Learning to program and learning to be a software designer & engineer are different things.

Here we focus on programming, but some C/C++ concepts are best understood in light of design & engineering.

Let’s contrast programming & software engineering:
Design principles

Programming project (e.g. for this class)

- Start with: formal algorithm, detailed requirements
- Work: by yourself or on small team
- Phases: just do it
- Expected software lifetime: often brief

Software engineering

- Start with: human input, vague requirements
- Work: on large team
- Phases: user interviews, design documents, prototype, product
- Expected software lifetime: years, decades
  - E.g. Y2K bug; that software was decades old
An electronic sign displaying the year incorrectly as 1900 on 3 January 2000; example of **Y2K bug**

Design principles

Readability: easy to read and understand code

Conciseness: no needlessly complex code, little repetition

Robustness: modifying one aspect shouldn't break another
To achieve these goals:

- *Separation of concerns*: distinct code units address distinct problems
- *Encapsulation*: implementation details are hidden; only a simple interface is exposed
Separation of concerns

Distinct code units address distinct problems

- Code units: source files, functions, classes, etc
- Also called *modularity*
Separation of concerns

Advantages:

- Promotes re-use
- Easier to develop separate code units separately
- Robust

Caveats:

- Concerns are often not perfectly separable
  - Chess: *some* part of the code needs to understand both Board and Piece to check if a move is legal
- Combining concerns can make code more efficient
  - Chess: say we want to determine whether a king is “in check” without iterating over all the pieces. Hard to do without substantially mixing Piece and Board concerns.
Encapsulation

Modular programming is an exercise in creating independent code units with good “interfaces”:

- classes have public members and protected/private members
- Functions have parameters, but they also have local variables

Part of the code unit is accessible (public members, arguments), part is hidden (protected/private, local variables)
I’m writing a class, but I don’t like to spend time debugging and dealing with compiler errors. Why not make everything public?

```cpp
class GradeList {
  public:
    void add(double grade);
    double percentile(double percentile);
    double mean();
    double median();
    std::vector<double> grades; // was private, NOW PUBLIC
    bool is_sorted; // was private, NOW PUBLIC
};
```
A prime duty of GradeList is to control when and how grades is sorted:

- If grades is sorted, it’s easy to answer percentile queries
- If it’s already sorted, it’s wasteful to try to sort it again

Once grades & is_sorted are public, user can modify them and there’s no guaranteed relationship between is_sorted & grades
Also, say we decide to switch from `vector<double>` to `multiset<double>`

- `multiset` can always be iterated over in sorted order
- ... so we can also get rid of `is_sorted`

Now user code that depended on `grades` being a `vector<double>` (or that depended on `is_sorted` existing) won’t compile
Encapsulation

An interface should be “as simple as possible, but no simpler”

- Have a few public members that handle everything users need
- Keep all details private
  - These can change later without breaking user code
  - class can maintain *invariants* over its private members
    (is_sorted is true if and only if grades is sorted) without fear that user will break them
Scott Meyers, a great C++ programmer & communicator, expresses prime design principle as: “Make interfaces easy to use correctly and hard to use incorrectly”

- bit.ly/meyers_iface
C++ design

For specific C++ design strategies:

[Image of Design Patterns book]

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch