C++ for numerical computing

Rupert Nash
r.nash@epcc.ed.ac.uk

13 June 2018

Outline

Introduction

Containers

Iterators

Object oriented C++

What this is not!

Writing efficient software, more than anything, requires you to choose an appropriate algorithmic approach for your problem. For lots on how to do this well, take Parallel Design Patterns next semester!

Here we want to take a lower-level approach and talk about how to implement patterns efficiently using C++.

What is “numerical computing”?

Both HPC and data science, when you actually come to running a program, are about getting a large amount of data from memory to a core, doing something useful to it, and storing it again.

This is why FORTRAN is still relevant! But it does force you to confront this all time.

I’ve mentioned previously than C++ is all about building abstractions and composing them.

👍 I will talk about a few today and give some suggestions of default rules

Outline

STL Containers

The standard library has 13 container template classes, but we'll only touch on a few.

- `vector` - a dynamically sized contiguous array
- `array` - a statically size contiguous array
- `list`/`forward_list` - a doubly/singly linked list
- `set` / `map`

vector - your new best friend

You will be using this a lot, because the elements are contiguous in memory.

```
#include <vector>

std::vector<int> primes(unsigned n) {
  std::vector<int> ans;
  for (auto i=2; i<n; ++i) {
    if (isprime(i))
      ans.push_back(i);
  }
 return ans
}
```

👍 Use by default - data locality often wins over algorithmic complexity

vector - your new best friend

Supports:

- copy
- move
- random element access by index
- resize (and pre-reserving memory)
- element insertion
- element deletion

Note that when it destructs, contained elements will also be destroyed (i.e. it owns them).
Also be aware that resizes may force reallocation and copying!

array

- Contiguous in memory but the size is fixed at compile time.
- Almost like a vector, but you can't change the size.
- Only difference is construction:

```
#include <array>
typedef std::array<int, 3> GridPoint;

GridPoint p1 = {1,2,3};
GridPoint p2{{5,2,5}};
// horrible extra brace can go in C++14
std::cout << p2.size() << std::endl; \\ 3
```

list (and forward_list)

- Almost always implemented as a doubly (singly) linked list.
- Elements are allocated one by one on the heap. Traversal requires pointer chasing.
- Fast element insertion and deletion (if you don't have to look for the element!)

👍 Use when you will be adding and removing from ends a lot and the contained objects are expensive to copy/move. Consider converting to `vector` if you have a build/access pattern.

set and map

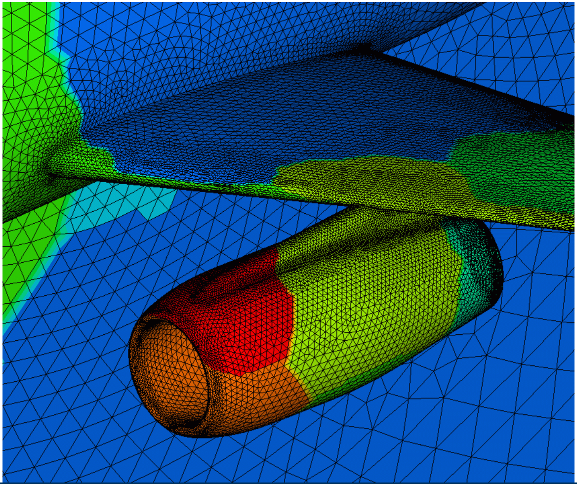
- These are associative containers implemented as sorted data structures for rapid search.
- `set` is just a set of keys, `map` is a set of key/value pairs (types can differ).
- You must have defined a comparison function for the key type.

