Part IX

Function and operator overloading

Outline

33 Function overloading

34 Member function wrappers

35 Member function overloading

36 Operator overloading

Function Overloading

- Different functions usually have different names, because they perform different tasks
- Sometimes, different functions may perform similar operations
- For example, a specialised printing function:

  ```
  Vector v;
  Matrix m;
  printVector(v);
  printMatrix(m);
  ```

- C++ allows us to have a consistent interface (somewhat like the implementation-hiding ideas of OOP), with a single function `print`:

  ```
  void print(const Vector& x);
  void print(const Matrix& a);

  print(v);
  print(m);
  ```

  where the correct function is called depending on its parameters.
- This is called function overloading.

Function overloading

- At its most basic, function overloading is obvious:
  - Determine exact types of function parameters
  - Test these against all visible overloads
  - Call the function that matches
- For more complex cases, C++ has rigorously defined rules
- Note that the return type of a function is never taken into account when determining overloads:

  ```
  int sum(double, double);
  double sum(double, double);
  ```

  will result in a compiler-error (ambiguous declaration) since the two functions cannot be separated by their parameter-types alone.

Outline

33 Function overloading

34 Member function wrappers

35 Member function overloading

36 Operator overloading

Member function wrappers

- It is not immediately trivial to create a wrapper of a member function:

  ```
  int MyClass::func(int i)const;
  ```

  has two extra features associated with it that an ordinary function does not have:
  - Firstly, it must know about a `MyClass` object
  - Secondly, it has a `const` modifier.
- These imply that the actual form of the function must be something like:

  ```
  int func(const MyClass* this, int i);
  ```

- This is never visible to the programmer, however.

Member function wrapper

- A member function wrapper can be created as:

  ```
  std::function<int (const MyClass&, int)> funcWrap =
      &MyClass::func;
  ```

  and used as:

  ```
  MyClass m;
  funcWrap(m, i);
  ```

- This construct can be used when there are multiple member functions of a class with the same signature.
- The & is required in this case (optional for ordinary functions).

Static member function wrappers

- Since static member functions do not require an object when calling, their type is not tied in to a class:

```
class MyClass{
public:
  static int f(int i);
};
std::function<int (int)> func = &MyClass::f;
j = func(i);
```

Outline

33 Function overloading

34 Member function wrappers

35 Member function overloading

36 Operator overloading

Member function overloading

- Member functions can be overloaded in the same way as ordinary functions:

```
class Matrix{
  void raiseToPower(int);
  void raiseToPower(double);
};
```

- Const-ness is taken into account when performing overload resolution:

```
class Vector{
  int& getElt(int);
  int getElt(int)const;
};
```

are distinct functions.

Member function overloading ctd

- The function called depends on the const-ness of the object on which the function is called:

```
const Vector v;
Vector x;
 // Calls non-const version, which returns by reference:
x.getElt(3) = 5;
 // Calls const-version, which returns by value:
int y = v.getElt(3);
 // Not allowed since tries to call const version, which
 // doesn't return by reference
v.getElt(3) = 5;
```

Overloading virtual functions

- Overloading virtual functions is allowed, but may need extra typing:

```
class MyBaseClass{
public:
  virtual double f(int);
  virtual double f(double);
};
class MyClass : public MyBaseClass {
public:
  virtual double f(int);
};
```

- `f` is overloaded within `MyBaseClass`
- The above works normally with polymorphism, i.e. referring to f(1.0) or f(1) through a pointer of type `MyBaseClass` calls either the second or third of the functions as expected.
- However, only the `f(int)` version is called by both:

```
MyClass a;
a.f(1.5);
a.f(1);
```

Solving virtual overloading

- So, within MyClass, only one version of f exists.
- g++ only warns about this if you specify -Woverloaded-virtual (not part of -Wall -Wextra).
- The correct approach is to use:

  ```
  class MyClass : public MyBaseClass {
  public:
    virtual double f(int);
    using MyBaseClass::f;
  };
  ```

  which brings the base-class's version of f into scope.
- MyClass now has two correctly overloaded functions f

Name mangling

- C++ was originally implemented using an intermediary compiler that converted C++ to plain C.
- C does not allow function overloading, so some way of distinguishing overloaded functions is required
- Function parameters are added to the function name to give a unique function name
- e.g. `void f(int, double)` might become `f_i_d`
- Name mangling is non-standard (between compilers)
- Only really see it at the linking stage.
- In Linux, use `c++filt -t MangledName` to recover actual function definition.
- This also works for types, although you will not see the use for this until later...

