Systems Programming

Lecture 19: Further C++

Stuart James

stuart.a.james@durham.ac.uk

Amir Atapour-Abarghouei: amir.atapour-abarghouei@durham.ac.uk

(Slide thanks to Amir Atapour-Abarghouei and Anne Reinarz)

Recap

https://PollEv.com/stuartjames

Recap: Example of a C++ class declaration

Recap: Example of a C++ class declaration

In [3]:

```
#include <iostream>
using namespace std;
class Rectangle {
   public:
      Rectangle (int w, int h) {     // Constructor
        width = w;
        height = h;
      }
      int Area(void) {
          return width*height;
      }                 // area method
      ~Rectangle (void) {}
                        //Destructor
   private:
        int width;
        int height;
}; // remember to have this!

int main(){
    Rectangle r = Rectangle(2,3);
    std::cout << "Area is " << r.Area() << std::endl;
}
```

```
Area is 6
```

Recap: Example of a C++ class declaration with 'Rule of Five'

Recap: Example of a C++ class declaration with 'Rule of Five'

In [1]:

```cpp
#include <iostream>

class Rectangle {
private:
    int* dimensions;  // Pointer to an array of two integers [width, height]

public:
    // Constructor
    Rectangle(int width, int height) {
        std::cout << "Constructor called\n";
        dimensions = new int[2];
        dimensions[0] = width;
        dimensions[1] = height;
    }

    // Destructor
    ~Rectangle() {
        std::cout << "Destructor called\n";
        delete[] dimensions;
    }

    // Copy Constructor
    Rectangle(const Rectangle& other) {
        std::cout << "Copy Constructor called\n";
        dimensions = new int[2];
        dimensions[0] = other.dimensions[0];
        dimensions[1] = other.dimensions[1];
    }

    // Copy Assignment Operator
```

Recap: Function overloading

- C++ allows multiple functions with the same name in the same scope
- Example from C: the `printf` function
  - behaviour depends on the argument, e.g. double or int
  - means we do not need a `print_int` and a `print_double` function etc.
- Here we define multiple constructors depending on input

Inheritence

Inheritence

Inheritance is the mechanism of basing a class on an existing class to retain similar implementation.

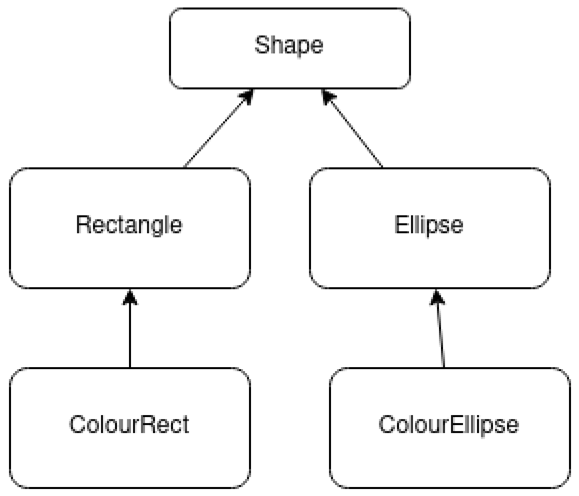
- We derive new classes (sub classes) from existing ones $\Rightarrow$ hierarchy of classes
- Allows code reuse

Inheritence

- Once we have started to work with classes we might want to start adding class hierarchies:
  - A superclass such as `Shape` could contain members that apply to all shapes
  - The class `Rectangle` would be a specialisation of that class
  - It would inherit the members of the class `Shape`

Inheritence

- Of course these hierarchies are not restricted to two levels
- We could have many hierarchy levels



Inheriting from `Rectangle`

Inheriting from `Rectangle`

- We acquire all data members and member functions from `Rectangle`

Inheriting from `Rectangle`

- We acquire all data members and member functions from `Rectangle`
- The constructor for `ColourRect` calls the constructor from the superclass `Rectangle`

