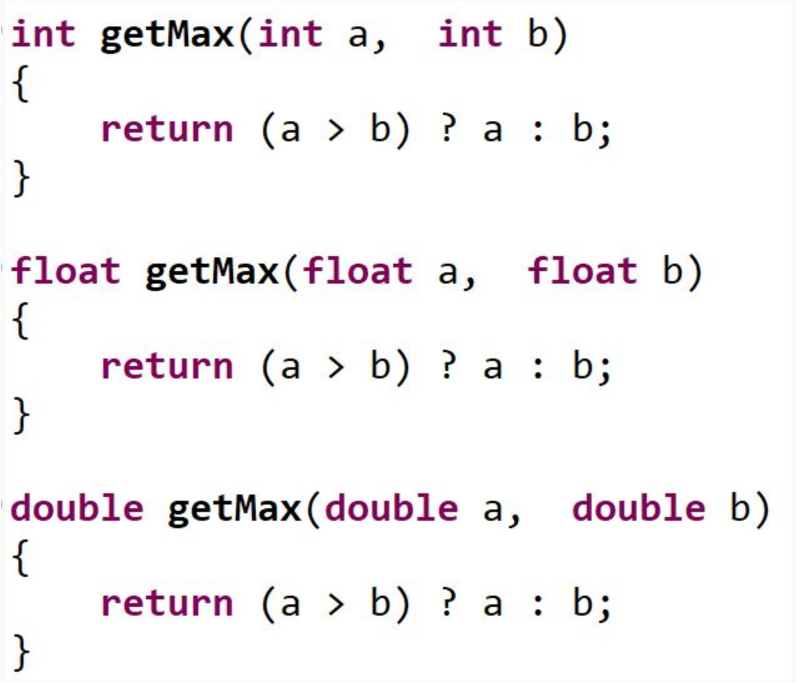
Generic Programming

- Generic programming
- C++ Templates
- Function templates
- Class templates
- STL quick overview

Example: same function, different data-types



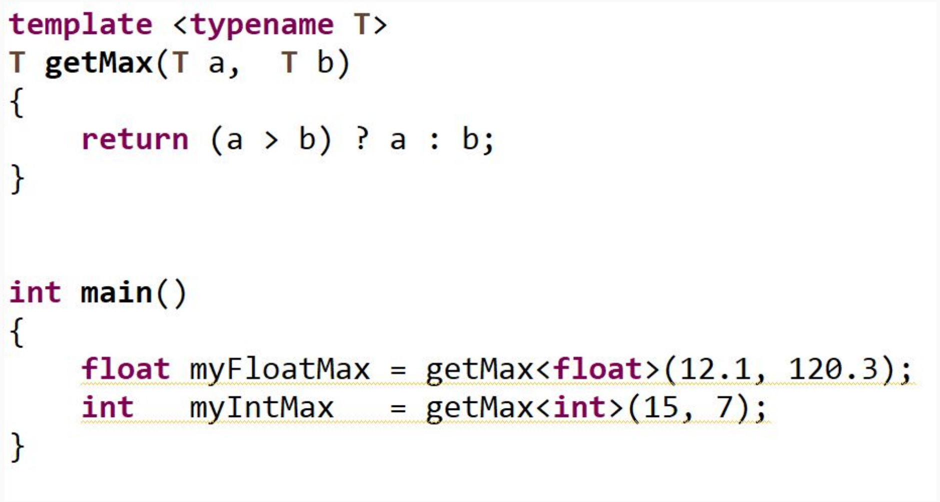
```
int getMax(int a,  int b)
{
    return (a > b) ? a : b;
}

float getMax(float a,  float b)
{
    return (a > b) ? a : b;
}

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

- **Same code is duplicated in order to support different data types**

What is generic programming?

- A programming paradigm in which types are abstracted and common code for all types is written only once
- Source code will be processed and transformed by the compiler before generating the object code
- Less work on the programmer’s side, more work on the compiler’s side
- C++ implements generic programming via the use of “Templates”
- Also called meta-programming (code that generates code)
- Java’s Generics is the closest thing to C++ templates

Example: same function, different data-types



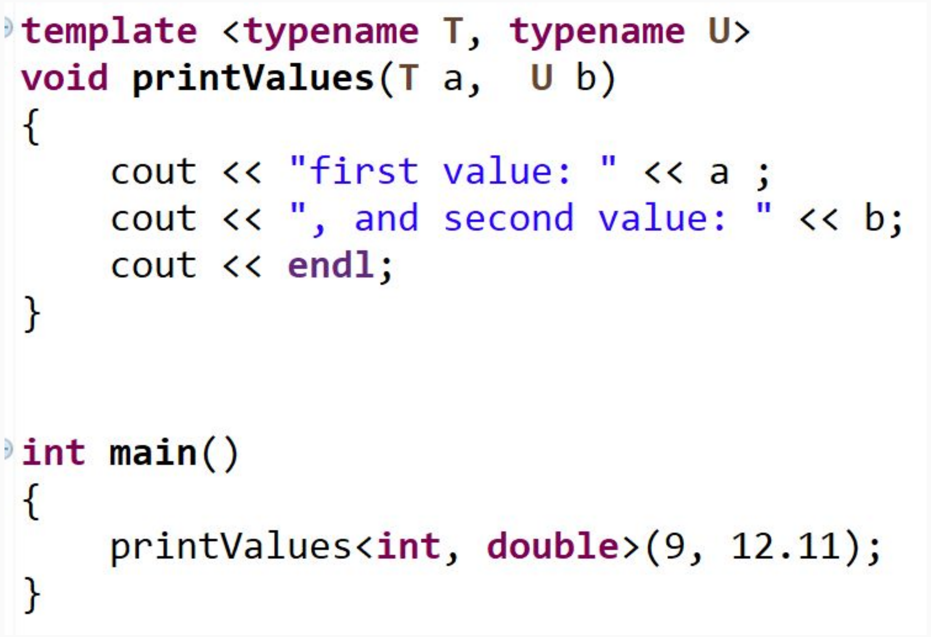
```
template <typename T>
T getMax(T a,  T b)
{
    return (a > b) ? a : b;
}


