Session 1: Advanced C++

Outline

- Namespaces
- Structs, classes, inheritance, virtual functions
- Header guards
- Templates
- Smart pointers
- Lambdas

Namespaces help to avoid name clashes

- STL is huge… how do we avoid giving a variable a name that already exists?
- Prepending a desired library object with namespace name followed by :: gives us access to that object
- Namespaces can be nested
  - Access like `outer::inner::object`
- In a local scope, can do `using namespace` to give access to objects in namespace without having to write the ::
  - Never do this in headers – every file that `#includes` it will be using the namespace!

```cpp
#include <vector>

// OK even though we can create a STL vector!
double vector = 1.0;

// Namespace object access.
std::vector<double> array;

// Example on how to make your own namespace.
namespace example {
    int example_integer;

    namespace inner_example {
        int inner_example_integer;
    }
}

// Accessing objects in nested namespaces.
example::inner_example::inner_example_integer = 4;

{
    using namespace example;
    // Now we can access anything in example namespace.
    example_integer = 3;
}
```

Structs are homemade composite datatypes

- A struct can contain an agglomeration of however many datatypes you want
  - Including other structs!
- After you define a struct, you can use it as though it were any other type
- When you create a struct, initialize all its **members** in curly braces
- Members can be accessed by a dot

`MyStruct` is a struct containing an integer and a double. Here I initialize an example called `important_data` with a 1 and a 2.0, respectively, then change the 1 to a 2

```
struct MyStruct {

int foo;
double bar;

};

MyStruct important_data{1, 2.0};

important_data.foo = 2;
```

Structs can also define member *functions*, or **methods**

- Member variables can be accessed inside methods without the dot
- Struct definitions are usually in headers, but we can still separate declarations from definitions
  - Note the use of namespace in the method definition
  - But inside the definition, we're effectively in the namespace of the struct, so we still have access to member variables

Here I just put the method definition in the struct definition

```
struct MyStruct {

int foo;

int doubleFoo() {
    int two_times_foo = 2*foo;
    return two_times_foo;
}

};
```

Here I separate the method declaration from the definition, so that the definition can be placed in a source file to reduce compile times

```
// Struct definition, in header
You, 1 second ago | 1 author (You)
struct SecondStruct {

int bar;

int doubleBar();
};

// Method definition, in source file
int SecondStruct::doubleBar() {
    int two_times_bar = 2*bar;
    return two_times_bar;
}
```

Classes are structs where we can restrict member access

- We can access and modify struct members from anywhere – dangerous!
- May want it to have member variables that only *it* can modify
- Three **access specifiers**:
  - `public:` Anyone can access
  - `private:` Only the class itself can access, i.e. inside methods
  - `protected:` Only the class itself and derived classes can access
    - We'll go over derived classes in a few slides

```
class Square {
// Everything underneath an access specifier
// has that access level.
public:
    double side_length;

private:
    double area_;

// Note that we can repeat access specifiers
private:
    void computeArea_() {
        area_ = side_length * side_length;
    }

public:
    double getArea() {
        computeArea_();
        return area_;
    }
};
```

What if we want outsiders to *read* a member variable, but not be able to *write* it?

Two options:

1. Declare the variable `public`, but `const`
    - This means that everyone can read it, but *nobody* can write it, not even the class itself
    - Can be useful if the member is set once and never needs to be touched afterward
2. Declare the variable `private/protected`and add a `public` **getter**
    - Everyone now has read access, but only the class itself can write it
    - Typically the best option

```
class ReadOnlyAccess {
public:
    const double option_1;

private:
    double option_2_;

public:
    double getOption2() {
        return option_2_;
    }
};
```

Classes can have multiple **constructors** (ctors)

- Ctors are functions that run when a class object is created
- Definition doesn’t return anything and has same name as class
- The body can be preceded by an **initializer list** that immediately initializes member variables before the body is run
    - Initializer list goes with method definition, not declaration, if separated
- The **default constructor** – ctor with no input – is automatically generated by the compiler if no default ctor is provided
- This is an example of **function overloading** – the compiler can distinguish between two functions with the same name by the differences in inputs

```
class Rectangle {
private:
    double length_;
    double width_;
    double area_;

public:
    Rectangle(double length, double width) {
        length_ = length;
        width_ = width;
        area_ = length * width;
    }

    Rectangle(double length) :
        // Initializer list to set some variables immediately.
        length_(length),
        width_(length)
    {
        area_ = length * length;
    }
};

// Constructs a 1x2 rectangle.
Rectangle use_first_ctor(1.0, 2.0);

// Constructs a 3x3 rectangle.
Rectangle use_second_ctor(3.0);
```

Classes can have children!

