Software Design (C++)

0. Introduction and overview

Juha Vihavainen
University of Helsinki

On classroom etiquette

- Please, silence your cell phone while in class.
- Please, do not take or make phone calls in class.
- Please, do not keep up discussions among each other during lectures
  - really, the lectures are voluntary and ongoing discussions disturb others wishing to listen
  - but, of course, do ask me questions . .
Rough course outline

- The basics
  - types, variables, strings, computations
  - simple IO, error handling (if any), exceptions, references, enums, overloading, etc.
  - small C++ code samples
- Data structures and algorithms
  - free store, pointers, and built-in arrays (directly from C)
  - user-defined types (classes)
  - on implementing vectors, lists, iterators..
- Also, safety and resource handling, templates, object-oriented programming & class hierarchies

A textbook *Programming - Principles & Practice with C++* (2. ed.) [Stroustrup, 2014; ~1300 pages]

Ch. 1 Computers, People, and Programming
Ch. 2 Hello, World!
Ch. 3 Objects, Types, and Values
Ch. 4 Computation
Ch. 5 Errors
Ch. 6 Writing a Program (Calculator)
Ch. 7 Completing a Program
Ch. 8 Technicalities: Functions, etc.
Ch. 9 Technicalities: Classes, etc.
Ch. 10 Input and Output Streams
Ch. 11 Customizing Input and Output
Ch. 12 A Display Model
Ch. 13 Graphics Classes
Ch. 14 Graphics Class Design
Ch. 15 Graphing Functions and Data
Ch. 16 Graphical User Interfaces
Ch. 17 Vector and Free Store
Ch. 18 Vectors and Arrays
Ch. 19 Vector, Templates, and Exceptions
Ch. 20 Containers and Iterators
Ch. 21 Algorithms and Maps
Ch. 22 Ideals and History
Ch. 23 Text Manipulation
Ch. 24 Numerics
Ch. 25 Embedded Systems Programming
Ch. 26 Testing
Ch. 27 The C Programming Language
App A Language Summary
App B Standard Library Summary
App C - E Visual Studio, FLTK, GUI code

Note the emphasis.
What kind of a language is C++?

- The next five slides give an overview of the evolution of C++
- But let’s hear how the father of C++, Bjarne Stroustrup explains the motivation and main features of the language (the first 20 mins.)
  - The Essence of C++ [GoingNative 2013, channel 9, url: http://channel9.msdn.com/Events/GoingNative/2013/OpenInKeynote-Bjarne-Stroustrup ]
- Modern C++ is not only for system programmers, but also for creative coding
  - OpenFrameworks [url: www.openframeworks.cc ]

What is C++? Some answers:

- An object-oriented programming language
- A hybrid language
- It’s C. Low level and too big.
- Supports generic programming
- A multi-paradigm programming language
- Template meta-programming
- Embedded systems programming language

**C++ and safety**

C++ is not statically type safe
C++ is not dynamically type safe

"a chain saw with all the safety guards removed" (Bob Gray)
Beginning with "C with Classes" ~ 1980

- C with Classes ~1980
- General abstraction mechanisms to cope with complexity
  - class from Simula
- General close-to-hardware machine model for efficiency
  - from C
- Became C++ in 1984
- Commercial release 1985
- ISO standard 1998 - a "corrected" version 2003
- 2nd ISO standard 2011
- See www.isocpp.org about C++ standardization activities
First version

*C with Classes* (1979-1983)

- originally implemented as a preprocessor (*C++ => C => native*)
- features already include
  - classes
  - derived classes (i.e., subclasses)
  - **public/private** access control
  - **friend** classes
  - constructors and destructors
  - more static type checking (than C)

- "Adding classes to the C language", *Software - Practice and Experience*, Feb 1983, pp. 139-161.

