Programmazione e Calcolo Scientifico

Matteo Cicuttin

Politecnico di Torino

April 9, 2024

Public service announcement

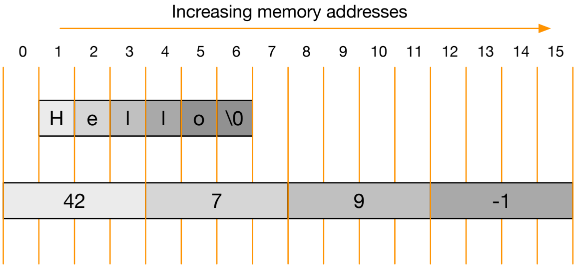
I was informed that there is some confusion between `float` and `double`.

- `float` is 32 bit wide and is made of 1 sign bit, 8 exponent bits and 23 mantissa bits
- `double` is 64 bit wide and is made of 1 sign bit, 11 exponent bits and 53 mantissa bits
- `float` is generally used in digital signal processing, image processing and video games
- `double` is for scientific computing
- if you don't know what you are doing, don't use `float`
- more importantly, DON'T MIX THEM
- for this course forget about the existence of `float` and use only `double`

Pointer arithmetic

Pointers support some arithmetic, but I won't go in the details. I just want you to know the following:

```
int vals[] = {1,2,3,4,5,6};
int* pval = &vals[1];
pval++;
/* now pval points to vals[2] */
```



- Remember that a pointer of type `T` contains an address to an object of type `T`.
- If you increment a pointer of type `T`, the contained address gets incremented by `sizeof(T)`.
- Therefore, if you point to an array and you increment the pointer, you get to the next element.

Iterators

A widely used concept in the STL is the iterator. Iterators allow to iterate on the elements of a container in an abstract way. For example, with a vector:

```
using namespace std;
vector<int> vals = {1,2,3,4,5,6};
for (vector<int>::iterator itor = vals.begin(); itor != vals.end(); itor++)
  int val = *itor;
```

- Iterators look like pointers, but they are not pointers.
- They allow you to iterate complex data structures (lists, trees) as if they were arrays/vectors

```
using namespace std;
list<int> vals;
/* fill the list */
for (list<int>::iterator itor = vals.begin(); itor != vals.end(); itor++)
  int val = *itor;
```

Notice how the for loops are the same despite the big difference between a vector and a list.

Automatic type deduction

You may have noticed that the type of the iterator is rather long...

- For vectors: `vector<int>::iterator`
- For lists: `list<int>::iterator`

Let's the compiler do the dirty work: meet `auto`

```
using namespace std;
list<int> vals;
/* fill the list */
for (auto itor = vals.begin(); itor != vals.end(); itor++)
  int val = *itor;
```

With `auto`, the compiler automatically deduces the type of `itor`. There is no magic involved, it just looks at the return type of `list<int>::begin()`.

Range-based for loops

Despite the `auto`, the for loop remains a bit annoying to write

```
using namespace std;
list<int> vals;
/* fill the list */
for (auto itor = vals.begin(); itor != vals.end(); itor++) {
  int val = *itor;
  cout << val << endl;
}
```

Meet the range-based for loop. The above code is equivalent to

```
using namespace std;
list<int> vals;
/* fill the list */
for (auto& val : vals) {
  cout << val << endl;
}
```

Implementing a singly-linked list

We have everything in place to implement a singly-linked list like the one in the STL.

Time to look at the code:

- `linked_list_example.cpp`
- `linked_list.hpp`

Safer memory management: unique_ptr

For this course you need to understand new/delete, but in modern C++ there is a much safer way to manage memory. Idea: use RAII to automatically manage object lifetime.

```
template<typename T>
struct unique_ptr {
  T* raw_ptr;

  unique_ptr(/*parameters*/) {
    raw_ptr = new T(/*parameters*/);
  }
  unique_ptr(const unique_ptr&) = delete;
  ~unique_ptr() {
    delete raw_ptr;
  }
};
```

```
/* old way */
T* p = new T(...);
...
/* if you forget delete or if you delete
 * two times you're in trouble */
delete p;

/* new way: when up goes out of scope,
 * memory is freed automatically */
unique_ptr<T> up = make_unique<T>(...);
```

The real-world implementation of unique_ptr is much more complex than this.

Implementing a singly-linked list with `unique_ptr`

Let's re-implement the list with `std::unique_ptr`.

The main concept to grasp here is the pointer ownership

- as we can't copy unique pointers, someone has to own the pointed resource
- `linked_list` owns and manages the lifetime of `list_head`
- each node owns and manages the lifetime of the next nodes

We will understand the concept with an example. Time to look at the code:

- `linked_list_example.cpp`
- `linked_list_up.hpp`