HMKANG@UMICH.EDU

MODULE 2.4

PASSING CLASSES AND FUNCTIONS BETWEEN R AND RCPP

Today

- **A quick review of important features in C++**
  - Classes
  - Operator overloading
  - Function objects
  - Abstract classes

- **To help organize your code better, both in C++ and Rcpp**
  - Define a general optimizer
  - Pass C++ functions as arguments

Disclamer

- This part is quite an advanced C++
- Some of this contents need to be used in the homework, so you are expected to have a basic understand of the material.
- However, but this will NOT be the part of the final exam. So it's okay if you didn't perfectly understand the material.

Defining a simple C++ class

```
class cPoint {
public:
  // member variables
  double x;
  double y;

  // constrcutor
  cPoint() { x = y = 0; }
  cPoint(double _x, double _y) { x = _x; y = _y; }

  // member functions
  double getRadius() { return sqrt(x*x + y*y); }
  double getAngle() { return x == 0 ? ( y > 0 ? 90 : -90 ) : atan(y/x) * 180 / M_PI; }
  double getX() { return x; }
  double getY() { return y; }
  void print() { // in C++, Rprintf() can be changed to printf()
    Rprintf("* x = %lg\n* y = %lg\n* radius = %lg\n* angle = %lg\n\n", getX(), getY(), ge
tRadius(), getAngle() );
  }
};
```

Using the C++ class

```
// [[Rcpp::export]]
void testPoint() {
  cPoint p1(3,4);
  Rprintf("p1.radius = %lg\n",p1.getRadius());
  Rprintf("p1.angle = %lg\n",p1.getAngle());
  p1.print();
  cPoint p2(4,4);
  Rprintf("p2.radius = %lg\n",p2.getRadius());
  Rprintf("p2.angle = %lg\n",p2.getAngle());
  p2.print();
}
```

Running examples

```
testPoint()
```

```
p1.radius = 5
p1.angle = 53.1301
* x = 3
* y = 4
* radius = 5
* angle = 53.1301

p2.radius = 5.65685
p2.angle = 45
* x = 4
* y = 4
* radius = 5.65685
* angle = 45
```

Using polar coordinate inside

```
class pPoint {
public:
  // member variables
  double radius;
  double degree;
  // constructor
  pPoint() { radius = degree = 0; }
  pPoint(double x, double y) {
    radius = sqrt(x*x + y*y);
    degree = ( x == 0 ) ? (y > 0 ? 90 : -90) : atan(y/x) * 180 / M_PI;
  }
  // member functions
  double getRadius() { return radius; }
  double getAngle() { return degree; }
  double getX() { return radius * cos(degree * M_PI / 180); }
  double getY() { return radius * sin(degree * M_PI / 180); }
  void print() { // in C++, Rprintf() can be changed to printf()
    Rprintf("* x = %lg\n* y = %lg\n* radius = %lg\n* angle = %lg\n\n", getX(),
 getY(), getRadius(), getAngle() );
  }
};
```

A modified usage

```
// [[Rcpp::export]]
void testPoint() {
  cPoint p1(3,4);
  Rprintf("p1.radius = %lg\n",p1.getRadius());
  Rprintf("p1.angle = %lg\n",p1.getAngle());
  p1.print();

  cPoint p2(4,4);
  Rprintf("p2.radius = %lg\n",p2.getRadius());
  Rprintf("p2.angle = %lg\n",p2.getAngle());
  p2.print();

  pPoint p3(4,4);
  Rprintf("p3.radius = %lg\n",p3.getRadius());
  Rprintf("p3.angle = %lg\n",p3.getAngle());
  p3.print();
}
```

```
testPoint()
```

```
p1.radius = 5
p1.angle = 53.1301
* x = 3
* y = 4
* radius = 5
* angle = 53.1301

p2.radius = 5.65685
p2.angle = 45
* x = 4
* y = 4
* radius = 5.65685
* angle = 45

p3.radius = 5.65685
p3.angle = 45
* x = 4
* y = 4
* radius = 5.65685
* angle = 45
```

Identical results to the results with cPoint()

Basics of C++ classes

- **Class is a user-defined data type**
- **… with member variables and functions**
- **Constructor defines how an instance of the class should be initialized**
  - Destructor defines what to do before being destroyed
- **An effective way to make an abstract object from a complex implementation**

Operator overloading

```
// [[Rcpp::export]]
void testPoint() {
  cPoint p1(1,2);
  cPoint p2(2,2);
  cPoint p3 = p1 + p2; // would this work?
  p3.print();
}
```

Defining operators for your classes

```
class cPoint {
public:
  // member variables
  double x;
  double y;

  // constructor
  cPoint() { x = y = 0; }
  cPoint(double _x, double _y) { x = _x; y = _y; }

  // member functions
  double getRadius() { return sqrt(x*x + y*y); }
  double getAngle() { return x == 0 ? ( y > 0 ? 90 : -90 ) : atan(y/x) * 180 / M_PI; }
  double getX() { return x; }
  double getY() { return y; }
  void print() { // in C++, Rprintf() can be changed to printf()
    Rprintf("* x = %lg\n* y = %lg\n* radius = %lg\n* angle = %lg\n\n", getX(), getY(), getRadius
(), getAngle() );
  }
```

This part remains the same..

