Computing and Statistical Data Analysis
Lecture 5

Pointers

Strings

Introduction to classes and objects:

- Declaring classes
- A **`TwoVector`** class
- Creating objects
- Data members
- Member functions
- Constructors
- Defining the member functions
- Pointers to objects

Pointers

A pointer variable contains a memory address. It 'points' to a location in memory. To declare a pointer, use a star, e.g.,

```
int* iPtr;
double * xPtr;
char *c;
float *x, *y;
```

Note some freedom in where to put the star. I prefer the first notation as it emphasizes that `iPtr` is of type "pointer to `int`".

(But in `int* iPtr, jPtr;` only `iPtr` is a pointer--need 2 stars.)

Name of pointer variable can be any valid identifier, but often useful to choose name to show it's a pointer (suffix `Ptr`, etc.).

Pointers: the `&` operator

Suppose we have a variable `i` of type `int`:

```
int i = 3;
```

We can define a pointer variable to point to the memory location that contains `i`:

```
int* iPtr = &i;
```

Here `&` means “address of”. Don’t confuse it with the `&` used when passing arguments by reference.

Initializing pointers

A statement like

```
    int* iPtr;
```

declares a pointer variable, but does not initialize it. It will be pointing to some “random” location in memory. We need to set its value so that it points to a location we’re interested in, e.g., where we have stored a variable:

```
    iPtr = &i;
```

(just as ordinary variables must be initialized before use).

Dereferencing pointers: the `*` operator

Similarly we can use a pointer to access the value of the variable stored at that memory location. E.g. suppose **`iPtr = &i;`** then

```
 int iCopy = *iPtr;    // now iCopy equals i
```

This is called 'dereferencing' the pointer. The * operator means "value stored in memory location being pointed to".

If we set a pointer equal to zero (or **`NULL`**) it points to nothing. (The address zero is reserved for null pointers.)

If we try to dereference a null pointer we get an error.

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.

Wrapping up lecture 5

We are now almost done with the part of C++ that resembles C, i.e., the part that doesn't deal with classes or objects.

We've seen arrays (static and dynamic).

We've introduced pointers (but not yet done much with them).

We've seen briefly "C-style" strings.

We've introduced classes -- these behave like a sort of user defined data type.

Objects are instances of classes. In addition to holding data they have a set of functions that can act on the data. This is what distinguishes object-oriented programming from "procedural programming".