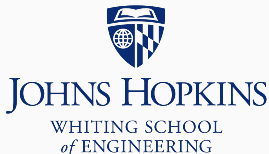
Template functions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Template functions

Templates allow us to write function or class once:

```
template<typename T>
void print_array(const T* a, int count) { ... }
```

but get a whole *family* of *overloaded specializations*:

```
void print_array(const int* a, int count) { ... }
void print_array(const float* a, int count) { ... }
void print_array(const char* a, int count) { ... }
void print_array(const linked_list_node* a, int count) { ... }
...
```

Template functions

This function sums even-indexed elements in a list:

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        ++it;
    }
    return total;
}
```

Works for `const vector<int>&`, but similar code could be used for other containers

Template functions

```
#include <iostream>
#include <vector>
#include <list>

int sum_every_other_vector(const std::vector<int>& ls) {
    //                                ^^^^^^
    int total = 0;
    for(std::vector<int>::const_iterator it = ls.cbegin(); it != ls.cend(); ++it) {
    //       ^^^^^^
        total += *it;
        ++it;
    }
    return total;
}

int sum_every_other_list(const std::list<int>& ls) {
    //                              ^^^^
    int total = 0;
    for(std::list<int>::const_iterator it = ls.cbegin(); it != ls.cend(); ++it) {
    //       ^^^^
        total += *it;
        ++it;
    }
    return total;
}
```

Template functions

```
int main() {
    std::vector<int> vec = {10, 7, 10, 7, 10, 7};
    int sum = sum_every_other_vector(vec);
    cout << "sum of every-other (vector): " << sum << endl;

    std::list<int> lis;
    lis.assign(vec.begin(), vec.end());
    sum = sum_every_other_list(lis);
    cout << "sum of every-other (list): " << sum << endl;
    return 0;
}
```

```
$ g++ -c seo_vec_list_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o seo_vec_list_1 seo_vec_list_1.o
$ ./seo_vec_list_1
sum of every-other (vector): 30
sum of every-other (list): 30
```

Template functions

Repetitive code is a sign of bad design

E.g. a correction for the `_vector` version also has to be made for the `_list` version (and any others we've made)

In fact, can you spot the error in our `sum_every_other_vector`/`sum_every_other_list`?

Template functions

Extra `++it` skips over `ls.cend()` when the container has odd # elements. Need another check:

```
int sum_every_other_vector(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        // now we can't skip over ls.cend()
        if(++it == ls.cend()) { break; } // that's better
    }
    return total;
}
```

Template functions

Template function:

```
template<typename T>
int sum_every_other(const T& ls) {
    int total = 0;
    for(typename T::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

If we pass `vector<int>`, compiler *instantiates* an appropriate function overload

- Same if we pass `list<int>`, `vector<double>`, ...

Template functions

```
int main() {
    vector<int> vec = {10, 7, 10, 7, 10, 7};

    // ** calls template function with T=vector<int> **
    int sum = sum_every_other(vec);

    cout << "sum of every-other (vector): " << sum << endl;

    list<int> lis;
    lis.assign(vec.begin(), vec.end());

    // ** calls template function with T=list<int> **
    sum = sum_every_other(lis);

    cout << "sum of every-other (list): " << sum << endl;
    return 0;
}
```

Template functions

```
$ g++ -c seo_vec_list_2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o seo_vec_list_2 seo_vec_list_2.o
$ ./seo_vec_list_2
sum of every-other (vector): 30
sum of every-other (list): 30
```

Template functions

The reason for typename here is subtle

```
for(typename T::const_iterator it = ls.cbegin();
//  ^^^^^^^^
    it != ls.cend(); ++it)
```

Without typename, compiler can't distinguish whether `T::const_iterator` is a type or a `static` field

Template functions

Without `typename` before `T::const_iterator` we get an error:

```
$ g++ -c seo_vec_list_3.cpp -std=c++11 -pedantic -Wall -Wextra
seo_vec_list_3.cpp: In function 'int sum_every_other(const T&)':
seo_vec_list_3.cpp:11:9: error: need 'typename' before 'T:: const_iterator'
because 'T' is a dependent scope
     for(T::const_iterator it = ls.cbegin();
         ^
seo_vec_list_3.cpp:11:27: error: expected ';' before 'it'
     for(T::const_iterator it = ls.cbegin();
                           ^~
seo_vec_list_3.cpp:12:9: error: 'it' was not declared in this scope
         it != ls.cend(); ++it)
         ^~
seo_vec_list_3.cpp:12:9: note: suggested alternative: 'int'
         it != ls.cend(); ++it)
         ^~
         int
seo_vec_list_3.cpp: In instantiation of 'int sum_every_other(const T&) [with T =
std::vector<int>]':
seo_vec_list_3.cpp:25:34:   required from here
seo_vec_list_3.cpp:11:43: error: dependent-name 'T:: const_iterator' is parsed
as a non-type, but instantiation yields a type
     for(T::const_iterator it = ls.cbegin();
                                           ^
seo_vec_list_3.cpp:11:43: note: say 'typename T:: const_iterator' if a type is
meant
seo_vec_list_3.cpp: In instantiation of 'int sum_every_other(const T&) [with T =
```