Defining operators for your classes

```
  // operator overloading
  cPoint operator+(const cPoint& rhs) {
    cPoint newPoint(x, y);
    newPoint.x += rhs.x;
    newPoint.y += rhs.y;
    return newPoint;
  }
};

// [[Rcpp::export]]
void testPoint() {
  cPoint p1(1,2);
  cPoint p2(2,2);
  cPoint p3 = p1 + p2; // would this work?
  p3.print();
}
```

More generally, for complex objects, you need to define copy constructor and assignment operators (not covered in this class)

Results

```
testPoint()
```

```
* x = 3
* y = 4
* radius = 5
* angle = 53.1301
```

Recap: operator overloading

- Instead of defining a new function, you may redefine an existing operator.
- Call the operator instead of calling a new function. (to give users some illusionary effect)
- Redefining copy constructor and assignment operator may be necessary if your member variables contain pointers and/or newly allocated objects inside the class.

Function objects

- **Passing a function in R / python is super-easy**
  - Use R/python function name as if it is a variable name when calling a function.

- **In C++, passing a function requires an additional tweaks**
  - Directly pass the function pointer
  - Define a function object and pass an instance

Passing a function pointer

```
double foo(double x, double y) {
  return sqrt(x*x + y*y);
}
```

```
double runFuncPtr(double (*f) (double, double), double x, double y) {
  double val = f(x,y);
  return val;
}
```

```
// [[Rcpp::export]]
double testPassingFunctionPtr(double x, double y) {
  return runFuncPtr(foo, x, y);
}
```

A running example

```
testPassingFunctionPtr(3,4)
```

```
[1] 5
```

Using operator overloading to define a function(-like) object

```
class bar {
public:
  double operator() (double x, double y) {
    return sqrt(x*x + y*y);
  }
};
```

```
// [[Rcpp::export]]
double testPassingFunctionObj(double x, double y) {
  bar barFunc;
  return barFunc(x,y);
}
```

A running example

```
testPassingFunctionObj(3,4)
```

```
[1] 5
```

Passing function pointer vs. object

```
double runFuncPtr(double (*f) (double, double), double x, double y) {
  double val = f(x,y);
  return val;
}

double runFuncObj(bar& f, double x, double y) {
  double val = f(x,y);
  return val;
}
```

Recap : function object

- C or C++ function can be passed as an argument in functions, but it is quite cumbersome to use
- Function object is a class that can be easily passed as an argument via operator overloading
- Function object is a class that can hold member variables, so it is an effective way to represent likelihood (with data)

An example C++ class

```
using namespace Rcpp;
using namespace std;

// abstract class -- cannot create an instance
class aPoint {
public:
  // these are pure virtual functions
  virtual double getRadius() = 0;
  virtual double getAngle() = 0;
  virtual double getX() = 0;
  virtual double getY() = 0;
  virtual string whoAmI() { return string("aPoint"); }
  void print() {
    Rprintf("* whoAmI = %s\n* x = %lg\n* y = %lg\n* radius = %lg\n* angle = %lg\n\n",
 whoAmI().c_str(), getX(), getY(), getRadius(), getAngle() );
  }
};
```

A derived class of the abstract class

```
class cPoint : public aPoint {
public:
  double x;
  double y;
  cPoint() { x = y = 0; }
  cPoint(double _x, double _y) { x = _x; y = _y; }
  double getRadius() { return sqrt(x*x + y*y); }
  double getAngle() { return x == 0 ? ( y > 0 ? 90 : -90 ) : atan(y/x) *
180 / M_PI; }
  double getX() { return x; }
  double getY() { return y; }
  string whoAmI() { return string("cPoint"); }
};
```

Another derived class of the abstract class

```
class pPoint : public aPoint {
public:
  double radius;
  double degree;
  pPoint() { radius = degree = 0; }
  pPoint(double x, double y) {
    radius = sqrt(x*x + y*y);
    degree = ( x == 0 ) ? (y > 0 ? 90 : -90) : atan(y/x) * 180 / M_PI;
  }
  double getRadius() { return radius; }
  double getAngle() { return degree; }
  double getX() { return radius * cos(degree * M_PI / 180); }
  double getY() { return radius * sin(degree * M_PI / 180); }
  string whoAmI() { return string("pPoint"); }
};
```

