CISC 3142
Programming Paradigms in C++

Ch1 - Notes to the Reader

(*Stroustrup – The C++ Programming Language, 4th Ed*)

The Design of C++

- Based on the idea of providing both of the following
  - Direct mapping of built-in operations and types to hardware
    - This ensures efficient memory use and low-level operations
  - Affordable and flexible abstraction mechanisms
    - This provides user-defined types that are closely matched to a problem to be solved
- Synthesis of programming styles that are both efficient and elegant
- Focused on concerns of systems programmers and those dealing with resource-constrained and high-performance systems
- Characterizations:
  - C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining light weight abstractions
  - Leaving no room for a lower-level language below C++
  - What you don’t use you don’t pay for it – zero-overhead principle

Programming Styles

- Stroustrup prefers “Programming Styles” over “Paradigms”
- C++ directly supports four styles:
  - Procedural programming (continuation of C)
  - Data abstraction (design of interface and implementation hiding)
  - Object-oriented programming (class hierarchies and polymorphism)
  - Generic programming (general algorithms, templates)
- But emphasis is on the support of effective combination of all those – as the best solution tends to be one that combines all styles
- More recent C++ (C++11) supports *move* semantics and can facilitate *functional programming* which discourages objects with memory identity

Type Checking

- Static types and compiler-time type checking is central to C++
- They ensure prevention of accidental data corruption during compiler-time analysis
- Requiring very little overhead or runtime support
  - Generally, no “housekeeping information” need to be stored in every object
  - Only `new`, `delete`, `typeid`, `dynamic_cast`, `throw` operators and the `try`-block, require runtime support

Learning C++

- No programming language is perfect, especially for a general-purpose one
- Learning a language should focus on mastering the native and natural styles for that language – not on understanding every little detail
- Go beyond the language features and make good use of libraries, as they:
  - Simplify the task of programming and increase the quality of our systems
  - Improve maintainability, portability, and performance
- For programmers from other languages, try to learn the idiomatic C++ programming style and technique instead of trying to apply techniques from other languages

Programming in C++ (tips)

[1] Represent ideas directly in code.
[2] Represent relationships among ideas directly in code (e.g., hierarchical, parametric, and ownership relationships).
[3] Represent independent ideas independently in code.
[4] Keep simple things simple (without making complex things impossible).

More specifically:
[5] Prefer statically type-checked solutions (when applicable).
[6] Keep information local (e.g., avoid global variables, minimize the use of pointers).
[7] Don’t overabstract (i.e., don’t generalize, introduce class hierarchies, or parameterize beyond obvious needs and experience).

Suggestions for C++ Programmers

[1] Use constructors to establish invariants (§2.4.3.2, §13.4, §17.2.1).

[2] Use constructor/destructor pairs to simplify resource management (RAII; §5.2, §13.3).

[3] Avoid ‘‘naked’’ **new** and **delete** (§3.2.1.2, §11.2.1).

[4] Use containers and algorithms rather than built-in arrays and ad hoc code (§4.4, §4.5, §7.4, Chapter 32).

[5] Prefer standard-library facilities to locally developed code (§1.2.4).

[6] Use exceptions, rather than error codes, to report errors that cannot be handled locally (§2.4.3, §13.1).

[7] Use move semantics to avoid copying large objects (§3.3.2, §17.5.2).

[8] Use **unique_ptr** to reference objects of polymorphic type (§5.2.1).

[9] Use **shared_ptr** to reference shared objects, that is, objects without a single owner that is responsible for their destruction (§5.2.1).

[10] Use templates to maintain static type safety (eliminate casts) and avoid unnecessary use of class hierarchies (§27.2).

Suggestions for Java Programmers

[1] Don’t simply mimic Java style in C++; that is often seriously suboptimal for both maintainability and performance.

[2] Use the C++ abstraction mechanisms (e.g., classes and templates): don’t fall back to a C style of programming out of a false feeling of familiarity.

[3] Use the C++ standard library as a teacher of new techniques and programming styles.

[4] Don’t immediately invent a unique base for all of your classes (an **Object** class). Typically, you can do better without it for many/most classes.

[5] Minimize the use of reference and pointer variables: use local and member variables (§3.2.1.2, §5.2, §16.3.4, §17.1).

[6] Remember: a variable is never implicitly a reference.

[7] Think of pointers as C++’s equivalent to Java references (C++ references are more limited; there is no reseating of C++ references).

[8] A function is not **virtual** by default. Not every class is meant for inheritance.

[9] Use abstract classes as interfaces to class hierarchies; avoid ‘‘brittle base classes,’’ that is, base classes with data members.

Suggestions for Java Programmers (cont')

[10] Use scoped resource management (''Resource Acquisition Is Initialization''; RAII) whenever possible.

[11] Use a constructor to establish a class invariant (and throw an exception if it can't).

[12] If a cleanup action is needed when an object is deleted (e.g., goes out of scope), use a destructor for that. Don't imitate **finally** (doing so is more ad hoc and in the longer run far more work than relying on destructors).

[13] Avoid ''naked'' **new** and **delete**; instead, use containers (e.g., **vector**, **string**, and **map**) and handle classes (e.g., **lock** and **unique_ptr**).

[14] Use freestanding functions (nonmember functions) to minimize coupling (e.g., see the standard algorithms), and use namespaces (§2.4.2, Chapter 14) to limit the scope of freestanding functions.

[15] Don't use exception specifications (except **noexcept**; §13.5.1.1).

[16] A C++ nested class does not have access to an object of the enclosing class.

[17] C++ offers only the most minimal run-time reflection: **dynamic_cast** and **typeid** (Chapter 22). Rely more on compile-time facilities (e.g., compile-time polymorphism; Chapter 27, Chapter 28).

Chapter-end Advice

[1] Represent ideas (concepts) directly in code, for example, as a function, a class, or an enumeration; §1.2.

[2] Aim for your code to be both elegant and efficient; §1.2.

[3] Don't overabstract; §1.2.

[4] Focus design on the provision of elegant and efficient abstractions, possibly presented as libraries; §1.2.

[5] Represent relationships among ideas directly in code, for example, through parameterization or a class hierarchy; §1.2.1.

[6] Represent independent ideas separately in code, for example, avoid mutual dependencies among classes; §1.2.1.

[7] C++ is not just object-oriented; §1.2.1.

[8] C++ is not just for generic programming; §1.2.1.

[9] Prefer solutions that can be statically checked; §1.2.1.

[10] Make resources explicit (represent them as class objects); §1.2.1, §1.4.2.1.

[11] Express simple ideas simply; §1.2.1.

[12] Use libraries, especially the standard library, rather than trying to build everything from scratch; §1.2.1.

[13] Use a type-rich style of programming; §1.2.2.

[14] Low-level code is not necessarily efficient; don't avoid classes, templates, and standard library components out of fear of performance problems; §1.2.4, §1.3.3.

[15] If data has an invariant, encapsulate it; §1.3.2.

[16] C++ is not just C with a few extensions; §1.3.3.