COMP6771
Advanced C++ Programming

Week 7.1
Templates Intro

In this lecture

Why?

- C++ is strongly-typed
- ensure correct storage and avoid leak & illegal operation
- C uses #define or void*
- Understanding compile time polymorphism in the form of templates helps understand the workings of C++ on generic types

What?

- Templates
- Non-type parameters
- Inclusion exclusion principle
- Classes, statics, friends
- major features and design rational

Recommended Reference:

C++ Templates the Complete guide (David Vandevoorde..2018)

The Past: Reuse with Cut&Paste

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
#include <iostream>

auto min(int a, int b) -> int {
        return a < b ? a : b;
}

auto min(double a, double b) -> double{
        return a < b ? a : b;
}

struct int_list{...};
struct double_list{...};

double int_list_append(...);
double double_list_append(...);

auto main() -> int {
        std::cout << min(1, 2) << "\n"; // calls min(int, in
        std::cout << min(1.0, 2.0) << "\n"; // calls min(dou
}
```

```cpp
#include<iostream>

auto min(int a, int b) -> int {
        return a < b ? a : b;
}

auto min(double a, double b) -> double{
        return a < b ? a : b;
}

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

demo701-functemp1.cpp

Explore how this looks in Compiler Explorer

Polymorphism & Generic Programming

- The problem has been around long time. 1970, specified later
- **Polymorphism:** Provision of a single interface to entities of different types.
- **Genering Programming:** Generalising software components to be independent of a particular type
  - Life algorithms and data structure from concrete examples to their most general and abstract form.
  - STL is a great example of generic programming
- 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 9

Function Templates

- Function template: not actually a function, generalization of algorithms
- Prescription (i.e. instruction) for the compiler to generate particular instances of a function varying by type
  - Single declaration that generates declarations
  - The generation of a templated function for a particular type T only happens when a call to that function is seen during compile time.

```cpp
#include <iostream>

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

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

Provide enough info to complier to generate a function body

demo702-functemp2.cpp

less runtime

type checking

(more efficient)

Still not producing any code: Just compile time information of a function.

Explore how this looks in Compiler Explorer

Some Terminology

template type parameter: a placeholder for a type argument

template parameter list: a placeholder for argument expression

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

Argument substitution happens at compile time: not run time

```
template <typename T>
T functionName(T parameter1, T parameter2, ...) {
    // code
}
```

Depending upon the context either:

1. **Compiler pass the argument at compile time or**
2. **program pass the argument at run time**

Function Template

- The act of generating a function definition from template is called template instantiation.
- function def. generated from template is instantiated function or instantiation.

```cpp
#include<iostream>

int const& max (int const& a, int const& b){
  std::cout<<"max(int, int)"<<std::endl;
  return a < b ? b : a;
}
template <typename T> // T's scope begins here..
T const& max (T const& a, T const& b){
  std::cout << "max(T, T)" << std::endl;
  return a < b ? b : a;
}       // T's scope ends here
template <typename T>
T const& max (T const& a, T const& b, T const& c) {
  std::cout << "max(T,T, T)" << std::endl;
  return max(max (a,b), c);
  //max(max<> (a,b), c);
}
int main() {
  max(15.0, 20.0);
  max('x', 'y');
  max(15,25);
  max<>(15,25);
  max<double>(15.0,25.0);
  max(15, 20, 25);
  max(15, 20, 25);
}
```

```cpp
// create a function template that prints the swap of two number

#include<iostream>

template<class T>
void swap(T &a,T &b) {
  T temp = a;
  a = b;
  b = temp;
}

int main() {
  int a = 10, b = 20;
  double x = 20.3, y = 55.3;

  std::cout << "Before Swap" << std::endl;
  std::cout << "A=" << a << "\t" << "B=" << b << std::endl;
  std::cout << "X=" << x << "\t" << "B=" << y << std::endl;

  swap(a, b);
  swap(x, y);

  std::cout << "After Swap: "<< std::endl;
  std::cout << "A=" << a << "\t" << "B=" << b << std::endl;
  std::cout << "X=" << x << "\t" << "B=" << y << std::endl;
}
```

Multitype Parameters

```cpp
// create a function template that prints the swap of two numbe

#include<iostream>

template<typename T1, typename T2>
void swap(T1 &a, T2 &b) {
  T2 temp = a;
  a = b;
  b = temp;
}

int main() {
  int a = 10, b = 20;
  double x = 20.3, y = 55.3;

  std::cout << "Before Swap" << std::endl;
  std::cout << "A=" << a << "\t" << "B=" << b << std::endl;
  std::cout << "X=" << x << "\t" << "B=" << y << std::endl;

  swap(a, b);
  swap(x, y);

  std::cout << "After Swap: "<< std::endl;
  std::cout << "A=" << a << "\t" << "B=" << b << std::endl;
  std::cout << "X=" << x << "\t" << "B=" << y << std::endl;
}
```

