

Programming Techniques for Scientific Simulations I

More on classes, Operators, function objects

Review: Classes

- Are a way to create new data types (“extended data structure”)
 - ♦ E.g., vector, matrix, genome, animal, frog, ... type
- Object oriented programming:
 - ♦ Instead of asking: “What are the subroutines?”
 - ♦ We ask:
 - What are the abstract entities?
 - What are the properties of these entities?
 - How can they be manipulated?
 - Then we implement these entities as classes
- Principles
 - ♦ High level of abstraction possible (**Abstraction**)
 - ♦ Hiding of representation dependent details (**Encapsulation**)
 - ♦ Basing one identity upon another retaining similar properties (**Inheritance**)
 - ♦ Single interface to identities of different types (**Polymorphism**)

Review: What are classes?

- Classes are collections of “members” representing one entity
- Members can be
 - ♦ Functions
 - ♦ Data
 - ♦ Types
- These members can be split into
 - ♦ **public**: accessible interface to the outside. Should not be modified later!
 - ♦ **private**: hidden representation of the concept. Can be changed without breaking any program using the class.
 - ♦ **protected**: like private but also available to derived types. Should be used only for member functions (not data!)
- Objects are instances of a class
- Objects of this type can be modified only through these member functions -> localization of access, easier debugging

Classes: basics

- Classes are specified as

```
class class_name {  
    access_specifier_1:  
        member1;  
    access_specifier_2:  
        member2;  
    ...  
} object_names;
```

← Functions, data, types

← Optional objects

- The access specifier are (in principle) optional
 - ♦ **class** keyword: default access private
 - ♦ **struct** keyword: default access public

Classes: basics

- A simple "point" example:

```
class Point {  
    public:  
        void set_values(double, double); // set Cartesian coordinates  
        double x() { return x_; }; // x coordinate  
        double y() { return y_; } // y coordinate  
        double abs(); // distance from origin aka polar radius  
        double angle(); // polar angle  
    private:  
        double x_, y_; // Cartesian coordinates  
};
```

```
Point p;
```

- How many members? Which ones are declared/defined?

Classes: basics

- A simple "point" example:

```
class Point {  
    public:  
        void set_values(double, double); // set Cartesian coordinates  
        double x() { return x_; }; // x coordinate  
        double y() { return y_; }; // y coordinate  
        double abs(); // distance from origin aka polar radius  
        double angle(); // polar angle  
    private:  
        double x_, y_; // Cartesian coordinates  
};  
  
Point p;
```

- How many members? Which ones are declared/defined?
 - ◆ 2 private data members, 5 public function members (3/2 declared/defined)

Classes: basics

- A simple "point" example:

```
class Point {  
    public:  
        void set_values(double, double); // set Cartesian coordinates  
        double x() { return x_; }; // x coordinate  
        double y() { return y_; }; // y coordinate  
        double abs(); // distance from origin aka polar radius  
        double angle(); // polar angle  
    private:  
        double x_, y_; // Cartesian coordinates  
};
```

```
Point p;
```

- Outside of class definition with the scope operator ::

```
void Point::set_values(double x, double y) {  
    x_ = x;  
    y_ = y;  
}
```

Classes: basics

- Difference between inside or outside of class definition?
 - ◆ Inside the function is considered implicitly `inline`
 - ◆ Outside not
- No difference in actual behavior, but just possible compiler optimizations
- Possibility to separate into header (declaration) and source files (implementation)
- Access object members with dot: `object.member_name`

```
p.set_values(1., 2.);  
std::cout << "p.x() = " << p.x() << '\n';  
std::cout << "p.y() = " << p.y() << '\n';
```


Classes: basics

- Define recurring type with

```
class Point {  
    public:  
  
        void set_values(double, double); // set Cartesian coordinates  
        double x() { return x_; }; // x coordinate  
        double y() { return y_; } // y coordinate  
        double abs(); // distance from origin aka polar radius  
        double angle(); // polar angle  
    private:  
        double x_, y_; // Cartesian coordinates  
};
```

Classes: basics

- Define recurring type with **typedef**

```
class Point {  
    public:  
        typedef double coord_t;  
        void set_values(coord_t, coord_t); // set Cartesian coordinates  
        coord_t x() { return x_; }; // x coordinate  
        coord_t y() { return y_; } // y coordinate  
        coord_t abs(); // distance from origin aka polar radius  
        coord_t angle(); // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};
```

- Access types with the scope operator **::**

```
Point::coord_t Point::abs() {  
    return std::sqrt(x_*x_ + y_*y_);  
}
```

Classes: basics

- Define recurring type with **using** (C++11)

```
class Point {  
    public:  
        using coord_t = double;  
        void set_values(coord_t, coord_t); // set Cartesian coordinates  
        coord_t x() { return x_; }; // x coordinate  
        coord_t y() { return y_; } // y coordinate  
        coord_t abs(); // distance from origin aka polar radius  
        coord_t angle(); // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};
```

- Access types with the scope operator **::**

```
Point::coord_t Point::abs() {  
    return std::sqrt(x_*x_ + y_*y_);  
}
```

Classes: basics

- Multiple objects (instances) of the class (the type)

```
Point p1, p2;
```

```
p1.set_values(1. , 2.);  
p2.set_values(7. , 3.14);
```

```
std::cout << "p1.angle() = " << p1.angle() << '\n';  
std::cout << "p2.abs() = " << p2.abs() << '\n';
```

- Each of them has his own members
- Since classes are types, they can be passed to functions, other objects, ... Object-oriented paradigm
- As opposed to something like
 - ♦ E.g., LAPACK (FORTRAN77) subroutine to compute eigenvalues (see [here](#) for doc.)

```
SUBROUTINE DSYTRD( UPLO, N, A, LDA, D, E, TAU, WORK, LWORK, INFO )
```

Classes: basics

- What's wrong?

```
Point p1, p2;
```

```
std::cout << "p1.angle() = " << p1.angle() << '\n';
```

```
std::cout << "p2.abs() = " << p2.abs() << '\n';
```

```
p1.set_values(1. , 2.);
```

```
p2.set_values(7. , 3.14);
```

Classes: basics

- What's wrong?

```
Point p1, p2;
```

```
std::cout << "p1.angle() = " << p1.angle() << '\n';  
std::cout << "p2.abs() = " << p2.abs() << '\n';
```

```
p1.set_values(1. , 2.);  
p2.set_values(7. , 3.14);
```

- Solution: constructor (ctor)!

Classes: basics

- Constructor (ctor)

```
class Point {  
    public:  
        using coord_t = double;  
  
        Point(coord_t, coord_t);    // ctor  
        coord_t x() { return x_; }; // x coordinate  
        coord_t y() { return y_; }; // y coordinate  
        coord_t abs();              // distance from origin aka polar radius  
        coord_t angle();            // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};  
  
Point::Point(coord_t x, coord_t y) {  
    x_ = x;  
    y_ = y;  
}
```

- Automatically called when an object of a class is created
- Like ordinary member function, but with same name as the class and no return

Classes: basics

- Constructor (ctor)

```
class Point {  
    public:  
        using coord_t = double;  
  
        Point(coord_t, coord_t);    // ctor  
        coord_t x() { return x_; }; // x coordinate  
        coord_t y() { return y_; }; // y coordinate  
        coord_t abs();              // distance from origin aka polar radius  
        coord_t angle();            // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};  
  
Point p1(1., 2.), p2(7., 3.14);  
std::cout << "p1.x() = " << p1.x() << '\n';  
std::cout << "p1.y() = " << p1.y() << '\n';  
Point p3;
```

- p1 and p2 are constructed by calling the ctor with its arguments
- What about p3?

Classes: basics

- Overloading constructors

```
class Point {
public:
    using coord_t = double;
    Point() { x_ = 0.; y_ = 0.; } // default ctor
    Point(coord_t, coord_t);      // ctor
    coord_t x() { return x_; }; // x coordinate
    coord_t y() { return y_; }; // y coordinate
    coord_t abs();                // distance from origin aka polar radius
    coord_t angle();              // polar angle
private:
    coord_t x_, y_; // Cartesian coordinates
};

Point p1(1., 2.), p2(7., 3.14);
std::cout << "p1.x() = " << p1.x() << '\n';
std::cout << "p1.y() = " << p1.y() << '\n';
Point p3;
```

- Constructor with no arguments is called the default ctor
- It's special because it's called when an object is declared

Classes: basics

- Overloading constructors

```
class Point {
public:
    using coord_t = double;
    Point() { x_ = 0.; y_ = 0.; } // default ctor
    Point(coord_t, coord_t);      // ctor
    coord_t x() { return x_; }; // x coordinate
    coord_t y() { return y_; }; // y coordinate
    coord_t abs();                // distance from origin aka polar radius
    coord_t angle();              // polar angle
private:
    coord_t x_, y_; // Cartesian coordinates
};

Point p1(1., 2.), p2(7., 3.14);
std::cout << "p1.x() = " << p1.x() << '\n';
std::cout << "p1.y() = " << p1.y() << '\n';
Point p3();
```

- The default ctor cannot be called directly!
- Why?

Classes: basics

- Overloading constructors

```
class Point {  
public:  
    using coord_t = double;  
    Point() { x_ = 0.; y_ = 0.; } // default ctor  
    Point(coord_t, coord_t);      // ctor  
    coord_t x() { return x_; }    // x coordinate  
    coord_t y() { return y_; }    // y coordinate  
    coord_t abs();                // distance from origin aka polar radius  
    coord_t angle();              // polar angle  
private:  
    coord_t x_, y_; // Cartesian coordinates  
};  
  
Point p1(1., 2.), p2(7., 3.14);  
std::cout << "p1.x() = " << p1.x() << '\n';  
std::cout << "p1.y() = " << p1.y() << '\n';  
Point p3();
```

- The default ctor cannot be called directly!
- Why? Because it looks like a function declaration! (“Most vexing parse”)

Classes: basics

- Uniform initialization (C++11)

```
class Point {  
public:  
    using coord_t = double;  
    Point() { x_ = 0.; y_ = 0.; } // default ctor  
    Point(coord_t, coord_t);      // ctor  
    coord_t x() { return x_; }   // x coordinate  
    coord_t y() { return y_; }   // y coordinate  
    coord_t abs();               // distance from origin aka polar radius  
    coord_t angle();             // polar angle  
private:  
    coord_t x_, y_; // Cartesian coordinates  
};  
  
Point p1(1., 2.), p2{7., 3.14};  
std::cout << "p1.x() = " << p1.x() << '\n';  
std::cout << "p1.y() = " << p1.y() << '\n';  
Point p3{};
```

Functional form

- Use braces / curly brackets
- (Mostly) Stylistic

Classes: basics

- Member initialization in constructors with initializer list

```
class Point {  
    public:  
        using coord_t = double;  
        Point() : x_(0.), y_(0.) {} // default ctor  
        Point(coord_t, coord_t);    // ctor  
        coord_t x() { return x_; }  // x coordinate  
        coord_t y() { return y_; }  // y coordinate  
        coord_t abs();              // distance from origin aka polar radius  
        coord_t angle();            // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};
```

- Insert a colon : and a list of initializations for class members (before ctor body)
 - member(value)
 - member{value} (C++11 uniform initialization) Can be a waste!
- For fundamental types no difference, class types default-ctor called

Classes: basics

- Sometimes it is necessary to use the initializer list

```
class Line {  
    public:  
        Line(Point p1, Point p2) : p1_(p1.x(), p1.y())  
                                   , p2_(p2.x(), p2.y())  
                                   , rp2(p2_) {}  
  
        // ... useful line stuff  
    private:  
        const Point p1_  
        Point p2_  
        Point& rp2;  
};
```

- Without the initializer list: no way of setting members that are `const`, reference or of a class type that does not have a default ctor!
- Stylistic advice: always use initializer list
- Order of member initialization: same order as declared (And not order in initializer list!)

Classes: basics

- Constructors are (special) functions

```
class Point {  
    public:  
        using coord_t = double;  
        Point() : x_(0.), y_(0.) {} // default ctor  
        Point(coord_t x = 0., coord_t y = 0.) : x_(x), y_(y) {} // (default) ctor  
        coord_t x() { return x_; }; // x coordinate  
        coord_t y() { return y_; } // y coordinate  
        coord_t abs(); // distance from origin aka polar radius  
        coord_t angle(); // polar angle  
    private:  
        coord_t x_, y_; // Cartesian coordinates  
};
```

- So we can use default arguments (and avoid the definition of the previous default ctor)

Classes: basics

■ Pointers to classes

```
class Point {  
    // ...  
};  
  
Point op(1., 2.);  
Point* pp1;  
Point* pp2 = new Point(3., 12.);  
Point ap1[] = {{1., 2.}, {3., 4.}};  
Point* ap2 = new Point[2] {{5., 6.}, {7., 8.}};  
  
pp1 = &op;  
  
std::cout << "op.x() = "           << op.x()           << '\n';  
std::cout << "pp1->y() = "         << pp1->y()         << '\n';  
std::cout << "(*pp2).y() = "       << (*pp2).y()       << '\n';  
std::cout << "ap1[0].abs() = "     << ap1[0].abs()     << '\n';  
std::cout << "ap2[1].angle() = "  << ap2[1].angle()  << '\n';  
// ...
```


Classes: basics

- The usual expressions

expression	means
<code>*x</code>	object pointed to by <code>x</code>
<code>&x</code>	address of object <code>x</code>
<code>x.y</code>	member <code>y</code> of object <code>x</code>
<code>x->y</code>	member <code>y</code> of object pointed to by <code>x</code>
<code>(*x).y</code>	member <code>y</code> of object pointed to by <code>x</code> (same as <code>-></code>)
<code>x[0]</code>	first object pointed to by <code>x</code>

See: https://en.cppreference.com/w/cpp/language/operator_member_access

Classes: Overloading operators

- Classes introduce new types and we would like to define operations on them:

```
Complex a(1., 2.);  
Complex b(2., 0.);  
Complex c = a + b;
```

```
Matrix A;  
Vector x;  
Vector b = A*x;
```

- C++ lets overload the following operators

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

Classes: Overloading operators

- The syntax is:

```
type operator OP ( arguments ) { /*... body ...*/ }
```

- Consider a simple complex numbers class

```
class Complex {  
    public:  
        using value_t = double;  
        Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor  
        value_t re() { return re_; } // real part  
        value_t im() { return im_; } // imaginary part  
  
    private:  
        value_t re_, im_; // real & imaginary parts  
};
```

- We want that: $a = 1 + 2i$ $b = 3 + 4i$
 $c = a + b = 4 + 6i$

Classes: Overloading operators

- The syntax is:

```
type operator OP ( arguments ) { /*... body ...*/ }
```

- Consider a simple complex numbers class

```
class Complex {  
public:  
    using value_t = double;  
    Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor  
    value_t re() { return re_; } // real part  
    value_t im() { return im_; } // imaginary part  
  
private:  
    value_t re_, im_; // real & imaginary parts  
};
```

- We want that:
Complex a(1., 2.);
Complex b(3., 4.);
c = a + b;
c = a.operator+(b);

Classes: Overloading operators

- The syntax is:

```
type operator OP ( arguments ) { /*... body ...*/ }
```

- Consider a simple complex numbers class

```
class Complex {  
    public:  
        using value_t = double;  
        Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor  
        value_t re() { return re_; } // real part  
        value_t im() { return im_; } // imaginary part  
        type operator OP ( arguments ) { /*... body ...*/ }  
  
    private:  
        value_t re_, im_; // real & imaginary parts  
};
```

- We want that:
 Complex a(1., 2.);
 Complex b(3., 4.);
 c = a + b;
 c = a.operator+(b);

Classes: Overloading operators

- The syntax is:

```
type operator OP ( arguments ) { /*... body ...*/ }
```

- Consider a simple complex numbers class

```
class Complex {  
    public:  
        using value_t = double;  
        Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor  
        value_t re() { return re_; } // real part  
        value_t im() { return im_; } // imaginary part  
        Complex operator + (Complex const&); // addition  
  
    private:  
        value_t re_, im_; // real & imaginary parts  
};
```

- We want that:
 Complex a(1., 2.);
 Complex b(3., 4.);
 c = a + b;
 c = a.operator+(b);

Classes: Overloading operators

- The syntax is:

```
type operator OP ( arguments ) { /*... body ...*/ }
```

- Consider a simple complex numbers class

```
class Complex {  
public:  
    using value_t = double;  
    Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor  
    value_t re() { return re_; } // real part  
    value_t im() { return im_; } // imaginary part  
    Complex operator + (Complex const&); // addition  
    int operator - (Complex const&) { return 0; } // WTF!  
private:  
    value_t re_, im_; // real & imaginary parts  
};
```

- Overloads are just regular functions... However, it is strongly recommended to keep some relation to their usual (mathematical) meaning!!!

Classes: Overloading operators

- A summary of the different operators and their “overload form” scope

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)
a@b	+ - * / % ^ & < > == != <= >= << >> && ,	A::operator@(B)	operator@(A, B)
a@b	= += -= *= /= %= ^= &= = <<= >>= []	A::operator@(B)	-
a(b, c...)	()	A::operator() (B, C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

"Inside" class def.

"Outside" class def.
(aka free function)

- See: <https://en.cppreference.com/w/cpp/language/operators>

Classes: Overloading operators (introducing friend)

- Implementing the minus operator as a non-member function

```
class Complex {
public:
    using value_t = double;
    Complex(value_t re = 0., value_t im = 0.) : re_(re), im_(im) {} // ctor
    value_t re() { return re_; } // real part
    value_t im() { return im_; } // imaginary part
    Complex operator + (Complex const&); // addition
    friend Complex operator - (Complex const&, Complex const&); // subtraction
private:
    value_t re_, im_; // real & imaginary parts
};

Complex operator - (Complex const& a, Complex const& b) {
    Complex tmp = Complex(a.re_ - b.re_, a.im_ - b.im_);
    return tmp;
}
```

- Grants a function or another class access to the private and protected members of the class
- What could be the benefits? (More on this later... Symmetric OPs in expressions with mixed types...)

Example: pretty printing Point

- We want

```
std::cout << "p0 = " << p0 << '\n';  
// prints: p0 = ( x, y)
```

- Pretty printing: overloading the "insertion" << operator

What would it mean to "copy"?

```
std::ostream& operator<<(std::ostream& out, Point const& p) {  
    out << "(" << p.x() << ", " << p.y() << " )";  
    return out;  
}
```

- Exercise: overload the "extraction" >> operator

Example: function objects

- We sometimes want to use an object like a function, e.g.,

```
Potential V;  
double distance;  
std::cout << V(distance);
```

- This works only if Potential is a function pointer, or if we define the function call operator

```
class Potential {  
    // ...  
    double operator()(double d) { return 1./d; }  
    // ...  
};
```

- Don't get confused by the two pairs of () ()
 - ◆ The first is the name of the operator
 - ◆ The second is the argument list

Example: function objects

- Assume a function with parameters, e.g., $f(x; \lambda) = \exp(-\lambda x)$

```
double func(double x, double lambda) {  
    return exp(-lambda*x);  
}
```

- Cannot directly be used with Simpson integrator from exercises!

- Solution: use a function object

```
class MyFunc {  
public:  
    MyFunc(double l) : lambda(l) {}  
    double operator()(double x) {  
        return exp(-lambda*x);  
    }  
private:  
    const double lambda;  
};  
MyFunc f(3.5);  
integrate(f, 0., 1., 1000);
```

- Uses object of type MyFunc like a function!

- Very useful and widely used technique

Function objects alternative: lambda (C++11)

- A lambda expression (or lambda function or simply lambda) is a simplified notation for defining and using an anonymous function object
- Particularly useful to define small local functions, e.g., to pass as an argument to an algorithm
- Syntax:

`[capture list] (parameter list) { body }`

Lambda expressions (C++11)

- Syntax: `[capture list] (parameter list) { body }`
- The capture list contains the variables from the enclosing scope that should be captured inside the lambda and how:

<code>[]</code>	Capture nothing
<code>[&]</code>	Capture any referenced variable by reference
<code>[=]</code>	Capture any referenced variable by making a copy
<code>[=, &foo]</code>	Capture any referenced variable by making a copy, but capture variable <code>foo</code> by reference
<code>[bar]</code>	Capture <code>bar</code> by making a copy; don't copy anything else
<code>[this]</code>	Capture the <code>this</code> pointer of the enclosing class

Example: lambda

- Assume a function with parameters, e.g., $f(x; \lambda) = \exp(-\lambda x)$

```
double func(double x, double lambda) {  
    return exp(-lambda*x);  
}
```

- Cannot directly be used with Simpson integrator from exercises!

- Solution: use a lambda

```
lambda = -3.5;  
auto f = [lambda] (double x) { return std::exp(-lambda*x); };  
integrate(f, 0., 1., 1000);
```

- Uses object f like a function!

- Very useful and widely used technique

Classes: `this`

- The keyword `this` is a pointer to the object whose member function is being executed.
- It's used to refer to yourself within a member function, e.g.,

```
Point::coord_t Point::x() {  
    return this->x_; // or (*this).x_ or simply x_  
}
```

- Often used in the assignment operators:

```
Point& Point::operator = (Point const& rhs) {  
    x_ = rhs.x_;  
    y_ = rhs.y_;  
    return *this;  
}
```

- Why?

Classes: `this`

- The keyword `this` is a pointer to the object whose member function is being executed.
- It's used to refer to yourself within a member function, e.g.,

```
Point::coord_t Point::x() {  
    return this->x_; // or (*this).x_ or simply x_  
}
```

- Often used in the assignment operators:

```
Point& Point::operator = (Point const& rhs) {  
    x_ = rhs.x_;  
    y_ = rhs.y_;  
    return *this;  
}
```

- Why? To allow, e.g., chains of assignments (like the built-in types)

```
a = b = c = 4; // "do as the ints do"
```

Classes: `static` members

- A static data member is one common variable for all the objects of that same class
- Also known as a “class variable”
- Why?

Classes: **static** members

- A static data member is one common variable for all the objects of that same class
- Also known as a “class variable”

- Example:

```
struct ImCounting {
    ImCounting() { cnt++; }
    static int cnt;
};
// initialize count to zero
int ImCounting::cnt = 0;

int main() {

    ImCounting a;
    ImCounting b;
    std::cout << ImCounting::cnt << '\n';
    ImCounting* p = new ImCounting[9];
    std::cout << ImCounting::cnt << '\n';
    delete[] p;
}
```

- What should that print?

Classes: **static** members

- Are **shared by all objects** of a type
- Act like global variables in a name space

- Exist even without an object, thus `::` notation used:

```
Genome::gene_number;  
Genome::set_mutation_rate(2);
```

- Static member functions can only access static member variables! Reason: which object's members to use?
- Must be declared and defined!
- Will not link otherwise
- See demo: IDGenerator

```
// in header file:  
class Genome {  
public:  
    Genome(); // ctor  
    // const static data member  
    static const unsigned short gene_number=64;  
    Genome clone() const;  
    static void set_mutation_rate(unsigned short);  
private:  
    unsigned long gene_;  
    static unsigned short mutation_rate_;  
};
```

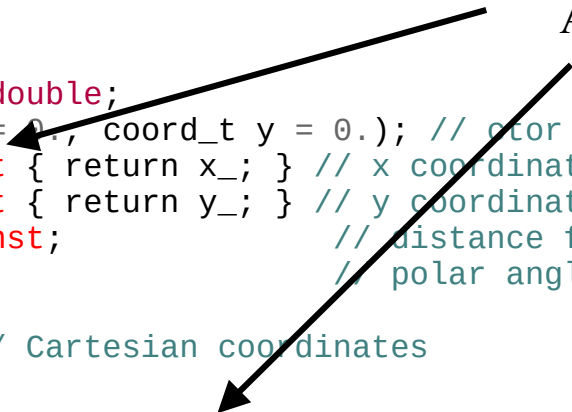
```
// in source file:  
// definition  
unsigned short Genome::mutation_rate_=2;
```

Classes: **const** member functions

- Member functions that don't change the state of an object should be qualified as such (by the keyword **const**):

```
class Point {  
public:  
    using coord_t = double;  
    Point(coord_t x = 0., coord_t y = 0.); // ctor defaults to (0,0)  
    coord_t x() const { return x_; } // x coordinate  
    coord_t y() const { return y_; } // y coordinate  
    coord_t abs() const; // distance from origin aka polar radius  
    coord_t angle(); // polar angle  
private:  
    coord_t x_, y_; // Cartesian coordinates  
};  
  
Point::coord_t Point::abs() const {  
    return std::sqrt(x_*x_ + y_*y_);  
}
```

After function (prototype)!



- Which member functions can be called on the following?

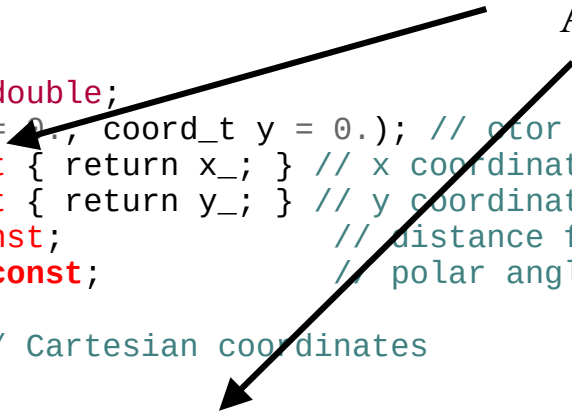
```
Point const cp(9., 10.);
```

Classes: **const** member functions

- Member functions that don't change the state of an object should be qualified as such (by the keyword **const**):

```
class Point {  
public:  
    using coord_t = double;  
    Point(coord_t x = 0., coord_t y = 0.); // ctor defaults to (0,0)  
    coord_t x() const { return x_; } // x coordinate  
    coord_t y() const { return y_; } // y coordinate  
    coord_t abs() const; // distance from origin aka polar radius  
    coord_t angle() const; // polar angle  
private:  
    coord_t x_, y_; // Cartesian coordinates  
};  
  
Point::coord_t Point::abs() const {  
    return std::sqrt(x_*x_ + y_*y_);  
}
```

After function (prototype)!



- Which member functions can be called on the following? All!
Only qualified const member function can be called on const objects

Classes: mutable

- Consider some class

```
class A {  
    public:  
        int func() const;  
    private:  
        int cnt_;  
};  
int A::func() const {  
    cnt_++;  
    return 42;  
}
```

- Problem: want to count
number of calls to
`func()` function

Classes: mutable

- Consider some class

```
class A {  
    public:  
        int func() const;  
    private:  
        int cnt_;  
};  
int A::func() const {  
    cnt_++; // ERROR: const!  
    return 42;  
}
```

- Problem: want to count number of calls to func() function

- Solution:

- ◆ mutable qualifier allows modification of member even in const object!

```
class A {  
    public:  
        int func() const;  
    private:  
        mutable int cnt_;  
};  
int A::func() const {  
    cnt_++; // OK!  
    return 42;  
}
```


Next week

- Special member functions
 - ◆ E.g., Copy constructor: Creates (new) copies of objects
`c(c const&); // copy ctor`
- Resource management: RAII
- Class templates
- Type traits (first contact with C++ template metaprogramming)