- …or at least inherit from other classes
- Inheritance also comes with an access specifier
  - `public`: All members of **base class** retain their access level. Usually what you want.
  - `protected`: Public and protected members of base class become protected
  - `private`: Public and protected members of base class become private
  - Recall that private members of base class are inaccessible by derived class!
- When a derived class is constructed, the base class is also constructed
  - Helpful to think of the derived class as actually storing the the base class as a secret member variable, under the hood

```cpp
class Rectangle {
private:
    double length_;
    double width_;

protected:
    double area_;

public:
    Rectangle(double length, double width) :
        length_(length),
        width_(width)
    {
        area_ = length*width;
    }
};

// Use public access specifier
class Square : public Rectangle {
public:
    Square(double length) :
        // It's a good idea to initialize the base class
        // ctor in the initializer list.
        // Otherwise, the default ctor will be used.
        Rectangle(length, length)
    {}

    // We can't access Rectangle's length_ or width_,
    // but we can access area_.
    double getArea() {
        return area_;
    }
};
```

Classes can inherit from more than one base class

- Derived class has access to public+protected members of *all* base classes
- Constructors should be called in the order that the classes are inherited

The `Mammal` class has a sound and the `Pet` class has a name. The `Dog` class inherits from both, so it has both a sound and a name. Its ctor initializes a Mammal with the “bark” sound and a Pet with the input name.

```
class Mammal {
protected:
    std::string sound_;

public:
    Mammal(std::string sound) : sound_(sound) {}
};

class Pet {
protected:
    std::string name_;

public:
    Pet(std::string name) : name_(name) {}
};

class Dog : public Mammal, public Pet {
public:
    Dog(std::string name) :
        Mammal("bark"),
        Pet(name)
    {}

    void speak() {
        std::cout << sound_ << std::endl;
    }
};

Dog my_dog("Fido");
// This will print "bark" to the console.
my_dog.speak();
```

Virtual functions enable considerable code modularity

- Example: given a list of shapes (triangles, rectangles, etc.) and their side lengths, what is the sum of their areas?
- We could check what kind of shape each object is and compute the area inside a for loop
  - But if we do this multiple times throughout a program, will need to copy this code everywhere
  - And if we add a new type of shape (trapezoid), must modify every single place
- What if we offload the responsibility for computing area to the shape itself?
- Key enabling feature: pointers to derived classes can be **cast** to pointers of base class
  - I.e., if I have a reference to a derived class, I can treat it as though it were a reference to a base class
  - This works because derived classes have all the same members as the base class

Virtual functions in derived classes override functionality of base class

- This effectively offloads class-specific work to the class itself
- Think about where this could be used in a PIC code
  - Particle boundaries: pass particle information to a boundary class and let it decide whether to absorb, reflect, etc.
  - Collision cross sections: give energy to cross section class and let it decide whether to use semi-empirical formulas, tables, etc.
- hPIC2 uses this just about everywhere. Get used to virtual functions!

```cpp
class Shape {
public:
    // The =0 means that this base class is "abstract";
    // you can't have just a Shape,
    // it must be one of the derived classes.
    virtual double getArea() = 0;
};

class Triangle : public Shape {
private:
    double base_, height_;

public:
    Triangle(double base, double height) : base_(base), height_(height) {}

    // Overrides definition in base class.
    double getArea() {
        return 0.5 * base_ * height_;
    }
};

int main() {
    Triangle triangle(2.0, 5.0);

    // We can store a pointer to the triangle
    // as a pointer to a Shape.
    Shape *shape = &triangle;

    // This will call the Triangle
    // version of getArea,
    // thereby returning 5 in this case!
    double area = shape->getArea();
}
```

