Lecture 2

Introduction to C++, function parameters, strings, streams

TDDD86: DALP

Utskriftsversion av Lecture in *Data Structures, Algorithms and Programming Paradigms*
September 6th, 2024

IDA, Linköping University

Content

Contents

1 C++ – introduction

C++ history: C

- C was introduced 1972 and became very successful
- C made it easier to write fast code for different platforms
- C was popular because it was simple:
  - Not much redundant syntax
  - Extremely fast execution
  - Available anywhere there is a C compiler (that is really everywhere)
- No objects or classes

Ken Thompson and Dennis Ritchie, creators of the C language

C++ history

1980 C with classes
1983 C++ was created by Bjarne Stroustrup:
... Introducing multiple inheritance, templates (generics) and exceptions
1998 ISO-standard, defining a standard library
2003 Bug fixes to improve consistency and portability
2011 **Major ISO-standard, C++11 (lambda, auto, threads...)**
2014 Bug fixes and small improvement
2017 typename in templates, nested namespace definitions, ...
2020 concepts, string litterals as template parameters, three-way comparison, ...
2023 literal siffix for size_t, simplifying lambda expressions...

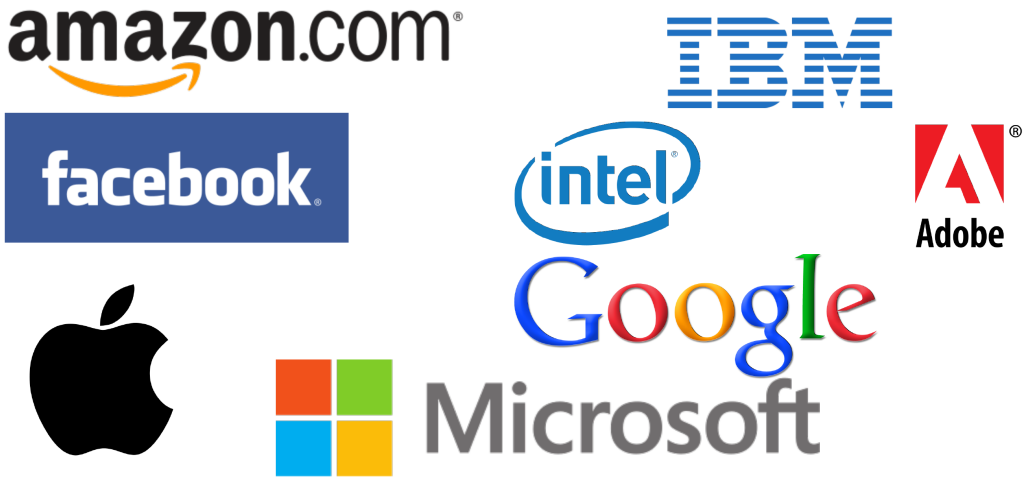
What is C++

- Almost all C-code is also valid C++
- What is valid C++ is defined in more than a thousand-pages standard `https://isocpp.org/std/the-standard`
- C++ is popular because it provides a good balance between performance and ease of development
- But not an easy language to start with

What is C++

- C ++ is a programming language that simplifies complex tasks without sacrificing performance
- Learning how to write “good C++” is a very good way to increase your understanding of programming in general

C++ users (corporations)



2 Base in C++

Hello world in C++

```
/*
 * hello.cpp
 * This program prints a welcome message
 * to the user.
```

```
 */

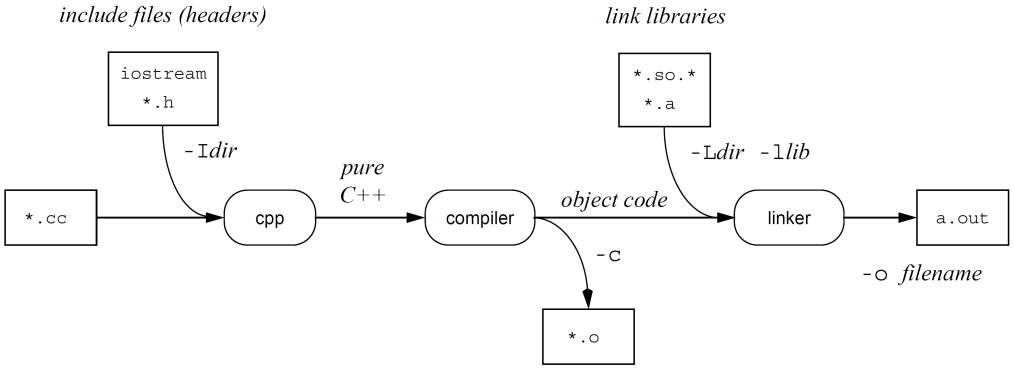
#include<iostream>

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

2.8

• Programs/files in C++

- C++ source code in `.cpp`-files
  - Additional declarations may be placed in “headerfiles", `.h`-filer
- Source is compiled into a binary *object file*
- Object files and libraries are linked together to form a program
  - Unlike **`.class`** in Java, program/objects are *platform dependent*



2.9

main function

