EDAF50 – C++ Programming

1. Introduction

Sven Gestegård Robertz
Computer Science, LTH

2021
1 About the course

2 Presentation of C++
   • History
   • Introduction
   • Functions
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
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
The compulsory course items are

- laborations
- project
- written examination

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

```cpp
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!

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::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 $x^2$.

```cpp
#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 something done in C++:
▶ call a *function*

▶ Declare before use
  A function must have been *declared* before it can be called
▶ A function declaration specifies
  ▶ name
  ▶ return type
  ▶ types of the parameters

▶ Example: function declarations

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

▶ A function *definition* contains the implementation
▶ Must only occur once

▶ The compiler ignores parameter names
▶ Give names if it increases readability
• 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

- Function declaration before use in main()
- Function definition elsewhere

Example: mean – variant 2

```cpp
# mean.h

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

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

# 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

```cpp
#include "mean.h"

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

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

```cpp
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")

```cpp
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
With overloaded functions, the compiler selects “the best” function (after implicit type conversion).
If two alternatives are “equally good matches” it is an error.

```c++
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)?
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

```cpp
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**

```cpp
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'*
A reference is *an alias* for a variable
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)*
References to sections in Lippman

**Functions**  6.1 (p 201–207)

**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)
References to sections in Lippman

Types, variables 2.1, 2.2, 2.5.2 (p 31–37, 41–47, 69)
Type aliases 2.5.1
Type deduction (auto) 2.5.2
Pointers and references 2.3
Scope and lifetimes 2.2.4, 6.1.1
const, constexpr 2.4
Arrays and pointers 3.5
Classes 2.6, 7.1.4, 7.1.5, 13.1.3
std::string 3.2
std::vector 3.3
enumeration types 19.3