New preprocessor directives: `ifdef, ifndef, endif`

- Recall that preprocessor directives start with # and tell the preprocessor to do something to the file before compilation
- Code can be optionally included or excluded based on whether a macro exists
  - `ifdef` enables code if macro is defined
  - `ifndef` enables code if macro is NOT defined
  - `endif` ends the block started by one of above
- Used to enable addons
  - E.g., MFEM code in hPIC2 is not compiled unless the user enables MFEM when compiling hPIC2

```
int main() {
    double something = 2.0;

#define FOO
#ifndef FOO
    // All this code is excluded
    std::cout << "FOO is defined!" << std::endl;
#endif

    return 0;
}
```

Circular inclusion can be a problem in big programs

Consider the following case

TypeA.hpp contents

```
#include "TypeB.hpp"

struct TypeA;

void foo(TypeB example);
```

TypeB.hpp contents

```
#include "TypeA.hpp"

struct TypeB;

void bar(TypeA example);
```

Each header declares a function that uses a struct defined in the *other* header

The preprocessor will keep trying to include each one in the other ad infinitum

Does this mean we have to be careful about ensuring inclusions are strictly hierarchical? ***No!***

Header guards prevent circular inclusion

TypeA.hpp contents

```
#ifndef TYPEA_HPP
#define TYPEA_HPP
#include "TypeB.hpp"

struct TypeA;

void foo(TypeB example);
#endif // TYPEA_HPP
```

TypeB.hpp contents

```
#ifndef TYPEB_HPP
#define TYPEB_HPP
#include "TypeA.hpp"

struct TypeB;

void bar(TypeA example);
#endif // TYPEB_HPP
```

- Take a second to realize how this works
  - When the preprocessor reads TypeA.hpp, it'll note that the `TYPEA_HPP` macro is undefined, so it'll define it and then try to `#include "TypeB.hpp"`
  - As it reads TypeB.hpp, it'll note that the `TYPEB_HPP` macro is undefined, so it'll define it and then try to `#include "TypeA.hpp"`
  - Now as it reads TypeA.hpp again, it'll note that `TYPEA_HPP` *is* defined, so it won't include any code and it'll return to TypeB.hpp again
  - The rest of TypeB.hpp is included as intended!

Header guards are used in every large project

- Every project standardizes guard macro names to prevent name clashes
- Typically, macro name is the filename in all caps with dots replaced by underscores
  - E.g., macro name for header guard of Types.hpp is `TYPES_HPP`
- The “pragma once” does the same thing and most compilers support it, but it’s technically not standard-compliant

```
#pragma once
#include "TypeA.hpp"

struct TypeB;

void bar(TypeA example);
```

Problem: writing type-agnostic function that squares input

- Function should work no matter what the type of the input is
- Naively, we should have a version of the function for every possible input type
- But if we make changes to one, we'll have to change *all* of them
  - Tedious, error-prone, no bueno!
- Can we automate this?

```
int squareInput(int x) {
    return x*x;
}

float squareInput(float x) {
    return x*x;
}

double squareInput(double x) {
    return x*x;
}

// ...
```

Yes, with **templates**!

Templates are like recipes for functions

- Or you can think of them as offloading some type deduction to the compiler
- To declare a template, prepend function or class definition with “template” followed by template arguments in <>
- To use, you can simply call the function with arguments in <>
  - In many cases you can even drop the <>
- Templates are particularly useful for making type-agnostic library code
- Template arguments are usually types (so that the template functionality can change depending on the type) but can also be integers, booleans, and more

```cpp
#include <vector>

template <typename T>
T squareInput(T x) {
    return x*x;
}

int main() {
    double foo = squareInput<double>(4.0);
    int bar = squareInput<int>(4);

    // Compiler can sometimes automatically deduce
    // template arguments based on input type.
    foo = squareInput(foo);

    // Now you know why std::vector uses <>:
    // the template argument is the contained datatype!
    std::vector<float> array_of_floats;
}
```

So much more to templates

- Classes can also be templates!
- Templates can take a variable number of arguments (**variadic templates**)!
- Templates can be used to make decisions about behavior at compile-time!
- Templates can do different things based on compilation options!
- We can constrain template behavior based on features of template arguments (**concepts**)!