```
int main()
{
    statement;
    statement;
    statement;
    ...
    return 0;
}
```

- `main`-function is a special function which indicate the start point of a program
  - Unlike in Java, in C++ (like in Python), functions do not need to be part of a class
  - `main` may call other functions
  - Unlike in Java, in C++ `main` returns an integer to indicate to the operating system if an error has occured
    - Return 0 to indicate no error
    - Can be printed from command line with “`echo $?`”

2.10

Typical syntax

```
int x = 42 + 7 * -5;              // variable, typed
double pi = 3.14159;
char c = 'Q';                     /* two styles of comments */
bool b = true;

for (int i = 0; i < 10; i++) {    // for-loop
    if (i % 2 == 0) {             // if-statement
        x += i;
    }
```

```
}

while (x > 0 && c == 'Q' || b) {  // while-loop, logic
    x = x / 2;
    if (x == 42) {
      break;
    }
}

fooBar(x, 17, c);                 // function call
```

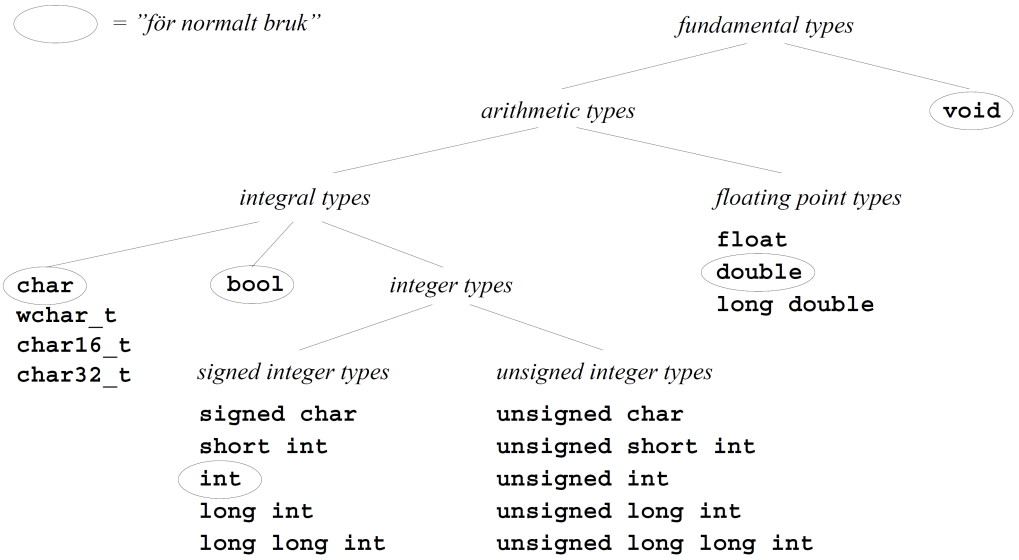
2.11

Data types in C++

- fundamental types
  - example **int**, **double**, **char**, **bool** and **void**
- compound types
  - example *class* (**struct**, **class**), array (**int**[]), pointer, references and functions

2.12

• Overview of fundamental types



2.13

Compound datatyper

- array is an indexed type with objects of the same type (you should use `std::array` or `std::vector` instead)
  - **int** `a[100]` (can be cast to **int***)
- pointer to an object of a specific type
  - **int** `*p`
- reference to an object of a specific type
  - **int** `&p`
- class (**struct**, **class**) consists of variables and even functions
  -
```
struct point
{
    int x;
    int y;
};
```
- functions have parameters of a given type and return **void** (nothing) or an object of a given type
  - **int** `max(int a, int b)` Type: **int** `(int, int)`

2.14

• Pointers

- Pointers contains the memory address to an other object

```
int i = 4711;    // int variable
int *p = &i;     // pointer to an int variable with address operator &
```

- Usage

```
cout << i << endl;              // 4711
cout << p << "␣->␣" << *p;      // 0xeffff828 -> 4711
```

The *address-of operator* `&` can give the address of

- a variable, an element in an array, a member of an object instance, ...

The *indirection operator* `*` can give the value pointed to by a pointer

- it can be used for an operation on the value in an expression

2.15

• Reference

