601.220 Intermediate Programming

Introduction to STL and templates

C++: STL

Standard Template Library (STL) is C++'s library of useful data structures & algorithms

- Like `java.util`/`java.lang`
- Like Python `sets`, `dictionaries`, `collections`

Templates are covered in detail later in the course; we'll give them a quick look now

The concept of templates

Templates are a way of writing an object (`Node`) or function (`print_list`) so they can work with *any* type

Defining a template is simultaneously defining a *family* of related objects/functions

The concept of templates

```
struct Node {
    T payload;  // 'T' is placeholder for a type
    Node *next;
};

void print_list(Node *head) {
    Node *cur = head;
    while(cur != NULL) {
        cout << cur->payload << " ";
        cur = cur->next;
    }
    cout << endl;
}
```

We could replace `T` with `int`, `float`, `char`, or `std::string` and this would compile and work

The concept of templates

Here's similar code using C++ templates

```
template<typename T>
struct Node {
    T payload;
    Node *next;
};

template<typename T>
void print_list(Node<T> *head) {
    Node<T> *cur = head;
    while(cur != NULL) {
        cout << cur->payload << " ";
        cur = cur->next;
    }
    cout << endl;
}
```

We write one `struct`/function, they work for *any* type `T` (almost)

C++: Templates

```
#include <iostream>
using std::cout; using std::endl;

template<typename T>
struct Node {
    T payload;
    Node *next;
};

template<typename T>
void print_list(Node<T> *head) {
    Node<T> *cur = head;
    while(cur != NULL) {
        cout << cur->payload << " ";
        cur = cur->next;
    }
    cout << endl;
}
```

```
int main() {

    Node<float> f3 = {95.1f, NULL};
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    print_list(&f1);

    Node<int> i2 = {239, NULL};
    Node<int> i1 = {114, &i2};
    print_list(&i1);

    return 0;

}
```

```
$ g++ -c ll_template_cpp.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ll_template_cpp ll_template_cpp.o
$ ./ll_template_cpp
24.3 48.7 95.1
114 239
```

C++: Standard Template Library (STL)

With STL we'll use types like:

- `vector<string>` – vector of `std::strings`
- `vector<float>` – vector of `floats`
- `map<string, int>` – map containing `std::strings` with associated `ints`

Similar to Java generics

C++: Standard Template Library (STL)

Incomplete list of STL classes:

- `array` – fixed-length array
- `vector` – dynamically-sized array
- `set` – set; an element can appear at most once
- `list` – linked list!
- `map` – associative list, i.e. dictionary
- `stack` – last-in first-out (LIFO)
- `deque` – double-ended queue, flexible combo of LIFO/FIFO

C++: `vector`

`vector` is an array that automatically grows/shrinks as you need more/less room

- Use `[ ]` to access elements, like array
- Allocation, resizing, deallocation handled by C++
- Like Java's `java.util.ArrayList` or Python's `list` type

`#include <vector>` to use it

`std::string` is like (but not same as) `std::vector<char>`

C++: vector

To declare a vector:

```
using std::vector;

vector<std::string> names;
```

To add elements to vector (at the back):

```
names.push_back("Alex Hamilton");
names.push_back("Ben Franklin");
names.push_back("George Washington");
```

To print number of items in vector, and first and last items:

```
cout << "Size =" << names.size()
     << ", first =" << names.front()
     << ", last =" << names.back() << endl;
```

C++: vector

The vector template handles memory for you

Behind the scenes, dynamic memory allocations are needed both to create strings and to add them to the growing vector:

```
names.push_back("Alex Hamilton");
names.push_back("Ben Franklin");
names.push_back("George Washington");
```

Allocations happen automatically; everything (vector and strings) is deallocated when names goes out of scope

