C++

Introduction to classes and objects

Inheritance (quick tour)

Class templates

STL and the Standard C++ Library

Memory allocation: static, automatic and dynamic

Glen Cowan
RHUL Physics

Adapted by Martin Erdmann, RWTH Aachen

Classes

A class is something like a user-defined data type such as **`int`**, **`double`**,.. 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;
    ...
};
```

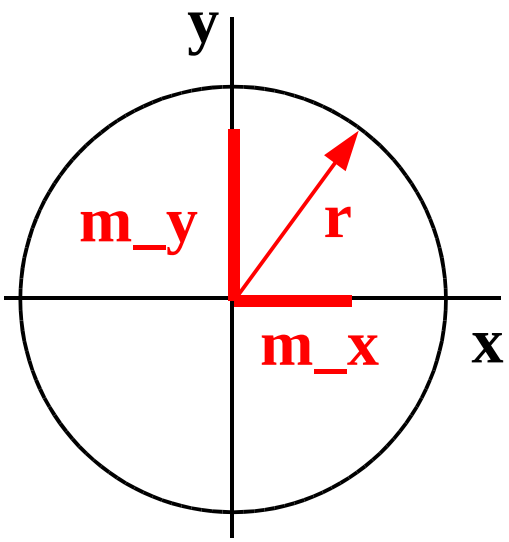
Typically this would be in a file called **`MyClassName.h`** and the definitions of the functions would be in **`MyClassName.cc`**.

Note the semi-colon after the closing brace.

A simple class: `TwoVector`

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

```
class TwoVector {
  public:
    // name(): "methods" ("member functions")
    TwoVector();              // ← "constructor"
    TwoVector(double x, double y);
    ~TwoVector();             // ← "destructor"
    double x() const;
    double y() const;
    double r() const;
    void setX(double x);
    void setY(double y);
    void setR(double r);
  private:
    double m_x;
    double m_y;          //"data members"
};
```



Class header files

To avoid multiple declarations, use the trick in **`TwoVector.h`** :

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H
class TwoVector {
  public:
    ...
  private:
    ...
};
#endif // TWOVECTOR_H
```

The header file must be included in files where the class will be used.

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

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 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 types 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.

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);
}
...
```

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 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 rho) {
  if ( this->r() != 0 ) {
    double cosTheta = m_x / this->r();
    double sinTheta = m_y / this->r();
    m_x = rho * cosTheta;
    m_y = rho * 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**.

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

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
...

v.setX(2.9);      // sets v's value of m_x to 2.9
cout << "vX = " << v.x() << endl; // prints vX = 2.9
...
```

Destructor

is 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(){ ... }
```

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

Inheritance

Often we define a class which is similar to an existing one. For example, we could have a class

```
class Animal {
  public:
    double weight();
    double age();
    ...
  private:
    double m_weight;
    double m_age;
    ...
};
```

Related classes

Now suppose the objects in question are dogs. We want

```
class Dog {
  public:
    double weight();
    double age();
    void bark();
  private:
    double m_weight;
    double m_age;
    ...
};
```

Dog contains some (perhaps many) features of the **Animal** class but it requires a few extra ones.

The relationship is of the form "X is a Y": a dog is an animal.

Inheritance

Rather than redefine a separate Dog class, we can derive it from Animal. To do this we declare in **Dog.h**

```
#include "Animal.h"
class Dog : public Animal {
  public:
    void bark();
    ...
  private:
    ...
};
```

Animal is called the "base class", **Dog** is the "derived class".

Dog inherits all of the public members of **Animal**. We only need to define **bark()**, etc.

Related classes: X has a Y

derive another class Fleas from Animal: **Fleas.h**

```
#include "Animal.h"
class Fleas : public Animal {
  public:
    int NumberOfFleas();
    ...
};
```

dogs have fleas:

```
#include "Animal.h"
#include "Fleas.h"
class Dog : public Animal {
  public:
    void bark();
    Fleas fleas;
};
```

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
    ...
  private:
    T m_x;
    T m_y;
};
```

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).