```
int i = 4711;
int &r = i;          // reference to variable, it is an alternate name for i
```

- A reference must *always* be initialized when defined and can not be changed

2.16

Include

- **`#include`**<**libraryname**>
  - When you want to use system pre-installed C++ headers
  - For.ex. **`#include`**`<iostream>` for I/O streams
- **`#include"`****header.h****`"`**
  - For libraries and headers in local folder
  - T.ex. **`#include`**`"lifeutils.h"` in lab 1

2.17

Using

- **`using namespace`** **name**
  - Many libraries use a *namespace* to separate their symbols (variables, functions, etc.) and not pollute the global namespace
  - A **using**-declaration import the symbols from the library into the global namespace
    - Example: **`using namespace`** `std;` to get all the standard library symbols `cout`, `cin`, `endl`, etc.
- **namespace : : identifier**
  - Witout **using**-declaration, symbols must be prefixed with the namespace and `::`
    - `std::cout << "Hello,␣world!"<< std::endl;`

2.18

3 Functions

3.1 Definition and declaration

Define functions

- Functions in C++ are similar to methods in Java. They have similar syntax but without the need for `public` or `private` keywords

```
type name(type name, type name, ..., type name)
{
    statement;
    statement;
    statement;
    ...
    statement;
    return expression;    // if we are not returning void
}
```

2.19

Example: function with parameters

```
// Return the biggest of two integers
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int bigger1 = max(17, 42);    // call the function
    int bigger2 = max(29, -3);    // call again
    int biggest = max(bigger1, bigger2);
    cout << "The biggest is " << biggest << "!!" << endl;
    return 0;
}
```


Order of declaration

- The program below does not compile
  - The compiler claims that it can not find the function `max`!

```
int main() {
    int bigger1 = max(17, 42);  // call the function
    return 0;
}

int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```


Functions prototype

- `type name(type name, type name, ..., type name);`
  - Declare the function without defining it at the top of the program
  - Now the compiler knows about the function and that it will be defined later
  - The prototype can be placed in a `.h`-file

```
int max(int a, int b);  // prototype declaration for max

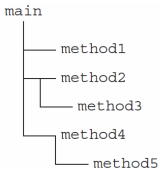
int main() {
    int bigger1 = max(17, 42);  // call the function
    return 0;
}

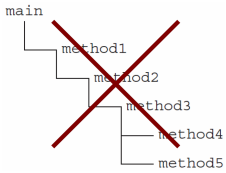
int max(int a, int b) {
    ...
}
```


Procedural degradation

- When solving a large problem, you will need to structure your code and divide tasks into functions
- Characteristics of a good function
  - Perform absolutely a well defined task
  - Do a small subset of the work
  - Is not unnecessary using other functions

  - Variables should be accessible from a narrower scope
- `main` should be a concise summary of the overall program
  - Most calls to other functions should come from `main`





2.23

3.2 Parameters

Value vs reference

- **value semantics**: In Java and C++, when a value with a basic type (**`int`**, **`double`**) is transferred as a parameter, its value is copied
  - Changing the value of the parameter variable does not affect the value in the call

```
void grow(int age) {
    age = age + 1;
    cout << "grow␣age␣is␣" << age << endl;
}

int main() {
    int age = 20;
    cout << "main␣age␣is␣" << age << endl;
    grow(age);
    cout << "main␣age␣is␣" << age << endl;
    return 0;
}
```

Output:

```
main age is 20
grow age is 21
main age is 20
```

2.24

Passing by reference

- **semantic of a reference**: If you declare a parameter with a `&` after the type in C++ this will link the variable in the calling code and in the function to the same area in memory
  - Value change in the function will affect the calling function

```
void grow(int& age) {
    age = age + 1;
    cout << "grow␣age␣is␣" << age << endl;
}

int main() {
    int age = 20;
    cout << "main␣age␣is␣" << age << endl;
    grow(age);
    cout << "main␣age␣is␣" << age << endl;
    return 0;
}
```

Output:

```
main age is 20
grow age is 21
main age is 21
```

2.25

Example

- Now you can write a swap-function!

```
/*
* Place a's value in b and vice versa.
*/
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

2.26

Benefits and drawbacks of reference parameters

- **benefits** of reference parameters:
  - a usual way to "return" more than one value
  - is often used with objects to avoid expensive copy
- **drawbacks** of reference parameters:
  - difficult to see in the call line if the value is passed by reference or not and if the value will be changed?
    - `foo(int& a, int& b); //can foo change the value of a or b? :-/`
  - (slightly) slower than passing by value (for ground types)
  - literals can not be transferred by reference
    - `grow(39); //fail`

2.27

Const reference

- **semantic of a const reference**: if you declare a parameter with **`const`** `type&` this will link the variable in the calling code and in the function to the same area in memory but the function will not be able to change the value

```
void grow(const int& age) {
    age = age + 1;
    cout << "grow␣age␣is␣" << age << endl;
}

int main() {
    int age = 20;
    cout << "main␣age␣is␣" << age << endl;
    grow(age);
    cout << "main␣age␣is␣" << age << endl;
    return 0;
}
```

2.28

Benefits and drawbacks of const reference

- **benefits** of const reference parameters:
  - no need to wonder if the values is passed by reference or not (does not affect the calling code)
  - literals can be transferred by const reference
    - `grow(39); //works`
    - Still slower if you want to pass base literals (only use for large objects!)
- **drawbacks** of const reference parameters:
  - the value cannot be changed

2.29

When to pass parameters by value, reference or const reference?

- pass fundamental types (**int**, **double**...) by value
- use references if you need to return several values

