

# SUPA Graduate C++ Course

## Lecture 3

---

Sarah Boutle  
University of Glasgow

Many thanks to Adrian Buzatu, S. Allwood-Spiers

# News

---

- **First assignment uploaded to course page**
  - Due: 31st October before the 2nd lab session
  - Please upload your source files to course page
- **Labs:** started this week
  - Monday afternoons, 2-5pm
  - Room 220a, Kelvin Building, University of Glasgow
  - Can be used to get help for the assignments, but not compulsory
  - Please indicate if you intend to come in person

**PLEASE UPDATE**

<http://doodle.com/poll/8n84fuqkt2bvzqtu>

  - Probably most useful to you to bring your own laptop, but there are (a limited number of) linux machines available
- **Next assignment will be available on Monday**
  - Watch out for the SUPA mail

# Last Week's Lecture

---

- More basic elements
  - Writing to a file
  - Formatting numbers
  - Arrays
  - Pointers
- Introduction to Classes
  - Simple example
  - Objects
  - Members
  - Constructors

# Class example so far

---

## TwoVector.h

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

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;
};

#endif
```

## TwoVector.cc

```
#include "TwoVector.h"
#include <cmath>
#include <iostream>

TwoVector::TwoVector() {
    m_x = 0;
    m_y = 0;
}

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

# Class example so far

---

## TwoVector.h

```
#ifndef TWOVECTOR_H
#define TWOVECTOR_H

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;
};

#endif
```

## TwoVector.cc

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

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;
}
```

# Class example so far

---

main.cc

```
#include <iostream>
#include "TwoVector.h"

using namespace std;

int main() {

    TwoVector v(1.5, 3.7);
    TwoVector u(1.4, 8.8);
    v.setX(2.9);
    u.setR(1.3);

    cout << "Vector 1, R=" << v.r() << ", Theta=" << v.theta() << endl;
    cout << "Vector 2, R=" << u.r() << ", Theta=" << u.theta() << endl;

    return 0;
};
```

# This Week's Lecture

---

- More classes
  - Constructing objects
  - Memory allocation
  - delete
  - Operator overloading
  - Static members
  - Destructors
  - Copy Constructors

# Pointers to objects

---

Last week I introduced pointers e.g. to type `int`

```
int* iPtr;
```

We can define a pointer to an object of any class too:

```
TwoVector* vPtr;
```

This doesn't create the object yet, this is done with:

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

now `vPtr` is a pointer to the object.

With an object pointer, we call member functions with a `->` not a `."`, e.g.

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

prints `vX = 1.5`



# Constructing objects

---

We have now seen two ways of constructing objects:

(i) by declaration:

```
TwoVector v(1.5, 3.7);  
int i;  
double myArray[5];  
TwoVector* vPtr;
```

(ii) using **new** (dynamic memory allocation):

```
vPtr = new TwoVector(1.5, 3.7);  
TwoVector* uPtr = new TwoVector();  
float* xPtr = new float(1.5);
```

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

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

# Memory allocation

---


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

Objects created this way go harmlessly out of scope, it's memory is automatically deallocated or popped off the stack

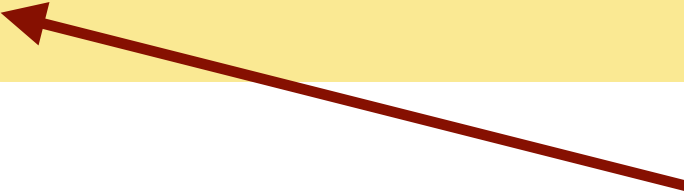
Any pointers to the object are then invalid

```
...  
{  
    int i = 3;  
    MyObject obj;  
    TwoVector v(1.5, 3.7);  
    ...  
}
```

memory is allocated  
for i and obj and v on  
the stack



i, obj and v go out of  
scope and the  
memory is freed



# Memory allocation

---

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