Smart pointers (`shared_ptr`) provide the flexibility of pointers with (most of) the safety of references

- Smart pointers ensure that the pointed object is deleted when all smart pointers referring to it have gone out of scope
  - This obviates the need to manually deallocate memory
- Still possible for smart pointers to be invalid, but very easy to check
- Dereferencing smart pointers works exactly the same as regular pointers
- Unlike references, can reassign smart pointers

```cpp
void squareInput(std::shared_ptr<float> x) {
    *x = (*x) * (*x);
}

int main() {
    {
        // Creating a smart pointer is verbose,
        // but worth it for the benefits!
        std::shared_ptr<float> x = std::make_shared<float>(2.0);
        squareInput(x);

        // Can check whether a smart pointer is valid
        // by simply treating it like a bool.
        if (not x) {
            std::cout << "Invalid smart pointer!" << std::endl;
        }
    }
    // The memory allocated for x has been deleted since all
    // smart pointers are out of scope.
    // No need to manually delete!

    return 0;
}
```

Smart pointers combine with virtual functions to enable clean modular code

- You can dereference a pointer to a class and access members in one fell swoop with a ->

```
class Shape {
public:
    virtual double getArea() = 0;
};

class Square : public Shape {
private:
    double length_;

public:
    Square(double length) : length_(length) {}

    double getArea() {
        return length_*length_;
    }
};

class Cube : public Shape {
private:
    double length_;

public:
    Cube(double length) : length_(length) {}

    double getArea() {
        return length_*length_*length_;
    }
};

int main() {
    std::vector<std::shared_ptr<Shape>> shapes;
    // Append 3 objects to shapes.
    shapes.push_back(std::make_shared<Cube>(1.0));
    shapes.push_back(std::make_shared<Square>(2.0));
    shapes.push_back(std::make_shared<Cube>(3.0));

    for (int i=0; i<shapes.size(); i++) {
        std::cout << shapes[i]->getArea() << std::endl;
    }

    return 0;
}
```

Lambdas are anonymous functions

- Unlike other functions, we can define them inside other functions and even store them like variables
- Lambdas are great for the lazy – you can use variables defined in that scope as though they were inputs
  - This is called **capture**
- Variables can be captured **by value** or **by reference**
  - If by value, variables are copied to a lambda-local scope
  - If by reference, variables can be modified in place
- Utility will be more obvious in Kokkos…

```cpp
int main() {
    float r = 0.1;
    float partial_sum = 0.0;

    auto for_each = [=, &partial_sum] (int i) -> float {
        float r_to_the_i = 1.0;
        for (int j=0; j<i; j++) {
            r_to_the_i *= r;
        }
        partial_sum += r_to_the_i;

        return r_to_the_i;
    };

    // Do partial sum up to 20.
    int limit = 20;
    for (int i=0; i<limit; i++) {
        std::cout << "iteration " << i
            << " adding " << for_each(i)
            << std::endl;
    }
    // partial_sum now contains that partial sum

    std::cout << partial_sum << std::endl;

    return 0;
}
```

Anatomy of a lambda definition

Let the compiler deduce the type

Store lambda in this variable

By default, capture variables by value, but capture `partial_sum` by reference

Integer input

Return a float

```
auto for_each = [=, &partial_sum] (int i) -> float {
```

Exercise

1. Write an abstract Shape base class with a pure virtual getArea function that takes no inputs and returns a double.
2. Write Triangle, Rectangle, and Circle classes that inherit from Shape and override getArea with the correct area formulas.
    a. The ctor for Triangle should accept a base and a height, which should be stored.
    b. The ctor for Rectangle should accept a length and a width.
    c. The ctor for Circle should accept a radius.
    d. The ctors may either precompute the area and store it, or save just the lengths and compute the area when getArea is called.
3. Using what you learned from Session 0, write a main function that converts command line input into a std::vector<std::string>.
4. Inside the main function, create an empty std::vector<std::shared_ptr<Shape>>.
5. Loop through the command line input. If the input is “triangle,” use std::make_shared to make a triangle with unit base and height, and similarly for the other shapes. Append them to your array of shapes.
6. Compute the sum of the areas of your array of shapes and print it to console.