Part II

C++11/14/17

Outline

Range-based for loops

- In C++03 we often had the following long syntax:

```
for(std::list<int>::const_iterator iter = lst.begin() ;
    iter != lst.end() ; ++iter)
```

- In C++11 we can use:

```
for( const int& i : lst ){
  std::cout << i << ", ";
}
for( int& i : lst ){
  i *= 2;
}
```

- We can even iterate over an in-place array:

```
for(double p : {1., 4., 9., 10., M_PI} ){
  std::cout << "sqrt(" << p << ") = " << sqrt(p);
}
```

- See `Examples/for-loops.C`.

Outline

noexcept

- In C++03 it was possible to indicate that certain functions would never throw an exception, using `throw()`.
- In C++11 this is replaced by `noexcept`:

```
Vector::Vector()noexcept{
  m_data = nullptr;
  m_size = 0;
}
```

- `noexcept` is part of the function signature in the same way as `const`, for example.
- If a `noexcept` function *does* throw an exception, the program will immediately terminate.
- This differs from default behaviour which would require that the exception propagate up the function stack until a matching `catch()` was found.

Exception function signatures

- From C++17 onwards, the `noexcept` forms part of a function signature:

```
void (*p)();
void (**pp)() noexcept = &p;
```

- The above will fail because `p` is not a `noexcept` function.
- See `Examples/noexcept.C`
- Also, the `throw( T )` syntax is no longer permitted. The syntax:

```
void q() noexcept(false);
```

  is permitted.

Outline

Return Value Optimization

- Consider the overloaded operator:

```
Vector operator+(const Vector& a, const Vector& b){
  Vector c(a.size());
  for(size_t i=0 ; i < a.size() ; i++){
    c[i] = a[i] + b[i];
  }
  return c;
}
Vector d = a + b;
```

- In C++03, with typical copy/assign-constructors, this can result in `c` being created, then copied into a newly created `d`.
- This seems wasteful, as `c` will immediately be destroyed.
- The Return Value Optimization allows compilers to avoid making this copy, and only creating the final destination `d`.
- In fact, `gcc` will do this at `-O0` unless you explicitly disable it via `-fno-elide-constructors`.
- See `Examples/move.C`, compiling in C++03 mode with and without `-fno-elide-constructors`.

Thwarting the Return Value Optimization

- However the RVO is easily thwarted by adding the following (not *entirely* unreasonable) code into `operator+`:

```
if(a.size() != b.size()){
  return Vector(0);
}
```

- There is no longer a single return point from the function, and the result `Vector` is not necessarily always the same one.
- Now, if we compile `move.C` with `-DFORCE_COPY` in C++03 mode, it includes the above code and we always have a copy-constructor call.
- Thus, adding the above will result in (possibly) slower code than before, for no particularly good reason.

Move constructor

- C++11 provides a way to *move* an object instead of copying it.
- This is used when we know the object being copied from will no longer be used (e.g. the returned `c` in the previous example).

```
Vector(Vector&& a){
  m_size = a.m_size;
  m_data = a.m_data;
  a.m_data = NULL;
  a.m_size = 0;
}
```

- Instead of copying the data element by element, we copy the data pointer itself, a constant-time operation.
- We cannot write the usual copy-constructor this way because then two `Vectors` would point to the same data.
- Note that since `a` is about to be destroyed, its contents must be something that the destructor will handle safely.
- Compiling `Examples/move.C` with C++11 support results in no slow copy functions being called.

Move into containers

- There are now functions to add elements to containers using move-syntax:

```
std::vector<Vector> vecs;
Vector a(10);
vecs.push_back(std::move(a));
```

- The `std::move` syntax comes from the `<utility>` header, and indicates that the variable should be moved rather than copied.
- Since we defined the `Vector::Vector(Vector&& v)` function to invalidate the contents of `v`, the size of `a` above will be zero after the `std::move() call`.
- You should not attempt to use `a` after the above has been called.
- See `Examples/moveContainer.C`.

Move into containers

- To reduce computational expense, `std` containers will attempt to move their elements, but only if the move operation is guaranteed not to throw an exception.
- (This follows the principles of Resource Allocation Is Initialization - see later lecture.)
- If a `std::vector` needs to copy all of its data into a new region of memory (due to a `push_back` for example), it will use the normal copy constructor unless it is guaranteed that the move-constructor will not throw an exception.
- So, we should declare the move-constuctor `noexcept`:

```
Vector(Vector&& a)noexcept { ... }
```

- Then, multiple insertions into an `std::vector` will not result in any slow copies.

Move into containers

- `Examples/moveContainer.C` demonstrates a copy-insertion, an in-place construct and push-back, and an explicit moving push-back:

```
vecs.push_back(a);
vecs.push_back(Vector(10));
vecs.push_back(std::move(a));
```

- With fully implemented move functions, only the first one performs a slow copy.
- If we omit the `noexcept`, then slow copies result if/when the `vector` has to reallocate its memory.

Emplace

- If move semantics had not been implemented for `Vector`, then:

```
vecs.push_back(Vector(10));
```

causes a construction and then copy-construct.

- Even with move semantics, it still causes a construct and move.
- However, an `emplace` will construct in-place:

```
vecs.emplace_back(10);
```

- The `10` corresponds to the parameters to be passed to the constructor.
- Other `emplace` functions are available for various containers, e.g:

```
std::list<int> numbers;
numbers.emplace(iter, 20)
```

where the iterator indicates the position before which to insert the new element.

```
std::map<std::string, int> ages;
ages.emplace("Tom", 10);
```

Move/Emplace performance

- As usual, you should consider readability before performance.
- The only reason that move-semantics give better performance is that there are non-trivial resources associated with a `Vector`.
- The `emplace` approach only avoids an extra `move` call.
- However, if we did not have move-semantics, and used `emplace`, it would save a copy-construct.
- Any resource allocation due to the container itself cannot be overcome using this method.
- Most containers support some form of `emplace` for data insertion.

