C++ - Pre-lecture 9

Advanced topics

Caterina Doglioni, PHYS30762 - OOP in C++ 2024

MANCHESTER
1824
The University of Manchester

Part 0.1: in this lecture

In this lecture

- **Part 1:** static data and functions [video 1, includes examples]
- **Part 2:** function and class templates [video 2]
- **Part 3:** namespaces [video 3]
- **Part 4:** recap of interface/implementation, new: how templates work in this case [video 4]
- **Part 5:** templates and friends [video 5, mainly an example for concepts from video 3 and previous lectures]
- Those are all **advanced features** of C++ (more in next pre-lecture)
  - Practicing some of these in the project will leads to extra marks
  - Be explicit in your report why you're using them / what design problem they solve

Part 1 / Video 1:
static data

Static data

- Recall: an **object** is an **instance** of a class
    - Each object has unique set of values for data members
- Object may not exist for the lifetime of the program (e.g. object destroyed when exiting function or loop - goes *out of scope)*
- *How to keep information around for all objects of a given class to modify?*
    - We may want all objects from a given class to share access to (and be able to modify) some data ("global data")
- Need to create **static data members** - memory is reserved for lifetime of program and can be accessed by all objects
- Here is how to implement one...

MANCHESTER 1824
The University of Manchester

Static data: implementation

```cpp
#include<iostream>
class my_class
{
private:
  int x{};
  static int n_objects;
public:
  my_class() : x{} {n_objects++;}
  my_class(int x_in) : x{x_in} {n_objects++;}
  ~my_class() {n_objects--;}
  void show() {std::cout<<"x="<<x<<", n_objects="<<n_objects<<std::endl;}
};
int my_class::n_objects{}; // define static data member outside class!
void test()
{
  my_class a3{3};
  a3.show();
}
int main()
{
  my_class a1{1};
  a1.show();
  my_class a2{2};
  a2.show();
  test();
  a1.show();
  return 0;
}
```

```
urania277@medram Prelecture9 % ./staticdata
x=1, n_objects=1
x=2, n_objects=2
x=3, n_objects=3
x=1, n_objects=2
```

Code on GitHub at: Prelecture9/staticdata.cpp

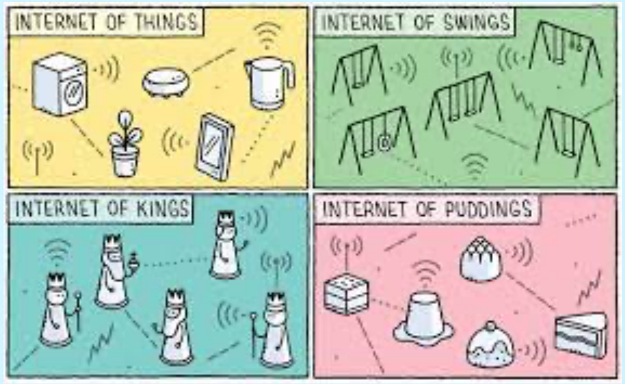
- We **declare** a data member within the class with the keyword `static`
- Then we **define** and initialise it outside the class
    - this is where memory is set aside
    - declaration: use `static`
      definition: do not use static
- Every object instantiated from our class can see the same `nobjects` and modify it
- In our example, we used it for the current number of objects (which changes in class constructor and destructor)
    - this is reflected in the output

MANCHESTER 1824
The University of Manchester

Static data: when to use/extra examples

- Static data within a class is used when:
  - you have a variable that needs to be accessed and kept track of by all the objects of a given class. Examples:
    - a *counter* (see previous code snippet)
    - a *state* that you need to keep track of, common to all objects
  - you want that variable to be only modifiable (/ accessible) by the class, but you need it to keep existing independently from the class
    - a global *state* that the class modifies and other code is accessing
    - do not use global variables! static variables are a much safer bet
  - you can also use static with functions, let's see a more advanced example…

Let’s code a laundry room simulator



T. Gauld

- Two classes, `washing_machine` and `dryer`, in a “smart home” (*Internet Of Things-powered!*)
  - once the cycle is over, your laundry machine/dryer disappears
    - Note: it would be really inconvenient if actual objects went out of scope in real life
- You are out and don’t want to spend too much electricity because it’s savings hour
  - you can’t go look in your laundry room so you can’t check if the machines are there

- So you ask your IoT device, and that checks the static variable that lets you know whether the laundry machine or the dryer is on
  - if it is, you go home to turn them off and save on your electricity bill!

