C/C++ **PRIMER**

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

Fabian Wermelinger
Harvard University

OUTLINE

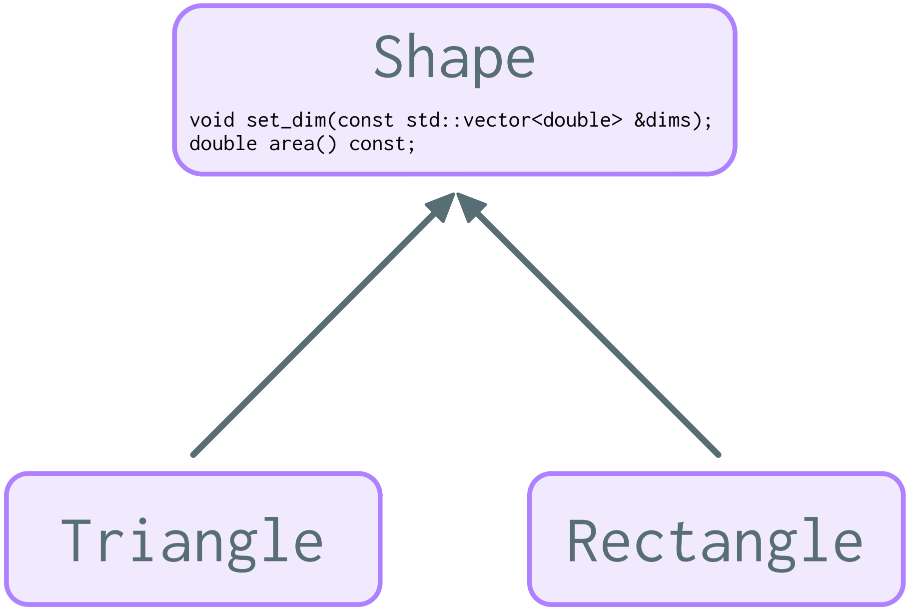
- `virtual` methods (continued)
- Abstract base classes and pure `virtual` methods
- Operator overriding
- `C++11` extension modules for `python`

`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 OPP 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:




```cpp
#include <cassert>
#include <vector>

class Shape
{
public:
    void set_dim(const std::vector<double> &dims) { dims_ = d
    virtual double area() const { return 0.0; }

protected:
    std::vector dims_;
};

class Triangle : public Shape
{
public:
    double area() const override
    {
        assert(2 == dims_.size());
        // assume the following:
        const double base = dims_[0];
        const double height = dims_[1];
        return 0.5 * base * height;
    }
};

class Rectangle : public Shape
{
public:
    double area() const override
    {
        assert(2 == dims_.size());
        return dims_[0] * dims_[1];
    }
};

```

`virtual` METHODS

Consider this design for shapes:

Example use case:

```
int main(void)
{
    Shape *s;
    if (user_request == "Triangle") {
        s = new Triangle;
    } else if (user_request == "Rectangle") {
        s = new Rectangle;
    }

    s->set_dim({1, 2});
    std::cout << s->area() << std::endl;
    delete s;
    return 0;
}
```

Assume you implement a new shape:

```
class Circle : public Shape
{
};
```

- 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:

```
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

```
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 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:

```cpp
#include <iostream>

class BaseSimulation
{
public:
    void run(const int steps)
    {
        for (int i = 0; i < steps; ++i) {
            step_(i);
        }
    }

protected:
    virtual void step_(const int i)
    {
        std::cout << "Base simulation: step ";
        std::cout << i << std::endl;
    }
};

class DerivedSimulation : public BaseSimulation
{
protected:
    void step_(const int i) override
    {
        std::cout << "Derived simulation: step ";
        std::cout << i << std::endl;
    }
};
```

```cpp
int main(void)
{
    DerivedSimulation sim;
    sim.run(10);
    return 0;
}
```

- 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 where the signature may change.
- 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`.

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++`. 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.

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

```
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:

```python
# import the module from our my_pybind11 package
from my_pybind11 import example

# add one and one together
two = example.add(1, 1)
print(two) # print: 2
```

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

```python
"""example module in pure python"""

def add(i, j):
    return i + j
```

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

C++11 EXTENSION MODULES FOR python

```cpp
// This pybind11 header is needed for the binding code below
#include <pybind11/pybind11.h>

// Here we implement the add function.  This is example is very trivial.
int add(int i, int j) { return i + j; }

// The following code is the python binding code that will create a module with
// the name 'example'. What we define inside the PYBIND11_MODULE macro below is
// similar to the code we have seen for the pure python module before.
//
// For this example we have the implementation of add and the binding code
// together in the same file.  Usually implementation and the binding code are
// in separate files.
PYBIND11_MODULE(example, m)
{
    // Optional module doc string
    m.doc() = "pybind11 example extension module";

    // Module function definition.  Note that we pass a reference to the add
    // function above.
    m.def("add", &add, "A function which adds two numbers implemented in C++");
}
```

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:

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

```
[build-system]
requires = ["mesonpep517"]
build-backend = "mesonpep517.buildapi"

[tool.mesonpep517.metadata]
author = "Fabian Wermelinger"
author-email = "fabianw@seas.harvard.edu"
classifiers = [
    "Intended Audience :: Education",
    "Intended Audience :: Science/Research",
]
platforms = "py3"
requires-python = "py3"
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:

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