

# C/C++ PRIMER

## LECTURE 8: ABSTRACT BASE CLASSES, PURE `virtual` METHODS, OPERATOR OVERRIDING

*Fabian Wermelinger*  
Harvard University

# OUTLINE

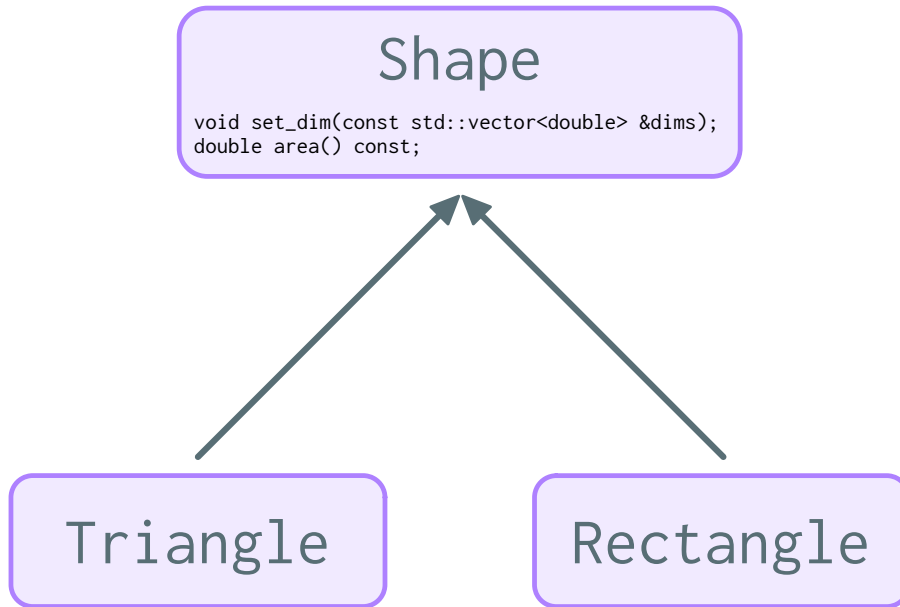
- `virtual` methods (continued)
- Abstract base classes and pure `virtual` methods
- Operator overriding
- C++11 extension modules for python

# virtual METHODS

- virtual methods define an *interface* that is common to all subclasses.
- The base class defines the interface.
- Runtime polymorphism allows to implement different behavior depending on the class instance.
- Powerful OOP technique for the development of advanced programs and libraries.
- Runtime polymorphism is not for free. Performance critical code should not suffer from overhead due to resolution of runtime polymorphism.

# virtual METHODS

*Consider this design for shapes:*



```
1 #include <cassert>
2 #include <vector>
3
4 class Shape
5 {
6 public:
7     void set_dim(const std::vector<double> &dims) { dims_ = d
8         virtual double area() const { return 0.0; }
9
10 protected:
11     std::vector<double> dims_;
12 };
13
14 class Triangle : public Shape
15 {
16 public:
17     double area() const override
18     {
19         assert(2 == dims_.size());
20         // assume the following:
21         const double base = dims_[0];
22         const double height = dims_[1];
23         return 0.5 * base * height;
24     }
25 };
26
27 class Rectangle : public Shape
28 {
29 public:
30     double area() const override
31     {
32         assert(2 == dims_.size());
33         return dims_[0] * dims_[1];
34     }
35 };
```

# virtual METHODS

*Consider this design for shapes:*

Example use case:

```
1 int main(void)
2 {
3     Shape *s;
4     if (user_request == "Triangle") {
5         s = new Triangle;
6     } else if (user_request == "Rectangle") {
7         s = new Rectangle;
8     }
9
10    s->set_dim({1, 2});
11    std::cout << s->area() << std::endl;
12    delete s;
13    return 0;
14 }
```

Assume you implement a new shape:

```
1 class Circle : public Shape
2 {
3 };
```

- You forgot to implement the `area()` method. (Here there is only one method and it is probably hard to forget about it. In reality there will be many virtual methods.)
- Will the code for `Circle` compile?
- We have assumed a default behavior for `area()` in the `Shape` base class:

```
1 virtual double area() const { return 0.0; }
```

Is this a good idea?