Part 2 / Video 2:
function and class templates

What is a template?

```
template<typename T>
class box {
    …
    T catContent = …;
}
```

- We have seen a number of these already, with a special syntax
    - most used example: `vector<int>`
    - it’s a container, and it can contain very different things
        - from numbers to strings to custom classes
- Conceptually, a template is a **blueprint for a generic function or class,** or a cookie-cutter to **build a family of similar functions or classes**
- The compiler deduces what kind of object you plug into the template
    - this can lead to very long and tortuous errors
- **Important implementation note:** templates do not divide header and implementation!

MANCHESTER 1824
The University of Manchester

Template functions

- Templates allow functions and classes to be created for **generic datatypes**

- Consider functions first - example (remember lecture 2)

```
double maxval(double a, double b) {return (a>b) ? a : b;}
int maxval(int a, int b) {return (a>b) ? a : b;}
```

- Used overloading to re-write function for integer parameters

- Second function performs identical task to first (maximum of two numbers) but with different data type

  - Used ternary operator (test ? iftrue : iffalse)— good for true-or-false tests returning an lvalue

- Lots of code duplication, when only the type is changing!

Template functions

- Overloading is good but laborious if you have to write a function for every type)
- Solution: write a **single function template**

```cpp
// PL9/functiontemplate.cpp
// Demonstration of function templates
// Niels Walet, Last modified 04/12/2019
#include<iostream>
template <class c_type> c_type maxval(c_type a, c_type b)
{
  return (a > b) ? a : b;
}
int main()
{
  double x1{1}; double x2{1.5};
  std::cout<<"Maximum value (doubles) = "<< maxval<double>(x1,x2)<<std::endl;
  int i1{1}; int i2{-1};
  std::cout<<"Maximum value (ints) = "<< maxval<int>(i1,i2)<<std::endl;
  return 0;
}
```

Code on GitHub at: Prelecture9/functiontemplate.cpp

```
urania277@medram Prelecture9 % ./functiontemplate
Maximum value (doubles) = 1.5
Maximum value (ints) = 1
```

How to write template functions

- Before function declaration/definition:

```
template <class c_type> c_type maxval(c_type a, c_type b)
```

placeholder for the templated type

return type

function name

arguments

- The statement `<class c_type>` tells the compiler the template is for a generic type `c_type` - known as a template parameter

- The remainder is like any function, with specific datatype is replaced with `c_type`

- NB: the compiler will not use the function template until an instance is created in the code (known as a template function)

  - a template function will be created, replacing `c_type` with `double`

MANCHESTER 1824
The University of Manchester

Template classes

- Same principle applies to classes
- Example class for a pair of integers
- What if I want another one with doubles?

```cpp
// PL9/twonum.cpp
// Define a class to hold a pair of numbers
// Niels Walet, Last modified 03/12/2019
#include<iostream>
class pair_of_numbers
{
private:
  int x;
  int y;
public:
  pair_of_numbers() : x{},y{} {}
  pair_of_numbers(int xx, int yy) : x{xx},y{yy} {}
  int add() {return x+y;}
  int sub() {return x-y;}
};
int main()
{
  int x{1},y{2};
  pair_of_numbers ip{x,y};
  std::cout<<"x+y="<<ip.add()<<std::endl;
  std::cout<<"x-y="<<ip.sub()<<std::endl;
  return 0;
}
```

Code on GitHub at: Prelecture9/twonum.cpp

MANCHESTER 1824 The University of Manchester

Template classes

```cpp
// PL9/twonum2.cpp
// Define a class template to hold a pair of numbers
// Niels Walet, Last modified 03/12/2019
#include<iostream>
template <class c_type> class pair_of_numbers {
private:
  c_type x,y;
public:
  pair_of_numbers() : x{},y{} {}
  pair_of_numbers(c_type xx, c_type yy) : x{xx},y{yy} {}
  c_type add() {return x+y;}
  c_type sub() {return x-y;}
};
int main()
{
  int x{1};
  int y{2};
  double a{-1.5};
  double b{-2.5};
  // Use class template for object representing pair of integers
  pair_of_numbers<int> ip{x,y};
  std::cout<<"x+y="<<ip.add()<<std::endl;
  std::cout<<"x-y="<<ip.sub()<<std::endl;
  // Now for  a pair of doubles
  pair_of_numbers<double> dp{a,b};
  std::cout<<"a+b="<<dp.add()<<std::endl;
  std::cout<<"a-b="<<dp.sub()<<std::endl;
  return 0;
}
```

