Computing and Statistical Data Analysis
Comp 4: finish pointers, intro to classes

Glen Cowan
Physics Department
Royal Holloway, University of London
Egham, Surrey TW20 0EX

01784 443452
`g.cowan@rhul.ac.uk`
`www.pp.rhul.ac.uk/~cowan/stat_course.html`

Why different kinds of pointers?

Suppose we declare

```
int* iPtr;        //   type "pointer to int"
float* fPtr;      //   type "pointer to float"
double* dPtr;     //   type "pointer to double"
```

We need different types of pointers because in general, the different data types (`int, float, double`) take up different amounts of memory. If declare another pointer and set

```
int* jPtr = iPtr + 1;
```

then the `+1` means “plus one unit of memory address for `int`”, i.e., if we had `int` variables stored contiguously, `jPtr` would point to the one just after `iPtr`.

But the types `float`, `double`, etc., take up different amounts of memory, so the actual memory address increment is different.

Passing pointers as arguments

When a pointer is passed as an argument, it divulges an address to the called function, so the function can change the value stored at that address:

```
void passPointer(int* iPtr){
  *iPtr += 2;           // note *iPtr on left!
}

...
int i = 3;
int* iPtr = &i;
passPointer(iPtr);
cout << "i = " << i << endl;    // prints i = 5
passPointer(&i);              // equivalent to above
cout << "i = " << i << endl;    // prints i = 7
```

End result same as pass-by-reference, syntax different. (Usually pass by reference is the preferred technique.)

Pointers vs. reference variables

A reference variable behaves like an alias for a regular variable. To declare, place `&` after the type:

```
int i = 3;
int& j = i;        // j is a reference variable
j = 7;
cout << "i = " << i << endl;  // prints i = 7
```

Passing a reference variable to a function is the same as passing a normal variable by reference.

```
void passReference(int& i){
  i += 2;
}

passReference(j);
cout << "i = " << i << endl;  // prints i = 9
```

What to do with pointers

You can do lots of things with pointers in C++, many of which result in confusing code and hard-to-find bugs.

One of the main differences between Java and C++: Java doesn't have pointer variables (generally seen as a Good Thing).

One interesting use of pointers is that the name of an array is a pointer to the zeroth element in the array, e.g.,

```
double a[3] = {5, 7, 9};
double zerothVal = *a;      // has value of a[0]
```

The main usefulness of pointers for us is that they will allow us to allocate memory (create variables) dynamically, i.e., at run time, rather than at compile time.

Strings (the old way)

A string is a sequence of characters. In C and in earlier versions of C++, this was implemented with an array of variables of type `char`, ending with the character `\0` (counts as a single 'null' character):

```
    char aString[] = "hello";  // inserts \0 at end
```

The `cstring` library (`#include <cstring>`) provides functions to copy strings, concatenate them, find substrings, etc. E.g.

```
    char* strcpy(char* target, const char* source);
```

takes as input a string `source` and sets the value of a string `target`, equal to it. Note `source` is passed as `const` -- it can't be changed.

You will see plenty of code with old "C-style" strings, but there is now a better way: the `string` class (more on this later).

Example with `strcpy`

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
  char string1[] = "hello";
  char string2[50];
  strcpy(string2, string1);
  cout << "string2:  " << string2 << endl;
  return 0;
}
```

No need to count elements when initializing string with `"   "`.

Also `\0` is automatically inserted as last character.

Program will print: `string2 = hello`

Classes

A class is something like a user-defined data type. The class must be declared with a statement of the form:

```
class MyClassName {
  public:
    public function prototypes and
    data declarations;
    ...
  private:
    private function prototypes and
    data declarations;
    ...
};
```

Typically this would be in a file called **`MyClassName.h`** and the definitions of the functions would be in **`MyClassName.cc`**.

Note the semi-colon after the closing brace.

For class names often use UpperCamelCase.

A simple class: **`TwoVector`**

We might define a class to represent a two-dimensional vector:

```
class TwoVector {
  public:
    TwoVector();
    TwoVector(double x, double y);
    double x();
    double y();
    double r();
    double theta();
    void setX(double x);
    void setY(double y);
    void setR(double r);
    void setTheta(double theta);
  private:
    double m_x;
    double m_y;
};
```

Class header files

The header file must be included (`#include "MyClassName.h"`) in other files where the class will be used.

To avoid multiple declarations, use the same trick we saw before with function prototypes, e.g., in `TwoVector.h` :

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

class TwoVector {
  public:
    ...
  private:
    ...
};

#endif
```

Objects

Recall that variables are instances of a data type, e.g.,

```
double a;     // a is a variable of type double
```

Similarly, objects are instances of a class, e.g.,

```
#include "TwoVector.h"
int main() {
  TwoVector v;  // v is an object of type TwoVector
```

(Actually, variables are also objects in C++. Sometimes class instances are called “class objects” -- distinction is not important.)

A class contains in general both:

variables, called “data members” and

functions, called “member functions” (or “methods”)

Data members of a `TwoVector` object

The data members of a `TwoVector` are:

```
...
private:
  double m_x;
  double m_y;
```

Their values define the “state” of the object.

Because here they are declared `private`, a `TwoVector` object’s values of `m_x` and `m_y` cannot be accessed directly, but only from within the class’s member functions (more later).

The optional prefixes `m_` indicate that these are data members. Some authors use e.g. a trailing underscore. (Any valid identifier is allowed.)

The constructors of a **`TwoVector`**

The first two member functions of the **`TwoVector`** class are:

```
...
public:
  TwoVector();
  TwoVector(double x, double y);
```

These are special functions called constructors.

A constructor always has the same name as that of the class.

It is a function that is called when an object is created.

A constructor has no return type.

There can be in general different constructors with different signatures (type and number of arguments).

The constructors of a `TwoVector`, cont.

When we declare an object, the constructor is called which has the matching signature, e.g.,

```
TwoVector u;    // calls TwoVector::TwoVector()
```

The constructor with no arguments is called the “default constructor”. If, however, we say

```
TwoVector v(1.5, 3.7);
```

then the version that takes two **`double`** arguments is called.

If we provide no constructors for our class, C++ automatically gives us a default constructor.

Defining the constructors of a `TwoVector`

In the file that defines the member functions, e.g., `TwoVector.cc`, we precede each function name with the class name and `::` (the scope resolution operator). For our two constructors we have:

```
TwoVector::TwoVector() {
  m_x = 0;
  m_y = 0;
}
TwoVector::TwoVector(double x, double y) {
  m_x = x;
  m_y = y;
}
```

The constructor serves to initialize the object.

If we already have a `TwoVector v` and we say

```
TwoVector w = v;
```

this calls a “copy constructor” (automatically provided).

The member functions of **`TwoVector`**

We call an object's member functions with the "dot" notation:

```
TwoVector v(1.5, 3.7);    // creates an object v
double vX = v.x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
...
```

If the class had public data members, e.g., these would also be called with a dot. E.g. if **`m_x`** and **`m_y`** were public, we could say

```
double vX = v.m_x;
```

We usually keep the data members private, and only allow the user of an object to access the data through the public member functions. This is sometimes called "data hiding".

If, e.g., we were to change the internal representation to polar coordinates, we would need to rewrite the functions **`x()`**, etc., but the user of the class wouldn't see any change.

Defining the member functions

Also in `TwoVector.cc` we have the following definitions:

```
double TwoVector::x() const { return m_x; }
double TwoVector::y() const { return m_y; }
double TwoVector::r() const {
  return sqrt(m_x*m_x  + m_y*m_y);
}
double TwoVector::theta() const {
  return atan2(m_y, m_x);           // from cmath
}
...
```

These are called “accessor” or “getter” functions.

They access the data but do not change the internal state of the object; therefore we include `const` after the (empty) argument list (more on why we want `const` here later).

