Operator Overloading and Templates

Operator Overloading

Overloading : printing

```
String
{
    char at(int index);
};

void print(ostream &output, const String &s)
{
   ...
}

ostream &operator <<(ostream &output, const String &s)
{
    print(output, s);
    return output;
}
```

Overloading : reading

```
String
{
    void push_back(char c);
};

String read(istream &input)
{
    ...
}

istream &operator >>(ostream &input, String &s)
{
    s = read(input);
    return input;
}
```

When to overload IO: “value” classes

- Some classes represent values
  - Their state completely captures them
  - They can be copied and duplicated
  - *Examples*: `int`, `complex`, `string`, bitmaps, audio data
- Reading and writing allows us to move values
  - *Through space*: write on one machine, read on another
  - *Through time*: write in the past, read in the future
  - It often makes sense to overload << and >>

When to overload IO: “real” things

- Some classes represent actual objects in the world
  - Their state identifies something outside the computer
  - They often cannot be copied and duplicated
  - *Examples*: cout, cin, a motor, a sensor, a display
- Overloading << may make sense for debug
  - *Robotic arm:* print the current angle and position
  - *Temperature sensor*: print sensor location + temperature
- Overloading >> is likely to be confusing

```
void f(RobotArm &arm)
{
  // What does this mean?
  cin >> arm;
}
```

```
void f(RobotArm &arm)
{
  arm.load_constraints(cin);
}
```

Overloading Assignment

Construction vs assignment

```
class String
{
private:
    int length;
    int capacity;
    char *data;
public:
    String();
    String(const char *s);
    String(const String &s);
};

int main()
{
    String a;
    String b("x");
    String c(b);
    String d=c;

    a = c;
}
```

A fresh string instance is being created
The constructor is called to initialise it.

An ***existing*** instance a is being assigned the value of the c. This is not construction.

Assignment in practise

```
struct MyStringVec
{
private:
    int length;
    int capacity;
    String *data;
public:
    void write(int index, const String &s)
    {
        data[index] = s;
    }
};
```

Overloading assignment : v1

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    length = s.length;
    capacity = s.capacity;

    data = new char[capacity];

    for(int i=0; i<length; i++){
      data[i] = s.data[i];
    }
    return *this;
  }
};
```

```
int f(String &x)
{
  String y;
  x = y;
}
```

Overloading assignment : v2

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    delete []data;

    length = s.length;
    capacity = s.capacity;
    data = new char[capacity];
    for(int i=0; i<length; i++){
      data[i] = s.data[i];
    }

    return *this;
  }
};
```

```
int f(String &x)
{
  x = x;
}
```

Overloading assignment : v3

```
class String
{
private:
  int length;
  int capacity;
  char *data;
public:
  String &operator=(const String &s)
  {
    if(this != &s){
      delete []data;
      length = s.length;
      capacity = s.capacity;
      data = new char[capacity];
      for(int i=0; i<length; i++){
        data[i] = s.data[i];
      }
    }
    return *this;
  }
};
```

Overloading assignment : v4

```
class String
{
private:
  vector<char> data;
public:
  String &operator=(const String &s)
  {
    data = s.data;
  }
};
```

Overloading assignment : v5

```
class String
{
private:
  vector<char> data;
public:
  // Use compiler generated
  // default assignment. Let the
  // vector class handle it.
};
```

Making a “full” type

- Certain operations should *always* be considered

```
T::T()
T::T(const T &x)
const T &T::operator=(const T &x)
```

  - These make an object look “normal”
  - Often the compiler default behaviour is fine
- Some operations make objects much more useful

```
bool T::operator<(const T &o) const;
bool T::operator==(const T &o) const;
```

  - Allows us to sort and order objects
  - Not meaningful for all classes
- Everything else depends on context

Making a restricted type

- Sometimes you don’t want people copying your type
  - E.g. Objects representing low-level resources; or
  - Trying to represent the idea of uniqueness
- *Simple*: make copy constructor and assignment private

- You are unlikely to want to do it in this course
  ... but maybe in Arduino

What you’re about to see won't work in Windows/Linux

```
/* WARNING: direct access to memory-map of
TR1 ZX80. Do not use on a desktop, it will crash.

DO NOT COPY THIS CLASS ONCE CONSTRUCTED
*/
class RawMotor
{
private:
  volatile int *m_peripheral;
public:
  RawMotor()
  {
    // WARNING: Deeply unsafe. This will only
    // work on the TR1 ZX80 robot. Demons here!
    m_peripheral = (int*) 0x80001000;
  }

  void set_speed(int degrees_per_second)
  {
    // WARNING: directly from TR1 ZX80 datasheet
    m_peripheral[1] = degrees_per_second;
  }
};
```

```
/* WARNING: direct access to memory-map of
TR1 ZX80. Do not use on a desktop, it will crash.*/
class RawMotor
{
private:
  volatile int *m_peripheral;

  RawMotor(const RawMotor &);
  const RawMotor &operator=(const RawMotor &);
public:
  RawMotor()
  {
    // WARNING: Deeply unsafe. This will only
    // work on the TR1 ZX80 robot. Demons here!
    m_peripheral = (int*) 0x80001000;
  }

  void set_speed(int degrees_per_second)
  {
    // WARNING: directly from TR1 ZX80 datasheet
    m_peripheral[1] = degrees_per_second;
  }
};
```

