Computing and Statistical Data Analysis
Lecture 6

Arrays of objects

Memory allocation: static, automatic and dynamic

Memory leaks and dangling pointers

Operator overloading

Static member functions

        digression on destructors, copy constructors

Class templates

Dynamic arrays

An array's name is a pointer to its first element. We can create a "dynamic array" using the **`new`** operator:

```
double* array;
int len;
cout << "Enter array length" << endl;
cin >> len;            // array length set at run time
array = new double[len];
  ...
```

When we're done (e.g. at end of function where it's used), we need to delete the array:

```
delete [] array;
```

Arrays of objects

Just as we had arrays of **`int`**, **`double`**, etc., we can have arrays of objects. For example, with our **`TwoVector`** class we could have:

```
TwoVector v[10];

cin >> n;
TwoVector u[n];     //  size determined at runtime

TwoVector* aPtr = new TwoVector[n];  // default const.
TwoVector* bPtr = new TwoVector[n](1.0, 2.0);
```

In the C++98 standard, the size of **`u`** and **`v`** must be known at compile time. In C99 (implemented by gcc), array length can be variable (set at runtime).

So when do we use **`new`**? (Question relevant not just to arrays.) This depends on where and how long we want the object to live.

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 `TwoVectors`. 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).

Wrapping up lecture 6

You can drop the words “heap” and “stack” at cocktail parties.

You can define static member functions (and static data members) which pertain to the class, not any particular object.

You can overload operators and define class templates (or at least you can recognize them when you see them).

In our final lecture we will take a quick tour through some advanced features and useful tools.