



# 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 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)*

# Don't Return References to Local Variables

- It's the same as returning a pointer to a local – it will go out of scope \*
- Instead:
  1. Prefer simply returning values (copy elision should work in most cases)
  2. In situations where that is not an option, use `unique_ptr` or `shared_ptr`
- Let's look at `"02_04_return_smart_pointers.cpp"`

\* Exception: Function local statics, will be discussed in the next chapter

# Smart Pointers

- Defined in the standard library `<memory>`
  - <http://en.cppreference.com/w/cpp/header/memory>
- 2 main types:
  - **unique\_ptr** – single owner, ownership can be transferred
  - **shared\_ptr** – potentially shared ownership
- Should *only* be used to model **ownership**
  - If you don't need to transfer ownership, use references or plain pointers



# Class Design and Class Hierarchies

# “class” vs “struct”

- Functionally the same, except that “struct” members are public by default
- Useful convention:
  - Use `class` if the class has an invariant
  - Use `struct` if the data members can vary independently
- In the examples not related to class design  
I mostly use structs, but that is only for brevity of the demonstration
- Note that “class” and “struct” can result in different mangled identifiers in the object code, so you need to ensure that you use them consistently!

A condition on the state of the class which needs to be established in order for the public member functions to execute correctly

# Class Hierarchies

- We'll investigate the example in "02\_05\_class\_hierarchies.cpp"
- Of note: `virtual`, `const` and pure virtual (`= 0`) qualifiers
- Implementation inheritance is possible
- Use the `override` keyword!

# Destructors in Class Hierarchies

- Base class destructors should be either
  - **Public and virtual**; *or*
  - **Protected and non-virtual**

*If the destructor is public and non-virtual, **undefined behaviour** will occur when an object of a derived type is destroyed through a pointer to its base type.*

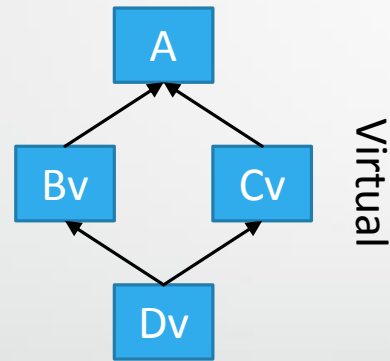
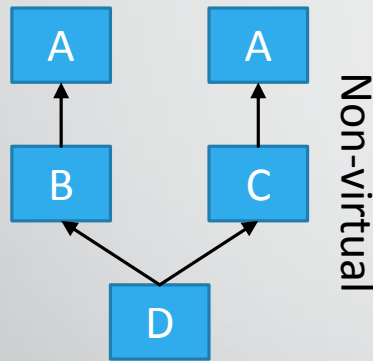
# Multiple Inheritance

- Let's start by looking at “02\_06\_multiple\_inheritance”
- C++ allows **multiple implementation inheritance**
  - Note that this doesn't mean you should use it all the time  
→ In many cases composition is a better choice
  - But it can be very useful for mixins, and reducing the amount of boilerplate code



# Diamond Inheritance Issue & Virtual Inheritance

- Consider “02\_07\_virtual\_inheritance.cpp”



- No duplication of data
- No ambiguity in upcasts
- Some overhead for virtual dispatch

- Often, linearizing your class hierarchy is a better choice
- But sometimes virtual inheritance allows for the cleanest solution

# Private Inheritance & Visibility in General

- Private inheritance is rarely useful – primarily if you want to provide *only part of an interface*
  - See “02\_08\_private\_inheritance.cpp”
- General visibility:
  - Use “public” for the interface, and “private” for implementation details
  - Don’t use “protected” data

# Operator Overloading

- Operator overloading is demonstrated in “02\_09\_operator\_overloading.cpp”
- Points to keep in mind:
  - Primarily use operator overloading to implement **conventional / expected usage**
  - Operators should be defined in the same namespace as the classes they operate on
    - *Note: namespaces and argument-dependent lookup*

# When to use Member Functions

1. If the function requires access to the internal state of a class
2. If it is a *virtual* function
3. If it is an operator that is required to be a member (“=”, “()”, “[ ]”, “->”)

In all other cases, a free-standing function should be preferred.

# Friends

- Friend declarations allow other functions/classes to access private data
- Let's look at an example in "02\_10\_friends.cpp"
- Friend declarations should **only** be used in exceptional situations
  - They introduce *tight coupling*, which you generally want to avoid
- Might be indicative of a design flaw
- "Long-distance" friendships are worse

[https://en.wikipedia.org/wiki/Coupling\\_\(computer\\_programming\)](https://en.wikipedia.org/wiki/Coupling_(computer_programming))



Conclusion

# Additional Resources

- Cpp Core Guidelines:
  - [I: Interfaces](#)
  - [F: Functions](#)
  - [C: Classes and class hierarchies](#)
- You *really* should read these, and refer to them when making design decisions in the future

# Summary

- Interface design is an essential programming skill
  - Functions:
    - Keep them tightly **focused**, **explicit**, and **precisely typed**
    - Know how to pass values in and out of functions in different circumstances
  - Classes:
    - Use “class” and “struct” as per common convention
    - You have access to virtual dispatch, multiple inheritance, virtual inheritance, operator overloading and friend declarations
- *But this doesn't mean that you always have to use all of them!*