Overloading : summary

- Overloading allows multiple functions with same name
  - Each overload must have different input types
- We have two types of overloading
  - Function overloading : multiple defn. of “normal” functions
  - Operator overload: adding new meanings to operators
- Constructor and assignment overloading are important
  - Needed to avoid surprises when working with raw pointers
- Overloading can be powerful, but don’t overuse it
  - Overloading should always “make sense”
  - Don’t overload `to_string` to return a float
  - Don’t overload `*` to mean divide

Templates (done fast)

templates : a whirlwind tour

- We've already used templates:
  - `vector<float>` vs `vector<string>`
  - `complex<float>` vs `complex<double>`
- You will often want to *use* templates
  - They are uses a lot in standard containers
- You *may* want to write templates
  - But you can often avoid it in your own code

templates : motivation

We often end up writing **functions** that look the same, except for the types:

```
// Ensures that a <= b
void put_pair_in_order(string &a, string &b)
{
    if(b < a){
        string tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : motivation

We often end up writing **functions** that look the same, except for the types:

```
// Ensures that a <= b
void put_pair_in_order(float  &a,  float &b)
{
    if(b < a){
        float  tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : motivation

We often end up writing **functions** that look the same, except for the types:

```
// Ensures that a <= b
void put_pair_in_order(int    &a,  int   &b)
{
    if(b < a){
        int    tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : motivation

We often end up writing **functions** that look the same, except for the types:

```
// Ensures that a <= b
void put_pair_in_order(vector<int> &a, vector<int> &b)
{
    if(b < a){
        vector<int> tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : motivation

We often end up writing **types** that look the same, except for the types:

```
struct MyVecOfInt
{
    int length;
    int capacity;
    float  *data;

    void push_back(float  x);
    float   at(int position) const;
};
```

templates : motivation

We often end up writing **types** that look the same, except for the types:

```
struct MyVecOfInt
{
    int length;
    int capacity;
    int    *data;

    void push_back(int    x);
    int    at(int position) const;
};
```

templates : motivation

We often end up writing **types** that look the same, except for the types:

```
struct MyVecOfInt
{
    int length;
    int capacity;
    string *data;

    void push_back(string x);
    string at(int position) const;
};
```

Manual solutions

- If we have a **function** where only the type varies:
  - We can manually create overloads for each type
  - Have to copy and paste the definition then modify
  - All functions have the same name
- If we have a **type** where another type varies
  - Have to create separate definitions of each type
  - Each type must have a different name
- These solutions have two big drawbacks
  1. There is a lot of copy, pasting, and renaming
  2. You have to know exactly which types could be used

templates : the solution

Write a single **function** for all cases,

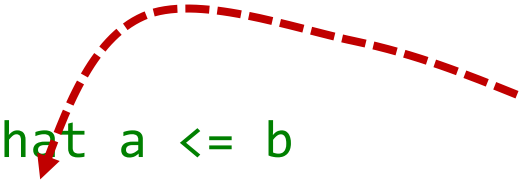
with a placeholder for the type:

```
// Ensures that a <= b
void put_pair_in_order(string &a, string &b)
{
    if(b < a){
        string tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : the solution

Write a single **function** for all cases,

with a placeholder for the type:



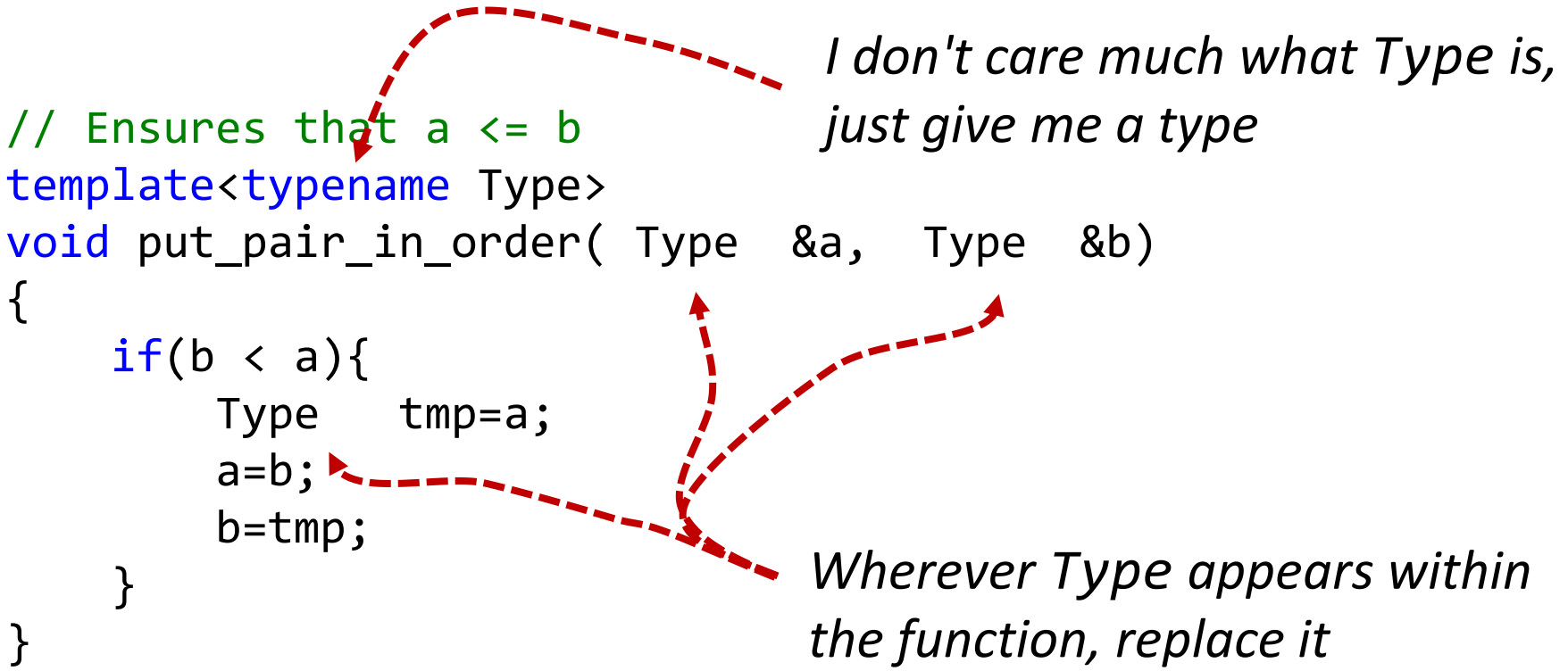
I don't care much what Type is, just give me a type