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;
}

template <class T>

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

With templates, class declaration must be in same file as function definitions, todays style: put definition in **`TwoVector.icc`**,

`#include "TwoVector.icc"`

at the end of the declarations in **`TwoVector.h`**

The C++ Standard Template Library (STL)

We've already seen parts of the standard library such as **`iostream`** and **`cmath`**. Here are some more:

What you `#include`	What it does
`<algorithm>`	useful algorithms (sort, search, ...)
`<complex>`	complex number class
`<vector>`	often used instead of arrays
`<map>`	container for key-value pairs
`<string>`	proper strings (better than C-style)

Most of these define classes using templates, i.e., we can have a STL vector of objects or of type **`double, int, float`**, etc. They form what is called the Standard Template Library.

Using STL **vector**

Here is some sample code that uses the **vector** class. Often a **vector** is better than an array.

```
#include <vector>
using namespace std;
int main() {
  vector<double> v;        // uses template
  double x = 3.2;
  v.push_back(x);          // element 0 is 3.2
  v.push_back(17.0);       // element 1 is 17.0
  vector<double> u = v;    // assignment
  int len = v.size();
  for (int i=0; i<len; i++){
    cout << v[i] << endl;    // like an array
  }
  v.clear();                 // remove all elements
  ...
```

Using `string`

Here is some sample code that uses the `string` class (much better than C-style strings):

```
#include <string>
using namespace std;
int main() {
  string a, b, c;
  string s = "hello";
  a = s;                 // assignment
  int len = s.length();      // now len = 5
  bool sEmpty = s.empty();  // now sEmpty = false
  b = s.substring(0,2);      // first position is 0
  cout << b << endl;         // prints hel
  c = s + " world";          // concatenation
  cout << s << endl;
  ...
```

Memory management

```
{
 static int K;
}
```

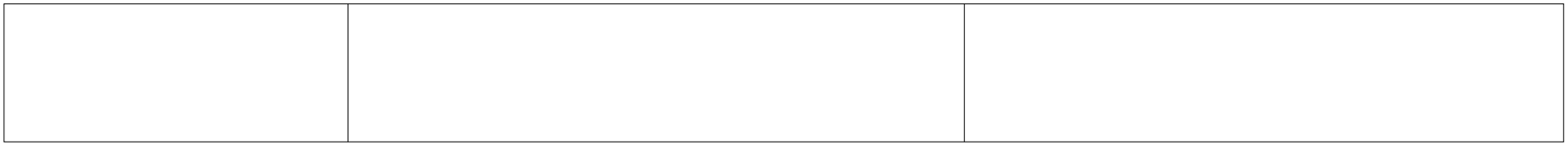
K *stays*

```
{
 TwoVector b;
}
```

b *is deleted at* }

```
{
 TwoVector* b
 =new TwoVector;
 ...
 delete b;
}
```

delete b *yourself*



„static“

„stack“
(automatic)

„heap“
(dynamic)

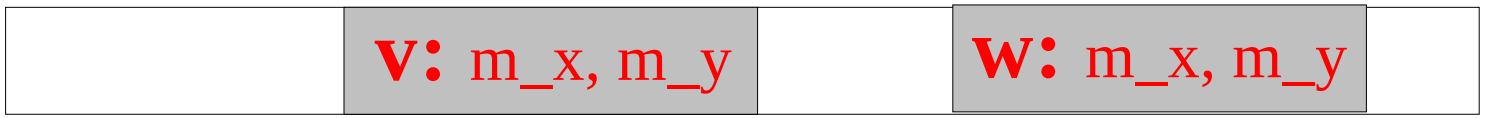
Objects

Recall that variables are instances ("alive in memory") of a data type, e.g.,

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

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

```
TwoVector v;  // v is an object of type TwoVector
TwoVector w;  // w is another object of the same type
```



new & delete

TwoVector* **b** = **new** TwoVector;

	pointer to **b**	m_x, m_y of **b**
„static“	**„stack“**	**„heap“**

delete b;

	pointer to **b**	

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 **->** (short cut for "**(*vPtr).**"), e.g.,

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

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
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.

Better way:

```
delete ptr;
ptr = 0;
if (ptr !=0 ) ptr->someMemberFunction(); // ok
```

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++.

Take away

Classes: behave like a sort of user defined data type. 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".

You can recognize: inheritance, class templates, standard C++ classes such as `vector` and `string` when you see them.

Automatic, dynamic, static memory allocation: you can drop the words "heap" and "stack" at cocktail parties.

Saaluebung this Thursday: Homework solutions & C++ Classes

Next Tuesday we start probability and statistical data analysis. This will give us many opportunities to develop and use C++ analysis tools.