Inheriting from `Rectangle`

- We acquire all data members and member functions from `Rectangle`
- The constructor for `ColourRect` calls the constructor from the superclass `Rectangle`

In [14]:

```
#include <iostream>
using namespace std;

class Rectangle {
   public:
      Rectangle (int w, int h): width(w), height(w) {        }
      int Area(void) { return width*height; }
      ~Rectangle (void) {}
   private:
        int width;
        int height;
};

class ColourRect: public Rectangle {
      public:
        ColourRect( int w, int h, int col): Rectangle(w,h), colour(col) {};
        int getColour() { return colour; } ;
      private:
        int colour;
};

int main(){
    ColourRect colRect(2,3,255);
```

Multiple Inheritance

- Suppose we already have Colour class

```
class Colour {
  private:
    int colour;
  public:
    int getColour();
};
```

- We can create `ColourRect` through multiple inheritance

```
class ColourRect: Colour, Rectangle {
  // Can have more data functions too
}
```

Multiple Inheritance

- Constructor:

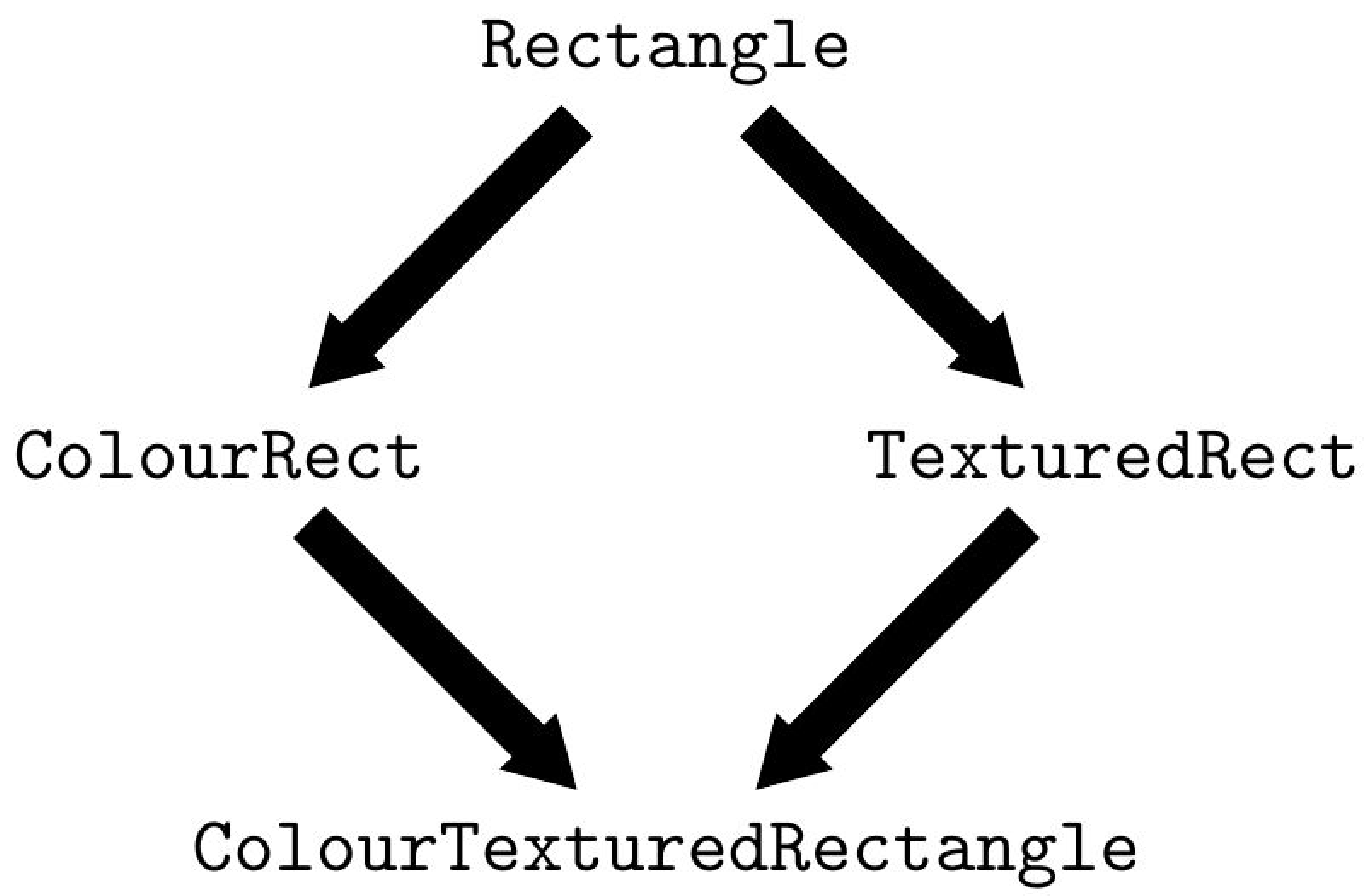
```
ColourRect::ColourRect (int w, int h, int col):
                         Rectangle(w,h), Colour(col) {
  // nothing else to do
}
```

Multiple Inheritance

- Problems with multiple inheritance:
  - What if two parents have a function with the same name?
  - This is why Java dropped multiple inheritance
  - Diamond Problems

Multiple Inheritance

Multiple Inheritance



- The two classes ColourRect and TexturedRect inherit from Rectangle, and ColourTexturedRect inherits both from ColourRect and TexturedRect
- If both ColourRect and TexturedRect override a method in Rectangle which should ColourTexturedRect use?

Possible solutions for multiple inheritance

- In python:
  - The list of classes is ordered and the "first" method is the list is used
- In C++
  - Virtual classes
  - Traits

Traits

Think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine "policy" or "implementation details". - Bjarne Stroustrup

Traits

- Many languages have traits:
  - C++
  - Python
  - Haskell
  - C#, PHP, Rust ...

Traits

- Many languages have traits:
  - C++
  - Python
  - Haskell
  - C#, PHP, Rust ...

- To understand traits, you need to understand Templates which we will get to later.

Virtual classes

- If two or more classes inherit virtually from a class, only one set of members from the base class will be available
- Virtual classes can act in a similar way to `interaces` in C# and Java

Virtual classes

Virtual classes

In [ ]:

```
class Rectangle {};

class ColourRect : virtual Rectangle {};

class TexturedRect : virtual Rectangle {};

class ColourTexturedRect : ColourRect, TexturedRect {
};
```

Virtual classes

Virtual classes

In [102]:

```
#include <iostream>
using namespace std;

class Rectangle {
   public:
      Rectangle (int w, int h): width(w), height(w) {   std::cout << "Rect:" << w << ','  << h << endl;    }
      int Area(void) { return width*height; }
      ~Rectangle (void) {}
   private:
      int width;
      int height;
};

class ColourRect: public virtual Rectangle {
      public:
        ColourRect( int w, int h, int col): Rectangle(w,h), colour(col) {   std::cout << "ColRect:" << w << ','  << h << end
        int getColour() { return colour; } ;
      private:
        int colour;
};

class TexturedRect : public virtual Rectangle {
      public:
        TexturedRect(int w, int h, int texture): Rectangle(w,h), texNum(texture) {   std::cout << "TexRect:" << w << ','  <<
      private:
        int texNum;
};

class ColourTexturedRect : public TexturedRect, public ColourRect {
    public:
        ColourTexturedRect( int w, int h, int col, int texture):
```

Virtual classes

- Still require great care of the scenarios
- Safest never to get yourself into this problem by using 'Purely Virtual' or Abstract classes

Virtual classes (Purely Virtual / Abstract)

Virtual classes (Purely Virtual / Abstract)

In [111]:

```
#include <iostream>
using namespace std;

class Rectangle {
public:
    Rectangle() {
        // Initialization code for Rectangle
    }
    virtual ~Rectangle() {} // Virtual destructor for proper cleanup
protected:
    int width;
    int height;
};

class TexturedRect : virtual public Rectangle {
public:
    TexturedRect() : Rectangle() {}

    virtual void applyTexture() = 0; // Pure virtual function
    virtual ~TexturedRect() {} // Virtual destructor
};

class ColourRect : virtual public Rectangle {
public:
    ColourRect() : Rectangle() {}

    virtual void applyColour() = 0; // Pure virtual function
    virtual ~ColourRect() {} // Virtual destructor
};

class ColourTexturedRect : public TexturedRect, public ColourRect {
```

Revisiting Public, Protected and Private (i.e. Access Specifiers)

- **Public**: members CAN be accessed from outside the class
- **Protected**: members CANNOT be accessed from outside the class, however, they CAN be accessed in inherited classes.
- **Private**: members CANNOT be accessed (or viewed) from outside the class

Templates

Template Meta-programming

- C++ templates are a great way to avoid duplicated code

Let's say we want to write a function that multiplies three numbers:

Template Meta-programming

- C++ templates are a great way to avoid duplicated code

Let's say we want to write a function that multiplies three numbers:

In [ ]:

```
int multiply3(int i, int j, int k) {
    return i*j*k;
}

std::cout << "Result is: " << multiply3(2,2,2) << std::endl;
```

Template Meta-programming

- C++ templates are a great way to avoid duplicated code

Let's say we want to write a function that multiplies three numbers:

In [ ]:

```
int multiply3(int i, int j, int k) {
    return i*j*k;
}

std::cout << "Result is: " << multiply3(2,2,2) << std::endl;
```

or with doubles:

Template Meta-programming

- C++ templates are a great way to avoid duplicated code

Let's say we want to write a function that multiplies three numbers:

In [ ]:
```
int multiply3(int i, int j, int k) {
    return i*j*k;
}

std::cout << "Result is: " << multiply3(2,2,2) << std::endl;
```

or with doubles:

In [ ]:
```
double multiply3(double i, double j, double k) {
    return i*j*k;
}

std::cout << "Result is: " << multiply3(2.0,2.0,2.0) << std::endl;
```

Templates

- Templates allow us to specify the type later:

Templates

- Templates allow us to specify the type later:

In [ ]:
```
template<typename T>
T multiply3(T i, T j, T k) {
    return i*j*k;
}
```

Templates

- We can call e.g. `multiply3(2.0,2.0,2.0)` or `multiply3(2,2,2)`
- or specify the type directly e.g. `multiply3<double>(2.0,2.0,2.0)`

Templates

- We can call e.g. `multiply3(2.0,2.0,2.0)` or `multiply3(2,2,2)`
- or specify the type directly e.g. `multiply3<double>(2.0,2.0,2.0)`

In [41]:

```cpp
#include<iostream>
using namespace std;

template<typename T>
T multiply3(T i, T j, T k) {
    return i*j*k;
}

int main(){
    cout << "Implicit type: (double) " << multiply3(2.0,2.0,2.0)  << " (int)" << multiply3(2,2,2) <<endl;
    cout << "Explicit type: (double) " << multiply3<double>(2.0,2.0,2.0) << endl;
    cout << "Explicit type: (int) with implicit cast " << multiply3<int>(2.0,2.0,2.0) << endl;
}
```

```
Implicit type: (double) 8 (int)8
Explicit type: (double) 8
Explicit type: (double) 8
```

Templates

Templates:

- Reduce code avoiding duplication
- Are easy to read (in general)
- They are handled by the Pre-Compiler, so as a rule are defined in header files

Templates can have one or more template parameters, which can be:

- Type parameters
- Non-Type parameters, for example an integer
- Template parameters, so you can have a templated template parameter

But,

- Can be problematic to debug

Templates

> *Reminder: Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones*

- You can initialise templates with defaults:

```cpp
template<int N, typename T = int>
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};
```

- if nothing else is spefified the type will be int

Template Type Parameters

- Template Type Parameters are template parameters that refer to a type

  `typename name` or `class name`

Template specialisation and overloading

What will go wrong with this:

Template specialisation and overloading

What will go wrong with this:

In [ ]:

```
template<int N, typename T>
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};
```

Template specialisation and overloading

What will go wrong with this:

In [ ]:

```
template<int N, typename T>
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};
```

- Compiler gets stuck as base cases are not handled.

Template specialisation and overloading

- We can change the behaviour for specific values:

Template specialisation and overloading

- We can change the behaviour for specific values:

In [ ]:

```
template<int N, typename T>
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};

template<typename T>
struct fib<1,T> {
    static constexpr T value = 1;
};

template<typename T>
struct fib<0,T> {
    static constexpr T value = 0;
};
```

Template specialisation and overloading

Template specialisation and overloading

In [51]:

```
#include <iostream>
using namespace std;

template<int N, typename T>
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};

template<typename T>
struct fib<1,T> {
    static constexpr T value = 1;
};

template<typename T>
struct fib<0,T> {
    static constexpr T value = 0;
};

int main() {
    constexpr int fib5 = fib<5,int>::value;   // fib5  will be 5
    constexpr int fib10 = fib<10,int>::value; // fib10 will be 55
    constexpr int fib1 = fib<1,int>::value;   // fib1  will be 1

    cout << "fib5: " << fib5 <<endl;
    cout << "fib10: " << fib10 <<endl;
    cout << "fib1: " << fib1 <<endl;
    return 0;
}
```

```
fib5: 5
fib10: 55
```

Template specialisation and overloading: constexpr

Why did we use `constexpr` in the example

```
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};
```

Template specialisation and overloading: constexpr

Why did we use `constexpr` in the example

```
struct fib {
    static constexpr T value = fib<N-1,T>::value + fib<N-2,T>::value;
};
```

- Tells the compiler it will not be changed as with `const` for variables.

Template Classes

Template Classes

In [42]:

```cpp
#include <iostream>
using namespace std;

template <typename T1, typename T2>
class Pair {
public:
    T1 first;
    T2 second;

    Pair(T1 a, T2 b) : first(a), second(b) {}

    void display() {
        std::cout << "(" << first << ", " << second << ")" << std::endl;
    }
};


int main() {
    Pair<int, double> myPair(5, 6.7);
    myPair.display();  // Output: (5, 6.7)

    Pair<string, char> anotherPair("Hello", 'A');
    anotherPair.display();  // Output: (Hello, A)

    return 0;
}
```

```
(5, 6.7)
(Hello, A)
```

The C++ Standard Template Library (STL)

The C++ Standard Template Library (STL)

- provides common programming data structures and functions
- It contains algorithms, e.g. for sorting or searching
- It contains containers, e.g. arrays, vectors or lists

Vector

- Essentially a C++ vector is only a wrapper around a C array

```
import <vector>
```

```
vector<int> a;
```

- Vectors can be resized dynamically
- Memory allocation/free is handled by the vector

The <vector> header

Some methods:

- `begin()` iterator pointing at first element of vector
- `end()` iterator pointing at last element of vector
- `size()` the number of elements in the vector.
- `resize(n)` resizes the vector to `n`
- `push-back` add element at the end
- ...

The <vector> header

The <vector> header

In [30]:

```cpp
#include <vector>
#include <iostream>

using namespace std;

int main(){
    vector<int> v(10);
    int n = v.size();
    cout << "The number of elements is: " << n << endl;

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

    v.push_back(15);
    n = v.size();
    cout << "The number of elements is: " << n << endl;
    cout << "The last element is: " << v[n - 1] << endl;
}
```

```
The number of elements is: 10
v = [ 0 0 0 0 0 0 0 0 0 0 ]
The number of elements is: 11
The last element is: 15
```

The <vector> header (Copying)

- What kind of copy is occuring here?

The <vector> header (Copying)

- What kind of copy is occuring here?

In [34]:

```
#include <iostream>
#include <vector>

class MyBigClass {
public:
    int value;
    MyBigClass(int v) : value(v) {}
    MyBigClass(const MyBigClass& other) : value(other.value) {}
};

int main() {
    // Original vector of objects
    std::vector<MyBigClass> original;
    original.push_back(MyBigClass(10));
    original.push_back(MyBigClass(20));

    // Deep copy
    std::vector<MyBigClass> copy = original;

    // Modifying the original vector's first element
    original[0].value = 100;

    // Only the original vector is affected
    std::cout << "Original first element: " << original[0].value << std::endl;
    std::cout << "??? copy first element: " << copy[0].value << std::endl;

    return 0;
}
```

The <vector> header (Copying)

- What kind of copy is occuring here?

In [34]:

```cpp
#include <iostream>
#include <vector>

class MyBigClass {
public:
    int value;
    MyBigClass(int v) : value(v) {}
    MyBigClass(const MyBigClass& other) : value(other.value) {}
};

int main() {
    // Original vector of objects
    std::vector<MyBigClass> original;
    original.push_back(MyBigClass(10));
    original.push_back(MyBigClass(20));

    // Deep copy
    std::vector<MyBigClass> copy = original;

    // Modifying the original vector's first element
    original[0].value = 100;

    // Only the original vector is affected
    std::cout << "Original first element: " << original[0].value << std::endl;
    std::cout << "??? copy first element: " << copy[0].value << std::endl;

    return 0;
}
```

The <vector> header (Copying)

- What kind of copy is occuring here?

The <vector> header (Copying)

- What kind of copy is occuring here?

In [36]:

```cpp
#include <iostream>
#include <vector>

class MyBigClass {
public:
    int value;
    MyBigClass(int v) : value(v) {}
};

int main() {
    // Original vector of pointers
    std::vector<MyBigClass*> original;
    original.push_back(new MyBigClass(10));
    original.push_back(new MyBigClass(20));

    // Shallow copy
    std::vector<MyBigClass*> copy = original;

    // Modifying the original vector's first element
    original[0]->value = 100;

    // Both vectors are affected
    std::cout << "Original first element: " << original[0]->value << std::endl;
    std::cout << "??? copy first element: " << copy[0]->value << std::endl;

    // Cleanup to avoid memory leaks
    for (MyBigClass* ptr : original) {
        delete ptr;
    }
```

The <vector> header (Copying)

- What kind of copy is occuring here?

In [36]:

```cpp
#include <iostream>
#include <vector>

class MyBigClass {
public:
    int value;
    MyBigClass(int v) : value(v) {}
};

int main() {
    // Original vector of pointers
    std::vector<MyBigClass*> original;
    original.push_back(new MyBigClass(10));
    original.push_back(new MyBigClass(20));

    // Shallow copy
    std::vector<MyBigClass*> copy = original;

    // Modifying the original vector's first element
    original[0]->value = 100;

    // Both vectors are affected
    std::cout << "Original first element: " << original[0]->value << std::endl;
    std::cout << "??? copy first element: " << copy[0]->value << std::endl;

    // Cleanup to avoid memory leaks
    for (MyBigClass* ptr : original) {
        delete ptr;
    }
```

The `string` class (not `string.h`)

- You may already have noticed that C strings can cause a lot of unexpected behaviour, the `string` class makes some of this easier

The `string` class (not `string.h`)

- You may already have noticed that C strings can cause a lot of unexpected behaviour, the `string` class makes some of this easier

In [117]:

```
#include <string>
#include <iostream>
using namespace std;

int main(){
    string str;
    str = "Hello world";
    cout << str << endl;
    str.push_back('s');
    cout << str << endl;
}
```

```
Hello world
Hello worlds
```

The `string` class (not `string.h`)

- `resize()` resizes your string, can shorten or lengthen it
- `length()` gives the length of your string
  - Can be very annoying in C to have to always pass the length of a `char*` around with it or search for `\0`
- `shrink_to_fit()` resize to correct length

Other useful C++ stl headers

- C++ library headers <algorithm>, <iomanip>, <list>, <queue>, <string>, <bitset>, <ios>, <locale>, <set>, <strstream>, <complex, <iosfwd>, <map>, <sstream>, <typeinfo>, <deque>, <iostream>, <memory>, <stack>, <utility>, <exception>, <istream>, <new>, <stdexcept>, <valarray>, <fstream>, <iterator>, <numeric>, <streambuf>, <vector>, <functional>, <limits>, <ostream>
- C++ library headers added in C++11 <array>, <condition_variable>, <mutex>, <scoped_allocator>, <type_traits>, <atomic>, <forward_list>, <random>, <system_error>, <typeindex>, <chrono>, <future>, <ratio>, <thread>, <unordered_map>, <codecvt>,<initializer_list>, <regex>, <tuple>, <unordered_set>
- Headers added in C++14 <shared_mutex>
- Headers added in C++17 <any>, <execution>, <memory_resource>, <string_view>, <variant>, <charconv>, <filesystem>, <optional>
- Headers added in C++20 <barrier>, <concepts>, <latch>, <semaphore>, <stop_token>, <bit>, <coroutine>, <numbers>, <source_location,

Other useful C++ aspects: auto and decltype

- The `auto` keyword uses the initialization expression of a declared variable to deduce its type

  `auto i = 33` will result in an `int`

  `auto i = 33.0` will result in an `double`

Example of (without) auto:

Example of (without) auto:

In [25]:

```
#include <vector>
#include <utility> // std::pair, std::make_pair
#include <iostream>
using namespace std;

int main() {
    // Creating a vector of pairs
    vector<pair<int, string>> vec;

    // Adding some pairs to the vector
    vec.push_back(make_pair(1, "Apple"));
    vec.push_back(make_pair(2, "Banana"));
    vec.push_back(make_pair(3, "Cherry"));

    // Iterating over the vector using a range-based for loop with using iterator
    for (vector<std::pair<int, string>>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << "Element: " << it->first << ", " << it->second << endl;
    }

    return 0;
}
```

```
Element: 1, Apple
Element: 2, Banana
Element: 3, Cherry
```

Example of (with) auto

Example of (with) auto

In [26]:

```cpp
#include <vector>
#include <utility> // std::pair, std::make_pair
#include <iostream>
using namespace std;

int main() {
    // Creating a vector of pairs
    vector<pair<int, string>> vec;

    // Adding some pairs to the vector
    vec.push_back(make_pair(1, "Apple"));
    vec.push_back(make_pair(2, "Banana"));
    vec.push_back(make_pair(3, "Cherry"));

    // Iterating over the vector using a range-based for loop with 'auto'
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        cout << "Element: " << it->first << ", " << it->second << endl;
    }

    return 0;
}
```

```
Element: 1, Apple
Element: 2, Banana
Element: 3, Cherry
```

Other useful C++ aspects: auto and decltype

- We can also use `decltype` to infer the types:

Other useful C++ aspects: auto and decltype

- We can also use `decltype` to infer the types:

In [29]:

```
#include <iostream>
using namespace std;

int main() {
    int i = 33;
    decltype(i) j = i * 2;
    cout << "i:" << i << ", j:" <<j <<endl;
    return 0;
}
```

```
i:33, j:66
```

Summary

- Class Inheritance
- Template functions and classes
- Standard Template Library
    - Vector
    - String
- `auto` and `decltype`