```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( Type  &a,  Type  &b)
{
    if(b < a){
        Type   tmp=a;
        a=b;
        b=tmp;
    }
}
```

templates : the solution

Write a single **function** for all cases,

with a placeholder for the type:

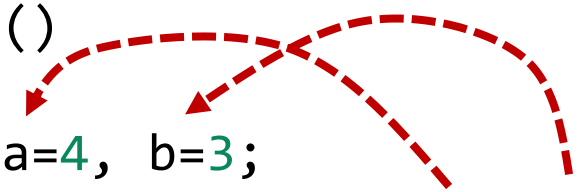


templates : expanded as needed

```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( Type  &a,  Type  &b)
{
    if(b < a){
        Type   tmp=a;
        a=b;
        b=tmp;
    }
}

int main()
{
    int a=4, b=3;
    put_pair_in_order( a , b );
    cout << a << " " << b << endl;
}
```

templates : expanded as needed

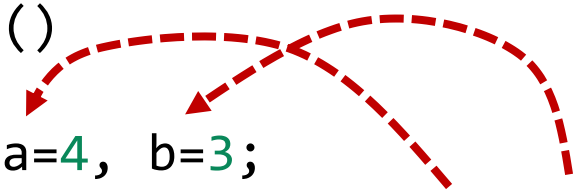
```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( Type  &a,  Type  &b)
{
    if(b < a){
        Type   tmp=a;
        a=b;
        b=tmp;
    }
}

int main()
{
    int a=4, b=3;
    put_pair_in_order( a , b );
    cout << a << " " << b << endl;
}
```



templates : expanded as needed

```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( Type  &a,  Type  &b)
{
    if(b < a){
        Type   tmp=a;
        a=b;
        b=tmp;
    }
}
int main()
{
    int a=4, b=3;
    put_pair_in_order( a , b );
    cout << a << " " << b << endl;
}
```

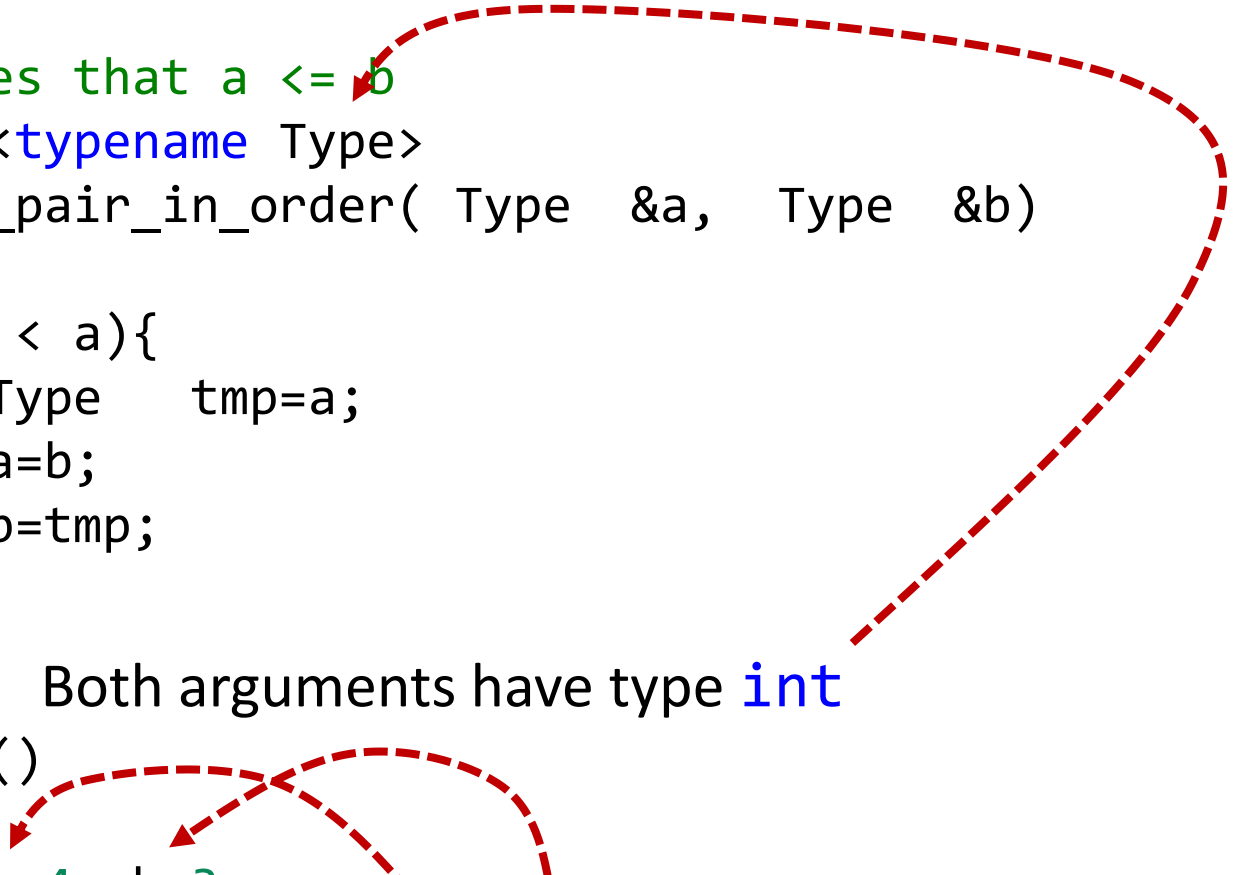
Both arguments have type `int`



templates : expanded as needed