Evolution of C++ from 1982

- **virtual** functions (already in *Simula*, 1967)
- function name and operator (++, *, ..) *overloading*
- **references** (&), in addition to old C pointers
- **const** variables and parameters
- user-controlled heap, via **new** and **delete** operations
- a version of multiple inheritance (used, e.g., by IO stream library)
- overloading of assignment and initialization ("=": value copy)
- **pure virtual** functions and **abstract** classes
- **const member functions** (non-mutating methods)
- **enumerations**
- **exceptions**: **throw**, and **try - catch** construct
- **templates** & Standard Template Library (STL)
- evolution and its standardization just goes on..
"C++11": even more features

- E.g., *auto*-typed variables, static assertions, scoped enums, lambdas, rvalue references, delegating constructors, defaulted and deleted functions, new function declarator syntax, inline namespaces, extern templates, local and unnamed types as template arguments, variadic template parameters, thread support library, std::initializer_list <T>, the "right angle brackets" problem, raw string literals, ..
  - some of these we will discuss as need arises

- "Every feature is implemented somewhere"

- New standard library components are shipping widely
  - e.g. GCC, Microsoft, Boost

What do we need to know about "C"?

- C++ is based on "C" - common features:
  - Direct manipulation of computer memory (pointers etc.)
  - Compilation and linking of programs as separate translation units (also the use of header files)
  - Syntax, semantics, and use of variables and functions
  - Structure of the statements and expressions
  - Primitive null-terminated strings, typedefs, assert-macro

- C and C++ code can co-exist in the same program (few caveats)

- The first exercises focus on the features of C that are necessary to understand for developing system-level code with C++
  - The course book is written without assuming prior knowledge of C, which should help in case C is unfamiliar
Tools

- **You need a compiler** that implements most of the features defined in the C++11 standard
  - For example, gnu gcc v. 4.8.4 or Microsoft Visual C++ 2012/2013 (see the "On-line resources" sub page at the home page of this course)
  - The Linux workstations at the department have all gcc 4.8.2 installed
  - There is also a remote machine running Linux and gcc 4.8.2 at pangolin.it.helsinki.fi (login with ssh and your CS account)

- You don't need an IDE (but you can use one, of course!)
  - The programs will be small and we are not building GUIs
  - A good text editor (like GNU Emacs on Linux or Notepad++ on Windows) and a compiler run from the command line are enough
  - You can even manage without make
  - It is good to understand what goes on "behind the scene"
  - IDEs are great and can help you a lot with bigger, more complex programs
    - Many IDEs provide also libraries (SDK) for services needed by applications (platform specific or cross-platform)
    - Examples: Eclipse, Visual Studio C++, Qt, Xcode
A first program – complete

// a first program:
#include <iostream>    // get the required IO facilities
int main () {        // where a C++ program starts
    std::cout << "Hello, world!\n";   // output the 13 characters,
            // followed by a new line
    return 0;                      // a value indicating success
}
// here, main () takes no arguments
// and returns an int to indicate success or failure
// "<iostream>" provides needed library headers (declarations)

A second program

// modified for a "console mode":
#include <iostream>    // get the required IO facilities
int main () {        // where a C++ program starts
    std::cout << "Hello, world!\n";   // output the 13 characters,
            // followed by a new line
    char c; std::cin >> c;             // wait for an input character
    return 0;                      // a value indicating success
}
// without std::cin >> c; the output window may be closed before you
// have a chance to read the output (on some implementations)
Compilation and linking (directly from C)

- The compiler translates what you wrote into object code (machine-level code)
- The linker links your code to system code needed to execute
  - e.g. input/output libraries, operating system code, and windowing code
- The result is an executable program
  - e.g. a .exe file on Windows or an a.out file on Unix

Note

- The code examples in the book Programming - Principles & Practice with C++ (2. ed.) include the header file "std_lib_facilities.h"
  - This header file contains declarations and definitions that make it easier for a novice programmer to write C++ programs
  - You can download the file from the home page of the book: http://www.stroustrup.com/Programming/ (look for standard library access header)
  - Using the header you can type in the examples from the book and compile and run them
- But: in the code examples on these slides we always use the standard headers directly
I/O library overview

Stream I/O

- `in >> x` Read from `in` into `x` according to `x`'s format
- `out << x` Write `x` to `out` according to `x`'s format
- `in.get(c)` Read a character from `in` into `c`
- `getline(in,s)` Read a line from `in` into the string `s`

I/O stream classes with multi-inheritance

```
#include <iostream>
#include <string>

int main () {
    std::cout << "Please enter your first name (followed by 'enter'):
    " << std::endl;
    std::string firstName;
    std::cin >> firstName;
    std::cout << "Hello, " << firstName << "!
    " << std::endl;
}
```

