EDAF50 – C++ Programming

1. Introduction

Sven Gestegård Robertz
Computer Science, LTH

2019

Outline

EDAF50: C++ programming, 7.5 hp

The course gives detailed knowledge about C++. Special emphasis is placed on the language constructs that make C++ a more advanced, and also more complex, language than Java.

Knowledge and understanding

- know about and be able to describe the differences between C++ and Java
- have detailed knowledge about C++ and the standard library STL

Competences and skills

- be able to choose the correct language construct to solve a given problem
- be able to use tools to develop C++ programs in a Unix environment

EDAF50: C++ programming , 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

EDAF50: C++ programming , 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.

EDAF50: C++ programming , 7.5 hp
Administration

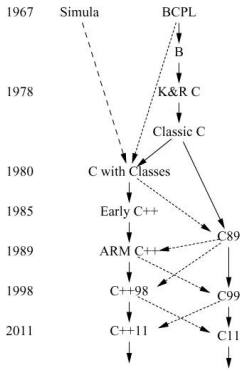
- Course plan
- Registration
- Sign up for labs
  - On the web - link from the course web page
  - Sign up for a group – same time all weeks

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

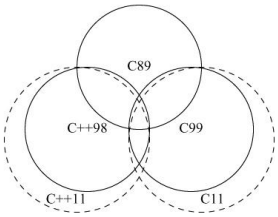
1967: Simula (Dahl & Nygaard)
1972: C (Dennis Ritchie)
1978: K&R C (Kernighan & Ritchie)
1980: C with Classes (Bjarne Stroustrup)
1985: C++ (Bjarne Stroustrup)
- ISO standard 1998

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

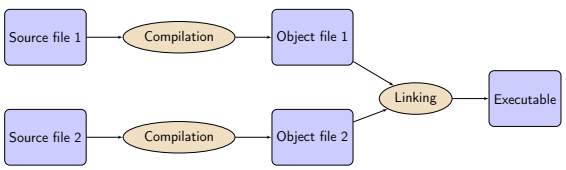
```
#include <iostream>
using std::cout;
using std::endl;

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


What is a program?

C++ is a compiled language

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



A C++ program

Example: compute and print x^2.

```
#include <iostream>

double square(double x)
{
  return x*x;
}

void print_square(double d)
{
    std::cout << "the square of " << d <<
                 " is " << square(d) << std::endl;
}

int main( )
{
    print_square(1.234);
    return 0;
}
```


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 only occur once

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, b=3.9;
   cout << mean(a, b) << endl;
}
```


Function definition
With forward declaration

- Fuction declaration before use in `main()`
- Fuction definition elsewhere

Example: mean – variant 2

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

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

mean.cc

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


Function definition
With forward declaration

- Fuction declaration before use in `main()`
- Fuction definition elsewhere

Example: mean – variant 2

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

```
#include "mean.h"

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

mean.cc

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

Functions
Function calls

The semantics of function argument passing is the same as copy initialization: *(Same as for primitive types in Java)*

- In a function call, the *values of the arguments* are
  - type checked, and
  - with implicit type conversion (if needed)
  - copied to the function parameters
- Example: with a function `double square(double d)`

```
double s2 = square(2);        // 2 is converted to double
                              // double d = 2;

double s3 = square("three"); // error
                              // double d = "three";
```


Functions
Function overloading

- Overloading ("överlagring")

```
void print(int);
void print(double);
void print(std::string);

void user()
{
    print(42);      // calls print(int);
    print(1.23);    // calls print(double);
    print(4.5f);    // calls print(double);
    print("Hello") // calls print(std::string);
}
```

  - Cannot differ only in return type
  - Must not be ambiguous
- Default arguments (sometimes) similar to overloading
  - `void print(int x, std::ostream& out = std::cout);`
  - The rules are complex. *Only use for trivial cases*
  - Risk of ambiguity if combined with overloading

Functions
Call - ambiguity

- With overloaded functions, the compiler selects "the best" function (after implicit type conversion)
- If two alternatives are "equally good matches " it is an error

```
void print2(int, double);
void print2(double, int);

void user()
{
    print2(0, 0);   // Error! ambiguous
}
```

- and also (with `print()` from last slide)

```
long l = 17;
print(l);        // Error! print(int) or print(double)?
```


Functions
Rule of thumb

Factor your code into small functions to

- give names to activities and document their dependencies
- avoid writing specific code in the middle of other code
- facilitate testing
- A function should perform a single task
- Keep functions as short as possible
- Rule of thumb
  - Max 24 lines
  - Max 80 columns
  - Max 3 block levels
  - Max 5–10 local variables
  - Inversely proportional to complexity

Call by value and call by reference
Call by value(*värdeanrop*)

In a 'normal' function call, the values of the arguments are copied to the formal parameters (which are local variables)

Example: swap two integer values

```
void swap(int a, int b)
{
   int tmp=a;
   a = b;
   b = tmp;
}
```

...and use:

```
int x = 2;
int y = 10;

swap(x, y);

cout << x ", " << y << endl;
```

2,10 x and y are not changed

Call by value and call by reference
Call by reference(*referensanrop*)

Use *call by reference* instead of *call by value*:

Example: swap two integer values

```
void swap(int& a, int& b)
{
   int tmp=a;
   a = b;
   b = tmp;
}
```

...and use:

```
int x = 2; int y = 10;

swap(x, y);
```

NB! The argument for a reference parameter must be an *lvalue*

The call `swap(x,15);` gives the error message

```
invalid initialization of non-const reference of type "int&"
from an rvalue of type 'int'
```

References

- A reference is *an alias* for a variable

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++)

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 set 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.

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*)

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)
```

Variables
Automatic type inference

`auto`: The compiler deduces the type from the initialization.

Declaration and initialization

```
auto x = 7;                 // int x
auto c = 'c';               // char c
auto b = true;              // bool b
auto d = 7.8;               // double d

std::vector<int> v;
auto it = v.begin();        // std::vector<int>::iterator it

double calc_epsilon();
auto ep = static_cast<float>(calc_epsilon());  // float ep
```

In `float ep = calc_epsilon();` *the narrowing is not obvious* NB!

with `auto` there is no risk of narrowing type conversion, so using `=` is safe.

Variables
Automatic type inference

Don't use `auto` if you need to be explicit about the declared type, e.g.

- if naming the type makes the code more readable.
- to specify the value range or precision (e.g., `int`/ `long` or `float`/ `double`)

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)