```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( Type  &a,  Type  &b)
{
    if(b < a){
        Type   tmp=a;
        a=b;
        b=tmp;
    }
}

int main()
{
    int a=4, b=3;
    put_pair_in_order( a , b );
    cout << a << " " << b << endl;
}
```

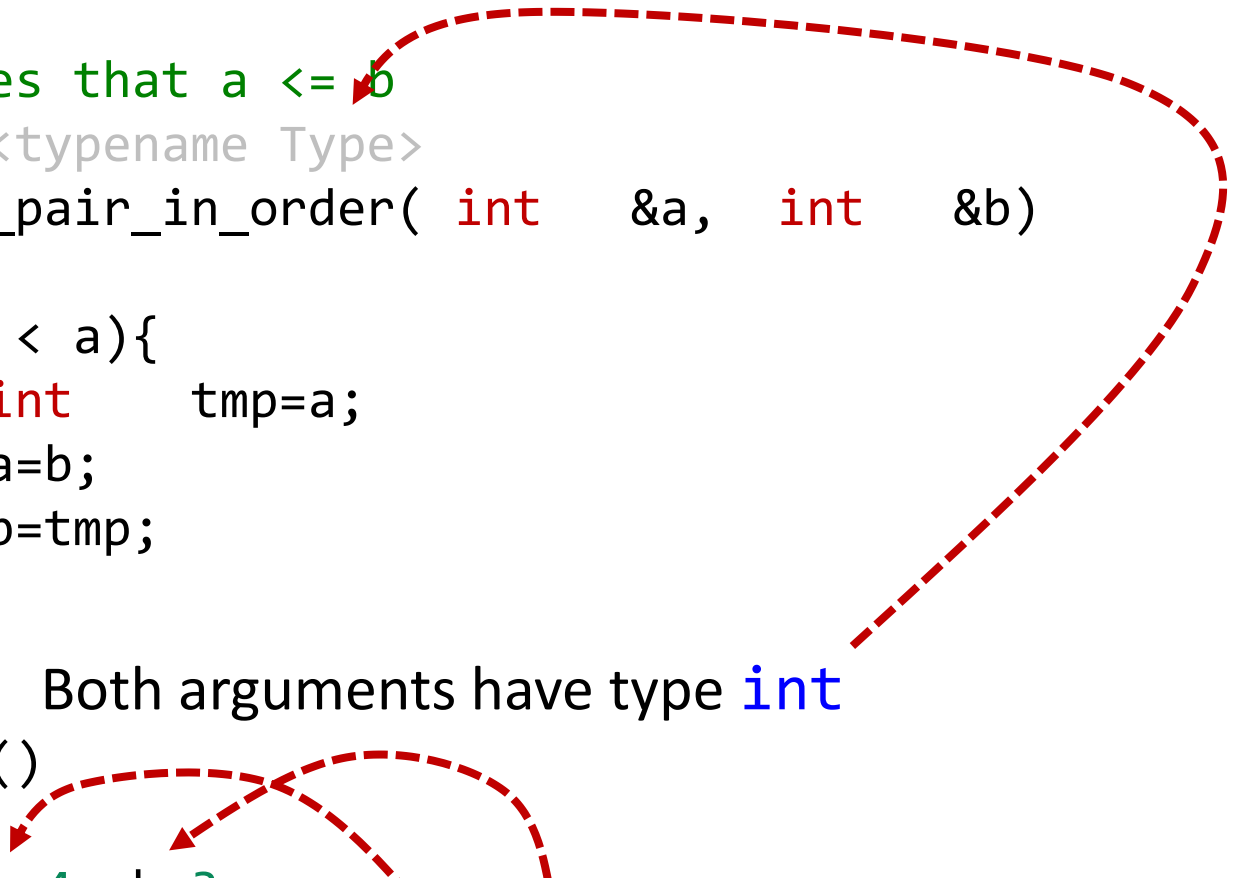
Both arguments have type `int`



templates : expanded as needed