Code on GitHub at: Prelecture9/twonum2.cpp

How to write template classes - I

- Like the function, the class template now has template parameters

```
template <class c_type > class pair_of_numbers
```

- Then replace appropriate data type in class with `T`, e.g. for parameterised constructor

```
pair_of_numbers(c_type xx, c_type yy) : x{xx},y{yy} {}
```

- Instances of the class are created as

```
pair_of_numbers<int> ip{x,y};
```

- Then for an object of double type, we write

```
pair_of_numbers<double> dp{a,b};
```

- Again, compiler uses class template to create t**wo instances (or template classes)**, one for each type, as required

- We have seen this already: `vector<double>` (`vector` is a class template and `vector<double>` creates a template class for vector of doubles)

MANCHESTER
1824
The University of Manchester

How to write template classes - II

- **How do I separate templates into interface and implementation?**
- If a member function contains a parameter that is an instance of a template class (i.e. object), must refer to its type as `twonum<c_type>`
- Compiler will then replace `c_type` with `int`, `double`, etc. as appropriate when creating template class
- Example: write a simple copy constructor

```
twonum(const twonum<c_type> &tn) x=tn.x; y=tn.y;
```

- For member functions defined outside class, we prototype inside class as before, e.g.

```
twonum(const twonum<c_type> &tn); // prototype
```

- Then we define the function itself as follows (including class name before ::)

```
template <class c_type> twonum<c_type>::twonum(const
twonum<T> &tn)
{
  x=tn.x; y=tn.y;
}
```

MANCHESTER 1824 The University of Manchester

Why not overloading/polymorphism…?

Let's start thinking of functions

- *Templates*: perform same action on different types
- *Overloading*: perform different action depending on types
- *Polymorphism*:
    - In general, there are a number of cases in which polymorphism is preferred to templates in terms of readability…
    - …but there are other cases in which you really have different types that have nothing to do with each other and you want to perform the same operations on them

```
class box {
…
cat* catContent = …;
}
```

Templates don't split interface and implementation

- Using the methods for splitting code in multiple files used so far will cause linker errors when using templates
- This is because template classes and functions are **generated on demand**
- There is a consequence: **compiler** needs to **see both declarations and definitions in the same file** as the code that uses the templates.
- The default 'house style' rule above was that there are no function definitions inside a header file. You are expected to **break this for templates**.

- What to do to put implementation outside the class (always good practice): below namespace (containing the class definition) in header file,
  - Add `using namespace my_namespace` (or equivalent) - see next video;
  - Then add all template function definitions;
  - Include this header file in any .cpp file where objects are instantiated from this class template.

Templates: no splitting (see video 4)

```cpp
// PL9/twonum3.h
// Header file to define a class template to hold a pair of numbers
// Niels Walet, Last modified 03/12/2019
#ifndef TWO_NUM_H // Will only be true the once!
#define TWO_NUM_H
namespace two_num
{
  template <class c_type> class pair_of_numbers {
  private:
    c_type x;
    c_type y;
  public:
  pair_of_numbers() : x{},y{} {};
  pair_of_numbers(const c_type xx, const c_type yy) : x{xx},y{yy} {};
    c_type add();
    c_type sub();
  };
}
using namespace two_num;
template<class c_type> c_type pair_of_numbers<c_type>::add() {return x+y;};
template<class c_type> c_type pair_of_numbers<c_type>::sub() {return x-y;};
#endif
```

Find this code at: Prelecture9/twonum3.h

- Add using namespace my_namespace (or equivalent);
- Then add all template function definitions;
- Include this header file in any .cpp file where objects are instantiated from this class template.

