```
void f(int x) { x++; }

int main() { int y=5; f(3*y+2); cout << y; }

//Output?
```

Call-by-Reference

- How can a function change the parameter?

```
void f(int &x) { x++; } //Notice the &
int main() { int y=5; f(y); cout << y; }
//Output?
```

- Now, code of previous slide does not compile!

Why call-by-ref?

- Allows function to change parameter values
- More efficient!
  - No copies are made
  - A “real” variable must be given...
- If I only care about efficiency...
  - `void f(const int &x) { ... }`
- Code that attempts to change x will not compile

References in general

```
int x = 5;
int &y = x;
x++;
cout << y << endl;
//Output? Without the &?
//Compare with Java for objects...
```

Another exercise

- Implement gcd(a,b) function which returns greatest common divisor of (ints) a,b
- Check that a,b>0
- Use the == operator to check if a == b
- Use the fact that for a>b,

  gcd(a,b) == gcd(a-b,b)

  - Recursive function!

Solution

```
int gcd(int a, int b)
{
        if(a<=0 || b<=0)
                return -1; //Error!
        if(a<b)
                return gcd(b,a); //Now a>b
        if(a==b)
                return a;
        return gcd(a-b,b);

}
```

Efficiency?

- How much space does this function use?
  - Think of **stack** of function calls...
- Can we implement this with a loop?

Solution 2

```
int gcd(int a, int b)
{
	if(a<=0 || b<=0)
		return -1; //Error!
	while(a!=b){
		if(a>b) a-=b;
		else b-=a;
	}
	return a;
}
```

Fibonacci

- The Fibonacci sequence is F(n+2)=F(n+1)+F(n)
- 1,1,2,3,5,8,13,21,34,...
- Write a recursive function that computes the F(n)
- Write a loop-y function that computes F(n)
- Which is better?

Solution 1

```
int Fib(int n)
{
        if(n<=0) return -1; //Error
        if(n<=2) return 1; //Base case
        return Fib(n-1)+Fib(n-2);
}
```

Solution 2

```
int Fib(int n)
{
	if(n<=0) return -1; //Error
	if(n<=2) return 1; //Base case
	int nm1=1, nm2=1;
	int nm;
	while(n-->2){
		nm = nm1 + nm2;
		nm2 = nm1;
		nm1 = nm;
	}
	return nm;
}
```

The preprocessor

- Recall the command #include <iostream>
- Generally, commands that begin with # are **preprocessor directives**
- The preprocessor is called on your program **before** the compiler, and changes the program in basic ways

Preprocessor commands

- #include <header.h>
- Most common preprocessor command
- Equivalent to “copy-paste”-ing all of the file header.h in the line where the command appears
- Most common use:
  - Declare classes, functions, global vars in a header file
  - Separate implementation (cpp) from declaration
  - Programming by contract

Preprocessor Commands

- #define MYPI 3.14
- Allows to define constants
- Meaning: replace MYPI with 3.14 everywhere **before** compilation
- Can also define “functions”

```
#define ABS(x) (x>0?x:-x)
```

- Careful! Not the same as function! (if called with parameter with side-effects)

Preprocessor Commands

- Main use for #define → make sure files are included only once

```
#ifndef MYCONST
#define MYCONST
...
#endif
```

- Above ensures that … will only happen once, no matter how many times file is included...

More exercises

- Function `int findSqRt(int x); //return rounded-down sqrt(x)`
- Implement with a loop
- Implement with binary search and function overloading(?)

Solution 1

```
int findSqRt(int x)
{
        if(x<0) return -1;
        for(int i=0; i<x; i++){
                if(i*i<=x && (i+1)*(i+1)>x)
                        return i;
        }
}
```

Solution 2

```
int findSqRt(int x, int l, int h);
int findSqRt(int x)
{
        if(x<0) return -1;
        return findSqRt(x,0,x);
}
```

Solution 2 cont'd

```
int findSqRt(int x, int l, int h)
{
        int t = (l+h)/2;
        if( t*t <= x && (t+1)*(t+1) > x) return t;
        if( t*t < x)
                return findSqRt(x,t+1,h);
        else
                return findSqRt(x,l,t);
}
```