```
// Ensures that a <= b
template<typename Type>
void put_pair_in_order( int   &a,  int   &b)
{
    if(b < a){
        int    tmp=a;
        a=b;
        b=tmp;
    }
}

int main()
{
    int a=4, b=3;
    put_pair_in_order( a , b );
    cout << a << " " << b << endl;
}
```

Both arguments have type int



templates : templated types

```
template<typename T>
class Vector
{
private:
    int length;
    int capacity;
    T *data;
public:
    int size();
    const T &at(int index) const;
    void append(const T &x);
};
```

templates : templated types

```
template<typename T>
class Vector
{
private:
    int length;
    int capacity;
    T *data;
public:
    int size();
    const T &at(int index) const;
    void append(const T &x);
};

int main()
{
    Vector<string> v;
    v.push_back("Hello");
}
```

templates : templated types

```
template<typename T>
class Vector
{
private:
    int length;
    int capacity;
    string *data;
public:
    int size();
    const string &at(int index) const;
    void append(const string &x);
};

int main()
{
    Vector<string> v;
    v.push_back("Hello");
}
```

T = string

templates : role of the compiler

- Templates are managed by the compiler
  - Everything happens at compile-time, like overloads
  - No decisions or choices at compile-time
- Each function or type is "specialised" on demand
  - The first time `vector<int>` is seen, the code is expanded
  - Following uses of `vector<int>` re-use the same code
- The template must "make sense" when specialised
  - Would it compile if you copied, pasted, and replaced types?
  - The compiler will give errors if a type can't be used

An example: vector<T>

We've happily used vector<T> many times

... it usually just works.

However, there are some requirements on T

- The type T must be "CopyAssignable":

```
void CopyAssignable(T &x, const T &y)
{
    x = y; // x should be an independent copy of y
}
```

An example: vector<T>

We've happily used vector<T> many times

... it usually just works.

However, there are some requirements on T

- The type T must be "CopyAssignable"
- The type T must be "CopyConstructible"

```
void CopyConstructible(const T &y)
{
    T x(y); // x should be an independent copy of y
    T z=y;  // z should be an independent copy of y
}
```

An example: vector<T>

We've happily used vector<T> many times

... it usually just works.

However, there are some requirements on T

- The type T must be "CopyAssignable"
- The type T must be "CopyConstructible"

Any type that meets those requirements should work

An example: min

```
template<class T>
const T &min(const T &a, const T &b)
{
    if( a < b ){
        return a;
    }else{
        return b;
    }
}
```

The type T must be "LessThanComparable":

```
void LessThanComparable(const T &a, const T &b)
{
    bool c = a < b;
}
```

Possible problems with templates

- Class and objects specify type requirements
  - "*Type T must be CopyConstructible*"
  - "*Type X must be LessThanComparable*"
  - These are specified in the documentation
- Type requirements are checked on specialisation
  - You won't see errors until you try to use the type
  - C++ compiler template errors are notoriously bad
- It is sometimes useful to "force" a template
  - Don't let the compiler pick the type automatically

templates : conflicting types

```
#include <algorithm>
#include <cmath>

using namespace std;

int main()
{
    int x = min( 0, max( 1.0f, sin(2.3) ) );
}
```

templates : conflicting types

```
int main()
{
    int x = min( 0, max( 1.0f, sin(2.3) ) );
}
```

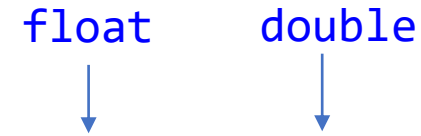
templates : conflicting types

```
int main()
{
    int x = min( 0, max( 1.0f, sin(2.3) ) );
}
```

```
source>: In function 'int main()':
<source>:8:41: error: no matching function for call to 'max(float, double)'
8 | int x = min( 0, max( 1.0f, sin(2.3) ) );
  |                                         ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:222:5: note: candidate: 'template<class _Tp> cons
222 | max(const _Tp& __a, const _Tp& __b)
  |  ^~~
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:222:5: note: template argument deduction/substi
<source>:8:41: note: deduced conflicting types for parameter 'const _Tp' ('float' and 'double')
8 | int x = min( 0, max( 1.0f, sin(2.3) ) );
  |                                         ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:268:5: note: candidate: 'template<class _Tp, class
268 | max(const _Tp& __a, const _Tp& __b, _Compare __comp)
  |  ^~~
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:268:5: note: template argument deduction/substi
```

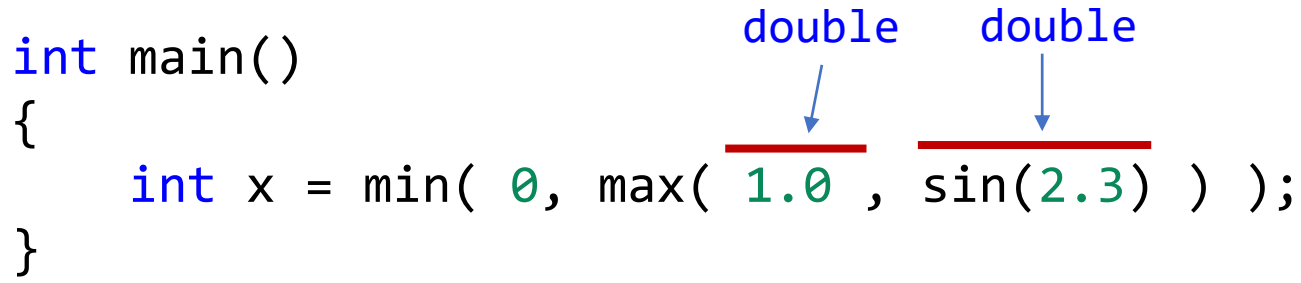
templates : conflicting types