More member functions

Also in `TwoVector.cc` we have the following definitions:

```
void TwoVector::setX(double x) { m_x = x; }
void TwoVector::setY(double y) { m_y = y; }
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

These are “setter” functions.   As they belong to the class, they are allowed to manipulate the `private` data members `m_x` and `m_y`.

To use with an object, use the “dot” notation:

```
TwoVector v(1.5, 3.7);
v.setX(2.9);      // sets v's value of m_x to 2.9
```

Pointers to objects

Just as we can define a pointer to type **`int`**,

```
int* iPtr;       //  type "pointer to int"
```

we can define a pointer to an object of any class, e.g.,

```
TwoVector* vPtr;  // type "pointer to TwoVector"
```

This doesn't create an object yet!   This is done with, e.g.,

```
vPtr = new TwoVector(1.5, 3.7);
```

`vPtr` is now a pointer to our object. With an object pointer, we call member functions (and access data members) with **`->`** (not with "**`.`**"), e.g.,

```
double vX = vPtr->x();
cout << "vX = " << vX << endl;  // prints vX = 1.5
```

Forgotten detail: the **`this`** pointer

Inside each object's member functions, C++ automatically provides a pointer called **`this`**. It points to the object that called the member function. For example, we just saw

```
void TwoVector::setR(double r) {
  double cosTheta = m_x / this->r();
  double sinTheta = m_y / this->r();
  m_x = r * cosTheta;
  m_y = r * sinTheta;
}
```

Here the use of **`this`** is optional (but nice, since it emphasizes what belongs to whom). It can be needed if one of the function's parameters has the same name, say, **`x`** as a data member. By default, **`x`** means the parameter, not the data member; **`this->x`** is then used to access the data member.

Memory allocation

We have seen two main ways to create variables or objects:

(1) by a declaration (automatic memory allocation):

```
int i;
double myArray[10];
TwoVector v;
TwoVector* vPtr;
```

(2) using **`new`**: (dynamic memory allocation):

```
vPtr = new TwoVector();       // creates object
TwoVector* uPtr = new TwoVector();  // on 1 line
double* a = new double[n];    // dynamic array
float* xPtr = new float(3.7);
```

The key distinction is whether or not we use the **`new`** operator.

Note that **`new`** always requires a pointer to the **`new`**ed object.

The stack

When a variable is created by a "usual declaration", i.e., without **`new`**, memory is allocated on the "stack".

When the variable goes out of scope, its memory is automatically deallocated ("popped off the stack").

```
...
{
  int i = 3;          // memory for i and obj
  MyObject obj;       // allocated on the stack
  ...
}                     // i and obj go out of scope,
                      // memory freed
```

The heap

To allocate memory dynamically, we first create a pointer, e.g.,

```
        MyClass* ptr;
```

`ptr` itself is a variable on the stack. Then we create the object:

```
        ptr = new MyClass( constructor args );
```

This creates the object (pointed to by `ptr`) from a pool of memory called the "heap" (or "free store").

When the object goes out of scope, `ptr` is deleted from the stack, but the memory for the object itself remains allocated in the heap:

```
   {
     MyClass* ptr = new MyClass();     // creates object
     ...
   }   // ptr goes out of scope here -- memory leak!
```

This is called a memory leak. Eventually all of the memory available will be used up and the program will crash.

Deleting objects

To prevent the memory leak, we need to deallocate the object’s memory before it goes out of scope:

```
{
  MyClass* ptr = new MyClass();   // creates an object
  MyClass* a = new MyClass[n];    // array of objects
  ...

  delete ptr;  // deletes the object pointed to by ptr
  delete [] a; // brackets needed for array of objects
}
```

For every **`new`**, there should be a **`delete`**.

For every **`new`** with brackets **`[]`**, there should be a **`delete []`** .

This deallocates the object’s memory. (Note that the pointer to the object still exists until it goes out of scope.)

Dangling pointers

Consider what would happen if we deleted the object, but then still tried to use the pointer:

```
MyClass* ptr = new MyClass();   // creates an object
...
delete ptr;
ptr->someMemberFunction();      //  unpredictable!!!
```

After the object's memory is deallocated, it will eventually be overwritten with other stuff.

But the "dangling pointer" still points to this part of memory.

If we dereference the pointer, it may still give reasonable behaviour. But not for long! The bug will be unpredictable and hard to find.

Some authors recommend setting a pointer to zero after the **`delete`**. Then trying to dereference a null pointer will give a consistent error.

Static memory allocation

For completeness we should mention static memory allocation. Static objects are allocated once and live until the program stops.