Find this code at: Prelecture9/twonum3.cpp

```cpp
// PL9/twonum3.cpp
// Define a class template to hold a pair of numbers (header file)
// Niels Walet, Last modified 03/12/2019
#include<iostream>
#include"twonum3.h"
using namespace two_num;
int main()
{
  int x{1},y{2};
  double a{-1.5},b{-2.5};
  // Use class template for object representing pair of integers
  pair_of_numbers<int> ip(x,y);
  std::cout<<"x+y="<<ip.add()<<std::endl;
  std::cout<<"x-y="<<ip.sub()<<std::endl;
  // Now for  a pair of doubles
  pair_of_numbers<double> dp(a,b);
  std::cout<<"a+b="<<dp.add()<<std::endl;
  std::cout<<"a-b="<<dp.sub()<<std::endl;
  return 0;
}
```

- In order to use the templated class in the main() function, still need to add using namespace my_namespace (or equivalent);
- Remember the health warnings as of earlier: always safer to use namespace::
- Can also use (better practice):
  - using two_num::pair_of_numbers

Part 3 / Video 3:
namespaces

Name collisions, and how to avoid them

- Imagine if we tried to include two classes with the same name:

```
#include<iostream>
class my_class
{
private:
  int x;
public:
  my_class() : x{} {}
  my_class(int xx) : x{xx} {}
  ~my_class(){}
  void show(){std::cout<<"x="<<x<<std::endl;}
};
class my_class
{
private:
  int x,y;
public:
  my_class() : x{},y{} {}
  my_class(int xx, int yy) : x{xx},y{yy} {}
  ~my_class(){}
  void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
};
int main()
{
  return 0;
}
```

Code on GitHub at: Prelecture9/namespacewrong.cpp

Name collisions, and how to avoid them

- Imagine if we tried to include two classes with the same name:

```
1   #include<iostream>
2   class my_class
3   {
4   private:
5     int x;
6   public:
7     my_class() : x{} {}
```

```
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture9/namespacewrong.cpp:12:7: error: redefinition of 'class my_clas
s'
   12 | class my_class
      |       ^~~~~~~~
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture9/namespacewrong.cpp:2:7: note: previous definition of 'class my
_class'
    2 | class my_class
      |       ^~~~~~~~

Build finished with error(s).
```

```
19    ~my_class(){}
20    void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
21  };
22  int main()
23  {
24    return 0;
25  }
```

Code on GitHub at: Prelecture9/namespacewrong.cpp

Namespaces help prevent collisions

- A namespace "contains" functions/classes/variables
  - need to prepend namespace name and scope resolution operator to name of needed function/class/variable (same as `std::cout`…)

```cpp
// PL9/namespaceright.cpp
// User defined namespaces and resolution
// Niels Walet, Last modified 06/01/2022
#include<iostream>
namespace namespace1 {
  const double ab{1.5};
  class my_class
  {
  private:
    int x;
  public:
    my_class() : x{} {}
    my_class(int xx) : x{xx} {}
    ~my_class(){}
    void show(){std::cout<<"x="<<x<<std::endl;}
  };
}
namespace namespace2
{
  const double ab{2.5};
  class my_class
  {
  private:
    int x,y;
  public:
    my_class() : x{},y{} {} // shorter method!
    my_class(int xx, int yy) : x{xx},y{yy} {}
    ~my_class(){}
    void show(){std::cout<<"x="<<x<<", y="<<y<<std::endl;}
  };
}
int main()
{
  namespace1::my_class c1{1}; // utilizes my_class from namespace1
  c1.show();
  namespace2::my_class c2{1,2}; // now different my_class from namespace2
  c2.show();
  return 0;
}
```

Code on GitHub at: Prelecture9/namespaceright.cpp

- can also use `using`

```cpp
int main()
{
  using namespace namespace1;
  my_class c1{1};
  c1.show();
  return 0;
}
```

- NB: don't use `using` with namespace `std`!
  - reason is [here](): libraries can declare functions with the same name as yours (and you don't know which ones)
  - if you use `using` then you will be using the library's function rather than yours!

