Brown University ENGN2912B Fall 2017
Scientific Computing in C++
Lecture 7 | Introduction to Classes

The concept of a class, or object

The C++ language supports several mechanisms for representing an *object*: **`class`**, **`struct`**, and **`union`**. This introduction to object-oriented design will focus on the **`class`** representation of an object. The characteristics of the other two object representations will be put off until a later lecture.

Recall from Lecture 1 that an *object* consists of both data and defined operations on its data. Moreover an object should represent an element of representation and behavior that can be used to build up a larger, complex, system where each component has an easily identifiable role.

To make the discussion less abstract, consider the problem of taking the square root of real number. The full solution requires the complex number field, since a real number can be negative. C++ does have a templated class to represent complex numbers, but for illustrative purposes here, it will be assumed that it is necessary to implement a complex number class called **`MyComplex`**. The class has to be able to represent the real and imaginary part of a complex number and to carry out appropriate operations.

To get started, it is necessary to create two files that represent the definition and implementation of the class, `MyComplex.hpp` and `MyComplex.cpp`. These files are structured in the CMakeLists.txt file as follows.

```
SET(lecture7_files MyComplex.cpp MyComplex.hpp Lecture7.cpp)

ADD_EXECUTABLE( lecture7 ${lecture7_files})
```

The main program file is called `Lecture7.cpp`. The last line of the CMakeLists.txt file associates these files with the resulting executable program, `lecture7`. You may call the main program anything, as well as the name of the executable. However it is standard coding practice to have the .h file for class specification, and the .cpp file for class implementation code, to have the same name as the class. This practice allows the programmer to easily find the source code associated with a given class.

The `MyComplex.hpp` file, which defines the *interface* to the class, appears as follows.

The class definition

```
#ifndef _MyComplex_hpp_
#define _MyComplex_hpp_

//=============================================================
// The following class represents complex numbers with double |
// precision.                                                 |
//=============================================================

class MyComplex {
public:
  // Constructors of the class
  MyComplex();
  MyComplex(double real, double imag);

  // The destructor of the class
  ~MyComplex();

  //Class methods
  // Accessors
  double re()const;
  double im()const;

  //Addition operator
  MyComplex operator + (MyComplex const& c) const;

  // The class members
private:
  double re_;
  double im_;

}; //note that there must be a terminating ;

#endif // _MyComplex_hpp_
```

It will be instructive to consider each line of this definition of the class. The first two lines define the *guard* that prevents multiple inclusion of the file.

The next section is a comment block that tells the user of this function some information about the purpose of the class and optionally some instructions on using the operations of the class. Some programmers make their comment blocks very fancy. The block here is made to look like a block.

The line,

```
class MyComplex
```

declares that the following scope will define the members and operations of the class, **`MyComplex`**.

The next line inside the scope, **`{}`**, is the keyword,

```
public:
```

This statement declares that the following operations or member variables will be visible to users of the class, i.e. publicly accessible. The operations and members of a class are **`private`** by default. **`private`** operations or members are only visible inside the class implementation. Class implementation is either done directly in the .h file or in a namespace that specifies the class methods in the .cpp file. (see the next section)

Class Constructors

The next line

```
MyComplex();
```

is a *constructor* of the class. A constructor is a function that generates a new *instance* of the class. An instance is a realization of the class in memory with storage for its specific member values. A class can have many instances but the structure of the instance is the same each time. The only difference from one instance to the next is the value of the member variables. In this case the constructor is called a *default* constructor. That is because there are no arguments to the constructor function so the class member values must be defined by default. The implementation of this default constructor in the .cpp file is as follows.

```
#include "MyComplex.hpp"
MyComplex::MyComplex() {
  re_ = 0;
  im_ = 0;
}
```

Note that there is a namespace **`::`** operator associated with a class to indicate that the code is being defined for the **`MyComplex`** class constructor method, **`MyComplex()`**.

The most reasonable default behavior is to set both the real and imaginary part of the complex number to zero, as is done here. A common coding convention is to indicate class members by a trailing underscore, i.e., **`re_`** vs. **`re`**.

The next constructor takes two arguments that are the real and imaginary parts of the complex number. The implementation of this constructor in the .cpp file is as follows.

```
MyComplex::MyComplex(double real, double imag) {
  re_ = real;
  im_ = imag;
}
```

This constructor has more power in that the user can directly specify what the real and imaginary parts of the complex number are to be. Note that member variables of the class are accessible within a class method, since they have been declared in the class specification in the .h file. It can be seen that the constructor is very much like a function, but a function that calls an instance of the class into existence.

The preferred approach to initializing class member variables in a constructor is to use a new form of the `':'` operator. The previous constructor can be also written as follows.

```
MyComplex::
MyComplex(double real, double imag): re_(real), im_(imag){}
```

The member variables are assigned by name with the `()` operator containing the assigned value. Multiple members can be assigned through a comma separated list. Note that the body of the constructor, `{}`, has to be present, even though nothing else is needed to create the class instance.
Keep in mind that constructors obey all the usual characteristics of functions. For example it is possible to define a default constructor by providing all default arguments.

```
MyComplex(double real=0.0, double imag=0.0):
re_(real), im_(imag){}
```

This definition will act the same way as the default constructor example above. However, this design is not a good idea since calling this constructor with no arguments is indistinguishable from the default constructor. In fact the complier won't let you do it anyway:

```
c: MyComplex.h(26) : warning C4520: 'MyComplex' : multiple default
constructors specified
```

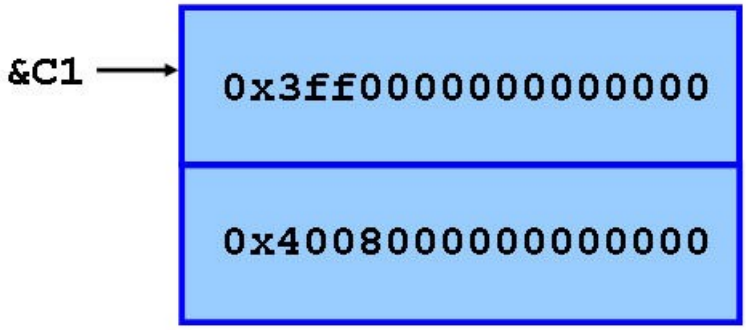
Memory allocation of a class instance

When the constructor is called, the class is allocated in memory, which requires that the double values, **`re_`** and **`im_`** be stored. For example if the class is constructed as follows.

```
MyComplex c1(1.0, 3.0);
unsigned n = sizeof(c1);// n == 16
```

The class constructor is called within a declaration or assignment statement. This statement declares **`c1`** to be of type **`MyComplex`**. Moreover the real and imaginary parts are 1.0 and 3.0 respectively. The size of a single **`MyComplex`** data type is 16 bytes, as expected for the storage of two **`double`** data types.

The layout of the class instance **`c1`** in memory is shown below. Note that there is no extra storage. A new data type has been defined that carries no overhead.



In this figure, the memory address of **`c1`** can be obtained by the use of the address operator **`&`**. In addition, C++ provides a special **`private`** class member called **`this`**, which is the address of the start of the class in memory. In the example **`&c1 == this.`**

Use of the default constructor

Also note that a declaration can use the default constructor for the class.

```
MyComplex c2;
```

In this case the default constructor is called and the real and imaginary parts of **`c2`** are both zero. It is not necessary to include the parentheses in a direct declaration. However if the default constructor is used in an assignment then the **`()`** must be included, i.e.,

```
MyComplex c_asign = MyComplex();
```

What constructor (if any) is called when an array is constructed in the stack?

```
MyComplex cArray[4];
```

Class destructor

The purpose of the class destructor

```
~MyComplex();
```

is to remove the class from memory when it is no longer needed. If the class is constructed on the stack then the class is removed after the scope where the class was constructed is exited. In the simple example here, both the class members, **`re_`** and **`im_`**, are constructed on the stack and are removed as soon as scope is exited so the destructor doesn't have to do anything specific. When a class creates storage on the heap, through **`new`**, it is usually necessary for the class to clean up the storage it created on the heap when the destructor executes. The destructor is called when the class is deleted from either the stack or heap.

The implementation of the destructor is,

```
// The destructor of the class, does nothing in this case
MyComplex:: ~MyComplex() {
}
```

On the other hand, suppose the class members are stored on the heap as follows:

```
// fragment of class MyComplex
…
private:
  double* re_;
  double* im_;
};
// the constructor implementation
MyComplex::
MyComplex(double real, double imag) {
  re_ = new double(real);
  im_ = new double(imag);
}
```

It is assumed that the rest of the class machinery is adjusted accordingly. For example, the destructor now has something to do:

```
MyComplex:: ~MyComplex() {
  delete re_;
  delete im_;
}
```

Note that users of **`my complex`** would not notice the internal redesign to pointers. The class interface "hides" the implementation. However, the discussion below will revert to the original class design using direct storage of **`re_`** and **`im_`**.

Member accessors

It is often necessary to get the values of the class members. The preferred way is to provide functions called accessors. These accessors isolate the member variables from direct manipulation by programmers.

```
// Accessors
  double re() const;
  double im() const;
```

The use of the **`const`** attribute indicates that accessing the variables through these functions does not change the state of the class. In this case the state is completely defined by the values of **`re_`** and **`im_`**. Each accessor function returns the appropriate class member as indicated by the function name. The implementation in the `.cpp` file is as follows.

```
double MyComplex::re()const {
      return re_;
 }

double MyComplex::im() const {
      return im_;
 }
```

Again, note that the compiler is told that these implementations belong to class methods of **`MyComplex`** through the use of the namespace operator, **`MyComplex::`**

A tip on style: some programmers like to name accessors as **`getRe()`** or **`getIm()`**, particularly when methods to set the class variables are necessary.

Accessing Class Elements

The next issue is how these accessor methods are called to return the member values. The following fragment indicates the syntax for calling, or *accessing*, a class method.

```
MyComplex c1(1.0, 3.0);
double vr = c1.re(), vi = c1.im();
```

Class access is indicated by the **`'.'`** operator, which accesses either a member or method of the class instance, **`c1`**. In the example, the class members are declared to be **`private`** so an attempt to access them such as,

```
double vr_ac = c1.re_, vi_ac = c1.im_;
```

will result in a complier error,

```
Error     1    error C2248: 'MyComplex::re_' : cannot
access private member declared in class 'MyComplex'
```

Another form of access is defined for a pointer to a class instance. For example,

```
MyComplex* cp = new MyComplex(1.0, 3.0);

double r = cp->re();//right arrow access to pointers, r == 1.0
```

In this example, a new symbol is introduced that serves the same role as **`'.'`**, but for pointers.

Access through `this`

For method implementations inside the class, the **`this`** pointer can be used to access class methods, e.g.,

```
double sum = (*this).re() + (*this).im();
```

The **`this`** pointer has to be dereferenced in order to use the **`'.'`** accessor operator. Because of this common form of access, C++ provides a convenient accessor operation that applies to a pointer to a class instance, the arrow operator as described above.

```
double sum = this->re() + this->im();
```

In this case, there is no need to dereference the pointer. The same format applies to member access,

```
double sum = this->re_ + this->im_;
```

Of course, since the **this** pointer is **private**, the form of access is only available inside class methods. One might ask, why use **this** access when the member variables and methods are directly accessible inside class methods anyway? The reason is the explicit style of coding makes clear that the methods and members are of "this" class. There could be other functions with similar names and this style makes it clear that the programmer wants to use the members and methods of the class being implemented. A specific case of this problem will be seen once more complex forms of class inheritance are taken up in the next lecture.

Operator Methods

It is also possible to define operators that apply to the data type defined by the class. In the example here of complex numbers, it is natural to define all the arithmetic operations that apply to the complex number field. It will suffice to demonstrate the process for addition. The specification of other operators is similar.

```
MyComplex operator + (MyComplex const& c) const;
```

This syntax indicates that there are two operands: the class instance itself; and another instance called **c**, which is supplied in the argument of the operator. Note that this argument is supplied as a **const** reference to insure that the operator cannot alter any members of **c**. The operator itself is **const**, which insures that the operation cannot alter the first operand class instance.

The symbol for the operator is defined to be **+** , which is chosen because it is universally associated with addition.

The operator returns a value of type **MyComplex** as one would expect. The

implementation of the operator in the .cpp file is as follows.

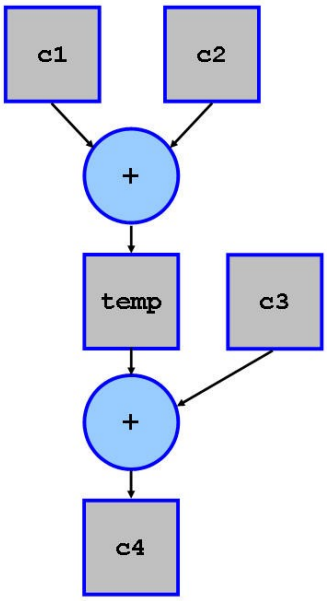
```
MyComplex MyComplex::operator + (MyComplex const& c) const {
  MyComplex cp(this->re_+c.re(), this->im_+c.im());
  return cp;
}
```

The implementation starts by constructing an instance of **MyComplex**, **cp**.
The constructor for **cp** is supplied the two arguments for the real and imaginary parts as sums of **this** class instance and the input class, **c**. Once the instance **cp** is constructed it can be returned, just as any other data type. Indeed the return type of the operator **+** is **MyComplex**. The operator is applied as shown in the following example.

```
MyComplex c1(1.0, 3.0);
MyComplex c2(2.0, 1.3);
MyComplex c3 = c1+c2;// c3.re()==3.0, c3.im() == 4.3
```

The operator is being accessed from instance **c1** and **c2** is supplied as the operator argument **c**. More precisely, the "**this**" in the implementation of the operator refers to class **c1** and **c** refers to class instance **c2**. Note that the operator can be applied as many times in a statement as desired. For example,

```
MyComplex c4 = c1+c2+c3;// c4.re()==6.0,    c4.im() == 8.6
```



In this example, the sum of **c1** and **c2** is performed first, resulting in a temporary instance of **MyComplex**, the temporary then applies its **+** operator to **c3**. This process is illustrated in the figure above. The temporary instance, **temp**, can be viewed as living on the stack only within the scope of the statement that computes **c4**, and is not visible.

The full set of C++ operator symbols that can be reimplemented ("overloaded") for new classes or for special behavior are shown below.

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

From http://www.cplusplus.com/doc/tutorial/classes2.html

Global functions

There are many other functions that one can think of that will be needed to process complex numbers. Indeed, the operation, **sqrt**, that caused trouble in the first place needs to be defined so that it can produce an imaginary number when the argument is a

negative real number. However, if one is going to the trouble of implementing a complex form of **sqrt** it might as well be defined for the entire complex number field.

It is a general design principle that the interface to a class be kept as simple and clean as possible. It is reasonable to define primitive operators, such as **+** and ***** on the class itself, but auxiliary functions should be declared in the `.h` file as global functions. These functions can be implemented in the same class `.cpp` file, but they are not member functions (methods) of the class. The following shows the top part of the class `.h` file where the declaration of **csqrt** is made.

```
#ifndef _MyComplex_hpp_
#define _MyComplex_hpp_

//Forward declare that the class MyComplex will exist somewhere
class MyComplex;

// compute the square root of a complex number
MyComplex csqrt(MyComplex const& c);

class MyComplex
{
public:
  // Constructors of the class
...
```

Note that it is necessary to declare that the class **MyComplex** will be defined at some point so that its data type can be used in the specification of the signature of the function **csqrt**, before **MyComplex** has been defined. Note also that the argument to **csqrt** is declared as a **const** reference so that the value will not be copied when passed to the function.

The implementation of **csqrt** will be typically carried out in the `MyComplex.cpp` file, since it is a companion function to the class itself. One implementation is as follows.

```
#include <math.h>
#include "MyComplex.hpp"
// compute the square root of a complex number
MyComplex csqrt(MyComplex const& c)
{
  double mag_sq = c.re()*c.re() + c.im()*c.im();
  double mag = sqrt(mag_sq);
  double arg = atan2(c.im(), c.re());
  double mag_sqrt = sqrt(mag);
  double arg_half = arg/2;
  return MyComplex(mag_sqrt*cos(arg_half),
                   mag_sqrt*sin(arg_half));
}
```

An example of calling this function is,

```
MyComplex sq = csqrt(MyComplex(-1,0));
// sq.re() == 6.1232339957367660e-017
```

```
// sq.im() == 1.0
```

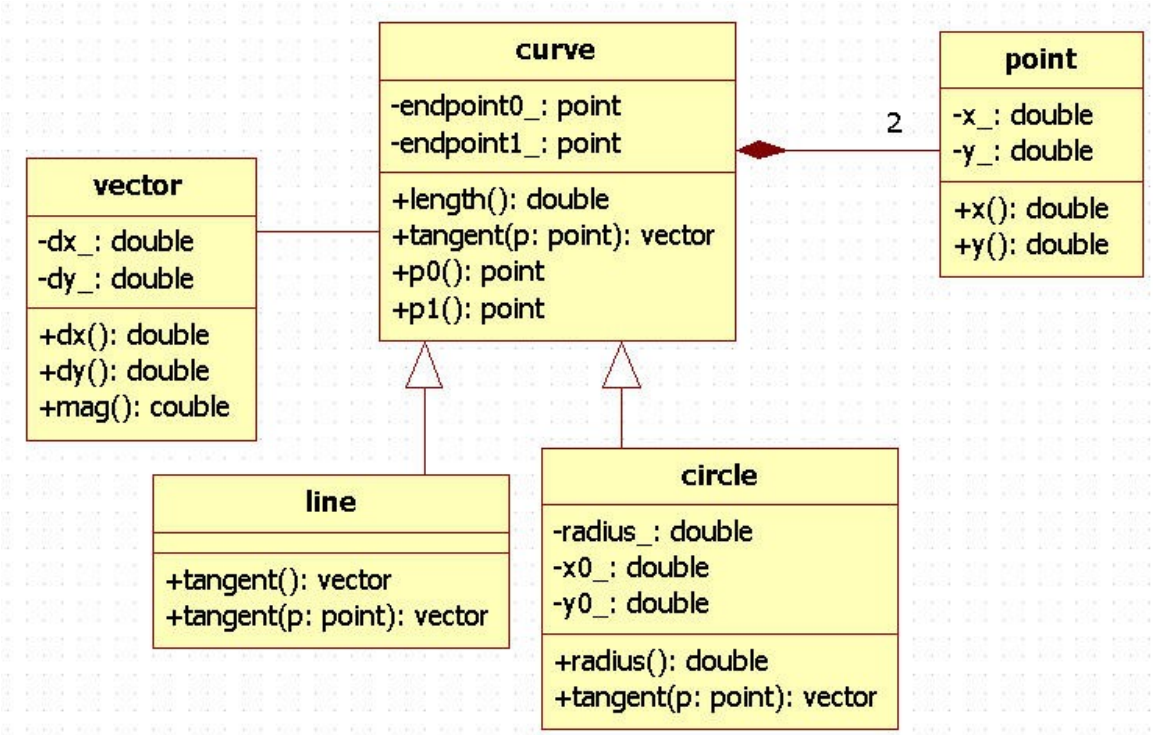
Note that it is possible to construct a **MyComplex** instance as an implicit assignment to the argument of the **csqrt** function. This instance, **MyComplex(-1,0)**, is created as a temporary that persists only during the call.

Note also that the function implementation above computed **mag** and **arg**, which should be methods on **MyComplex**. There is a subtle dividing line that separates **csqrt** from these more basic methods. The key is that **mag** and **arg** are fundamental views of the complex number, while **csqrt** is merely one of hundreds of operations that apply to complex numbers.

An even more compelling reason to implement global functions is the "dispatch problem." Consider the following function, $\|z-w\|$, where z and w are complex numbers, i.e. the "distance" between two complex numbers. This function should not be implemented as a method on **MyComplex** because each argument shares in the resulting distance. It is not clear which should hold the method called with respect to the other.

It could be argued that **+** has the same indifference to order, but its more fundamental role wins out. That is, **+** is considered to add to the existing class value upon which it operates.

Inheritance and Structure



Perhaps the two most powerful mechanisms that human beings have for organizing complexity is *inheritance* and *structure*. Inheritance is a process of abstraction where concepts are related by generality of expression and meaning, e.g. a **`line`** *is a* **`curve`**. Structure is the composition of a complex system from components, which themselves may be made up of components. These mechanisms are illustrated in the design for types of curves, shown by the UML diagram above.

Note that the signature of a class method in UML notation is

`+f(x:double):double`.

The **`+`** indicates the method is **`public`** (**`-`** and **`#`** indicate **`private`** and **`protected`** respectively). The **`':'`** indicates that the type of argument **`x`** is **`double`**, the second **`':'`** indicates that the return type of the method **`f`** is **`double`**. Since UML transcends any particular programming language, this use of **`':'`** has nothing to do with the use of the symbol in C++, so don't mix up the two languages.

The most general concept is **`curve`** which denotes a point set in 2-d space where each point has a one-dimensional neigborhood, i.e. there are at most two neighboring points for any point in curve set. At this level, only a few properties can be specified such as the endpoints and accessors such as **`length`** and **`tangent`**. (actually these latter methods are on the boderline of making it as class methods rather than global methods). The **`length`** method will be ignored for the following discussion.

Structural composition is illustrated by the use of the **`point`** class as a *part of* the **`curve`** class, i.e. its endpoints.

Inheritance and structure are often referred to as *is-a* and *part-of* relations, respectively.

The implementation of the curve class is put in the file `curve.hpp` and appears as follows.

```
#ifndef _curve_hpp_
#define _curve_hpp_
#include "vector.hpp"
#include "point.hpp"
class curve {
public:
  // default constructor.
  // {} means no implementation required
  curve() { };
  // destructor, also no implementation required
  ~curve() { };
  double length() const;
  virtual vector tangent(point const& p) const = 0;
  point p0() const;
  point p1() const;
protected: // described later
  point endpoint0_;
  point endpoint1_;
};
#endif      // _curve_hpp_
```

This implementation is pretty much what has already been covered with the complex number example. In this case, the class members are themselves instances of a class, i.e., **`point`**. A key element to note is the method **`tangent`**. It has the keyword **`virtual`** preceding the return type, **`vector`**. The keyword **`virtual`** indicates that subclasses of curve *can* implement their own version of the computation necessary to produce the tangent. Such methods are called *virtual* methods.

The **`curve`** class represents an abstract concept of a curve and so there is no way to know how to implement tangent at a point without further constraints. The class **`curve`** requires the subclasses *must* implement the **`tangent`** method. This strict requirement is indicated by the **`= 0`** statement at the end of the method declaration. When a method declaration ends in **`= 0`**, that method is called a *pure virtual* method.

It should be clear that if a class defines a pure virtual method, it cannot be constructed as itself, only as a subclass.

The specification of the class **`point`** is very simple, and is declared in the file `point.hpp`

```
#ifndef _point_hpp_
#define _point_hpp_
class point {
public:
  point();
  point(double x, double y);
  ~point();
  double x() const;
  double y() const;
private:
  double _x;
  double _y;
};
#endif // _point_hpp_
```

The more interesting example is the specification for the class **`line`**, in the file `line.hpp`

```
#ifndef _line_hpp_
#define _line_hpp_
#include "curve.hpp"
class line : public curve {
public:
  line();
  line(point const & point0, point const & point1);
  virtual vector tangent(point const & p) const;
  vector tangent() const;
};
#endif // _line_hpp_
```

The statement

```
class line : public curve
```

indicates that the class **`line`** is a subclass of class **`curve`** and that all **`public`** members of curve are publicly accessible from instances of **`line`**. If the **`public`** keyword is left off then the **`public`** members of curve are considered **`private`** to **`line`**. Of course **`private`** members of **`curve`** are not accessible by **`line`** even with the **`public`** designation.

There is one other possibility and that is that a class can declare some of its member values and methods as **`protected`**. This designation means that **`protected`** members and values are accessible by a subclass but not by anything else. In this example, the endpoints are given the status **`protected`** so the curve subclasses can access them.
A summary of the effects of access restriction on members and restriction on inheritance is given in the following table.

base class member restriction	`private`	`public`	`protected`
`public` inheritance	no access by subclass	member is **`public`**	member is **`protected`**
`private` inheritance	no access by subclass	member is **`private`**	member is **`private`**
`protected` inheritance	no access by subclass	member is **`protected`**	member is **`protected`**

Note also that the **`virtual tangent`** method is inherited from the **`curve`** class and must be implemented by **`line`**. Another t**`angent`** method particular to **`line`** is also defined since the tangent to a line is the same everywhere and doesn't require a point to be specified. The two **`tangent`** methods can have the same name and return type since their argument specification differs.