```cpp

#include<iostream>

int const& max (int const& a, int const& b){
  std::cout << "max(int, int)" << std::endl;
  return a < b ? b : a;
}
template <typename T1, typename T2>
T1 const& max (T1 const& a, T2 const& b) {
  std::cout << "max(T1, T2)" << std::endl;
  return a < b ? b : a;
}
template <typename T1, typename T2>
T1 const& max (T1 const& a, T2 const& b, T2 const& c) {
  std::cout << "max(T1,T2, T2)" << std::endl;
  return max(max(a,b), c);
  // max(max<>(a,b), c);
}
int main() {
  max(15.0, 20.0);
  max('x', 'y');
  max(15,25);
  max<>(15,25); // Explicit instantiation
  max<double>(15.0,25.0);
  max(15, 20, 25);

  // max<double, double, double> (20.0, 15.0, 20.0);
}
```

Explicit Specialisation

```
#include <iostream>

template <typename T>
void fun(T a) {
  std::cout << "The main template fun(): "
            << a << std::endl;
}

template<> // may be skipped, but prefer overloads
void fun(int a) {
  std::cout << "Explicit specialisation for int type: "
            << a << std::endl;
}

int main() {
  fun<char>('a');
  fun<int>(10);
  fun<float>(10.14);
}
```

```
#include<iostream>
#include<sstream>
#include<vector>

template<typename T>
T add_all(const std::vector<T> &list) {
        T accumulator = {};
        for (auto& elem:list){
        accumulator += elem;
    }

    return accumulator;
}

template<>
T add_all(const std::vector<std::string> &list) {
        std::string accumulator = {};
        for (const std::string& elem : list)
        for (const char& chr : elem)
                accumulator += elem;
    }

        return accumulator
}

int main() {
  std::vector<int> ivec = {4,3,2,4};
  std::vector<double> dvec = {4.0,3.0,2.0,4.0};
  std::vector<string> svec = {"abc", "bcd"};
  std::cout << add_all(ivec) << std::endl;
  std::cout << add_all(dvec) << std::endl;
  std::cout << add_all(svec) << std::endl;
}
```

Type and Non-type Template Parameters

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

template type para

template non-type para

```
#include <array>
#include <iostream>

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

auto main() -> int {
	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 = " << find_min(x) << "\n";
	std::cout << "min of x = " << find_min(y) << "\n";
}
```

demo703-nontype1.cpp

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

Type and Non-type Template Parameters

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

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

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

demo704-nontype2.cpp

Class Templates

- How we would currently make a Stack type
- Issues?
  - Administrative nightmare
  - Lexical complexity (need to learn all type names)
  - Compiler can not deduce the template parameter type for class template: We have to tell data type we are using.

```cpp
class int_array {
  int array[15];
public:
  void initialize(int value) {
    for(int i = 0; i < 15; i++) {
        array[i] = value;
    }
  }
  int& at(int index) {
    return array[index];
  }
```

Text

```
Same behaviour with double

..

..

..

..
```

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

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

Class Templates

```cpp
//template <template T> //for function
// template < parameter-list > class-declaration
template <class T>
class ClassName {
  private:
    T var;
    ... .. ...
  public:
    T function_name(T arg);
    ... .. ...
};
```

```cpp
ClassName<dataType> class_object;
//For Example
ClassName<int> class_object;
ClassName<float> class_object;
ClassName<string> class_object;
```

```cpp
#include <iostream>

// Class template
template <class T>
class Number {
   private:
    // Variable of type T
    T num;

   public:
    Number(T n) : num(n) {}   // constructor

    T get_num() {
        return num;
    }
};

int main() {

    // create object with int type
    Number<int> number_int(7);

    // create object with double type
    Number<double> number_double(7.7);

    std::cout << "int Number = " << number_int.get_num() << std::endl;
    std::cout << "double Number = " << number_double.get_num() << std::endl;

    return 0;
}
```

Class Templates

```cpp
template <class T>
class ClassName {
    ... .. ...
    // Function prototype
    ReturnType function_name();
};

// Function definition
template <class T>
returnType ClassName<T>::function_name() {
    // code
}
```