# ABSTRACT BASE CLASSES

An *abstract base class* is a type that *can not be instantiated* but can be used as a base class.

- An abstract base class declares member functions that are *pure virtual*.
- A pure virtual member takes the form

```
1 virtual return_type method_name(argument_list) [virt_specifier] = 0;
```

The [virt\_specifier] is optional and can either be `override` or `final`.

- Pure virtual members are used to define the interface in the base class and enforce implementation in derived classes.

# ABSTRACT BASE CLASSES

*A better design for shapes:*

```
1 #include <vector>
2
3 class Shape
4 {
5 public:
6     void set_dim(const std::vector<double> &dims) { dims_ = dims; }
7     virtual double area() const = 0; // pure virtual member function
8
9 protected:
10     std::vector<double> dims_;
11 };
12
13 class Circle : public Shape
14 {
15     // this will not work because Circle is still abstract
16 };
17
18 int main(void)
19 {
20     Shape *s = new Circle;
21     delete s;
22     return 0;
23 }
```

Creating just a pointer in line 20 would work alright, calling the new operator with a Circle argument fails.

- Because the `area()` computation depends on a geometry, it is more precise to leave it abstract in the base class `Shape`.
- The code on the left will not compile because you can not create an instance of an abstract base class.
- Creating pointers of abstract class types is legal because this does not create a new instance.

# ABSTRACT BASE CLASSES

*Virtual member functions are also called when invoked through regular member functions:*

```
1 #include <iostream>
2
3 class BaseSimulation
4 {
5 public:
6     void run(const int steps)
7     {
8         for (int i = 0; i < steps; ++i) {
9             step_(i);
10        }
11    }
12
13 protected:
14     virtual void step_(const int i)
15     {
16         std::cout << "Base simulation: step ";
17         std::cout << i << std::endl;
18     }
19 };
20
21 class DerivedSimulation : public BaseSimulation
22 {
23 protected:
24     void step_(const int i) override
25     {
26         std::cout << "Derived simulation: step ";
27         std::cout << i << std::endl;
28     }
29 };
```

```
1 int main(void)
2 {
3     DerivedSimulation sim;
4     sim.run(10);
5     return 0;
6 }
```

- You can use a wrapper method like `void run(const int steps)` that implements a basic algorithmic framework.
- If the `step_` method was not declared virtual on the left, what would be the output if the code is run by a `DerivedSimulation` instance like in the main function above?



# OPERATOR OVERRIDING

- We have seen that in runtime polymorphism we *override* operators as opposed to operator overloading which applies for different function signatures in the same scope.
- We can also specify a method as **final**. A final method can no longer be overridden in derived classes.
- We can use the final keyword in the same way for inheritance. We can no longer inherit from a class that is **final**.

```
1 class Base
2 {
3     virtual void f_() {}
4 };
5
6 class Derived1 : public Base
7 {
8     void f_() final {}
9 };
10
11 class Derived2 : public Derived1
12 {
13     // error: f() is final in Derived1
14     void f_() override {}
15 };
```

```
1 class Base
2 {
3     virtual void f_() {}
4 };
5
6 class Derived1 final : public Base
7 {
8     void f_() {}
9 };
10
11 // error: Derived1 is final
12 class Derived2 : public Derived1
13 {
14     void f_() {}
15 };
```

# C++11 EXTENSION MODULES FOR python

Here we are going to discuss the creation of C++11 extension modules for python code. Extension modules are similar to pure python modules except that the underlying module code is implemented in C/C++. This is especially useful for performance critical code in a python library. We are going to use the `pybind11` header-only C++ library for this purpose here.

Our goal is to write a simple function called `add` that is part of an extension module called `example` which is in turn part of a package called `my_pybind11`. The `add` function adds together two integers according to:

```
1 int add(int i, int j) { return i + j; }
```

This example follows the tutorial in [the first steps of the pybind11 documentation](#).