```
int main()
{
    int x = min( 0, max( 1.0f, sin(2.3) ) );
}
```




```
source>: In function 'int main()':
<source>:8:41: error: no matching function for call to 'max(float, double)'
8 | int x = min( 0, max( 1.0f, sin(2.3) ) );
  |                                       ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
                 from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:222:5: note: candidate: 'template<class _Tp> cons
222 | max(const _Tp& __a, const _Tp& __b)
  | ^~~
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:222:5: note: template argument deduction/substi
<source>:8:41: note: deduced conflicting types for parameter 'const _Tp' ('float' and 'double')
8 | int x = min( 0, max( 1.0f, sin(2.3) ) );
  |                                       ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
                 from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:268:5: note: candidate: 'template<class _Tp, class
268 | max(const _Tp& __a, const _Tp& __b, _Compare __comp)
  | ^~~
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:268:5: note: template argument deduction/substi
```

templates : conflicting types



templates : conflicting types

```
int main()
{
    int x = min( 0, max( 1.0,  sin(2.3) ) );
}
```


```
source>: In function 'int main()':
<source>:8:42: error: no matching function for call to 'min(int, const double&)'
8 | int x = min( 0, max( 1.0, sin(2.3) ) );
  |                                          ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:198:5: note: candidate: 'template<class _Tp> co
198 | min(const _Tp& __a, const _Tp& __b)
  |     ^~~
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:198:5: note: template argument deduction/sub
<source>:8:42: note: deduced conflicting types for parameter 'const _Tp' ('int' and 'double')
8 | int x = min( 0, max( 1.0, sin(2.3) ) );
  |                                          ^
In file included from /opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/algorithm:61,
from <source>:1:
/opt/compiler-explorer/gcc-9.2.0/include/c++/9.2.0/bits/stl_algobase.h:246:5: note: candidate: 'template<class _Tp, cla
246 | min(const _Tp& __a, const _Tp& __b, _Compare __comp)
  |     ^~~
```

templates : conflicting types

```
int main()
{
    int x = min<int>( 0, max( 1.0,  sin(2.3) ) );
}
```

Force a particular specialization of the function