Input and output

```
#include <iostream> // get the required IO facilities
#include <string>   // get the standard std::string

int main () {
    std::cout << "Please enter your first name (followed " "by 'enter'):\n    ";
    std::string firstName;
    std::cin >> firstName;
    std::cout << "Hello, " "firstroke << firstName << "!
    " << std::endl;
}
```

// note how literal strings can be concatenated
// note how several values can be output by a single statement
// the final return 0 is optional in main ()
// - but you may need to include it to pacify your compiler
Input and type

- We read into a variable of type `std::string`
- The type of a variable determines what operations we can do on it
- Here, "`std::cin >> firstName;`" calls an overloaded "`>>`"
  - first skips any leading white space, then
  - reads characters in a word until a white-space character is seen (`space, tab, newline,..`)
  - the input text ("word") can be of any size, and the string "grows" in size as needed (allocated space in a buffer)

---

String input plus concatenation

```cpp
int main () {
    // read first and second name:
    std::cout << "Please enter your first and second names\n";
    std::string first, second;
    std::cin >> first >> second;       // read two strings
    std::string name = first + ' ' + second;   // concatenate,
                                                   // separated by a space
    std::cout << "Hello, " << name << "\n";
}
```

Integer input

- The same IO operations >> and << work for different types, e.g., for integer values
- Later we make them work similarly for user-defined types, too

```cpp
// read name and age:
int main () {  
    std::cout << "Please enter your first name and age\n";  
    std::string firstName;  // string variable
    int age;  // integer variable
    std::cin >> firstName >> age;  // read both
    std::cout << "Hello, " << firstName << " age " << age << \n';
}
```

Integers and strings

- The type of a variable determines which operations (names) are valid and what their meanings are for that type
- uses "overloading" or "operator overloading"

- `std::string`
  - `cin >>` reads until whitespace
  - `cout <<` writes the string
  - `+` concatenates
  - `+= s` adds the string `s` at end
  - `++` is a compile-time `error`
  - `-` is a compile-time `error`
- `...`

- `Integer and floating point number`
  - `cin >>` reads a number
  - `cout <<` writes the number
  - `+` adds
  - `+= n` increments by the int `n`
  - `++` increments by 1
  - `-` subtracts
  - `...`
A simple computation

```cpp
int main () {   // inch-to-cm conversion
    const double cm_per_inch = 2.54;   // centimeters per inch
    int length = 1;   // length in inches
    while (length != 0) {   // 0 is used to exit
        std::cout << "Please enter a length in inches: ";
        std::cin >> length;
        std::cout << length << " in.  = "
                    << cm_per_inch * length << " cm.\n";
    }
}
```

Types and literals

- Built-in types
  - Boolean type
    - `bool`
  - Character types
    - `char`, `char16_t`, `char32_t`
- Integer types
  - `int`, `short`, `long`, `long long`
- Floating-point types
  - `float`, `double`, `long double`
- Standard-library types
  - `std::string`
  - `std::complex <ScalarType>`
- Most types are "standalone"

- boolean literals
  - `true` `false`
- character literals
  - `'a', 'x', '4', '\n', '$'
- integer literals
  - `0, 1, 123, -6, 0x34, 0xa3, 1234576L, 123456789LL`
- floating point literals
  - `1.2, 13.345, .3, -0.54, 1.2e3, .3F, .3F, 13.345L`
- string literals "asdf" (C-style string)
- complex "literals" (constructor calls)
  - `complex <double> (12.3, 99)`
  - `complex <float> (1.3F)`
  - `{ 12.3, 99 }`   // C++11
C++ types

- C++ provides a set of built-in types
  - represent the native types of the underlying hardware
  - e.g. bool, char, int, double, long long, long double, etc.
- C++ programmers can define new types (struct, class, enum)
  - called “user-defined types” = abstract data types/subclasses
  - we'll get to that later
- The C++ standard library provides types (in std namespace)
  - e.g. string, vector, complex, list, map, istream, ostream
  - in principle, these are “user-defined types”, too
    - i.e., they are built using facilities available to every user: classes, overloading, templates, exceptions.

