Lecture 9

- Operator overloading
- Templates
Operator overloading

We have seen examples already of operators acting on classes:

```cpp
std::string s1("hello "), s2("world");
std::string s3 = s1 + s2; // + concatenates strings

std::vector<int> v(10);
v[0] = 5; // [] accesses elements
```

This is called *operator overloading*.

Operator overloading is achieved by special member functions:

```cpp
class MyClass
{
  double operator[](int i) // member function for [] operator
  {
    return ...; // implement operator behaviour
  }
};
```
Operator overloading

Operator overloading can be used to define:

- mathematical operators +, −, ∗ etc. for mathematical objects (e.g. a Matrix class).
- relational operators <, > etc. to define an ordering
- subscript operators [ ] for container classes
- input and output operators >> and << for use with streams (std::cout etc.)
- the function call operator () to give objects the syntax of functions

For details, see example sheet 5; C++ Primer Chapter 14.
**Example:** We have previously defined a `Point2D` class, but trying to add two `Point2D` objects fails:

```c++
Point2D a(1.0, 2.0), b(3.0, 4.0);
Point2D c = a + b; // error!
```

We can define a function

```c++
Point2D operator+(Point2D a, Point2D b) // "a + b"
{
    return Point2D(a.x + b.x, a.y + b.y);
}
```

that specifies how the `+` operator works for `Point2D.

The two components of `Point2D` add as components of a vector.
Operator overloading

Open the example code ‘Operator overloading 1’

1. Compile and run the code to verify that the `operator+` function allows `a+b` to be calculated.

2. Write an analogous function `operator-` that allows you to call `a-b`. Test that this new function works.

3. Create a new function:
   
   ```cpp
   Point2D operator*(double v, Point2D a) that allows you to multiply a Point2D by a scalar,
   ```
   
   ```cpp
   Point2D f(1.0, 2.0);
   Point2D g = 3.5 * f;
   ```
Operator overloading

**Example:** recall our *Integrand* base class, where we defined a pure virtual function

```cpp
virtual double Evaluate(double x) = 0;
```

We can instead overload `operator()` in *Integrand*:

```cpp
virtual double operator()(double x) = 0;
```

and implement `operator()` in our derived class

```cpp
virtual double operator()(double x) {
    /* return our integrand as function of x */
}
```

then call (references/pointers to) *Integrand* objects as functions:

```cpp
double Integral(double a, double b, Integrand &i) {
    // Trapezium rule with one trapezium
    return (b-a)*0.5*(i(a)+i(b));
}
```
Open the example code ‘Operator overloading 2’:

1. Compile and run the code to verify that the syntax \( \text{si}(1.0) \) on line 22 calls the \texttt{operator()} implementation on line 13.

2. Change the code so that \texttt{operator()}:
   
   - takes two \texttt{double} arguments, \( x \) and \( y \)
   - evaluates to \( \sin(x) \cos(y) \)

   (remember to change both the base and derived class). Verify that \( \text{si}(1.0, 2.0) \) returns \(-0.35018\ldots\)
Templates

- Templates allow functions and classes to have type parameters.
- This is useful if we want to write code without specifying exactly the type of objects it acts on.
- For example, `std::vector` has a template to specify the type of object it contains:

```cpp
std::vector<int> v(50);
```

- Template parameters are denoted by triangular brackets `< >`
- We can apply templates to functions, and to classes
Function templates

How can we implement a C++ function to bound a value $x$ between two limits $a$ and $b$ (with $a < b$):

$$(x, a, b) = \min(\max(x, a), b)$$

double Bound(double x, double a, double b)
{
    if (x<a) return a;
    else if (x>b) return b;
    return x;
}
Function templates

How can we implement a C++ function to bound a value $x$ between two limits $a$ and $b$ (with $a < b$):

$$(x, a, b) = \min(\max(x, a), b)$$

Exactly the same function could be written with `int`...

```cpp
int Bound(int x, int a, int b) {
    if (x < a) return a;
    else if (x > b) return b;
    return x;
}
```
Function templates

How can we implement a C++ function to bound a value $x$ between two limits $a$ and $b$ (with $a < b$):

$$(x, a, b) = \min(\max(x, a), b)$$

...or with `float`:

```cpp
float Bound(float x, float a, float b)
{
    if (x<a) return a;
    else if (x>b) return b;
    return x;
}
```
Function templates

How can we implement a C++ function to bound a value $x$ between two limits $a$ and $b$ (with $a < b$):

$$(x, a, b) = \min(\max(x, a), b)$$

We can define this function for all types with a template:

```cpp
template<typename T>
T Bound(T x, T a, T b)
{
    if (x<a) return a;
    else if (x>b) return b;
    return x;
}
```

The compiler deduces the type $T$ from the type of the arguments passed. See example code Function templates 1
# Function templates

```cpp
#include <iostream>

template <typename T>
T Bound(T x, T a, T b)
{
    if (x<a) return a;
    else if (x>b) return b;
    return x;
}

int main()
{
    // works with double arguments
    std::cout << Bound(1.5, 0.0, 1.0) << std::endl;
    // works with int arguments
    std::cout << Bound(-2, 0, 1) << std::endl;
    // ambiguous template type; specify explicitly
    std::cout << Bound<double>(1.5, 0, 1) << std::endl;
    return 0;
}
```
Function templates

```cpp
#include <iostream>
template <typename T>
void Recurrence(T coef1, T coef2)
{
    T b=1.0/3.0, a=1.0, c;
    for (int i=0; i<20; i++) {
        c = coef1*b+coef2*a;
        std::cout << a << std::endl;
        a=b; b=c;
    }
}

int main()
{
    Recurrence<float>(19.0/3.0, -2.0);
    Recurrence<double>(19.0/3.0, -2.0);
    Recurrence<float>(4.0/3.0, -1.0/3.0);
    Recurrence<double>(4.0/3.0, -1.0/3.0);
    return 0;
}
```

See example code **Function templates 2**
C++ supports template parameters for classes, as well as functions.

Recall our **Point2D** class:

```cpp
class Point2D
{
public:
    void SetXY(double x_, double y_)
    {
        x = x_; y = y_; 
    }
    double GetX() {return x;}
    double GetY() {return y;}
private:
    double x, y;
};
```
C++ supports template parameters for classes, as well as functions.

We can generalise this to use any type for the coordinates:

```cpp
template<typename T>
class Point2D
{  
  public:
    void SetXY(T x_, T y_)
    {
      x = x_; y = y_; 
    }  
    T GetX() {return x;}
    T GetY() {return y;}
  private:
    T x, y;
};
```
We declare instances of our templated class with

```cpp
Point2D<double> p1;
Point2D<int> p2;
```

in the same manner as `vector`.

See example code **Class templates**

Templates can be used for much more. See

- **C++ Primer, Chapter 16**
- **Modern C++ Design (A. Alexandrescu)** for more complex usage (so-called `template metaprogramming`)**