int main()
{
    float myFloatMax = getMax<float>(12.1, 120.3);
    int   myIntMax   = getMax<int>(15, 7);
}
```

- By using a template, the compiler will generate the needed code to support all needed types
- The type is abstracted as T
- Templates applied to functions are called function templates

C++ Templates

- both declaration and definition of a template must reside in the same file (either .h or .cpp)
- Compiler generates object code for generic source code on an “as-needed” basis
- **template <typename T>** is equivalent to **template <class T>**
    - You can use **typename** or **class** interchangeably

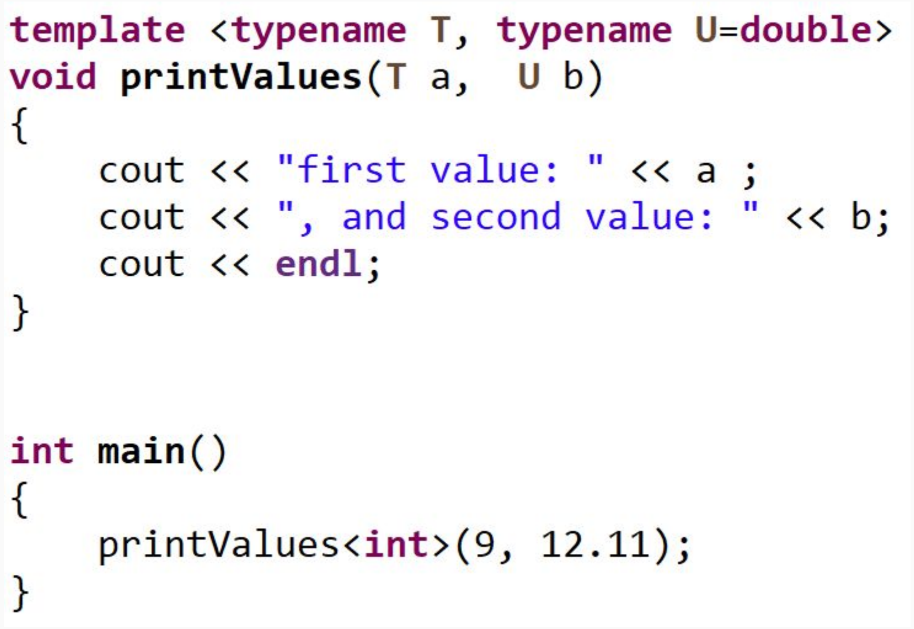
Templates can support multiple types



```
template <typename T, typename U>
void printValues(T a,  U b)
{
    cout << "first value: " << a ;
    cout << ", and second value: " << b;
    cout << endl;
}


int main()
{
    printValues<int, double>(9, 12.11);
}
```

- T and U may or may not have the same type
- Templates can support as many different typenames as needed

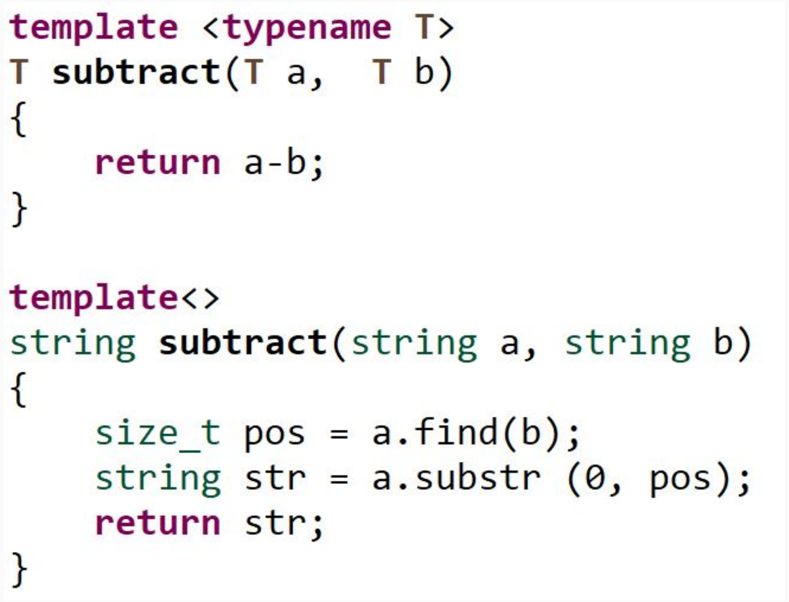
Templates can be provided a default type



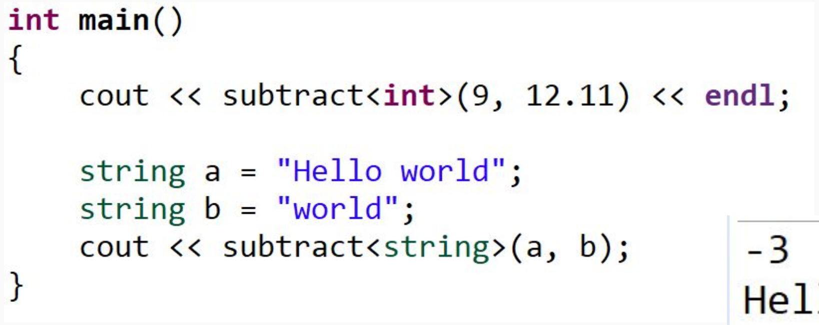
```
template <typename T, typename U=double>
void printValues(T a,  U b)
{
    cout << "first value: " << a ;
    cout << ", and second value: " << b;
    cout << endl;
}


int main()
{
    printValues<int>(9, 12.11);
}
```

- **U will be substituted by “double” if no type input was provided to the template**

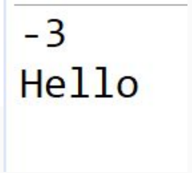
Templates can be specialized for specific types



```cpp
template <typename T>
T subtract(T a,  T b)
{
    return a-b;
}

template<>
string subtract(string a, string b)
{
    size_t pos = a.find(b);
    string str = a.substr (0, pos);
    return str;
}
```



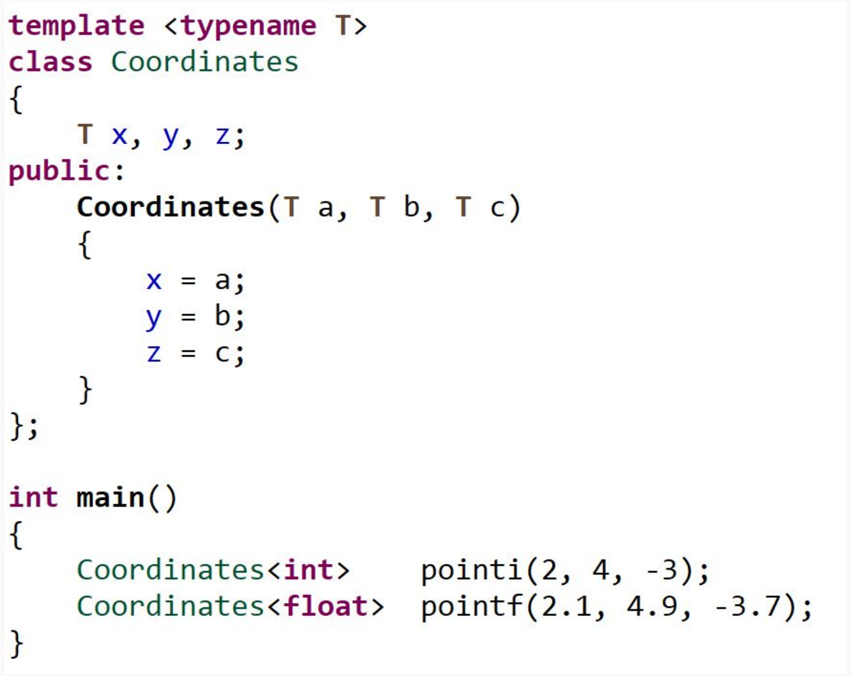
```cpp
int main()
{
    cout << subtract<int>(9, 12.11) << endl;

    string a = "Hello world";
    string b = "world";
    cout << subtract<string>(a, b);
}
```



```
-3
Hello
```

- **The subtract function template has different implementation in the case of string type**

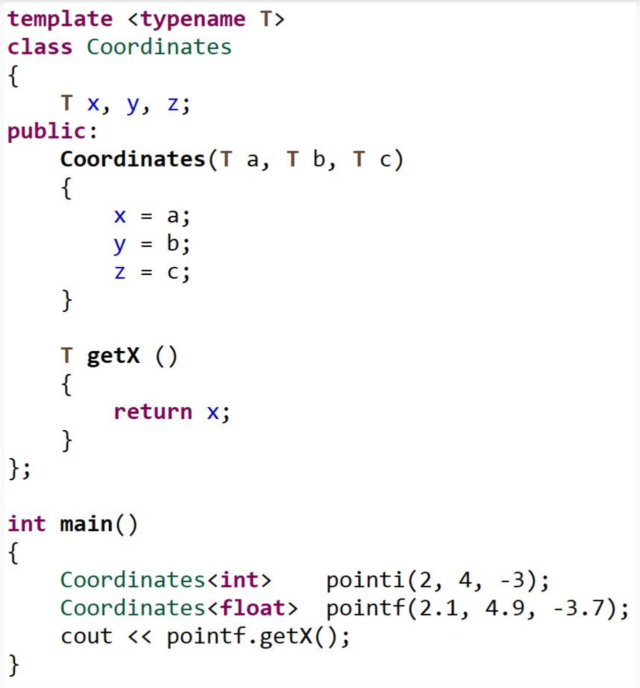
Class Templates



```
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }
};

int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float>  pointf(2.1, 4.9, -3.7);
}
```

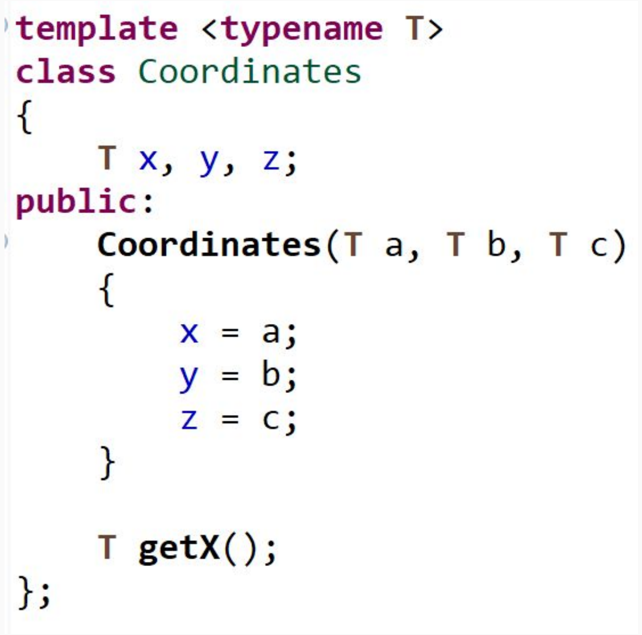
- **Templates can be applied to classes**

Class Templates

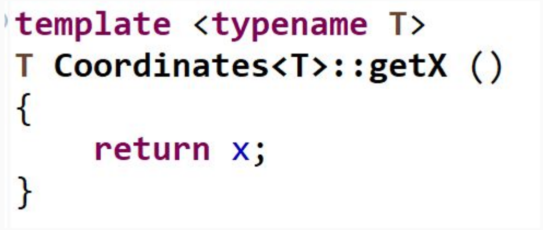

```cpp
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }

    T getX ()
    {
        return x;
    }
};

int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float>  pointf(2.1, 4.9, -3.7);
    cout << pointf.getX();
}
```

- **Method members can be defined within the class**

Class Templates



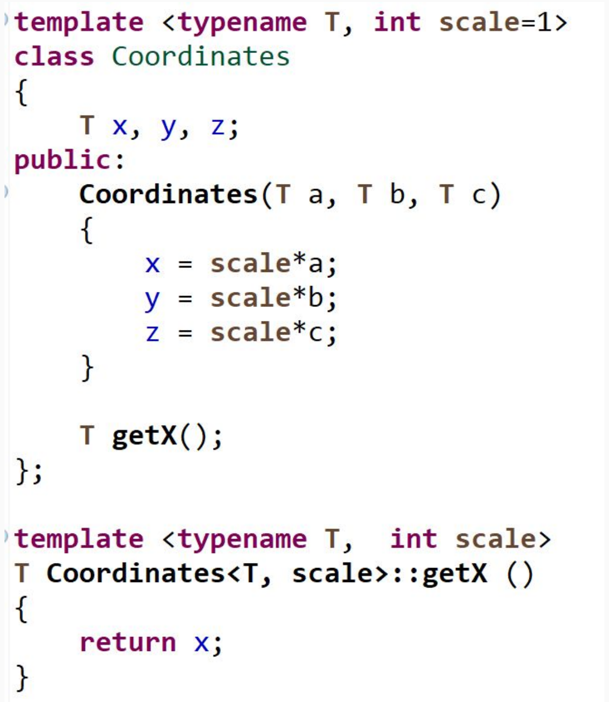
```
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }

    T getX();
};
```



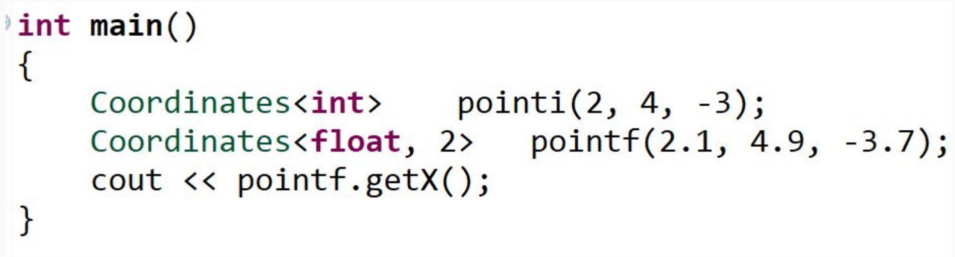
```
template <typename T>
T Coordinates<T>::getX ()
{
    return x;
}
```

- **Method members can be defined outside of the class, but be careful with the syntax**

Class Templates: non-type parameters



```cpp
template <typename T, int scale=1>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = scale*a;
        y = scale*b;
        z = scale*c;
    }

    T getX();
};

template <typename T,  int scale>
T Coordinates<T, scale>::getX ()
{
    return x;
}
```



```cpp
int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float, 2>   pointf(2.1, 4.9, -3.7);
    cout << pointf.getX();
}
```

- Templates can also accept regular parameters (similar to the way functions behave)

Templates: pros

- Avoid code duplication
  - which reduces development time
  - And also provides more safety when updating and maintaining the code
- Preferable to C-macros because they provide type safety
- Preferable to deriving classes when performance is on stake
  - In other terms, compile time polymorphism is prefered to run time polymorphism when efficiency is terribly needed

Templates: cons

- Heavy use of templates may result in “code bloating” which is the generation of huge object files
  - Which also means long compilation and build time
  - Which also means a huge executable files
- You lose code hiding when using templates because you have to provide the implementation in the header file
- Templates had the reputation of being hard to debug due to cryptical error messages provided by compilers
  - Since the compiler generates additional code for templates, it is hard to locate during run time the origin of an error when debugging
  - However, compilers had made a lot of enhancements since …

Templates debugging

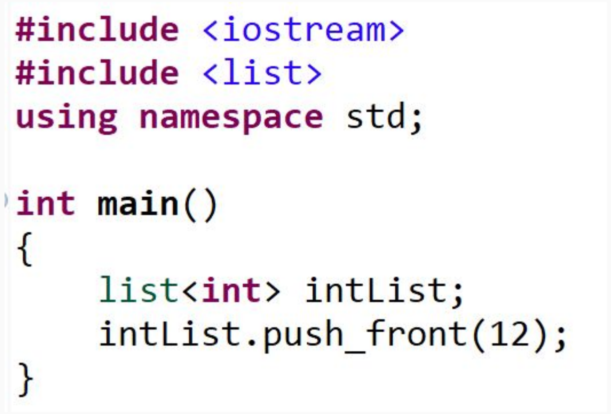
- This is exactly how you would look like when debugging code with heavy class template usage

The Standard Template Library (STL)

- STL's design and architecture is credited largely to Alexander Stepanov
  - Alex is a computer scientist and generic programming advocate
- STL was not part of the original standard library (it was added later on)
- STL provides a set of class templates implementing the basic data structures
- STL main components are: containers, iterators, algorithms and functions

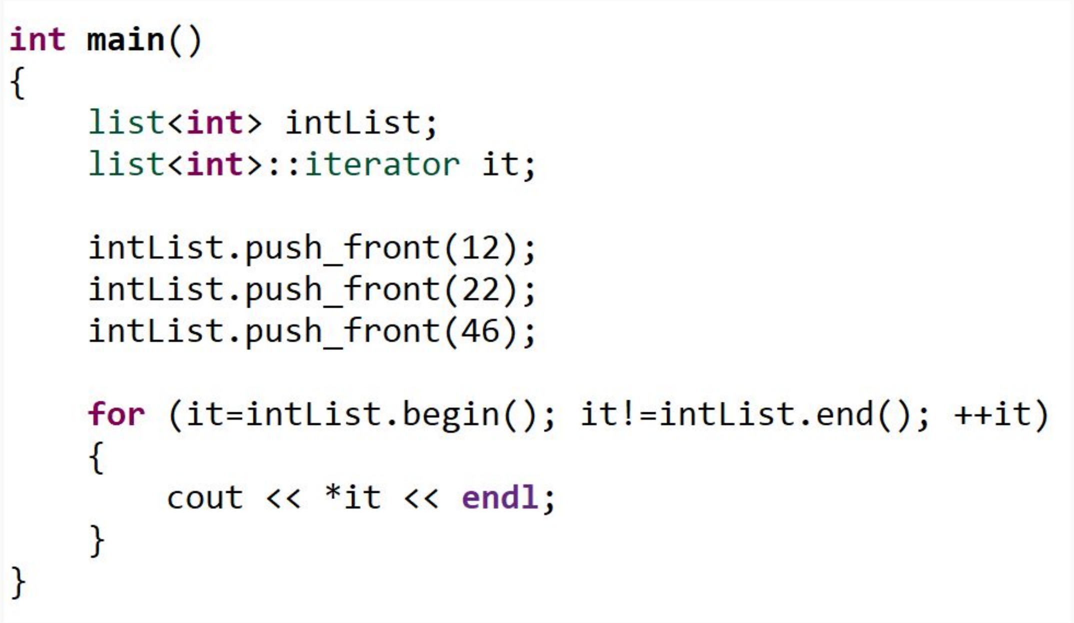
STL components: Containers

- Generic classes to store objects and data
- Sequential containers:
  - list
  - Vector
  - arrays (since C++11)
  - ...
- Associative containers:
  - map
  - set
  - ...



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

int main()
{
    list<int> intList;
    intList.push_front(12);
}
```

STL components: Iterators



```
int main()
{
    list<int> intList;
    list<int>::iterator it;

    intList.push_front(12);
    intList.push_front(22);
    intList.push_front(46);

    for (it=intList.begin(); it!=intList.end(); ++it)
    {
        cout << *it << endl;
    }
}
```

- **Iterators make it possible to iterate over containers and accessing their values**
- **Iterators are abstraction to access different types of containers**

STL components: Algorithms

- STL provides a large collection of algorithms to be applied on containers
  - Sorting
  - Searching
  - Comparing
  - ...

STL components: Functions

- STL offers classes that overload the function call operator: operator()
- This is a C++ techniques that is also known under the name of: Functors
    - Functor = function object
    - The idea is that regular functions don't keep state. Objects on the other hand do, so by overloading the function call operator() we'll be imitating the function behavior while keeping the object's state