Outline

Tuples

- As well as a `std::pair<A, B>` we can now have a generic tuple of values of different types `Examples/tuple.C`

```
std::tuple<int, double, std::string> a(42, 3.141, "Douglas");
std::cout << std::get<0>(a) << " "
          << std::get<1>(a) << " "
          << std::get<2>(a) << std::endl;
```

- This can be used to return more than one value from a function:

```
std::tuple<int, double, std::string> getNameAndNumber(){
  return std::make_tuple(54, 1.0, "Arthur");
}
int id;
double real;
std::string name;
std::tie(id, real, name) = getNameAndNumber();
```

New containers: `forward_list`

- As an alternative to `std::list`, which is bidirectional, `forward_list` is singly-linked.
- It is more space-efficient than `std::list`.

New containers: `unordered_map`

- Recall that a `std::map<Key, Value>` relies on an ordering on the `Key` type.
- This allows a new element to be inserted with complexity $O(\log(N))$ where N is the number of elements in the map.
- It is likely that the map is implemented as a binary-tree so that searching for the correct insertion point requires searching down tree branches.
- (There is the option to insert using an iterator as a hint where to put the new element.)
- What if you have a `Key` with no ordering?
- C++11 now has a hashed map, called `std::unordered_map`.

New containers: `unordered_map`

- `std::unordered_map<Key, Value, Hash>` relies on the existence of a hash function from `Key` to `size_t`.
- The cost of insertion is now O(1) (ish).
- The third template parameter is a functional which defaults to `std::hash<Key>` and is defined for all basic types, `string`s and a few other types (not containers).
- The hash functional maps a `Key` type to a `size_t` and should map different `Key`s to different values as far as possible.
- If a hash-collision occurs, then a "bucket" is created to hold all `Key`s that hash to this value.
- A poor hash function can therefore reduce the efficiency of a `std::unordered_map` to have $O(N)$ complexity for insertion if many hash-collisions occur.

Creating a new hash functional

Suppose we have a 2D coordinate type (harder to create an ordering)

```
struct Coord{
  Coord(int i, int j) : x(i), y(j){ }
  bool operator==(const Coord& b)const{
    return (x == b.x) && (y == b.y);
  }
  int x;
  int y;
};
```

Create a functional:

```
struct hashCoord{
  size_t operator()(const Coord& a)const{
    return std::hash<int>()(a.x) ^ std::hash<int>()(a.y);
  }
};
```

Note that `std::hash<int>` is a type, so `std::hash<int>()` is an object of that type, which has a function `operator()(int)`

Creating a new hash functional

- The ^ bitwise exclusive OR operator ensures that the results of the underlying hash functions are combined so as to give a result which is also `size_t`.
- Now we can use the `Coord` as a key as follows:

```
std::unordered_map<Coord, double, hashCoord> cellValues;
Coord a(1,3);
cellValues[a] = 3.0;
```

See `Examples/unordered_map.C`

New containers: unordered_map

- There are other related new containers:

```
std::unordered_set
std::unordered_multiset
std::unordered_multimap
```

- These may be more or less useful depending on what algorithm you are implementing.
- I advise checking the complexity of various operations on the containers before using them.

New containers: array

- C++11 now has a fixed-size array type, essentially containing a C-like array:

```
#include <array>
std::array<double, 5> a{1,4,9,16,25};

a[2] = 36;
std::sort(a.begin(), a.end());

for(size_t i=0 ; i < a.size() ; i++){
    std::cout << "a[" << i << "] = " << a[i] << std::endl;
}
```

- Importantly, this does not decay to a `double*` when being passed to a function.
- See `Examples/array.C`
- Note that the initializer above is not an `std::initializer_list` but aggregate initialization.

Movement between containers

- From C++17 it is possible to transfer elements from one container to another:

```
std::map<std::string, int> map1;
std::map<std::string, int> map2;
map1.merge(map2);
map2.insert(map1.extract("Arthur"));
```

- This moves all elements from `map2` into `map1`, and then moves element "Arthur" from `map1` into `map2`.
- This kind of operation is available on the other `*map` containers.
- See `Examples/container_transfer.C` for full code.

Allocators

- The various STL containers all have an Allocator template parameter.
- By default this is the `std::allocator<T>` which has essentially two methods:

```
std::allocator<double> a;
int * data = a.allocate(1000);
a.deallocate(data);
```

- Other methods do exist but are deprecated by C++17 (and removed in C++20).
- This seems pointless; can't we just use `new` and `delete`?
- Yes, but consider a `std::map` that is frequently updated; elements being added and removed.
- This results in 1,000s of calls to `new`, one for each element. This can be slow.
- It might be more efficient if you could allocate a large block of memory and the `std::map` used/reused small blocks from a single block of memory.

Allocators

- Writing your own efficient allocator requires some effort, and you need to consider whether contiguous chunks of memory are more important, or many smaller ones.
- The Boost Pool library helps in this case, and you can use:

```
std::vector<double, boost::pool_allocator<double> > myVec;
std::list<int, boost::fast_pool_allocator<int> > myList;
```

- These containers cannot be converted automatically to ones using the default allocator, so you need to use `typedef` to shorten definitions and reduce the number of places the allocator is specified.
- As with all optimizations: only do this if you find that this *is* your bottleneck.
- Changing your data-structure may be more profitable.