```
TwoVector* vPtr;
```


then `vPtr` is a variable on the stack. Then we create an object:

```
vPtr = new TwoVector();
```

This creates the object from a pool of memory called the "**heap**" or free store.

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

```
{  
    TwoVector* vPtr = new TwoVector();  
    .....  
}
```



`vPtr` goes out of scope, Memory Leak!

Eventually all the memory available will be used up and the program will crash

# Deleting objects

---

To prevent the memory leak we must deallocate the object's memory before it goes out of scope, this returns the memory to the heap:

```
{  
    TwoVector* vPtr = new TwoVector();  
    TwoVector* a = new TwoVector[n];  
    .....  
  
    delete vPtr;  
    delete [] a;  
}
```

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: the pointer to the object still exists until it goes out of scope.

# Dangling pointers

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

```
TwoVector* vPtr = new TwoVector();  
.....  
  
delete vPtr;  
vPtr->SetR(1.2);
```

This is unpredictable.....

After the objects memory is deallocated, it will be eventually overwritten with other stuff.

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

Some recommend to set pointer to zero after **delete**

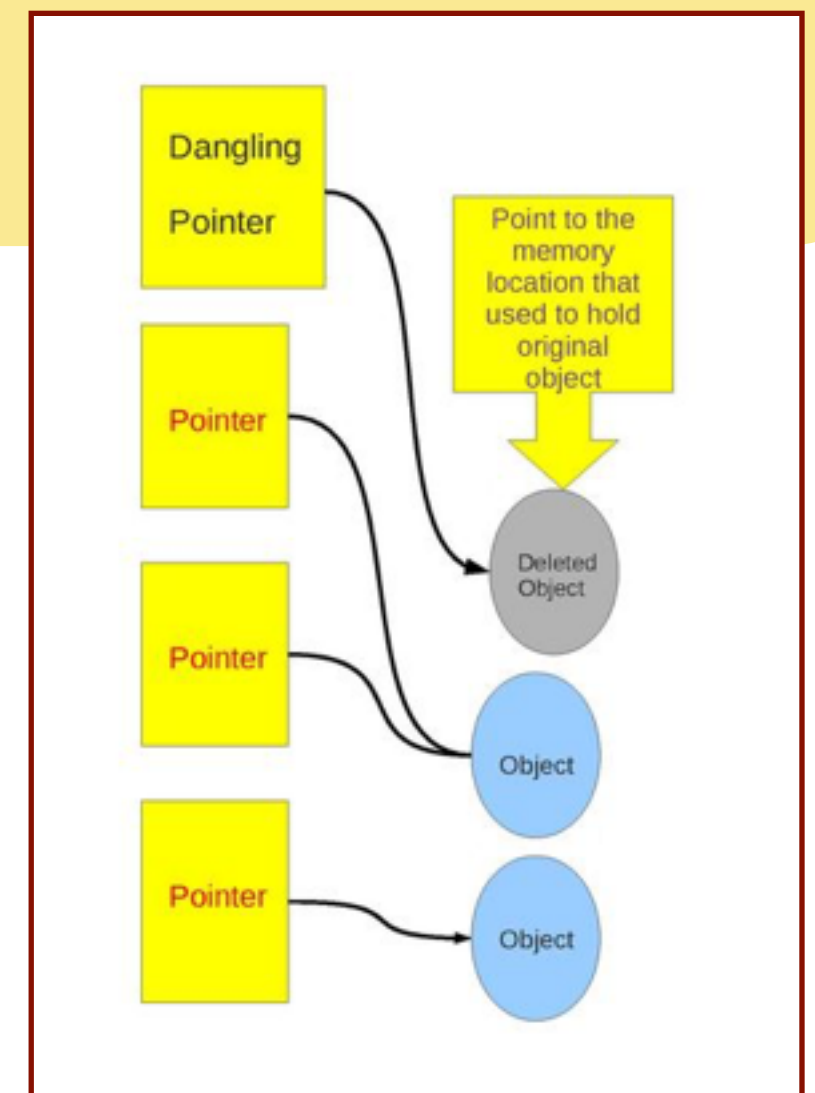


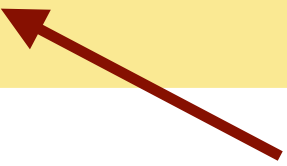
image from wikipedia

# Dangling pointers

Other examples:

```
int* iPtr = NULL;


{
    int i = 2;
    iPtr = &i;
}
```



*i goes out of scope, iPtr is dangling*

```
int* func()
{
    int num = 123;
    return &num;
}
```

*returns address of a stack-allocated local variable: once the called function returns, the space for this variables gets deallocated*



Attempts to read from the pointer may still return correct value for a while after calling the function but eventually it will be overwritten

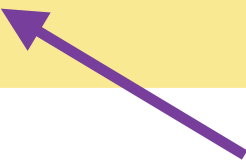
# Dangling pointers

---

Other examples:

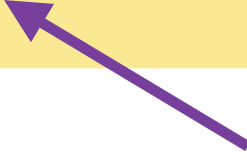
```
int* iPtr = NULL;

{
    int i = 2;
    iPtr = &i;
}
```



solution, set: `vPtr = NULL` before exiting block

```
int* func()
{
    int num = 123;
    return &num;
}
```




one solution: if a pointer to `num` must be returned, `num` must have scope beyond the function—it might be declared as **static**.

# Static memory allocation

Static objects are allocated once, and live until the program stops:

```
int* func() {  
    static int num = 123;  
    return &num;  
}
```



num goes out of scope,  
but is still alive

```
void func() {  
    static int x = 0;  
    x++;  
    std::cout << x << std::endl;  
}
```

```
int main() {  
    func(); // prints 1  
    func(); // prints 2  
    func(); // prints 3  
    func(); // prints 4  
    func(); // prints 5  
    return 0;  
}
```

Each time we enter the function, it remembers the previous value of the variable x.

Not very elegant initialization but it does work.



# Operator overloading

---

Simple arithmetic and other functionality can be implemented in a class

```
float x=0, y=5, z=3;  
x = ++y * z;  
x = x/2.0;
```

We might want to do:

```
TwoVector v1(2.0, 3.0);  
TwoVector v2(2.0, 3.0);  
TwoVector v3 = v1 + v2;
```

We could define a 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;  
}
```

```
TwoVector v3 = v1.add(v2);
```

# Operator overloading

But actually, we can overload operators:

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

put the keyword `operator` before  
the operator to be overloaded

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

When we say:

```
TwoVector v3 = v1 + v2;
```

`v1` calls the function and `v2` is the argument

# Operator overloading

---

The define the overloaded operator along with the other member functions 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;  
}
```

This function adds the x and y components of the object that called it to those of the argument.

Then it returns another one with those summed components.

```
TwoVector v1(2.0, 3.0);  
TwoVector v2(2.0, 3.0);  
TwoVector v3 = v1 + v2;
```

# Overloaded operators: asymmetric arguments

---

Suppose we want to overload `*` to multiply 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;  
}
```

Consider:

```
TwoVector v1(2.1, 5.5);  
double s = 2;  
  
v = v*s;  
v = s*v;
```

we cannot say `s*v` as `s` is not a `TwoVector` object and cannot call the appropriate member function.....

# Overloading operators as non-member functions

---

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

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

It makes sense to put this in `TwoVector.h` since it is related to the class but outside the class declaration.