Outline

33 Function overloading

34 Member function wrappers

35 Member function overloading

36 Operator overloading

Operator overloading

- One of the very useful (and often abused) features of C++is the ability to overload operators
- For example, if you have a Matrix type you've created, you can allow the following:

```
Matrix a, b;
// Initialize matrices a and b
Matrix c = a * b;
std::cout << "A x B = " << c << std::endl;
```

- We need to extend the definition of the * and << operators to allow Matrix objects as arguments.
- If you think of an operator as just another function, this is the obvious extension of function overloading.

Operator overloading

```
class Matrix{
  public:
  Matrix operator*(const Matrix& b)const{
    Matrix result;
    // Use data and b.data to create result.data
    return result;
  }
private:
  std::array<std::array<float,3>,3> data;
};
```

- Note that a member function of `Matrix` has access to private members of other `Matrix` objects.
- It is best to have the arguments as passed by `const &` since this allows the compiler to optimize out the copy constructor.

More Operator overloading

- It is also possible to create operations that take other types:

```
class Matrix{
  public:
    Matrix operator*(float x)const;
    Matrix& operator*=(float x){
      data[0][0] *= x;
      return *this;
    }
};
```

- which will allow operations of the form

```
Matrix a; float x;
a = a*x;
a *= x;
```

  but NOT

```
a = x * a;
```

- Note that the first function does not return by reference, whereas the second one does.

Non-member operator overloading

- Operators may also be defined outside classes:

```
Matrix operator*(const Matrix& a, const Matrix& b){
  Matrix result = a;
  result *= b;
  return result;
}
Matrix operator/(const Matrix& a, float x){
  return result;
}
```

- All operators defined outside classes must take at least one argument of class-type
- Note that these do not return by reference
- To reduce copy-paste errors, it is useful to define binary operators in terms of operate-and-assign operators as above.

Multiple operator overloads

- You will need to define multiple versions of the overload:

```
Matrix operator*(float a, const Matrix& b);
Matrix operator*(const Matrix& a, float b);
Matrix operator*(const Matrix& a, const Matrix& b);
```

- as well as any others for multiplication by a `double`, for example.
- This may be the point at which you resort to macros (or, better, templates).

Output overloading

- The following (defined outside the class) allows a Matrix to be output to a stream.

```
std::ostream& operator<<(std::ostream& os, const Matrix& m){
  os << m[0][0] << " " << m[0][1] ; // Etc.
  return os;
}
```

- A stream is modified when output is sent to it, so must be passed by reference.
- A stream needs to be returned so that the following works:

```
Matrix m;
std::cout << "My matrix is" << m << std::endl;
```

  or, equivalently:

```
(((std::cout << "My matrix is ") << m) << std::endl);
```

  where the parentheses are solely to make the separate function calls clearer.

Other operators

- Increment and decrement operators are overloaded as

```
class Number{
  public:
    Number& operator++(); // prefix ++
    Number operator++(int); // post-fix ++
};
```

- The extra `int` does not take any value when called, it is only a dummy parameter to distinguish the overloads.
- Note that the prefix ++ returns `*this` by reference, whereas post-fix does not return by reference.

Overloading []

- The element access operator can be overloaded, taking a single parameter.

```
class My5Array{
  public:
  int operator[](int i)const{
    return data[i];
  }
  int& operator[](int i){
    return data[i];
  }
  private:
  std::array<int,5> data;
};

My5Array a;
int g = a[4];
a[3] = 5;
```

Overloading ()

- The function call operator () can be overloaded, taking any number of parameters at all:

```
class My2DArray{
  int operator()(int i, int j)const{
    return data[i][j];
  }
};

My2DArray a;
std::cout << a(2,2) << std::endl;
```

Overloading casting

- You can also allow your own classes to be cast to other types, including built-in types.

```
class Rational{
public:
  operator double()const{
    return m_numerator / (double)m_denominator;
  }
}
```

- which would allow a use-case such as:

```
Rational r(1, 3);
std::cout << r << " ~= " << (double)r << std::endl;
```

- Giving:

```
1/3 ~= 0.33333333
```

Operator overloading warnings

- Always consider whether the syntax that will result is clear
  The overloading of left-shift `<<` for output is a good example.
- Operator precedence cannot be changed; if operator precedence makes sense for your class, the overloads should follow that
- The following operators cannot be overloaded:
  `?:` ternary operator
  `.` member-access
  `::` scope-resolution
  `.*` pointer-to-member
- Just because you can use overloading, doesn't mean that you should.
- You cannot alter the number of arguments that an operator takes.