Declaration and initialization

```cpp
int a = 7;
ing b = 9;
char c = 'a';
double x = 1.2;
std::string s1 = "Hello, world";
std::string s2 = "1.2";
```

(a: 7  b: 9  c: 'a'  x: 1.2  s1: "Hello, world"  s2: "1.2" (conceptually "containers") )
Objects

- "Object" is a technical term in C++
  - some memory that holds a value of a given type (built-in/user-defined)
- A variable is a named or "declared" object
  - since a declaration (usually) names an object

```
int a = 7;
char c = 'x';
std::complex<double> z(1.0, 2.0);
std::string s = "qwerty";
```

```
7
'x'
1.0  2.0
"qwerty"
```

Type safety

- Language rules try to enforce or support type safety
  - every object is used only according to its type, i.e., only operations defined for the object's type will be applied
  - each operation is (hopefully) programmed to leave the object with a valid value (or indicates an error..)
- Ideal: **static type safety**
  - a program that has a type violation will not even compile
  - the compiler reports each violation (in an ideal system)
    - no need to run, test, and determine code coverage
- Ideal: **dynamic type safety**
  - type violations detected (prevented) at run time (at the latest)
  - some code ("the run-time system") reports violations not found by the compiler; usually via exceptions
Type safety and C++

- Type safety is a very big deal: “the compiler is your best friend”
  - but it won’t always feel like that when it rejects your “correct” code
- C++ is not *statically* type safe
  - no practical language is (completely) statically type safe
  - absolute static type safety would interfere with our ability to express run-time computations (say, using dynamic index values..)
- C++ is not *dynamically* type safe
  - dynamic type checking may make code bigger and slower
  - in C++, may have to insert checks of our own (*assert()*, *check()*)
- Java and C# compensate for lack of total static safety with dynamic checks => such languages are called *strongly typed* (prevent type errors)

Most of what you’ll be taught here is type safe

We’ll try to specifically mention anything that is not

---

A type-safety violation ("implicit narrowing")

```c++
// beware: C++ does not prevent you from trying to put a large value
// into a small variable (though a compiler may possibly warn)

int main () { // illustrative code
    int a = 20000;
    char c = a; // or "(char)a"
    int b = c;
    if (a != b) // of the same type (int)
        std::cout << "oops!: " << a << " != " << b << "\n";
    else
        std::cout << "Wow! We have very large characters\n";
}
```

in C++, we don't (necessarily) know what will happen ("c = 20000")
- try it to see what value *b* gets on your machine
- warnings depend on compiler options (e.g., VC++ not by default)
Type-safety violation: uninitialized variables

// beware: C++ does not prevent you from trying to use a variable
// before you have initialized it (though a compiler may try to warn)

int main () {
    int x;     // x gets a "random" initial value
    char c;    // c gets a "random" initial value
    double d;  // d gets a "random" initial value
    double dd = d; // potential failure, in some hardware:
                    // can't copy invalid floating-point values
    std::cout << " x: " << x << " c: " << c << " d: " << d << 'n'; // warn?
}

- not every bit pattern is a valid floating-point (as seen by the hardware)
- beware: "debug mode" may initialize by some default values - or not
- always initialize your variables
  - one possible exception to this rule: use as input variable (nearby)

About efficiency

- C++ is derived from C, another systems programming language
- still provides direct access to hardware (memory/instructions)
- C++’s built-in types map directly to computer main memory
  - an int is stored in a memory word
  - a double fits in a floating-point machine register
- C++’s built-in operations map directly to machine instructions
  - an integer + is implemented by an integer add operation
  - an integer = is implemented by a simple move operation
- C++ may help to build safer, more elegant, and efficient new types and operations using these built-in types and operations
  - e.g., std::string, std::vector, std::list
- eventually, we’ll show some of how that’s done
- For now, concentrate on correctness and simplicity of code
A bit of philosophy

- One of the ways that programming resembles other kinds of engineering is that it involves a lot of tradeoffs.

- We have ideals or requirements, but they often conflict, so must decide what matters for a given program under specific circumstances:
  - type safety (static/dynamic)
  - run-time performance (vs. e.g., dynamic checks for type safety)
  - ease of construction
  - ease of maintenance (which may make construction harder...)
  - ability to run on our given platform
  - ability to run on multiple platforms with same results (portability)
  - compatibility with 3rd-party code, libraries, and legacy systems

- Don't cut corners on correctness or testing.
- By default, aim for type safety and portability (not so trivial in C++)