```
const int &min(const int &x, const int &y);
```

Templates mostly work as expected

- The best mental model for templates is:

  The compiler will copy and paste on demand,
  then search and replace for the template type

- There are some more complex aspects we won't cover
  - *Partial specialization, variadic templates, template templates, ...*

- Template errors can be a bit complex and *very* long
  - Try to avoid huge changes to your code: be incremental
  - See if being explicit about types helps
    - Force input arguments to a specific type
    - Force the template arguments to a specific tyoe

Templates open up the C++ STL

- STL = Standard Template Library
  - The library of classes and algorithms provided with C++
  - e.g. string, vector, complex, sort, ...

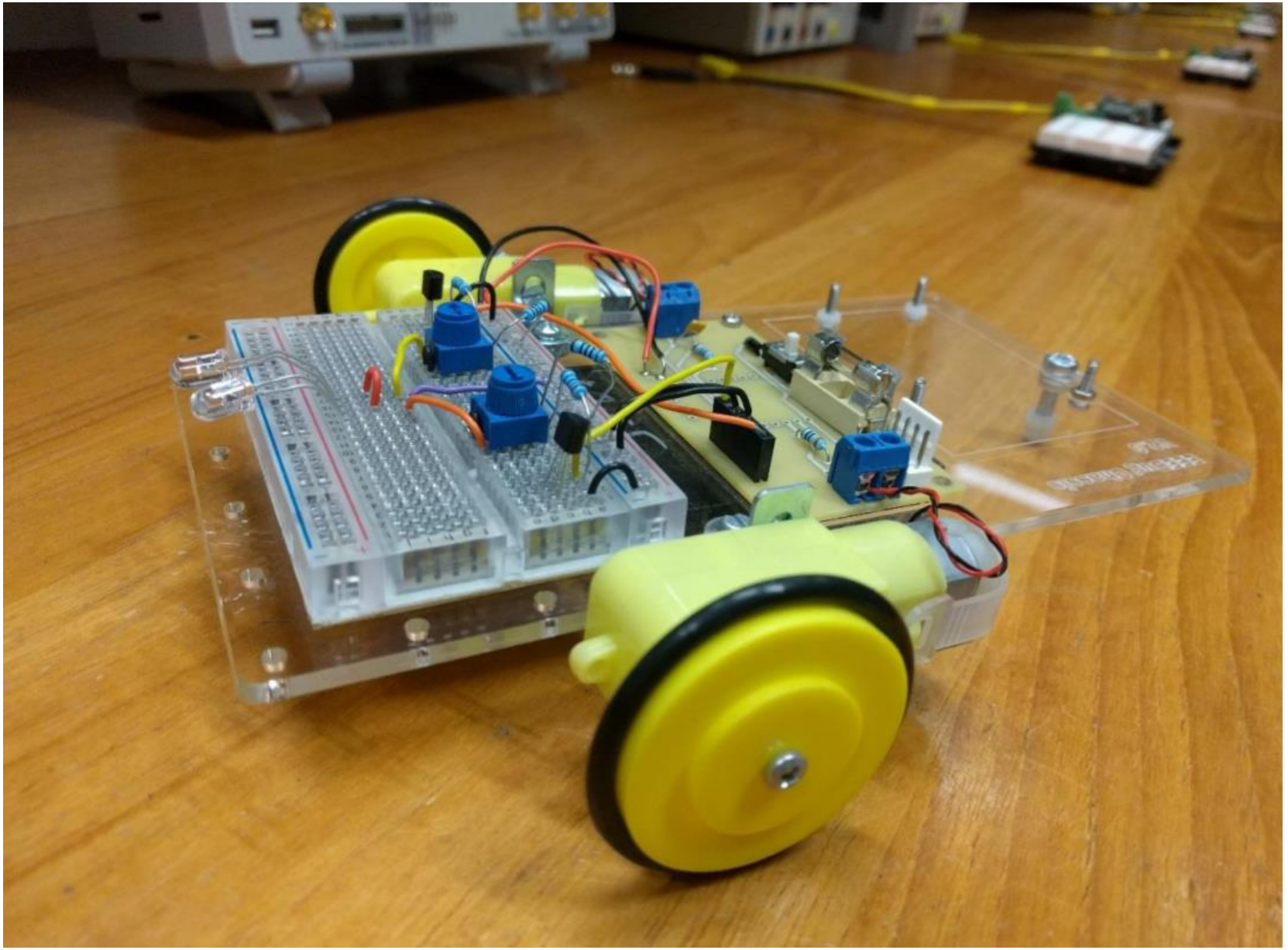
- We get more complicated data-structures
  - lists and slists
  - maps and sets : remember sorted trees?
  - queue, stack, priority_queue, ...

- We also get algorithms
  - sort, accumulate, random numbers, search, ...

Modelling real things

Objects usually model things

- Value/data types
  - Integers, reals, matrices, sets, vectors
  - Images, audio data, graphs, ...
  - Modifying the object changes the value it represents
- Physical objects
  - Robots, UAVs, computers, network connections, ...
  - Modifying the object changes the world
- Simulations of physical objects
  - Circuits, filters, mechanical models, ...
  - Robots, UAVs, computers, network connections, ...
  - Modifying the object *simulates* how the world would change



Modelling the EEBug

- We could model it at many levels
  - Detailed electrical model: sensors actuators
  - Physics model: e.g. acceleration, mass, rotation rate...
  - Functional level: move forwards, rotate, read light levels

- An object model could serve multiple purposes
  - Direct control: actual code that runs inside the bug
  - Indirect control: code that controls bug over a network