👍 Use if you either

  - have a large key space that mostly lacks value, or
  - will be looking up unpredictable values a lot or frequently adding/removing values.

set and map

For example, describing your communication pattern between MPI ranks with a domain decomposed problem



set and map

```
std::map<int, BoundaryComm> rank2comms;
for (auto p =0; p != MPI_COMM_SIZE; ++p) {
  if (ShareBoundaryWithRank(p)) {
    rank2comms[p] = BoundaryComm(my_rank, p);
  }
}
// later
for (auto iter = rank2comms.begin(),
          end = rank2comms.end();
     iter != end; ++iter) {
  auto& bc = iter->second;
  bc->SendData(local_data);
}
```

Outline

Introduction

Containers

Iterators

Object oriented C++

Iteration

C programmers are used to:

```
unsigned n = 100;
double* data = GetData(n);
for (auto i=0; i != n; ++i) {
  data[i] *= 2;
}
```

More old-skool C programmers will prefer this:

```
unsigned n = 100;
double* start = GetData(n);
double* stop = start + n;
for (auto ptr = start; ptr != stop; ++ptr) {
  *ptr *= 2;
}
```

Iteration

These three humble pointers can implement the concept of traversing every element in the array (in order).

They also model the concept of an iterator which is a vital for using the standard library effectively.

There are a few different categories of iterator (forward, backward, random, etc) but they all can traverse the elements of something (e.g. a container, data in a file, input from keyboard) and provide access to them.

Iteration

A C++ equivalent of the previous might be:

```
std::vector<double> data = GetData(n);
for (std::vector<double>::iterator iter
                                      = data.begin();
     iter != data.end();
     ++iter) {
   *iter *= 2;
}
```

Iteration

A C++ equivalent of the previous might be:

```
std::vector<double> data = GetData(n);
for (std::vector<double>::iterator iter
                                   = data.begin();
     iter != data.end();
     ++iter) {
  *iter *= 2;
}
```

Or equivalently

```
std::vector<double> data = GetData(n);
for (auto iter = data.begin();
     iter != data.end(); ++iter) {
  *iter *= 2;
}
```

Er why?

What do we gain?

Er why?

What do we gain? Separation of concerns!

We can separate the data and how it's stored from the way we're traversing it, and also from the operations we apply to it.

Er why?

What do we gain? Separation of concerns!
We can separate the data and how it's stored from the way we're traversing it, and also from the operations we apply to it.

```
template <class ItT>
void doubleInPlace(ItT start, ItT end) {
  for (auto iter = start; iter != end; ++iter)
    *iter *= 2;
}

std::vector<double> data = GetData(100);
doubleInPlace(data.begin(), data.end());

std::list<HugeMatrix> mats = GetMatrices();
doubleInPlace(mats.begin(), data.end());
```

Container iterators

All the STL containers contain two iterator types, for example:

- `std::list<char>::iterator` - instance must be non-`const` - get with `begin()` or `end()`.
- `std::list<char>::const_iterator` - if the instance is `const` you get one of these from `begin()`/`end()`, if non-const, you can get one with `cbegin()`/`cend`.

You can also get iterators from e.g. `std::map<KeyT, ValT>::find(search_key)`, which will give and iterator pointing to the element you want or to the `end()`.

Note that an iterator pointing to the end is not valid! Dereferencing it may have undefined behaviour...

Implementing your own iterator

To define your own iterator, you need to create a class with several overloads (exactly which one depends on the category of iterator you need).

- derefence operator (`*it`) - you have to be able to get a value (either to read or write)
- pre-increment (`++it`) - you have to be able to "go to the next one" [1]
- assigment - you need to bind it to name
- inequality comparison (`it`= end!) - you need to know when you are done

[1] Why not post-increment? Because this has to return the value of `it` from before it was incremented. This usually means a copy.

Range for loop

Any class instance with `begin()` and `end()` member functions that return iterators can be used in a range based for-loop.

```
std::vector<int> primes = getPrimes(5);
for (auto p : primes) {
  std << p << "␣" << std::endl;
}
// 2 3 5 7 11
```

Almost “pythonic”?

Range for loop

The compiler will translate this for us into something approximating the following

```
{
  auto&& _range = <range expression>;
  for (auto _begin = _range.begin(),
            _end = _range.end();
       _begin != _end;
       ++_begin) {
    <range declaration> = *_begin;
    <loop body>
  }
}
```

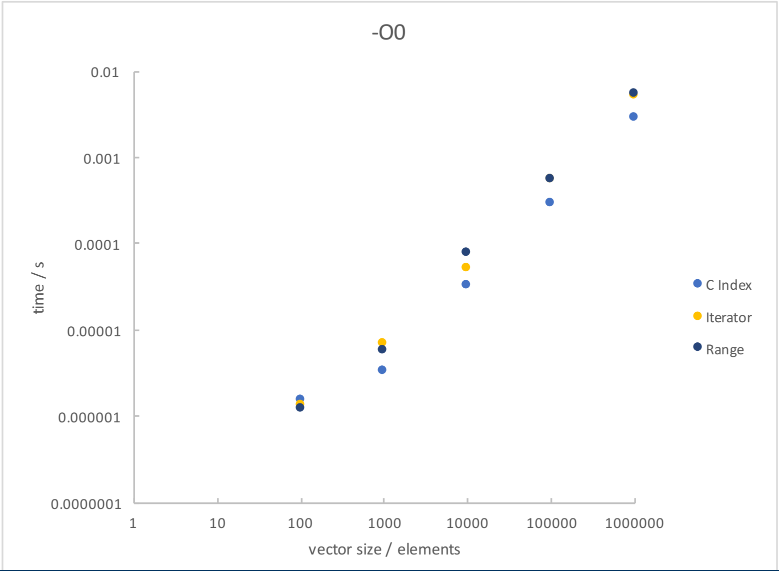
Is there any overhead?

Going to quickly compare three implementations

- C-style array indexing
- Standard vector with iterator
- Standard vector with range based for-loop

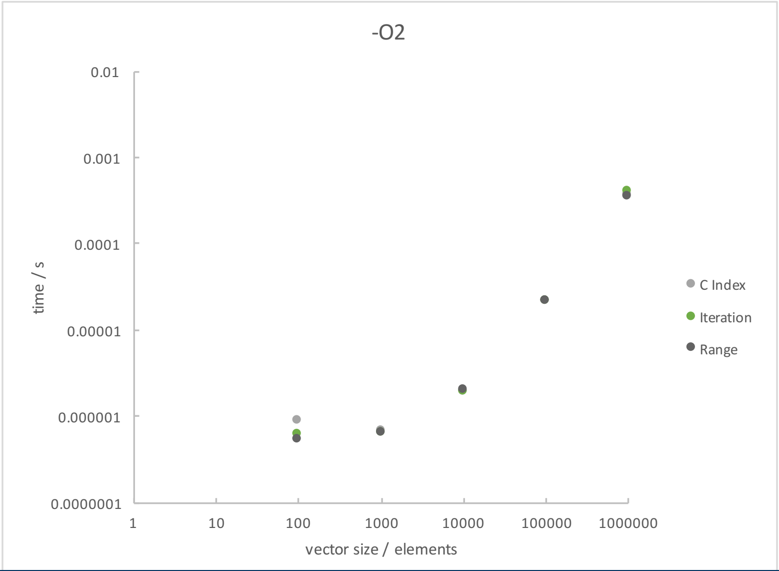
```
int main(int argc, char** argv) {
  int size = std::atoi(argv[1]);
  std::vector<float> data(size);
  for (auto& el: data)
    el = rand(1000);
  Timer t;
  scale(data.data(), data.size(), 0.5);
  std::cout << size << ", "
            << t.GetSeconds() << std::endl;
}
```

Results



-O0
0.01
0.001
0.0001
0.00001
0.000001
0.0000001
time / s
1
10
100
1000
10000
100000
1000000
vector size / elements
C Index
Iterator
Range

Results



-O2
0.01
0.001
0.0001
0.00001
0.000001
0.0000001
time / s
1
10
100
1000
10000
100000
1000000
vector size / elements
C Index
Iteration
Range

Assembly

Just showing the main loops:

```
LBB4_7:
        movups  -16(%rdx), %xmm2
        movups  (%rdx), %xmm3
        mulps   %xmm1, %xmm2
        mulps   %xmm1, %xmm3
        movups  %xmm2, -16(%rdx)
        movups  %xmm3, (%rdx)
        addq    $32, %rdx
        addq    $-8, %rcx
        jne     LBB4_7
```

```
LBB4_8:
        movups  -48(%rsi), %xmm2
        movups  -32(%rsi), %xmm3
        mulps   %xmm1, %xmm2
        mulps   %xmm1, %xmm3
        movups  %xmm2, -48(%rsi)
        movups  %xmm3, -32(%rsi)
        movups  -16(%rsi), %xmm2
        movups  (%rsi), %xmm3
        mulps   %xmm1, %xmm2
        mulps   %xmm1, %xmm3
        movups  %xmm2, -16(%rsi)
        movups  %xmm3, (%rsi)
        addq    $64, %rsi
        addq    $-16, %rdi
        jne     LBB4_8
```

Outline

Introduction

Containers

Iterators

Object oriented C++

Object oriented programming is one of the major paradigms supported by C++

> *OOP is based on the concept of "objects", which may contain data and code. A feature of objects is that an object's procedures can access and often modify the data of the object with which they are associated.*

We briefly covered how to create classes - today we'll go a little deeper.

Inheritance

- Inheritance is a method for deriving a new, related class from another one (called the base class, parent class, or super class).
- This relationship says that the derived object also is an object of the base class too!
- The new class (derived, child, sub) has all the data and function members of its parent, but you can add new ones and override existing ones.
- The derived class member functions can access the base class members that are public, but not the private ones. There is a third access specifier `protected` that allows derived classes to access the member.
- It's use should be minimal as you are promising to all subclasses that this interface will not change!

Inheritance

Suppose you had to process a lot of image files. You might start with a JPEG file:

```
class JpegFile {
  string _fn;
  int _nx, _ny;
  unique_ptr<char> _pixeldata;
public:
  JpegFile(string fn) : _fn(fn) {
     // read _nx/_ny/_ncols from header
    _pixeldata = new char[_nx*_ny*3];
    // decompress data from file
  }
  char& GetPixel(int x, int y) {
    return _pixeldata[x*_ny + y];
  }
}
```

Inheritance

But then you have to add PNG, and GIF, and ...

Inheritance

But then you have to add PNG, and GIF, and ...
Might want to do the same for each one, but:

- code duplication :(
- the types are totally unrelated :(

Inheritance

So instead create a base class and several derived classes

```
class ImageFile {
  string _fn;
protected:
  int _nx;
  int _ny;
  unique_ptr<char> _pixeldata;
public:
  ImageFile(string fn);
  char& GetPixel(int x, int y);
};
```

Inheritance

So instead create a base class and several derived classes

```
class JpegFile : public ImageFile {
public:
  JpegFile(string fn) : ImageFile(fn) {
    // read _nx/_ny/_ncols from header
    _pixeldata = new char[_nx*_ny*3];
    // decompress data from file
  }
};

class PngFile : public ImageFile {
public:
  PngFile(string fn);
};
```

Pointers to base class

One important thing to know is that a pointer to a derived class (JpegFile*) is type compatible with a pointer to the base class (ImageFile*).

```
JpegFile jpg("cat.jpg");
ImageFile* img = &jpg;
// also works with references
ImageFile& im_ref = jpg;
```

Dynamic polymorphism

- What if we have some behaviour, like writing the image data to file, that varies between the subclasses?
- We ideally want to have a uniform interface and when we call it as run time the pointer-to-base knows which subclass method to call.
- Enter virtual functions!

Dynamic polymorphism

```
class ImageFile {
public:
 virtual void Write(string fn} = 0;
};

class JpegFile : public ImageFile {
public:
    virtual void Write(string fn) {
      // write header
      // compress + write data
    }
};
ImageFile* img = new JpegFile("cat.jpg");
img->Write("notdog.jpg");
```

How does this work?

- Through a virtual function table.
- Each class has a static table of function pointers that point to the code of its virtual functions.
- Each instance of the class has a pointer to the table that belongs to its actual class (filled in by the compiler in the constructor).
- To call, the object's "vtable" pointer is followed, the offset for the method added, and the function called by pointer. Clearly slower than a simple function call by compile-time constant! Worse it make inlining of the function impossible.
- You really don't want to use virtual functions in an inner loop! (By all means use them outside!)