And then define both:

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

# Operator overloading

---

Operators you can overload:

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

Operators you can't overload:

```
.    .*    ?:    ::
```

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

# Class declaration: in header file

```
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);  
    TwoVector operator+ (const TwoVector&);  
    TwoVector operator- (const TwoVector&);  
  
private:  
  
    double m_x;  
    double m_y;  
  
};
```

REMINDER

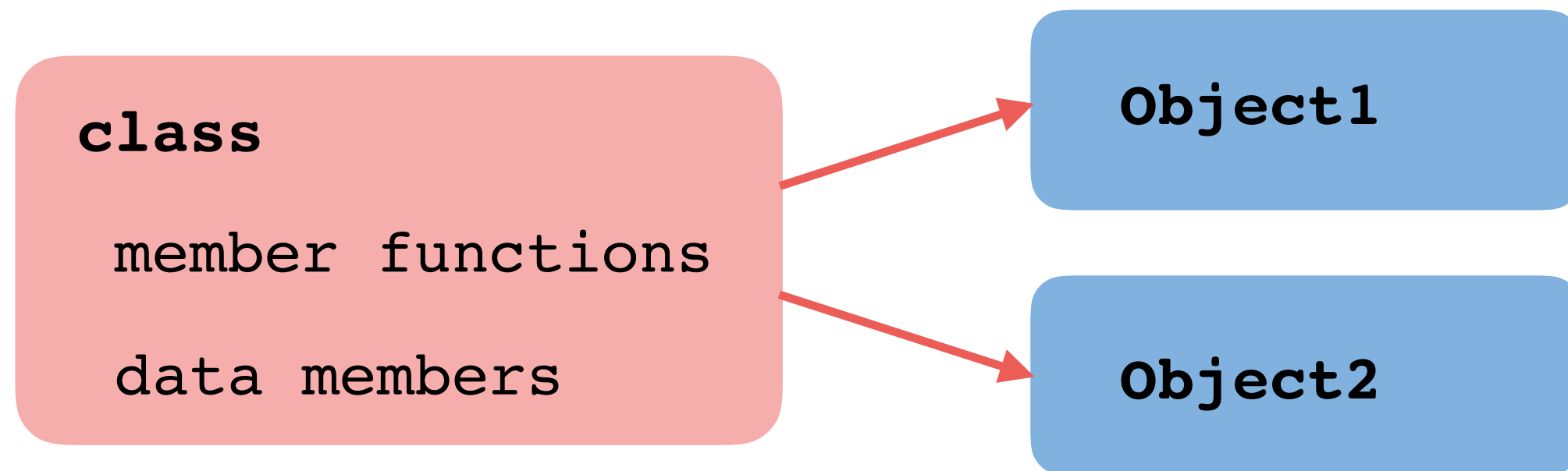
Member functions

Data members

# Static members

---

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



For example, you might want to have a variable which counts the number of objects of the class that have been created

These are called **static** member functions/variables



# Static members: example

---

TwoVector.h

```
class TwoVector {
```

```
public:
```

```
    TwoVector();
```

```
    TwoVector(double x, double y);
```

```
    double x();
```

```
    double y();
```

```
    double r();
```

```
    .
```

```
    .
```

```
    static int totalTwoVecs();
```



```
private:
```

```
    double m_x;
```

```
    double m_y;
```

```
    static int m_counter;
```



```
};
```

# Static members: example

TwoVector.cc

```
int TwoVector::m_counter = 0;
```

← initialize static data member

```
TwoVector::TwoVector() {
```

```
    m_x = 0;
```

```
    m_y = 0;
```

```
    m_counter++;
```

```
}
```

← inside every constructor

```
TwoVector::TwoVector(double x, double y){
```

```
    m_x = x;
```

```
    m_y = y;
```

```
    m_counter++;
```

```
}
```

← inside every constructor

```
int TwoVector::totalTwoVecs() {
```

```
    return m_counter;
```

```
}
```

← define static member function

# Static members: example

---

Now we can use it to count the `TwoVector` objects:

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

prints 3

---

In this example, `totalTwoVecs` doesn't really work well.

The number is incremented every time a constructor is executed.

Recall, e.g. the overloaded `+` operator....

```
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;  
}
```

← constructor  
executed!

# Destructors

---

We can remedy this by defining a **destructor**.