  - Simulation: simulate the actions of the bug over time

Public API: key questions

- What are the observable properties and state?
  - What fixed properties are associated with the thing?
  - State/properties that change as a result of actions?
  - State/properties that changes independent of action?
- What are the actions that one can take?
  - What verbs, actions or behavior does the thing have?
  - What input information do those actions need?
  - What kind of output do they produce?
  - How will they affect the observable state

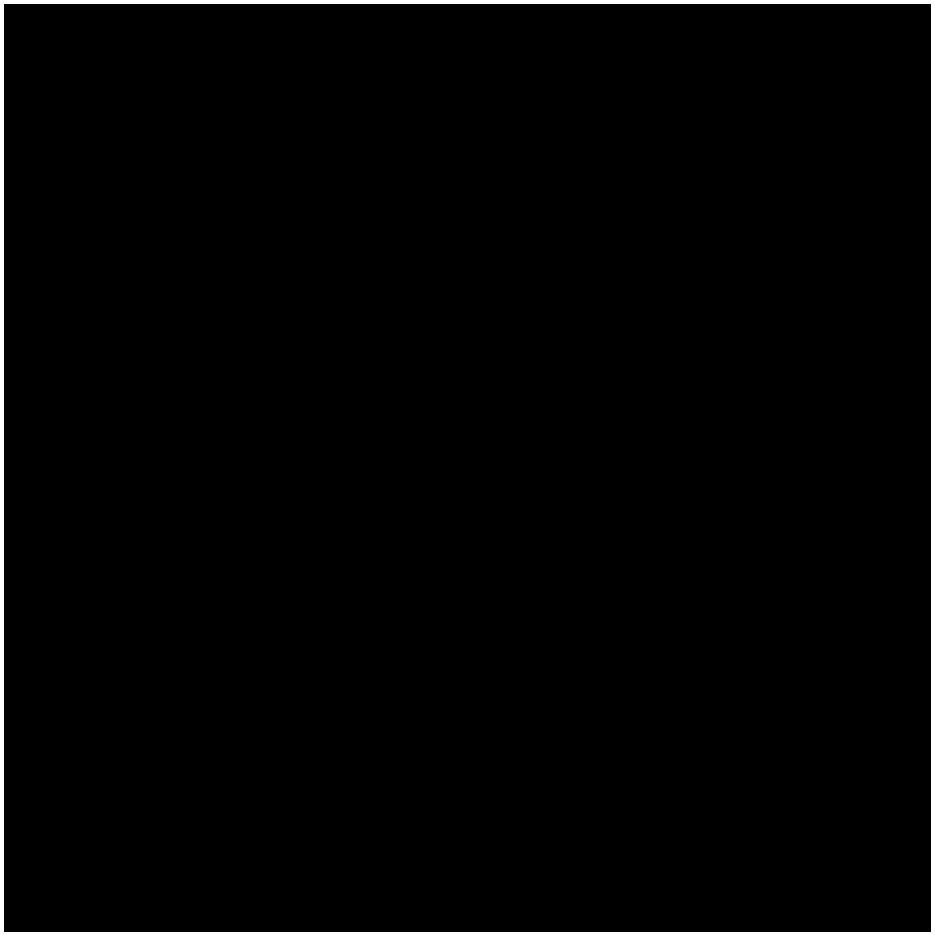
These are independent of whether you are controlling the "real" thing or a simulation of it

Next two labs model the EEBug

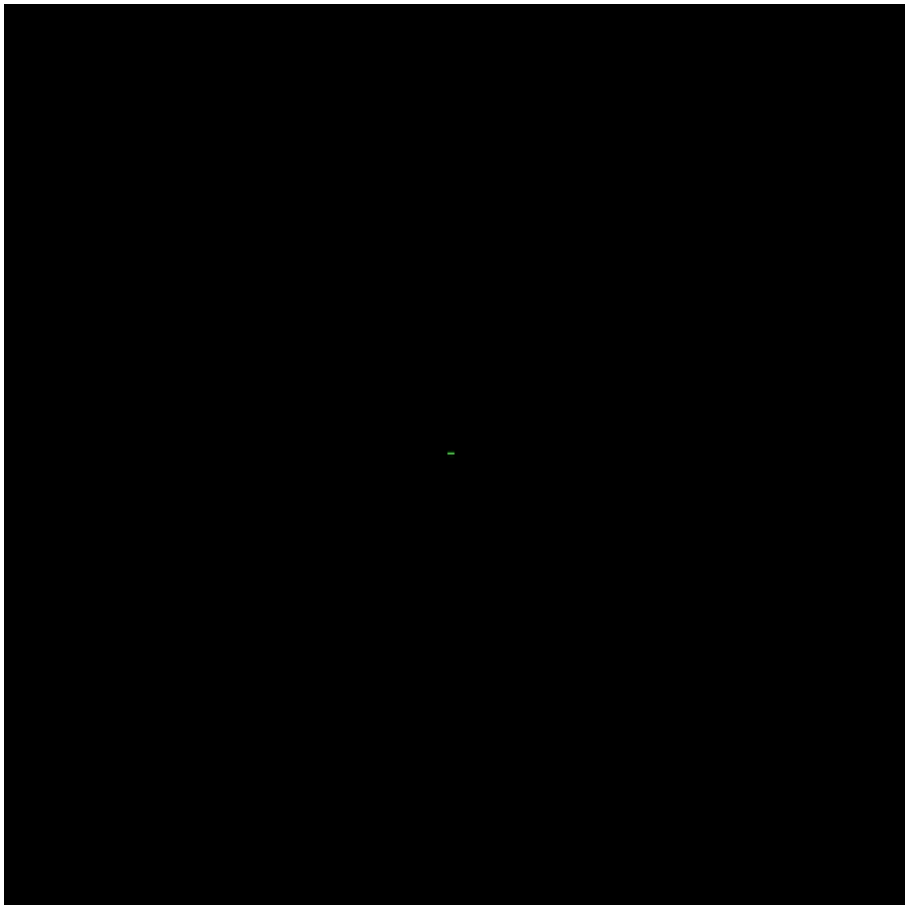
- We want one object API that can do two things:
  - Control a simulation of the EEBug
  - Control a "remote" EEBug
- Multiple controllers should be able to use the API
  - Controller should not care what it is controlling

- This week: using plain objects
- Next week: using basic inheritance

```
class Rover
{
public:
    float get_time() const;
    vector2d get_position() const;
    float get_speed() const;
    float get_angle() const;
    bool get_pen_down() const;

    void set_speed(float speed);
    void set_angle(float angle);
    void set_pen_down(bool pen_down);

    void advance_time(float dt);
};
```



```
Rover r;
r.set_angle(PI/4);
r.set_speed(1);
r.advance_time(sqrt(2));

r.set_pen_down(true);
float angle=PI;
float time_step=2;
while(time_step > 1e-3){
    r.set_angle(angle);
    r.advance_time( time_step );

    angle = angle + PI/2 + 0.05;
    time_step=time_step*0.95;
}
```



```
Rover r;

r.set_pen_down(true);

float dt=0.1;
while(r.get_time() < 100){
    float dx = normal_dist(rng);
    float dy = normal_dist(rng);

    float angle = atan2(dx, dy);
    float distance = sqrt(dx*dx + dy*dy);

    r.set_speed( distance/dt );
    r.set_angle(angle);
    r.advance_time( dt );
}
```