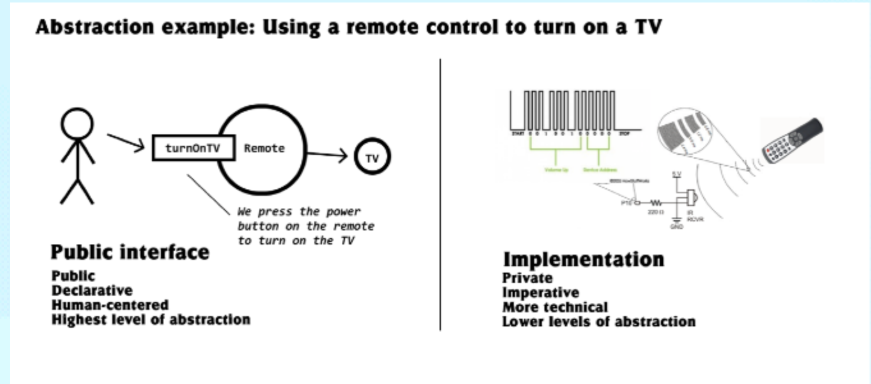
Code on GitHub at: Prelecture9/namespaceright2.cpp

Part 4 / Video 4:
headers & multiple files

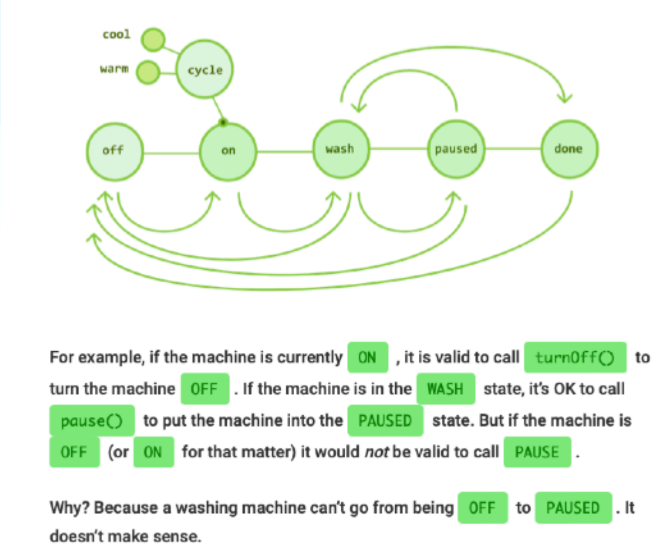
We have already covered this in lecture 4, since we're reusing videos from last year, consider this as a recap that also includes the last slide about what to do with templates

A reminder of OOP

Most describe it in terms of these 4 principles:



- **Abstraction**
  separate interface
  and implementation
  (remote control example)



- **Encapsulation:**
  keep data private, alter properties
  via methods only (washing machine example)

- **Inheritance:**
  classes can be based on other classes to avoid code duplication

- **Polymorphism:**
  can decide **at run-time** what methods to invoke for a certain class, based on the object itself

MANCHESTER
1824
The University of Manchester

Headers and implementation

A reminder from prelecture 4

- C++ classes are usually split into two different parts, often in different files
  - *Header* (file extension: .h, .hpp) = where things are defined
    - Think of it as the index of a book
    - Usually headers contain *interfaces* that help you get an overview of what something (e.g a function) does, without the clutter of the full implementation
  - *Implementation* (file extension: .cxx, .cpp) = where things are implemented
    - Think of it as the book content
    - The implementation of the class's functions go here

Our particle class, split in .cxx and .h

Compilation will have to change slightly…

```cpp
#ifndef PARTICLE_H
#define PARTICLE_H

#include<iostream>
#include<string>
#include<cmath>
class particle
{
private:
  std::string type {"Ghost"};
  double mass {0.0};
  double momentum {0.0};
  double energy {0.0};
public:
// Default constructor
  particle() = default ;
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {};
  ~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor (in-line)
  double gamma() {return energy/mass;} // One-line functions are OK in-line
  void print_data();
};

#endif
```

Find this code at: Prelecture4/particle.h

```cpp
#include<iostream>
#include "particle.h"

void particle::print_data()
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<type<<","<< mass
      <<","<<momentum<<","<<energy<<"]"<<std::endl;
  return;
}
```

Find this code at: Prelecture4/particle.cpp

```cpp
#include<iostream>
#include<string>
#include<cmath>
#include "particle.h"

int main()
{
  // Set values for the two particles
  particle electron("electron",5.11e5,1.e6);
  particle proton("proton",0.938e9,3.e9);
  // Print out details
  electron.print_data();
  proton.print_data();
  // Calculate Lorentz factors
  std::cout.precision(2);
  std::cout<<"Particle 1 has Lorentz factor gamma="
      <<electron.gamma()<<std::endl;
  std::cout<<"Particle 2 has Lorentz factor gamma="
      <<proton.gamma()<<std::endl;
  return 0;
}
```