This is a special member function which gets called automatically just before its object dies.

`TwoVector.h`

```
class TwoVector {  
  
public:  
  
    TwoVector();  
    TwoVector(double x, double y);  
    ~TwoVector();  
    .  
    .  
}
```

~ precedes class name  
no arguments or return  
type



`TwoVector.cc`

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

Destructors are good  
places for cleaning up.

# Copy constructors

Back to static members.... `totalTwoVecs` still doesn't really work well. Consider:

```
TwoVector v;           ← increments m_counter
TwoVector u = v;       ← m_counter stays the same
```

The assign statement here, just calls the **copy constructor**, which by default just makes a copy.

If we want `m_counter` to increment properly, we need to write our own copy constructor

## TwoVector.h

```
class TwoVector {
public:
    TwoVector();
    TwoVector(double x, double y);
    TwoVector(const TwoVector&);
    ~TwoVector();
}
```

## TwoVector.cc

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

# Review

---

- More classes
  - Constructing objects
  - Memory allocation
  - delete
  - Operator overloading
  - Static members
  - Destructors
  - Copy Constructors

# 2nd Assignment

---

Due Monday 7th November before 3rd lab session

The file input.txt contains a list of planets (name, mass in kg in scientific notation, average distance to the Sun in astronomic units, where 1.0 is the distance from the Earth to the sun).

1. Print on each line the name of the planets, in alphabetical order.
2. Print on each line the name of the planet followed by its mass, in order of the mass. Same for distance to the sun.
3. Print the planet with the smallest and largest mass. Same for distance to the sun.
4. Compute the weighted average distance to the Sun of all the planets, defined as sum over the planets of mass times distance, and all divided by the sum of the masses of all planets

Tips:

1. scientific notation, useful for large numbers:

<http://www.cplusplus.com/forum/general/9616/>

2. `std::map<std::string, double>` to store pairing between a word and a number, when iterating over a map, the items come ranked, and if they are words, that means alphabetical order.

3. Use algorithms on the STL containers, like getting max and min from an `std::vector<double>`  
<http://stackoverflow.com/questions/10158756/using-stdmax-element-on-a-vectordouble>

# Standard C++ library

---

We've already seen some of the standard library, e.g. `iostream` and `cmath`. Here are some more:

What you <code>#include</code>	What it does
<code>&lt;algorithm&gt;</code>	useful algorithms (sort, search...)
<code>&lt;complex&gt;</code>	complex number class
<code>&lt;list&gt;</code>	a linked list
<code>&lt;stack&gt;</code>	a stack (push, pop, etc)
<code>&lt;string&gt;</code>	strings
<code>&lt;vector&gt;</code>	an alternative to arrays

Most of these define classes using templates, i.e. we can have a vector of objects or of type `double`, `int`, etc....



# Using vector

---

Often a `vector` is better than an `array`

- Larger than an array.
  - Require header information to keep track of elements
- Flexible size
  - Container manages memory allocation
- Elements can be accessed with an index `[i]` or via an iterator.

# Using vector

---

```
#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
    ...
```

# Sorting elements of a vector

---

```
#include <vector>
#include <algorithm>

using namespace std;

bool descending(double x, double y){ return (x>y); }

int main() {
    ...
    // u, v are unsorted vectors; overwritten by sort.
    // Default sort is ascending; also use user-
    // defined comparison function for descending order.

    sort(u.begin(), u.end());
    sort(v.begin(), v.end(), descending);
}
```

# Iterators

---

To loop over the elements of a vector `v`, we could do:

```
vector<double> v = ... // define vector v

for (int i=0; i<v.size(); i++){
    cout << v[i] << endl;
}
```

Or, we can use an iterator, which is defined by the vector class  
It's a type of smart pointer....

```
vector<double> v = ... // define vector v

vector<double>::iterator it;

for (it = v.begin(); it != v.end(); ++it){
    cout << *it << endl;
}
```