```cpp
template <class T>
class A
{
  static int i;
};

template <class T>
int A<T>::i=0;
```

```cpp
#include <iostream>

// Class template
template <typename T>
class Number {
   private:
    // Variable of type T
    T num;

   public:
    Number(T n) : num(n) {}   // constructor
        T get_num();
  //  T get_num() {
  //       return num;
  //  }
};

// Member template definition
template <typename T>
T Number<T>::get_num() {
    return num;
}

int main() {

    // create object with int type
    Number<int> number_int(7);

    // create object with double type
    Number<double> number_double(7.7);

    std::cout << "int Number = " << number_int.get_num() << std::endl;
    std::cout << "double Number = " << number_double.get_num() << std::endl;

    return 0;
}
```

Class Templates

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

#include <iostream>
#include <vector>

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

private:
        std::vector<T> stack_;
};

#include "./demo705-classtemp.tpp"

#endif // STACK_H
```

demo705-classtemp-main.h

```cpp
#include "./demo705-classtemp.h"

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

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

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

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

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

demo705-classtemp-main.tpp

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

Class Templates

```cpp
#include <iostream>
#include <string>

#include "./demo705-classtemp.h"

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.
}
```

(int, string)

2 instancated

demo705-classtemp-main.cpp

Class Template: Array

```
#include <iostream>
class int_array {
  int array[10];
public:
  void fill(int value) {
    for (int i = 0; i < 10; i++)
      array[i] = value;
  }
 int& at(int index) {
    return array[index];
  }
};
class string_array {
public:
  std::string array[10];
  void fill(std::string value)
  {
    for (int i = 0; i < 10; i++)
      array[i] = value;
  }
 std::string& at(int index)
  {
    return array[index];
  }
};

int main()
{
  int_array<int> int_arr;
  int_arr.fill(2);
  std::cout << "int_array[4]: " << int_arr.at(4) << std::endl;
  string_array<std::string> str_arr;
  str_arr.fill("abc");
  str_arr.at(6) = "123";
    for (int i = 0; i < 8; i++)
    std::cout << "str_arr[" << i << "]: " << str_arr.at(i) << std::endl;
   return 0;
}
```

```
#include <iostream>

template <typename T, std::size_t length>
class Array {
  T array[length];

public:
  void fill(T value) {
    for (int i = 0; i < length; i++)
      array[i] = value;
  }

  // returns a reference to array element of type T@ given index
  T& at(int index) {
    return array[index];
  }
};


int main() {
  Array<int, 5> int_arr;
  int_arr.fill(2);
  std::cout << "int_array[4]: " << int_arr.at(4) << std::endl;
  Array<std::string, 8> str_arr;

  str_arr.fill("abc");
  str_arr.at(6) = "123";

  for (int i = 0; i < 8; i++)
    std::cout << "str_arr[" << i << "]: " << str_arr.at(i) << std:::

  return 0;
}
```

Class Templates

Default rule-of-five (you don't have to implement these in this case)

The rule of 5 states that if a class has a user-declared destructor, copy constructor, copy assignment constructor, move constructor, or move assignment constructor, then it must have the other 4.

Have <T> when referring to type.

```
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() { }
```

Class Template Specialisation

```cpp
#include <iostream>

template <class T>
class Test {
// Data members of test
public:
  Test() {
    // Initialization of data members
    cout << "General template object \n";
  }

  // Other methods of Test
};

template <>
class Test<int> {
public:
  Test() {
    // Initialization of data members
    std::cout << "Class template specialisation\n";
  }
};

int main() {
  Test<int> a;
  Test<char> b;
  Test<float> c;

  return 0;
}
```

Inclusion compilation model

- What is wrong with this?
- g++ min.cpp main.cpp -o main

min.h

```
template <typename T>
auto min(T a, T b) -> T;
```

#include "./min.h"

min.cpp

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

#include "./min.h"

main.cpp

```
#include <iostream>

auto main() -> int {
        std::cout << min(1, 2) << "\n";
}
```

Inclusion compilation model

- When it comes to templates, we include definitions (i.e. implementation) in the .h file
  - This is because template definitions need to be known at **compile time** (template definitions can't be instantiated at link time because that would require an instantiation for all types)
- Will expose implementation details in the .h file
- Can cause slowdown in compilation as every file using min.h will have to instantiate the template, then it's up the linker to ensure there is only 1 instantiation.

min.h

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

main.cpp

```cpp
#include <iostream>

auto main() -> int {
  std::cout << min(1, 2) << "\n";
}
```

Inclusion compilation model

- Alternative: Explicit instantiations
- Generally a bad idea

min.h