C++: vector

```cpp
// names_1.cpp:
#include <iostream>
#include <vector>
#include <string>
using std::vector; using std::string;
using std::cin;    using std::cout;   using std::endl;
int main() {
  vector<string> names = {"Alex Hamilton", "Ben Franklin"};
  names.push_back("George Washington");
  cout << "First name was " << names.front() << endl;
  cout << "Last name was " << names.back() << endl;
  // names.front()<->names[0], names.front()<->names[names.size()-1]
  return 0;
} // names goes out of scope and memory is freed
```

C++: vector

```
$ g++ -c names_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o names_1 names_1.o
$ ./names_1
First name was Alex Hamilton
Last name was George Washington
```

C++: vector

Two ways to print all elements of a `vector`. With indexing:

```
for(size_t i = 0; i < names.size(); i++) {
    cout << names[i] << endl;
}
```

With an *iterator*:

```
for(  vector<string>::iterator it = names.begin();
      it != names.end();
      ++it) {
    cout << *it << endl;
}
```

C++: `vector`

`Iterators` are "clever pointers" that know how to move over the components of a data structure

Structure could be simple (vector, linked list) or complicated (tree)

They are safer and less error-prone than pointers; pointers cannot generally be used with STL containers

C++: iterators

For STL container of type `T`, iterator has type `T::iterator`

```
for(vector<string>::iterator it = names.begin();
    it != names.end();
    ++it) {
    cout << *it << endl;
}
cout << endl;
```

Here, iterator type is `vector<string>::iterator`

C++: iterators

Looking harder at the loop:

```
for(vector<string>::iterator it = names.begin();
    it != names.end();
    ++it)
```

First line: declares `it`, sets it to point to first element initially

Second: stops loop when iterator has moved past `vector` end

Third: tells iterator to advance by 1 each iteration

- `++it` isn't really pointer arithmetic; ++ is "overloaded" to move forward 1 element *like* a pointer

C++: iterators

Looking harder at the body:

```
cout << *it << endl;
```

`*it` is *like* dereferencing; `*` is "overloaded" to get the element currently pointed to by the iterator

For `vector`, `*it`'s type equals the element type, `string` in this case

C++: vector

```cpp
// names_2.cpp:
#include <iostream>
#include <vector>
#include <string>
using std::cin;    using std::cout;   using std::endl;
using std::vector; using std::string;
int main() {
  vector<string> names = {"Alex Hamilton", "Ben Franklin"};
  names.push_back("George Washington");
  for(vector<string>::iterator it = names.begin(); it != names.end(); ++it) {
    cout << *it << endl;
  }
  return 0;
}
```

C++: vector

```
$ g++ -c names_2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o names_2 names_2.o
$ ./names_2
Alex Hamilton
Ben Franklin
George Washington
```

C++: vector

Iterate in *reverse* order by using `T::reverse_iterator`, `.rbegin()` and `.rend()` instead:

```
for(vector<string>::reverse_iterator it = names.rbegin();
    it != names.rend();
    ++it) {
    cout << *it << endl;
}
```

C++: vector

```
#include <iostream>
#include <vector>
#include <string>
using std::cin;    using std::cout;   using std::endl;
using std::vector; using std::string;
int main() {
  vector<string> names = {"Alex Hamilton", "Ben Franklin"};
  names.push_back("George Washington");
  for(vector<string>::reverse_iterator it = names.rbegin();
    it != names.rend(); ++it) {
    cout << *it << endl;
  }
  return 0;
}
```

```
$ g++ -c names_3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o names_3 names_3.o
$ ./names_3
George Washington
Ben Franklin
Alex Hamilton
```

C++: `vector`

See C++ reference for more `vector` functionality

- www.cplusplus.com/reference/vector/vector/

Don't miss:

- `front`/`back` – get first/last element
- `pop_back` – return and delete final element
- `erase`, `insert`, `clear`, `at`, `empty` – just like `string`
- `swap` – swap elements
- `begin`/`end` – iterators for beginning/end
- `rbegin`/`rend` – `reverse_iterators` for beginning/end
- `cbegin`/`cend` – `const_iterators` for beginning/end (more about these soon)