EDAF30 – Programming in C++

1. Introduction

Sven Gestegård Robertz
Computer Science, LTH

2021

Outline

EDAF30: Programmering i C++, 7.5 hp
Syfte och mål

Kursens syfte är att ge kunskaper i objektorienterad programmering i C++.

Kunskap och förståelse:

- känna till och kunna beskriva skillnaderna mellan C++ och Java
- vara förtrogen med språket C++ och standardbiblioteket STL
- kunna förklara grundläggande begrepp inom objektorienterad C++-programmering
- förstå och kunna förklara de olika typerna av funktionsanrop
- kunna tolka, analysera och förklara befintlig C++-kod.

Färdighet och förmåga:

- kunna utveckla ett fungerande C++-program från en given specifikation
- kunna felsöka metodiskt i C++-kod.

EDAF30: programming in C++ , 7.5 hp
Important differences to Java

New or extended concepts in C++
(compared to Java / introductory courses):

- Pointers and memory management
- Functions: call-by-value and call-by-reference
- Polymorphism: both static and dynamic
  (compare *templates* to *generics*)
- Operator overloading

And also

- The tool chain

EDAF30: programming in C++ , 7.5 hp
Examination details

The compulsory course items are

- laborations
- project
- written examination

The final grade is based on the result of the written examination.

EDAF30: programming in C++ , 7.5 hp
Administration

- Course plan
  - Canvas
  - Zoom
- Registration
- Sign up for labs
  - Work in pairs
  - On the web - link from the course web page
  - Sign up for a group – same time all weeks

History
C++ is a descendent of Simula and C.

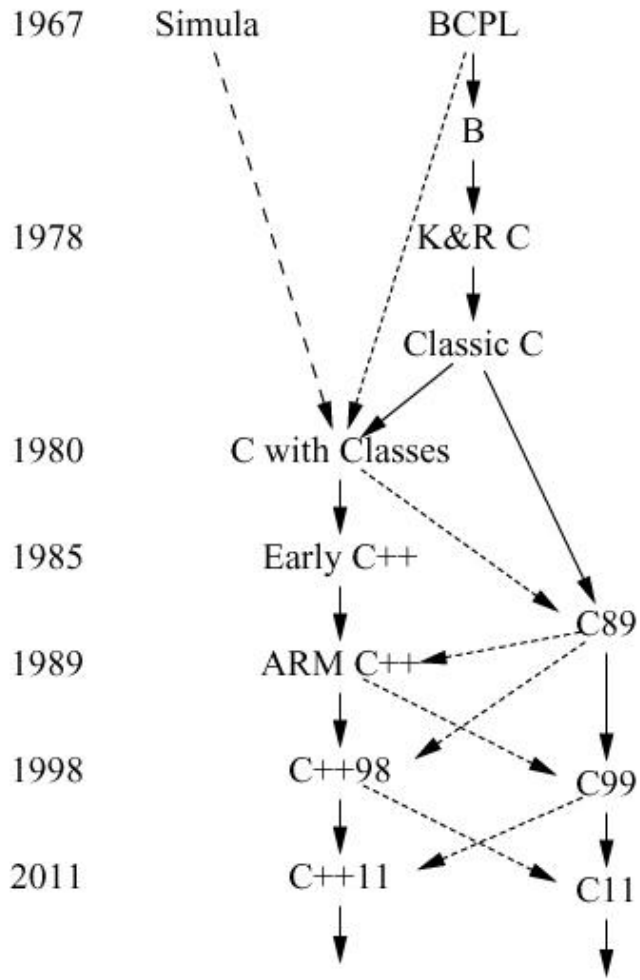
1967: Simula (Dahl & Nygaard)
1972: C (Dennis Ritchie)
1978: K&R C (Kernighan & Ritchie)
1980: C with Classes (Bjarne Stroustrup)
1985: C++ (Bjarne Stroustrup)

- ISO standard 1998

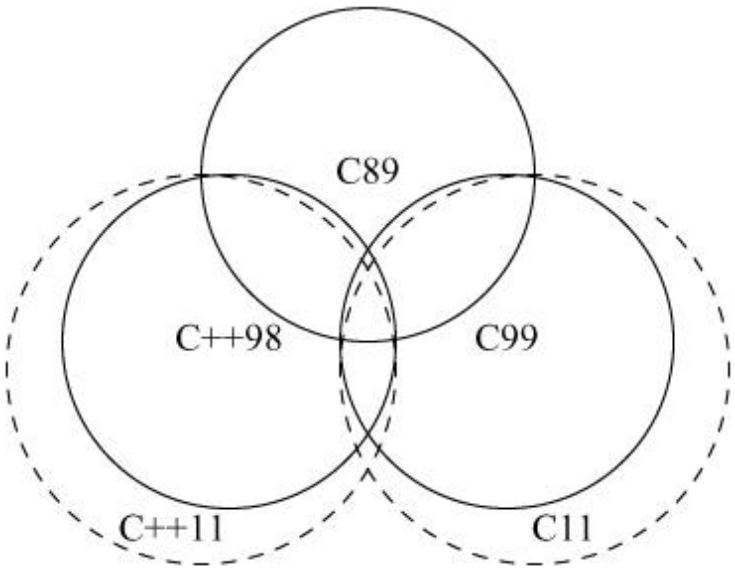
Other relatives:

1995: Java (James Gosling et al.)
2000: C# (Anders Hejlsberg)

- virtual machine
- automatic memory management
- *safe* languages



C++ is not a pure extension of C



- both ISO C and ISO C++ are descendants of K&R C, and are "siblings"
- some details are incompatible between ISO C och C++
- Areas are not to scale

In general: Don't write C++ as if it were C

What is C++?

The ISO standard for C++ defines two things

- *Core language features*, e.g.,
  - data types (e.g., **`char`**, **`int`**)
  - control flow mechanisms (e.g., **`if`** and **`while`** statements).
  - rules for declarations
  - templates
  - exceptions
- *Standard-library components*, e.g.,
  - Data structures (e.g., `string`, `vector`, and `map`)
  - Operations for in- and output (e.g., << and `getline()`)
  - Algorithms (e.g., `find()` and `sort()`)

The standard library is written in C++

- Example of what is possible

A minimal program in C++

empty.cc

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

- has no parameters
- does nothing
- the return value of `main()` is interpreted by the system as an error code
  - non-zero means error
  - no explicit return value is interpreted as zero (NB! only in `main()`)
  - rarely used in Windows
  - often used on Linux/Mac

The first C++ program
Hello, World!

hello.cc

```
#include <iostream>
int main( )
{
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

hello.cc

```
#include <iostream>
using std::cout;
using std::endl;

int main( )
{
   cout << "Hello, World!" << endl;
   return 0;
}
```

Statements

Mostly the same syntax as in Java:

- **`if`**, **`switch`**
- **`for`**, **`while`**, **`do while`**
- **`break`**, **`continue`**

but **`goto`** *is spelled differently:*

- No **`break`** to a label
- **`goto`** (used in C, rarely used in C++)

Operators

Operators and expressions quite similar to Java

The same as in Java

E.g., `+ - * / % ++ -- += -= *= && || & |` etc., and `[] . ?:`

The trinary operator `?:` (like in Java)

```
z = (x>y) ? x : y;
```

```
if (x>y)
  z=x;
else
  z=y;
```

Many more, including

Pointer operators: `* & ->`

Input and output: `<< >>` (*overloaded shift operators*)

`sizeof`, **`decltype`** (*compile-time*)

Functions
Declaration and definition

The main way of getting sonething done in C++:

- call a *function*
  - Declare before use
    A function must have been *declared* before it can be called
  - A function declaration specifices
    - name
    - return type
    - types of the parameters
  - Example: function declarations

```
int random();
void exit(int);
double square(double);
int pow(int x, int exponent);
```

    - The compiler ignores parameter names
    - Give names if it increases readability

- A function *definition* contains the implementation
  - Must not occur more than once (*One Definition Rule*)

Difference from Java
Function and variable declarations

- In Java functions and variables
  can only be declared inside a class.
- In C++, functions and variables
  can exist independently of classes.
  - free functions do not belong to a class
  - member functions in a class
  - global variables
  - member variables

Function declaration
Example

- Declaration and definition

Example: Mean value – variant 1

```
double mean(double x1, double x2) // Declaration and definition
{
   return (x1+x2)/2;
}

int main()
{
   double a=2.3;
   double b=3.9;
   cout << mean(a, b) << endl;
}
```

Function definition
With previous declaration

- *Forward declaration*
- Fuction definition after `main()`

Example: mean – variant 2

```
double mean(double, double);          // declaration (prototype)

int main()
{
   double a=2.3;
   double b=3.9;
   cout << mean(a, b) << endl;        // use
}

double mean(double x1, double x2)     //definition
{
   return (x1+x2)/2;
}
```

Function declaration and definition
in separate files

Header file with declaration (mean.h)

```
double mean(double, double);          // declaration (prototype)
```

Main source file