Find this code at: Prelecture4/main_class4.cpp

- [yourCompilerDir]/g++-11 -fdiagnostics-color=always -g [yourdir]/main_class4.cpp [yourdir]/particle.cpp -o [yourdir]/main_class4

More details on header files

- When our code grows large, we must **divide code across files for readability**
- First thing to consider is where to put **constants**, **class definitions** and **function declarations**
- The best place to do this is in the header file
  - This is what the user wants to know - part of the ***interface***
- We include the contents of header files as follows

```
#include<iostream> // system include file (C++ standard library)
#include<cmath> // another one (from C library)
#include "myheader.h" // our include file
```

  - Note differences between system header files and our own
- We can then include this header file in every .cpp file that makes up our program
  - Health warning: the compiler will literally insert the code of the included .h files on top of the cpp file! And repetition causes clashes…we will see a solution soon
- If header files are for class definitions and function declarations: where should we put function definitions?

More details on cxx/cpp files

- **Function definitions** (what functions actually do) usually go **in a .cpp file**, especially when substantial
- We can create a second .cpp file to hold these, separate from the main function
  - This is what the user doesn't need to know - part of the ***implementation***
- We now have 3 files: `myclass.h`, `myclass.cpp` and `myproject.cpp`
- We name files as appropriate; **the house style requires the same name for header and implementation** (.h or .cpp extension)
- Important for easy compilation: keep all these files in projects folder

Important note on definitions

- Important: **definitions can be made only once.**
- *Functions* in .cpp file OK - included only once.
- *Headers* (containing class definitions) may be included more than once (e.g., included in multiple other headers)
  ⇒we need a **header guard** to prevent multiple definition.
- We can use pre-processor directives to ensure this. #ifndef/#define
- See the header file of `particle.h` for an example:

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include<iostream>
#include<string>
#include<cmath>
class particle
{
```

…

```
  void print_data();
};

#endif
```

Find this code at: Prelecture4/particle.h

What about templates? no splitting!

- Using the method for splitting code in multiple files discussed above will cause linker errors when using templates.
- Template classes and functions are **generated on demand**
- There is a consequence: **compiler** needs to **see both declarations and definitions in the same file** as the code that uses the templates.
- The default 'house style' rule above was that there are no function definitions inside a header file. You are expected to **break this for templates**.

- What to do to put implementation outside the class (always good practice): below namespace (containing the class definition) in header file,
  - Add `using namespace my_namespace` (or equivalent);
  - Then add all template function definitions;
  - Include this header file in any .cpp file where objects are instantiated from this class template.

Part 5 / Video 5:
templates and friends

Templates and friends

- You need to be specific about relationship between a template class and friends (as template functions).
- This is particularly important for the insertion operator <<.
- Here's how to do it:
  - **Before** the class declaration, add the following lines so that the compiler knows about friends:

```
// Forward declaration of class
template <class c_type> class myclass;
// So that we can declare friend function as a template function
template <class c_type > std::ostream & operator<<(std::ostream &os, const
myclass<c_type> myobject);
```

  - Then in body of class, declare friend as follows:

```
friend std::ostream & operator<< <c_type> (std::ostream &os, const myclass<c_type>
&myobject);
```

  - Finally, define operator<< (may have to refer to class' namespace if there is one defined)

```
// Function to overload << operator
template <class c_type >

std::ostream & myns::operator<<(std::ostream &os, const myclass<c_type> &myobject)
{ .... return os; }
```

An example of templates with friends

- Let's take the *cardboard_box* that is every cat's dream
  - Note that *bobcat* and *housecat* would easily be able to enter the *cardboard_box* via polymorphism, no need for templates
  - but what if *dog* wants to get in the box too?
- At some point both *dog* and *housecat* (maybe even bobcat if you live in a zoo) will want to enter the box
  - one will *kick_out()* the other, with potentially unsafe consequences
    - can you do this with templates?
  - In any case we need a method to remove the pet inside to keep the peace, and this method needs to access (private) content of the box

peptic_ulcer on Flicker

Wikimedia commons

Bluebike on Flicker

MANCHESTER
1824
The University of Manchester