```
void aFunction(){
  static bool firstCall = true;
  if (firstCall) {
    firstCall = false;
    ...                       // do some initialization
  }
  ...
}     // firstCall out of scope, but still alive
```

The next time we enter the function, it remembers the previous value of the variable `firstCall`. (Not a very elegant initialization mechanism but it works.)

This is only one of several uses of the keyword `static` in C++.

Operator overloading

Suppose we have two **TwoVector** objects and we want to add them. We could write an **add** member function:

```
TwoVector TwoVector::add(TwoVector& v){
  double cx = this->m_x + v.x();
  double cy = this->m_y + v.y();
  TwoVector c(cx, cy);
  return c;
}
```

To use this function we would write, e.g.,

```
TwoVector u = a.add(b);
```

It would be much easier if would could simply use **a+b**, but to do this we need to define the **+** operator to work on **TwoVector**s.

This is called operator overloading. It can make manipulation of the objects more intuitive.

Overloading an operator

We can overload operators either as member or non-member functions. For member functions, we include in the class declaration:

```
class TwoVector {
  public:
    ...
    TwoVector operator+ (const TwoVector&);
    TwoVector operator- (const TwoVector&);
    ...
```

Instead of the function name we put the keyword **`operator`** followed by the operator being overloaded.

When we say **`a+b`**, **`a`** calls the function and **`b`** is the argument.

The argument is passed by reference (quicker) and the declaration uses **`const`** to protect its value from being changed.

Defining an overloaded operator

We define the overloaded operator along with the other member functions, e.g., in `TwoVector.cc`:

```
TwoVector TwoVector::operator+ (const TwoVector& b) {
  double cx = this->m_x + b.x();
  double cy = this->m_y + b.y();
  TwoVector c(cx, cy);
  return c;
}
```

The function adds the *x* and *y* components of the object that called the function to those of the argument.

It then returns an object with the summed *x* and *y* components.

Recall we declared `x()` and `y()`, as `const`. We did this so that when we pass a `TwoVector` argument as `const`, we're still able to use these functions, which don't change the object's state.

Overloaded operators: asymmetric arguments

Suppose we want to overload `*` to allow multiplication of a `TwoVector` by a scalar value:

```
TwoVector TwoVector::operator* (double b) {
  double cx = this->m_x * b;
  double cy = this->m_y * b;
  TwoVector c(cx, cy);
  return c;
}
```

Given a `TwoVector v` and a `double s` we can say e.g. `v = v*s;`

But how about `v = s*v;`   ???

No! `s` is not a `TwoVector` object and cannot call the appropriate member function (first operand calls the function).

We didn't have this problem with `+` since addition commutes.

Overloading operators as non-member functions

We can get around this by overloading `*` with a non-member function.

We could put the declaration in `TwoVector.h` (since it is related to the class), but outside the class declaration.

We define two versions, one for each order:

```
TwoVector operator* (const TwoVector&, double b);
TwoVector operator* (double b, const TwoVector&);
```

For the definitions we have e.g. (other order similar):

```
TwoVector operator* (double b, const TwoVector& a) {
  double cx = a.x() * b;
  double cy = a.y() * b;
  TwoVector c(cx, cy);
  return c;
}
```

Restrictions on operator overloading

You can only overload C++'s existing operators:

Unary:
```
+   -   *   &   ~   !   ++   --   ->   ->*
```
Binary:
```
+   -   *   /   &   ^   &   |   <<   >>
+=  -=  *=  /=  %=  ^=  &=  |=  <<=  >>=
<   <=  >   >=  ==  !=  &&  ||  ,    []   ()
new    new[]    delete    delete[]
```

You cannot overload: `.` `.*` `?:` `::`

Operator precedence stays same as in original.

Too bad -- cannot replace `pow` function with `**` since this isn't allowed, and if we used `^` the precedence would be very low.

Recommendation is only to overload operators if this leads to more intuitive code. Remember you can still do it all with functions.

A different "static": static members

Sometimes it is useful to have a data member or member function associated not with individual objects but with the class as a whole.

An example is a variable that counts the number of objects of a class that have been created.

These are called static member functions/variables (yet another use of the word static -- better would be "class-specific"). To declare:

```
class TwoVector {
  public:
    ...
    static int totalTwoVecs();
  private:
    static int m_counter;
  ...
};
```

Static members, continued

Then in `TwoVector.cc` (note here no keyword `static`):

```
int TwoVector::m_counter = 0;  // initialize
TwoVector::TwoVector(double x, double y){
  m_x = x;
  m_y = y;
  m_counter++;   // in all constructors
}

int TwoVector::totalTwoVecs() { return m_counter; }
```

Now we can count our `TwoVector`s. Note the function is called with ***`class-name::`*** and then the function name. It is connected to the class, not to any given object of the class:

```
TwoVector a, b, c;
int vTot = TwoVector::totalTwoVecs();
cout << vTot << endl;        // prints 3
```

Oops #1: digression on destructors

The `totalTwoVec` function doesn't work very well, since we also create a new `TwoVector` object when, e.g., we use the overloaded `+`. The local object itself dies when it goes out of scope, but the counter still gets incremented when the constructor is executed.

We can remedy this with a destructor, a special member function called automatically just before its object dies. The name is `~` followed by the class name. To declare in `TwoVector.h`:

```
public:
  ~TwoVector();      // no arguments or return type
```

And then we define the destructor in `TwoVector.cc` :

```
TwoVector::~TwoVector(){  m_counter--;  }
```

Destructors are good places for clean up, e.g., deleting anything created with `new` in the constructor.

Oops #2: digression on copy constructors

The `totalTwoVec` function still doesn't work very well, since we should count an extra `TwoVector` object when, e.g., we say

```
TwoVector v;        // this increments m_counter
TwoVector u = v;    // oops, m_counter stays same
```

When we create/initialize an object with an assignment statement, this calls the copy constructor, which by default just makes a copy.

We need to write our own copy constructor to increment `m_counter`. To declare (together with the other constructors):

```
TwoVector(const TwoVector&);  // unique signature
```

It gets defined in `TwoVector.cc` :

```
TwoVector(const TwoVector& v) {
  m_x = v.x();  m_y = v.y();
  m_counter++;
}
```

Class templates

We defined the **`TwoVector`** class using **`double`** variables. But in some applications we might want to use **`float`**.

We could cut/paste to create a **`TwoVector`** class based on **`float`**s (very bad idea -- think about code maintenance).

Better solution is to create a class template, and from this we create the desired classes.

```
template <class T>       // T stands for a type
class TwoVector {
  public:
    TwoVector(T, T);     // put T where before we
    T x();               // had double
    T y();
    ...
};
```

Defining class templates

To define the class's member functions we now have, e.g.,

```
template <class T>
TwoVector<T>::TwoVector(T x, T y){
  m_x = x;
  m_y = y;
  m_counter++;
}

template <class T>
T TwoVector<T>::x(){ return m_x; }

template <class T>
void TwoVector<T>::setX(T x){
  m_x = x;
}
```

With templates, class declaration must be in same file as function definitions (put everything in `TwoVector.h`).

Using class templates

To use a class template, insert the desired argument:

```
   TwoVector<double> dVec;  // creates double version

   TwoVector<float> fVec;   // creates float version
```

`TwoVector` is no longer a class, it's only a template for classes.

`TwoVector<double>` and **`TwoVector<float>`** are classes (sometimes called "template classes", since they were made from class templates).

Class templates are particularly useful for container classes, such as vectors, stacks, linked lists, queues, etc. We will see this later in the Standard Template Library (STL).