```
void compute(int& result1, int& result2) {
     result1 = foo(...);
     result2 = bar(...);
}

int main() {
    int result1=0;
    int result2=0;
    compute(result1, result2);
    std::cout << result1 << "␣" << result2 << std::endl;
    return 0;
}
```

- pass compound object as const reference
- Expected style guildelines: `https://www.ida.liu.se/~TDDD86/info/style.sv.shtml`

2.30

Default parameters

- You can make a parameter optional by providing a default value
  - Parameters with default values must come last in the parameter list

```
// Prints a range of characters with a specified width
void printLine(int width = 10, char letter = '*') {
    for (int i = 0; i < width; i++) {
        cout << letter;
    }
}

...

printLine();          // **********
printLine(5);         // *****
printLine(7, '?');    // ???????
```

2.31

4 Strings

Strings

```
#include<string>
...
string s = "hello";
```

- A string is a (possibly empty) sequence of characters
- Strings in C ++ are conceptually similar to strings in Java
  - Several small differences:
    - Different names for similar approaches
    - Different behaviour similar methods
  - And some really big differences:
    - There are two types of strings in C ++
    - In C++ strings are mutable

2.32

Character

- Characters are variable of type **char**, with 0-based index:
  - `string s = "Hi␣D00d!"`

index	0	1	2	3	4	5	6	7
character	'H'	'i'	' '	'D'	'0'	'0'	'd'	'!'

- Individual characters can be accessed by indexing operator or method `at`:
  - **char** `c1 = s[3]; //D`
  - **char** `c2 = s.at(1); //i`

2.33

Operators

- Like Java, you can concatenate strings:
  - ```
    string s1 = "ka";
    s1 += "nin"         // "kanin"
    ```

- Unlike Java, you can compare strings with relational operators:
  - ```
    string s2 = "apa";
    if (s1 > s2 && s2 != "kaka") {    // true
    ...
    }
    ```

- Unlike Java strings are mutable and can be changed!
  - ```
    s1.append("␣krubbar")  // s1 == kanin krubbar
    ```

2.34

Strings from C vs C ++

- Technically speaking, C ++ has two kinds of strings:
  - C-strings (“array” of **`char`**), inherited from the C language
  - `string`-object, comes from the C++ standard library
  - If possible, declare your variable with `string`
- All string literals such as `"hi␣there"` are C-strings
  - C-strings have no members
- Converting between string types
  - `string("text")` convert C-string to a string object
  - `s.c_str()` returns a C-string from a C++ string object

2.35

Bugs related to using C-strings

- This does not compile:

```
// print the double of a number
void printDouble(string s) {
    cout << s * 2 << endl;
}
```

- Does this?

```
// print a number appended with 4
void appendFour(int n) {
    cout << n +  "4" << endl;
}
```

2.36
```

5 Streams

Basic concepts behind streams

"Designing and implementing a general input/output facility for a programming language is notoriously difficult" --Bjarne Stroustrup

2.37

Writing to the console: cout

- `cout` << **expression**
  - Sends the specified value to the console standard output
  - << can be chained to form a more complex output
    - `cout << "You␣are␣"<< age << "␣years␣old!␣\n";`

- `endl`
  - A variable which means “go to next line and flush the output"
    - `cout << "You␣are␣"<< age << "␣years␣old!"<< endl;`

2.38

Input from the console: cin

- `cin` >> **expression**
  - Read from the console and store in the variable

- Note that `cout` use << but `cin` use >>
  - << >> are the “pilars” of data flow (streams)

2.39

Strings as input

- `cin` can read a string, word by word

```
string name;
cout << "Type␣your␣name:␣";          // Type your name: John Doe
cin >> name;
cout << "Hello,␣" << name << endl;   // Hello, John
```

- The function `getline` read a full line

```
string name;
cout << "Type␣your␣name:␣";          // Type your name: John Doe
getline(cin, name);
cout << "Hello,␣" << name << endl;   // Hello, John Doe
```

2.40

Reading from files

- **#include**<fstream>
  - Introduce class `ifstream` and `ofstream` for reading/writing from/to a file

```
ifstream input;
input.open("poem.txt");
string line;
getline(input, line);
```

- `cin` is a variable of type `ifstream`, `cout` has type `ofstream`
  - Reading and writing from file works like `cin`/`cout`

```
string filename ="data/docs/bank.txt";
ifstream input;
input.open(filename.c_str());
string line;
while (getline(input, line)) {
    cout << line << endl;
}
input.close();
```