```cpp
template <typename T>
T min(T a, T b);
```

min.cpp

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

template int min<int>(int, int);
template double min<double>(double, double);
```

main.cpp

```cpp
#include <iostream>

auto main() -> int {
        std::cout << min(1, 2) << "\n";
        std::cout << min(1.0, 2.0) << "\n";
}
```

Inclusion compilation model

- Lazy instantiation: Only members functions that are called are instantiated
  - In this case, pop() will not be instantiated
- Exact same principles will apply for classes
- Implementations must be in header file, and compiler should only behave as if one Stack<int> was instantiated

main.cpp

```
auto main() -> int {
	stack<int> s;
	s.push(5);
}
```

stack.h

```
#include <vector>

template <typename T>
class stack {
public:
	stack() {}
	auto pop() -> void;
	auto push(const T& i) -> void;
private:
	std::vector<T> items_;
}

template <typename T>
auto stack<T>::pop() -> void {
	items_.pop_back();
}

template <typename T>
auto stack<T>::push(const T& i) -> void {
	items_.push_back(i);
}
```

Static Members

```cpp
#include<iostream>

template<typename T>

void print(const T &x) {
  static int value = 10;
  std::cout << ++value
            << std::endl;
}

int main() {
  print(1);
  print('x');
  print(2.5);
  print(2);
  print(3);
}
```

```cpp
#include <vector>

template<typename T>
class stack {
public:
        stack();
        ~stack();
        auto push(T&) -> void;
        auto pop() -> void;
        auto top() -> T&;
        auto top() const -> const T&;

private:
        static int num_stacks_;
        std::vector<T> stack_;
};

template<typename T>
int stack<T>::num_stacks_ = 0;

template<typename T>
stack<T>::stack() {
        num_stacks_++;
}

template<typename T>
stack<T>::~stack() {
        num_stacks_--;
}
```

demo706-static.h

Each template instantiation has it's own set of static members

```cpp
#include <iostream>

#include "./demo706-static.h"

auto main() -> int {
        stack<float> fs;
        stack<int> is1, is2, is3;
        std::cout << stack<float>::num_stacks_ << "\n";
        std::cout << stack<int>::num_stacks_ << "\n";
}
```

1

3

demo706-static.cpp

Friends

Each stack instantiation has one unique instantiation of the friend

```cpp
#include <iostream>
#include <vector>

template<typename T>
class stack {
public:
	auto push(T const&) -> void;
	auto pop() -> void;

	friend auto operator<<(std::ostream& os, stack<T> const& s) -> std::ostream& {
		return os << "My top item is " << s.stack_.back() << "\n";
	}

private:
	std::vector<T> stack_;
};

template<typename T>
auto stack<T>::push(T const& t) -> void {
	stack_.push_back(t);
}
```

demo707-friend.h

```cpp
#include <iostream>
#include <string>

#include "./stack.h"

auto main() -> int {
	stack<std::string> ss;
	ss.push("Hello");
	std::cout << ss << "\n":

	stack<int> is;
	is.push(5);
	std::cout << is << "\n":
}
```

demo707-friend.cpp

Two Phase Translation

Compiler processes each template into two phases:

1 - When compiler reaches the definition

(happen once for each template)

2 - When compiler instantiates the template for particular combination of type arguments.

(happen once for each instantiation)

(Unrelated) Constexpr

- We can provide default arguments to template types (where the defaults themselves are types)
- It means we have to update all of our template parameter lists

Constexpr

- Either:
  - A variable that can be calculated at compile time
  - A function that, if its inputs are known at compile time, can be run at compile time

```cpp
#include <iostream>

constexpr int constexpr_factorial(int n) {
	return n <= 1 ? 1 : n * constexpr_factorial(n - 1);
}

int factorial(int n) {
	return n <= 1 ? 1 : n * factorial(n - 1);
}

auto main() -> int {
	// Beats a #define any day.
	constexpr int max_n = 10;
	constexpr int tenfactorial = constexpr_factorial(10);

	// This will fail to compile
	int ninefactorial = factorial(9);

	std::cout << max_n << "\n";
	std::cout << tenfactorial << "\n";
	std::cout << ninefactorial << "\n";
}
```

demo708-constexpr.cpp

Constexpr (Benefits)

- Benefits:
  - Values that can be determined at compile time mean less processing is needed at runtime, resulting in an overall faster program execution
  - Shifts potential sources of errors to compile time instead of runtime (easier to debug)

Feedback