601.220 Intermediate Programming

Template functions

Template functions

Templates allow us to write function or class once:

```
template<typename T>
void fun(const T& input) { ... }
```

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

```
void fun(const int& input) { ... }
void fun(const float& input) { ... }
void fun(const char& input) { ... }
void fun(const MyClass& input) { ... }
...
```

Template functions

Recall: two functions are overloaded if they have same name & return type, but different parameter types

```
void print_array(const int* array, int count) {
    //                   ^^^
    for(int i = 0; i < count; i++) {
        cout << array[i] << " ";
    }
}

void print_array(const double* array, int count) {
    //                   ^^^^^^
    for(int i = 0; i < count; i++) {
        cout << array[i] << " ";
    }
}
```

C/C++ design: function templates

Q: When should you consider using function templates?

When you find yourself writing functions with essentially the same body, but different types.

Template functions

This function sums even-indexed elements in a `std::vector`:

```
using std::vector;

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, e.g. a `std::list`

Template functions

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

using std::vector;  using std::list;

int sum_every_other(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(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

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

using std::vector; using std::list;
using std::cout;   using std::endl;

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

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

Template functions

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

using std::vector; using std::list;
using std::cout;   using std::endl;

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;
}

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

int main() {
    vector<int> vec = {10, 7, 10, 7, 10, 7};
    int sum = sum_every_other(vec);
    cout << "sum of every-other (vector): "
         << sum << endl;
    list<int> lis;
    lis.assign(vec.begin(), vec.end());
    sum = sum_every_other(lis);
    cout << "sum of every-other (list): "
         << sum << endl;
    return 0;
}
```

```
$ g++ total.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./a.out
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, we do have an error in our `sum_every_other` function.

Have you spotted it?

Template functions

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

```
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;
        // now we can't skip over ls.cend()
        if(++it == ls.cend()) { break; } // that's better
    }
    return total;
}
```

Template functions

```
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

```cpp
// seo_vec_list_2.cpp:
#include <iostream>
#include <vector>
#include <list>

using std::vector; using std::list;
using std::cout;   using std::endl;

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;
}
```

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

    // 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
```

Quiz!

Which definition of the `print` function is correct?

A.

```
template<class T> void print (const T &a) {
  for (typename T::const_iterator it = a.cbegin(); it != a.cend(); it++) { cout << *it << endl; }
}
```

B.

```
template<class T> void print (const T &a) {
  for (T::const_iterator it = a.cbegin(); it != a.cend(); it++) { cout << *it << endl; }
}
```

C.

```
template<typename T> void print (const T &a) {
  for (T::const_iterator it = a.cbegin(); it != a.cend(); it++) { cout << *it << endl; }
}
```

D.

```
template<typename T> void print (const typename T &a) {
  for (T::const_iterator it = a.cbegin(); it != a.cend(); it++) { cout << *it << endl; }
}
```

E. None of the above