COMP6771 Week 6.2

Function Templates, Class Templates

Polymorphism & Generic Programming

- **Polymorphism:** Provision of a single interface to entities of different types
- Two types - :
  - Static (our focus):
    - Function overloading
    - Templates (i.e. generic programming)
      - std::vector<int>
      - std::vector<double>
  - Dynamic:
    - Related to virtual functions and inheritance - see week 8
- **Genering Programming:** Generalising software components to be independent of a particular type
  - STL is a great example of generic programming

Function Templates

Without generic programming, to create two logically identical functions that behave in a way that is independent to the type, we have to rely on function overloading.

```cpp
1  int min(int a, int b) {
2    return a < b ? a : b;
3  }
4  double min(double a, double b) {
5    return a < b ? a : b;
6  }
7
8  int main() {
9    std::cout << min(1, 2) << "\n"; // calls line 1
10   std::cout << min(1.0, 2.0) << "\n"; // calls line 4
11 }
```

Explore how this looks in Compiler Explorer

Function Templates

- Function template: Prescription (i.e. instruction) for the compiler to generate particular instances of a function varying by type
  - The generation of a templated function for a particular type T only happens when a call to that function is seen during compile time

```
template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}

int main() {
  std::cout << min(1, 2) << "\n"; // calls int min(int, int)
  std::cout << min(1.0, 2.0) << "\n"; // calls double min(double, double)
}
```

Explore how this looks in Compiler Explorer

Some Terminology

template type parameter

template parameter list

```
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

Type and Nontype Parameters

- **Type parameter:** Unknown type with no value
- **Nontype parameter:** Known type with unknown value

```
#include <iostream>

template <typename T, int size>
T findmin(const std::array<T, size> a) {
  T min = a[0];
  for (int i = 1; i < size; ++i) {
    if (a[i] < min) min = a[i];
  }
  return min;
}

int main() {
  std::array<int, 3> x{ 3, 1, 2 };
  std::array<double, 4> y{ 3.3, 1.1, 2.2, 4.4 };
  std::cout << "min of x = " << findmin(x) << "\n";
  std::cout << "min of x = " << findmin(y) << "\n";
}
```

Compiler deduces **T** and **size** from **a**

Type and Nontype Parameters

- The above example generates the following functions at compile time
- What is "code explosion"? Why do we have to be weary of it?

```
int findmin(const std::array<int, 3> a) {
  int min = a[0];
  for (int i = 1; i < 3; ++i) {
    if (a[i] < min) min = a[i];
  }
  return min;
}

double findmin(const std::array<double, 4> a) {
  double min = a[0];
  for (int i = 1; i < 4; ++i) {
    if (a[i] < min) min = a[i];
  }
  return min;
}
```

Class Templates

- How we would currently make a Stack type
- Issues?
  - Administrative nightmare
  - Lexical complexity (need to learn all type names)

```cpp
class IntStack {
 public:
  void push(int&);
  void pop();
  int& top();
  const int& top() const;
 private:
  std::vector<int> stack_;
};
```

```cpp
class DoubleStack {
 public:
  void push(double&);
  void pop();
  double& top();
  const double& top() const;
 private:
  std::vector<double> stack_;
};
```

Class Templates

- Creating our first class template

```cpp
// stack.h
#ifndef STACK_H
#define STACK_H

#include <iostream>
#include <vector>

template <typename T>
class Stack {
 public:
  friend std::ostream& operator<<(std::ostream& os, const Stack& s) {
    for (const auto& i : s.stack_) os << i << " ";
    return os;
  }
  void push(T&);
  void pop();
  T& top();
  const T& top() const;
 private:
  vector<T> stack_;
};

#endif // STACK_H
```

https://en.cppreference.com/w/cpp/language/friend#Template_friends

```cpp
// stack.h (continued)

template <typename T>
void Stack<T>::push(const T &item) {
  stack_.push_back(item);
}

template <typename T>
void Stack<T>::pop() {
  stack_.pop_back();
}

template <typename T>
T& Stack<T>::top() {
  return stack_.back();
}

template <typename T>
const T& Stack<T>::top() const {
  return stack_.back();
}

template <typename T>
bool Stack<T>::empty() const {
  return stack_.empty();
}
```

Class Templates

Main function

```cpp
#include "stack.h"

#include <iostream>
#include <string>

int main() {
  Stack<int> s1; // int: template argument
  s1.push(1);
  s1.push(2);
  Stack<int> s2 = s1;
  std::cout << s1 << s2 << '\n';
  s1.pop();
  s1.push(3);
  std::cout << s1 << s2 << '\n';
  s1.push("hello"); // Fails to compile.

  Stack<std::string> string_stack;
  string_stack.push("hello");
  string_stack.push(1); // Fails to compile.
}
```

Class Templates

Default rule-of-five

```
template <typename T>
Stack<T>::Stack() { }

template <typename T>
Stack<T>::Stack(const Stack<T> &s) : stack_{s.stack_} { }

template <typename T>
Stack<T>::Stack(Stack<T> &&s) : stack_(std::move(s.stack_)); {

template <typename T>
Stack<T>& Stack<T>::operator=(const Stack<T> &s) {
  stack_ = s.stack_;
}

template <typename T>
Stack<T>& Stack<T>::operator=(Stack<T> &&s) {
  stack_ = std::move(s.stack_);
}

template <typename T>
Stack<T>:: ~Stack() { }
```

default

https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/defaulted_functions_in_c_11?lang=en

```cpp
class Stack {
 public:
  // Why won't this work if we uncomment the line below?
  // Stack(int i) { stack_.push_back(i); }
  void push(int&);
  void pop();
  int& top();
  const int& top() const;
 private:
  std::vector<int> stack_;
};

int main() {
  Stack s;
}
```