

# 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 specifications:

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

# Template functions

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`:

```
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;  
}
```

Works for `const vector<int>&`, but similar code could be used for other containers, e.g. a `std::list`.

# Template functions

```
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

```
// template1.cpp
```

```
1  #include "sum_every_other.h"
2
3  int main() {
4      std::vector<int> vec = {10, 7, 10, 7, 10, 7};
5      int sum = sum_every_other(vec);
6      std::cout << "sum of every-other (vector): " << sum << std::endl;
7
8      std::list<int> lis;
9      lis.assign(vec.begin(), vec.end());
10     sum = sum_every_other(lis);
11     std::cout << "sum of every-other (list): " << sum << std::endl;
12     return 0;
13 }
```

```
$ g++ -o template1 template1.cpp -std=c++11 -pedantic -Wall -Wextra
```

```
$ ./template1
```

```
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 or modification for one of them also has to be made for 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 number of elements. We need another check:

```
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;
        // 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 `std::vector<int>`, compiler **instantiates** an **appropriate function overload**

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

# Template functions

```
// template2.cpp
1  #include "sum_every_other_template.h"
2  #include <iostream>
3  #include <vector>
4  #include <list>
5
6  int main() {
7      std::vector<int> vec = {10, 7, 10, 7, 10, 7};
8      int sum = sum_every_other(vec);
9      std::cout << "sum of every-other (vector): " << sum << std::endl;
10
11     std::list<int> lis;
12     lis.assign(vec.begin(), vec.end());
13     sum = sum_every_other(lis);
14     std::cout << "sum of every-other (list): " << sum << std::endl;
15     return 0;
16 }
```

```
$ g++ -o template2 template2.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./template2

sum of every-other (vector): 30
sum of every-other (list): 30
```

## Quiz - answers

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