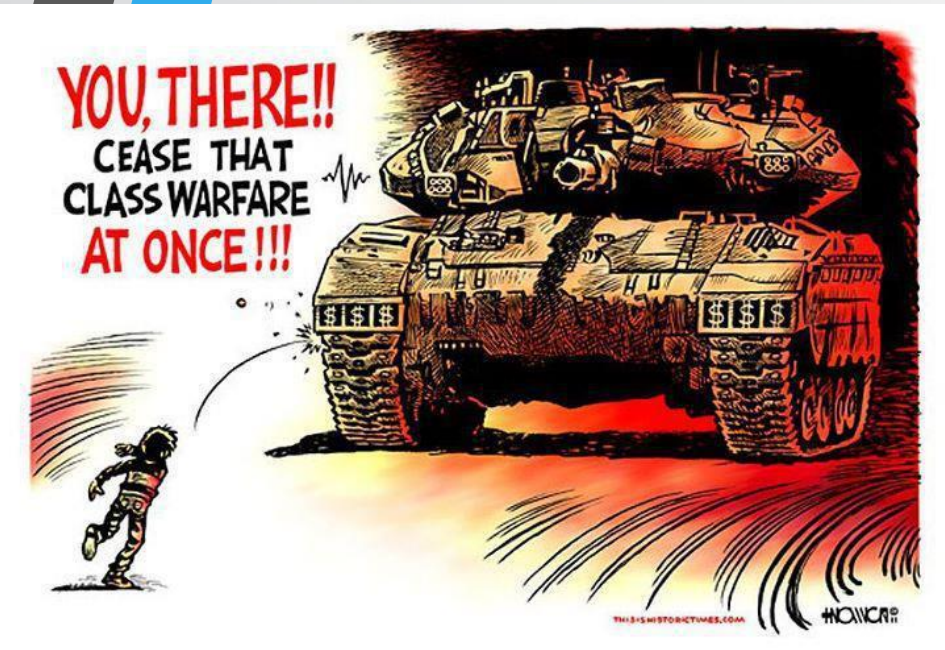

Advanced C++ Programming

Classes and Interfaces

Preliminaries

Overview & Goals

- As we established in the first lecture, all of you have used Java before
  - As such, I will not explain the basic concepts of object-oriented programming
  - However, I don't believe that all of you are good at basic interface design
- We will focus on the specifics of C++, but also general programming design rules which apply to other languages as well
- *Note*: "Interfaces" in the title doesn't just mean a special type of (base) class, it is concerned with how to design your functions and classes in general

Types of Interfaces

There are four main types of interfaces you can offer in C++:

- **Functions**
  - Which operate on some inputs and produce an output
- **Classes**
  - Which group operations and the data they operate on
- **Function Templates** and **Class Templates**, which will be the topic of a later lecture

Function Interface Design

Basics of good Interface Design in C++

- Interfaces should be
  - **Explicit**: avoid non-local or implicit state
  - **Precisely Typed**: more specific types at the interface level allow
    - Better *error-checking* at compile time
    - Better *optimization*
    - And they are more *self-documenting*
- Let's examine "02_01_basic_interfaces.cpp"

Function Size and Number of Parameters

- Individual functions should only perform **a single task**
- They should also be small
  - A good general rule of thumb: *if your function is larger than one screen, it is too large*
- Too many parameters, especially of the same type, usually indicate a design issue
- Some examples are shown in “02_02_function_size.cpp”

Parameter Passing and Return Values

- For multiple return values, use tuples or structs
  - I prefer structs, due to field names serving as documentation
- Selecting whether to use basic values, references, pointers or something else for parameters is important
  - There are some good basic guidelines
- We'll study these points in "02_03_parameters_and_retvals.cpp"

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

From the C++ Core Guidelines