Example usages

Even if it is represented as a pointer to the abstract class, the virtual functions are executed based on the correct derived classes

```
// [[Rcpp::export]]
void testPoint() {
  cPoint p1(3,4);
  Rprintf("p1.radius = %lg\n",p1.getRadius());
  Rprintf("p1.angle = %lg\n",p1.getAngle());
  p1.print();

  aPoint* p2 = new cPoint(4,4);
  Rprintf("p2.radius = %lg\n",p2->getRadius());
  Rprintf("p2.angle = %lg\n",p2->getAngle());
  p2->print();

  aPoint* p3 = new pPoint(4,4);
  Rprintf("p3.radius = %lg\n",p3->getRadius());
  Rprintf("p3.angle = %lg\n",p3->getAngle());
  p3->print();
}
```

Results

```
testPoint()
```

```
p1.radius = 5
p1.angle = 53.1301
* whoAmI = cPoint
* x = 3
* y = 4
* radius = 5
* angle = 53.1301
```

```
p2.radius = 5.65685
p2.angle = 45
* whoAmI = cPoint
* x = 4
* y = 4
* radius = 5.65685
* angle = 45
```

```
p3.radius = 5.65685
p3.angle = 45
* whoAmI = pPoint
* x = 4
* y = 4
* radius = 5.65685
* angle = 45
```

Passing a function in C++ and Rcpp

- In the next lectures, we will develop general algorithms for multi-dimensional optimization.

- Passing a "general" C++ function to an C++ function (without calling R each time) is the most efficient way to repetitively call the function many times.

- We can achieve this using a function pointer, abstract class, and `Rcpp::XPtr`

Defining an abstract function pointer

```cpp
#include <Rcpp.h>
#include <cmath>

using namespace std;
using namespace Rcpp;

class abstractFunc {
public:
  virtual double operator() (double x, double y) = 0;
  // virtual destructor is required to use XPtr
  virtual ~abstractFunc() {}
};
```

Defining a normal likelihood function

```
class normalLLK : public abstractFunc {
public:
  NumericVector data;
  normalLLK(NumericVector& _data) { data = _data; }
  double operator() (double mu, double sigma2) {
    int n = data.size();
    double logLik = 0;
    for(int i=0; i < n; ++i) {
      logLik += (( data[i] - mu ) * (data[i] - mu ));
    }
    logLik = -0.5 * logLik / sigma2;
    logLik -= ( n / 2.0 * log(2*M_PI) );
    return logLik;
  }
};
```

Defining a beta likelihood function

```
class betaLLK : public abstractFunc {
public:
  NumericVector data;
  betaLLK(NumericVector& _data) { data = _data; }
  double operator() (double alpha, double beta) {
    int n = data.size();
    double logLik = 0;
    for(int i=0; i < n; ++i)
      logLik += R::dbeta(data[i], alpha, beta, 1);
    return logLik;
  }
};
```

Getting functions, and using functions

```cpp
// [[Rcpp::export]]
XPtr<abstractFunc> getLLKFunc(string type, NumericVector obs) {
  abstractFunc* fptr = NULL;
  if ( type == "normal" )
    fptr = new normalLLK(obs);
  else if ( type == "beta" )
    fptr = new betaLLK(obs);
  else
    stop("Cannot recognize the LLK function type ");
  return XPtr<abstractFunc>(fptr);
}
```

```cpp
// [[Rcpp::export]]
double calculateLLK(XPtr<abstractFunc> p, double param1, double param2) {
  abstractFunc& f = *p;
  return f(param1,param2);
}
```

Need to use reference(or pointer) type to access virtual functions

Example use

```r
x <- runif(100);
y <- rbeta(100,2,2);
fnormx <- getLLKFunc("normal",x)
fbetax <- getLLKFunc("beta",x)
fnormy <- getLLKFunc("normal",y)
fbetay <- getLLKFunc("beta",y)
print(calculateLLK(fnormx, 0, 1))
```

```
[1] -107.116
```

More examples

```
print(calculateLLK(fbetax, 1, 1))
```

```
[1] 0
```

Hide

```
print(calculateLLK(fbetax, 2, 2))
```

```
[1] -7.971546
```

Hide

```
print(calculateLLK(fbetay, 2, 2))
```

```
[1] 3.799747
```

Summary

- Class is a user-defined data type
- Class gives a level of abstraction to users
- Operator overloading allows class to be used like a function
- Function objects can be easily passed as argument in C++
- Abstract classes and virtual functions allows to pass different types of classes (and function objects) as pointers.