# C++11 EXTENSION MODULES FOR python

- We would like to be able to write the following python code:

```
1 # import the module from our my_pybind11 package
2 from my_pybind11 import example
3
4 # add one and one together
5 two = example.add(1, 1)
6 print(two) # print: 2
```

- We can easily accomplish this with pure python code if we write a module `example.py` with the following content:

```
1 """example module in pure python"""
2
3 def add(i, j):
4     return i + j
```

- We are interested in a C++ implementation of the function `add` instead.

# C++11 EXTENSION MODULES FOR python

```
1 // This pybind11 header is needed for the binding code below
2 #include <pybind11/pybind11.h>
3
4 // Here we implement the add function. This is example is very trivial.
5 int add(int i, int j) { return i + j; }
6
7 // The following code is the python binding code that will create a module with
8 // the name 'example'. What we define inside the PYBIND11_MODULE macro below is
9 // similar to the code we have seen for the pure python module before.
10 //
11 // For this example we have the implementation of add and the binding code
12 // together in the same file. Usually implementation and the binding code are
13 // in separate files.
14 PYBIND11_MODULE(example, m)
15 {
16     // Optional module doc string
17     m.doc() = "pybind11 example extension module";
18
19     // Module function definition. Note that we pass a reference to the add
20     // function above.
21     m.def("add", &add, "A function which adds two numbers implemented in C++");
22 }
```

# C++11 EXTENSION MODULES FOR python

- A C++ extension module consists of:
  1. Code that implements your extension
  2. Binding code that generates a an extension module (shared library) that can be loaded in python code using the `import` statement.
- Our simple example code can be compiled with a single command line:

```
1 g++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11 --includes) \  
2     add.cpp -o example$(python3-config --extension-suffix)
```

This command is however already quite lengthy and involves some python dependencies. Automating the module build with a build system should be preferred here even for very small extension modules.

- We will use [meson](#) in the following as it provides very nice tools for building python extension modules. (**Recall:** we did an exercise with meson in the first class.)

# C++11 EXTENSION MODULES FOR python

## *Build system setup:*

- [meson](#) is a very powerful build system used to automatically compile and link code with a powerful dependency resolution.
- It comes with a [PEP517](#) plugin that makes development of python extension straight forward. The plugin is [documented here](#).
- All we need is a [pyproject.toml](#) file that defines the backend for pip and optionally other project related meta data.

# C++11 EXTENSION MODULES FOR python

## *Build system setup:*

- All we need is a `pyproject.toml` file that defines the backend for pip and optionally other project related meta data.

```
1 [build-system]
2 requires = ["mesonpep517"]
3 build-backend = "mesonpep517.buildapi"
4
5 [tool.mesonpep517.metadata]
6 author = "Fabian Wermelinger"
7 author-email = "fabianw@seas.harvard.edu"
8 classifiers = [
9     "Intended Audience :: Education",
10    "Intended Audience :: Science/Research",
11 ]
12 platforms = "py3"
13 requires-python = "py3"
14 summary = "Demo extension module for pybind11"
```

- This is already enough for our simple `example` extension module for python. Let's see how this all works with a demo.

# HANDS-ON: WRITE AN EXTENSION MODULE TO COMPUTE THE SUM OF A `numpy` ARRAY

In this hands-on we want to write a custom `sum` function that should behave similar to `np.sum` (<https://numpy.org/doc/stable/reference/generated/numpy.sum.html>) for a 1D array input (flat array). We want to benchmark the performance of our custom implementation with respect to the `numpy` version and a pure python implementation. Follow the further instructions in `hands-on/01/README.md`.

## Expected benchmark output:

```
1 $ python -m sum_bench
2 len(x) = 1000000
3 numpy      : result=499999500000.0  7.51018524e-04 seconds
4 pybind11:  result=499999500000.0  1.12009048e-03 seconds
5 pure       : result=499999500000.0  1.16991997e-01 seconds
```