SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming
C++ Metaprogramming and STL

Giorgio Grisetti

Metaprogramming

Metaprogramming is a technique that minimizes the code replication by generating stubs (templates) that get instantiated to specific code instances at **compile time.**

If properly used, instantiation at compile time allows for hard optimizations (loop unrolling, simd extensions etc), that provide significant runtime improvements.

Template Metaprogramming

Well known algorithms and data structures to be adapted for a compiled language require a specific instantiation that depends

- on the data types
- on algorithms

Examples:

List/Stack/Heap/Tree of (integers, float, string, classes)

Sorting algorithm (container to be sorted, comparison operator)

Matrices/Vectors (float, double,complex)

Templates

Overall sintax:

add

```
template <Arguments>
```

before a class or a function.

Inside the function/class, use the arguments as if they were going to be substituted (in fact they are, at compile time).

<Arguments> can be either “typenames” or values

Templates, Type and Value arguments

```
struct Vec3f {
  static const int size=3;
  float values[size];
  float& at(int pos){
    return values[pos];
  }
};

struct Vec4d {
  static const int size=4;
  double values[size];
  double& at(int pos){
    return values[pos];
  }
};
Vec3f v3_1, v3_2;
Vec3f v4_1;
```

```
template <typename T, int s>
struct VecT_ {
  static const int size=s;
  T values[size];

  T& at(int pos){
    return values[pos];
  }
};
…
VecT_<float, 3> v3_1, v_3_2;
VecT_<double, 4> v4_1;
```

with templates we
can define the stuff once
and use it many times

Template functions

Templates can be also be used in functions, to make them parametric w.r.t the data type

A generic min function, supporting all types that define a comparison operator is the following

```
template <typename T>
const T& myMinTemplate(const T&a,
                       const T&b) {
  if (a<b) return a;
  return b;
}

int main(){
  cout << myMinTemplate(3,4) <<
endl;
  cout << myMinTemplate(4.5,3.8) <<
endl;
}
```

Code Example: Merge Sort

In sort_1.cpp we see the plain merge sort (for arrays)

In sort_2 we generalize it to arrays of arbitrary data types

Function Objects

```
float timesTree(float f){
  return 3*f;
}

…
float v=3;
v=timesThree(v);
```

```
struct TimesThree {
  float operator()(float f) {
    return 3*f;
  }
};
…
float v=3;
TimesThree timesThree; // declare object
v=timesThree(v);
```

By defining the () operator, one can construct an object that behaves syntactically similar to a function. Function Objects can be passed as parameters, or at type parameters.

Function Objects

```
struct CompareInt {
  bool operator()
   (const int a, const int b) const
  {
    return a<b;
  }
};
```

```
struct CompareFloat {
  bool operator()
   (const float a, const float b)
const
  {
    return a<b;
  }
};
```

Function objects are typically used to pass “traits” about types to algorithms, think about “compare” in sort.

Code Example: Merge Sort

In sort_3.cpp we generalize the sort algorithm to take arbitrary ordering operations

We do so by delegating the comparison to a function object.

Using and Constexpr

using defines a type alias; It helps the eyes when dealing with long template expressions.

```
using <name> = <type>;
using MyInt= long int;
using V3=Vec_<float, 3>;
MyInt i; // equivalent to long_int i;
V3 v;    //equivalent to Vec_<float, 3>;
```

constexpr serves to specify that an expression is a compile time constant and forces its evaluation once. This helps the compiler and the template system.

Templates (declaration/definition)

Similar to the “plain c++” also template functions can be defined outside the body of a class. The mechanism is the same, and the template clause should be repeated before each method definition

Standard Template Library

Is a library of data structures and algorithms that comes together with the compiler

STL is not part of the language, but it is pretty standard (like the libc is not directly part of the compiler)

Headers of the STL do not need the .h extension (map.h → map)

Prominent aspects are

- Containers
- Algorithms
- Streams
- Threads

STL Containers

Containers are types that define collections of objects of some basic type

In the STL, a container might have some “interfaces” such as

- Random Accessible
- Copy Constructible
- Assignable
- Iterable
- and so on.

They offer a symmetric interface to iterate over the elements, insert, remove and in general manipulate the data structure.

STL containers are templates defined by the contained type and potentially some other entity (e.g. comparison operator)

Containers: iterators

A standard way to access the elements of a container is through an “iterator”

An iterator of type **<ContainerType>::iterator** or **<ContainerType>>::const_iterator**, can be usually moved forward and backwards in the container.

The iterator to the first element is typically obtained by the **begin()** member function.

The iterator to the last element is the end().