```
#include "mean.h"
int main()
{
   double a=2.3;
   double b=3.9;
   cout << mean(a, b) << endl;        // use
}
```

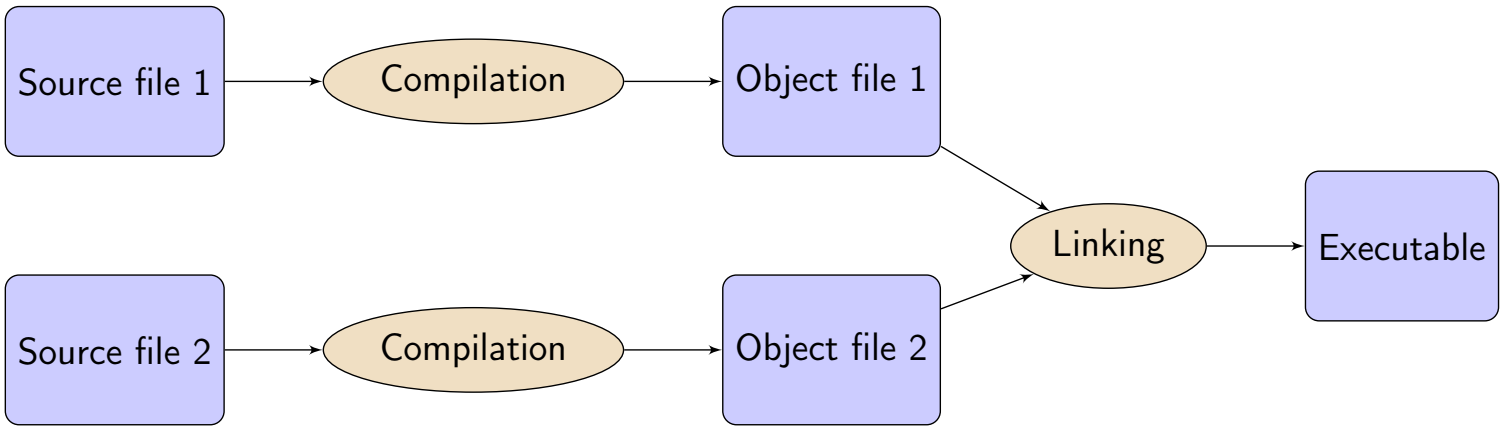
Library source file (mean.cc)

```
double mean(double x1, double x2)    //definition
{
   return (x1+x2)/2;
}
```

What is a program?

C++ is a compiled language

- Source code
- Object file(s)
- Executable file



Data types and variables

- Every name and every expression has a type
- some concepts:
  - a *declaration* introduces a *name* (and gives it a *type*)
  - a *type* defines the set of possible values and operations (for an *object*)
  - an *object* is a place in memory that holds a *value*
  - a *value* is a sequence of bits interpreted according to a *type*.
  - a *variable* is a named *object*

An object has

- a *value* and
- a *representation*

Unnamed objects

Unnamed objects include

- temporary values
- objects on the heap (allocated with **new**)

Data types
Primitive types

- Integral types: **char**, **short**, **int**, **long**, **long long**
  - **signed** (as in Java)
  - **unsigned** (*modulo* 2^N "non-negative" numbers, not in Java)
- Floting point types: **float**, **double**, **long double**
- **bool** (boolean in Java)
  - integer values are implicitly converted to **bool**
  - zero is **false**, non-zero is **true**
- The type **char** is "the natural size to hold a character" on a given machine (often 8 bits). Its size (in C/C++) is called "a byte" regardless of the number of bits.
- **sizeof**(**char**) $\equiv 1$ (1 byte)
- The sizes of all other data types are multiples of **sizeof**(**char**).
  - sizes are *implementation defined*
  - **sizeof**(**int**) is commonly 4.

Variables
Declaration and initialization

Declaration without initialization (avoid)

```
int x;          // x has an undefined value (if local)
                // (as local variables in Java)
```

Declaration and initialization

```
int x{7};       // C++ style (recommended if unsure)
int y = {7};    // C++ with extra =
int z = 7;      // C style

vector<int> v{1,2,3,4,5};
```

C style: Beware of implicit type conversion

```
int x = 7.8;    // x == 7. No warning
int y {7.8};    // Gives a warning (or error with -pedantic-errors)
```

Suggested reading

References to sections in Lippman

Functions	6.1 (p 201–207)
Types, variables	2.1,2.2,2.5.2 (p 31–37, 41–47, 69)
Type aliases	2.5.1
Arithmetic	4.1-4.5, 4.11
Constants	2.4 2.4.4 (p 59–60, 65–66)
Pointers and references	2.3 (p 50–59)

Next lecture
Function calls. Pointers. User-defined types

References to sections in Lippman

Literals	2.1.3
Pointers and references	2.3
std::string	3.2
std::vector	3.3
Arrays and pointers	3.5
Classes	2.6, 7.1.4, 7.1.5, 13.1.3
Scope and lifetimes	2.2.4, 6.1.1
I/O	1.2, 8.1–8.2, 17.5.2