A classical loop to iterate over the elements of a container is

```
for (ContainerType::iterator it=c.begin(); it!=c.end(); ++it) {
  cout << *it << endl; // note the dereference.
}
```

Standard library: list

The STL list defines a double linked list. Elements in the list do not need to be sortable. As all containers the list implements deep copy, and copy construction.

```
#include <list>
using namespace std;
using IntList=list<int>; // list of integers;
int main() {
  IntList l;
  l.push_back(1);
  l.push_back(2);
  l.push_back(3);
  l.push_front(5);

  // use of keyword auto to determine the type of a value from the
  // return type of an expression
  for (auto it=l.begin(); it!=l.end(); ++it)
   cerr <<  *it << endl;
  auto l2=l; // deep copy, l1 and l are different objects
}
```

Standard library: list

Removing an elements in the list can be done with an iterator. Let's remove the element at odd positions

```
int count=0;
for (it=l.begin(); it!=l.end(); ++it){
  if (count%2) {
   auto it_copy=it;
   ++it;
   l.erase(it_copy);
  }
  ++count;
}
```

A common mistake is to "increment" an iterator after erase. This is wrong. The element has beed destroyed.

```
++it;
l.erase(it);
```

STL: vector

The vector class defines a variable size array, that supports random access by index. Erasing and removing, done as in the case of list

```
#include <vector>
using IntVector=std::vector<int>; // vector of integers;
int main() {
  IntVector v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);
  v.push_front(5);
   // use of keyword auto to determine the type of a value from the  retur type of an expression
  for (auto it=v.begin(); it!=v.end(); ++it);
   cerr <<  *it << endl;
  auto v2=v;
  v[1]=3; // element access to the second component (first is at pos 0)
}
```

STL: vector

Removal in a vector is safely done with iterators (as in the list)

```
auto it=v.begin()+2; // get iterator to 2nd element
v.erase(it);

// this “compacts the vector”
```

STL: iterating on containers

For list and vectors, the type <ContainerType>::value_type specifies the type of the items stored in the container.

If a class implements begin() and end() consistently one could use also the compact version of for loop.

On the right you see 3 versions of for loop on containers

```
IntVector v;
...
for (int val: v) // with copy
  cerr << val << endl;


for (int& val: v) // without copy, modify the element
  val+=2;


for (const int& val: v) { // without copy, no modifications
  cerr << val << endl;
}
```

STL: sorting

Some containers can be sorted, provided that we define an ordering.

The ordering is defined either by

- **the <operator, defined in the item class**
- by providing a function object that tells which of two elements comes first
- by defining a lambda within the call to sort.

```
#include <algorithm>

IntVec v;

…

std::sort(v.begin(), v.end());

//sorts ascending
```

STL: sorting

Some containers can be sorted, provided that we define an ordering.

The ordering is defined either by

- the <operator, defined in the item class
- **by providing a function object that tells which of two elements comes first**
- by defining a lambda within the call to sort.

```
#include <algorithm>
struct CompDescending{
bool operator()(const int&a, const int& b){
   return b<a;
}
};
IntVec v;
…
std::sort(v.begin(), v.end(), CompDescending);
//sorts descending
```

STL: sorting

Some containers can be sorted, provided that we define an ordering.

The ordering is defined either by

- the <operator, defined in the item class
- by providing a function object that tells which of two elements comes first
- **by defining a lambda within the call to sort**.

```
#include <algorithm>
IntVec v;
…
std::sort(
 v.begin(),
 v.end(),
 [&](const int&a,
      const int& b)→bool {
      return b<a;
  }
);
//sorts ascending
```

STL: set

A set is a container that supports unique values. The elements in a set should define an ordering, to support comparison.

You can either define a < operator in the item class, or define a comparison function object.

The comparison object/operator becomes part of the type;

```
#include <set>
struct MyS{
 MyS(int _a=0, int _b=0):
 a(_a), b(_b){}
 int a; float b;
};
struct MySCompare {
 bool operator()(const MyS& s1,
const MyS& s2) {
  return s1.a<s2.a || (s1.a==s2.a &&
s1.b < s2.b);
 }
};
using MySSet =
  set<MyS, MySCompare>;
```

STL: set

Elements can be inserted/removed from the set using the **`insert`** and **`erase`** methods.

The elements in a set are sorted.

Iterating on a set is similar to iterating on a list or an array.

Non sorted sets include std::hash_set, more efficient in many cases. You might need to provide a hash function.

```
MySSet s;
s.insert(MyS(1,2.3));
s.insert(MyS(0,3.5));
s.insert(MyS(2,2.1));
auto it=s.find(MyS(0, 3.5));
if (it!=s.end()){
 cerr << “element in set” << endl;
 s.erase(it);
}
```

STL: map

Maps are associative containers in the form **<key, value>**. They look pretty much like arrays, however both keys and values can be anything.

The element of a map is a **pair <key_type, value_type>** (not a value alone).

Keys should be “ordered” (same as in a set).

Default maps elements are sorted by key.

A unique “value type” can exist for a given key.

Changing the value of the second element in a set does not alter the structure of a map (the ordering). Changing the key does (requires removal/insertion).

STL: map

```
#include <map>
using IntFloatMap=std::map<int, float>;

IntFloatMap m;
…
m.insert(std::make_pair(3,0.7));
auto it=m.find(3); // seek for element 3
it.second = 0.5;   // we change from 0.7 to 0.5;
m.erase(it);

//alternative
m[5]=3.7; // insert new element
m[-1]=2.8; // insert new element
m[5]=4.5; // change value of existing element at key 5;
```

STL: string

And yes, the standard library defines a string object that supports comparison, assignment and all the sugar you need.

Beware that operating on strings rolls back on heap manipulation, and this might lead to performance issues;

```
#include <string>
#include <map>
using StringFloatMap=std::map<string, float>;
StringFloatMap sm;
sm[“giorgio”]=0.5;
sm[“luca”]=0.8;
auto it = sm.find(“giorgio”);
*(it.second)=0.9;
```

STL: streams and files

We have been using streams so far to display the output of our programs on standard output. Files behave much like the same;

Reading from a stream

```
<stream_name> >> <variable>
```

Writing to a stream

```
<stream_name> << <value>
```

Opening a file for write

```
#include <fstream>

ofstream os;

os.open("my_file");

os << "hello world on file" << endl;

os << 0.5 << endl;

os.close();
```

STL: streams and files

We have been using streams so far to display the output of our programs on standard output. Files behave much like the same;

Reading from a stream

<stream_name> >> <variable>

Writing to a stream

<stream_name> << <value>

Opening a file for read

```
#include <fstream>
ifstream is;
is.open("my_file");
std::string s;
is >> s;
cerr << s;
float f;
is >> f ;
cerr << f;
os.close();
```

Streams, read a data file

```
#include <fstream>
#include <list>
#include <static_vec.h> // remember that?
using Vec3f = Vec3_<float,3>;
using Vec3fList = std::list<Vec3f>
using namespace std;

int main() {
  ifstream is(“my_input.txt”);
  Vec3fList l;
  while (is.good()) {
    Vec3f p;
    for (int i=0; i<p.dimension(); ++i)
     is >> p.at(i);
    l.push_back(p);
  };
}
```

STL: smart pointers

As C programmers you might have learned that dealing with dynamic memory can be an issue.

C++ provides a set of “managed” pointers to automate the allocation and deallocation

Smart pointers are classes defined on top of the standard pointers, that implement a reference counter and some sharing mechanism.

Overriding copy constructors, destructors and assigment operator, smart pointers free the expert user from the burden.

Smart pointers come in three flavors

- shared pointers
- unique pointers
- weak pointers

STL: shared pointers

Shared pointers are the easiest. They can be assigned, copied and the system behaves consistently. Their use is pretty much like the Java references.

The main issue is that you can create “loops” with shared pointers.

In this case, the corresponding memory is never deallocated.

Rationale: use shared pointers if you are sure you have no loops in your structures. (double linked list?)

```
#include <memory>
using MySPtr =std::shared_ptr<MyS>;
{
  MySPtr p;
  {
    MySPtr p1(new MyS);
    p=p1; // p and p2 point to the same object;
  } // destroyed p1, object still exists
} // destroyed p, if it was the last reference the object is deleted
```

STL: weak pointers

How to create loops with managed pointers.

Unique pointers are of no use in this case.

We need to extend the shared pointers, with a mechanism that does not retain ownership: The weak pointers.

```
struct L{
   std::shared_ptr<L> next;
   std::weak_ptr<L> prev;
};
```

A weak pointer can be constructed from a shared pointer.

Prior using it, you need to “lock” the weak pointer in a temporary shared ptr, use it, and finally throw away the ptr.

```
std::shared_ptr<L> L1(new L);
std::shared_ptr<L> L2(new L);
L1→next=l2;
L2→prev=L1; // assignment from shared ptr
…
```

want to access prev field of l2

```
std::shared_ptr<L> temp_prev=l2→prev.lock();
l2→prev…. // do stuff
```

STL: unique pointers

Unique pointers refer to instances of objects that exist only once.

They cannot be assigned, and can be passed only by reference.

They are more efficient than shared pointers, while they share the same mechanism of destruction when out of scope.

```
#include <memory>
using MySUPtr =std::unique_ptr<MyS>;
{
  MySUPtr p;
  {
    MySPtr p1(new MyS);
    p=p1; // not legal!!!!
  } // here the object is destroyed

}
```

Exercises

- We will shrink the code of our simulator by adding some stl functions. Stay posted and check the exercise on the web