C/C++ PRIMER

LECTURE 8: GENERIC PROGRAMMING AND C++ TEMPLATES

Fabian Wermelinger
Harvard University

OUTLINE

- Generic programming
- `C++` templates
- Implementation approaches for templated code

GENERIC PROGRAMMING

- ***Conceptual:*** programs encode algorithms using data structures with some given types.
- Algorithms are generic, they ***do not*** depend on a particular data type. ***Example:*** the quicksort algorithm can sort any sequence of entities among which a *pairwise comparison operator* is defined.
- ***Limitation:*** any classical implementation of an algorithm is limited to one specific data type. *Type aliases can be used but the compiler will still only generate one binary representation for the aliased type.* ***Example:*** the quicksort algorithm is only compiled for `double` arrays.
- ***Generic algorithms:*** allow for a *type template* that creates different instantiations for different types. The compiler is our friend, let it do the work.

GENERIC FUNCTIONS IN C++

- Let us look a generic C++ function to compute the absolute value:

```cpp
template <typename T>
T abs(T value)
{
    if (value < 0) {
        return -value;
    } else {
        return value;
    }
}
```

- The ***data type*** is templated by a template parameter `T`.
- For this function, the compiler can *automatically infer* what the type `T` should be at the time the function is being called:

```cpp
int main(void)
{
    std::cout << abs(-10) << std::endl;    // int
    std::cout << abs(-10.0) << std::endl;  // double
    std::cout << abs(-10l) << std::endl;   // long int
    std::cout << abs(-10.0f) << std::endl; // float
    return 0;
}
```

GENERIC FUNCTIONS IN C++

- For this function, the compiler can *automatically infer* what type `T` should be at the time the function is being called:

```
1 int main(void)
2 {
3     std::cout << abs(-10) << std::endl;    // int
4     std::cout << abs(-10.0) << std::endl;  // double
5     std::cout << abs(-10l) << std::endl;   // long int
6     std::cout << abs(-10.0f) << std::endl; // float
7     return 0;
8 }
```

This is very convenient. You can think of it as *"function overload on demand"* as the compiler generates code for all four `abs` instances above. They are four different instances in binary but all of them are generated from the same *function template*. This is called ***compile-time polymorphism*** (at the cost of increased compilation time).

GENERIC FUNCTIONS IN C++

```cpp
template <typename T>
T abs(T value)
{
    if (value < 0) {
        return -value;
    } else {
        return value;
    }
}
```

```cpp
int main(void)
{
    std::cout << abs('a') << std::endl;
    std::cout << abs("a") << std::endl;
    return 0;
}
```

- Since the compiler generates code by inferring the type of the argument automatically, what do you thing happens here?
- The compiler complains with this error:

```
main.cpp: In instantiation of ‘T abs(T) [with T = const char*]’:
main.cpp:16:21:   required from here
main.cpp:6:15: error: ordered comparison of pointer with integer zero
                      (‘const char*’ and ‘int’)
    6 |     if (value < 0) {
      |         ~~~~~~^~
main.cpp:7:17: error: wrong type argument to unary minus
    7 |         return -value;
      |                 ^~~~
```

- Both the `<` and unary sign flip must be defined for any type `T`.

GENERIC FUNCTIONS IN C++

```cpp
template <typename T>
T abs(T value)
{
    if (value < 0) {
        return -value;
    } else {
        return value;
    }
}
```

```cpp
int main(void)
{
    std::cout << abs('a') << std::endl;
    std::cout << abs("a") << std::endl;
    return 0;
}
```

- The type `'a'` is a `char` which is just a one byte *signed integer*. We can compare that type using " < " and flip the sign with " – ". The compiler will not complain.
- The type `"a"` is a *literal string* which is of type `const char *`. We can use " < " on a pointer but we ***can not*** change the sign of a pointer. Therefore the compiler complains.
- The generated error in this case is well readable. Errors due to template code can be in general *very difficult* to interpret.

GENERIC FUNCTIONS IN C++

Basic form of a function template:

```
1 template <typename T>
2 return_type func_name(argument list)
3 {
4     // The type parameter T can be used here as well as
5     // for the return type or any type in the argument list
6 }
```

Properties of function templates:

- The type parameter `T` is replaced by the required type.
- Instead of using the keyword `typename`, you can also use `class`.
- You can have multiple type parameters: `template <typename U, typename V>`
- Type parameter can be built-in integral types too: `template <int X, bool Y>`
- You can call a function template with explicit type, e.g., `func_name<int>(args)` and you must use it like that if the compiler can not automatically infer the type.

GENERIC FUNCTIONS IN C++

Basic form of a function template:

```
template <typename T>
return_type func_name(argument list)
{
    // The type parameter T can be used here as well as
    // for the return type or any type in the argument list
}
```

Properties of function templates:

- Templates can be *specialized* for a particular type:

```
template <>
int func_name<int>(argument list)
{
    // The type parameter T can be used here as well as
    // for the return type or any type in the argument list
}
```

- This would generate a specialization for the type `int` of the generic template on the top of the slide. This is useful to handle special exceptions or manually optimize instantiations for that type.

TEMPLATES: MULTIPLE TYPE PARAMETERS

```
1 template <typename T1, typename T2>
2 T1 getmax(T1 x1, T2 x2)
3 {
4     if (x1 < x2) {
5         return x2;
6     } else {
7         return x1;
8     }
9 }
```

- You can add more than only one template parameter.
- You can also mix type parameter and integral template parameter:

```
1 template <int X, typename U, typename V>
```

- Recall that you can also use the `class` keyword instead of `typename`:

```
1 template <int X, typename U, class V>
```

It is good practice to use `typename` for single type template parameter.

We see later that we can also have templated template parameter where using both `class` and `typename` can help reading the code.

TEMPLATES: SPECIALIZATION EXAMPLE

```
 1 template <int N>
 2 double power(const double d)
 3 {
 4     double res = 1.0;
 5     for (int i = 0; i < N; ++i) {
 6         res *= d;
 7     }
 8     return res;
 9 }
10
11 // specializations for powers of 2 and 3
12 template <>
13 double power<2>(const double d)
14 {
15     return d * d;
16 }
17
18 template <>
19 double power<3>(const double d)
20 {
21     return d * d * d;
22 }
23
24 int main(void)
25 {
26     const double d = 2.5;
27     std::cout << power<2>(d) << std::endl;
28     std::cout << power<3>(d) << std::endl;
29     std::cout << power<10>(d) << std::endl;
30     return 0;
31 }
```

- In this example we use an integer template argument `N`.
- Templates are expanded at compile-time. This allows us to exploit some optimization techniques explicitly via *specializations*.
- On the left we specialize the function for powers of 2 and 3. We optimize them by removing the loop overhead all together and just return the corresponding powers.
- Note that for this template, the compiler has no way to figure out `N`. We must specify it explicitly like we do in `line 27, 28, 29.`

GENERIC CLASSES IN C++

- Of course we can also create *class templates* in addition to the function templates discussed before.
- The syntax and properties remain the same as for function templates:

```
1 template <typename T>
2 class ClassName
3 {
4     // class definition
5 };
```

- A class template is *a family of types*. You create an instance of one of the types from the family with:

```
1 ClassName<double> c;
```

Here `ClassName<double>` is a ***type***, whereas `ClassName` is not a type, it is a class template.

EXAMPLE: GENERIC CLASSES IN C++

This class is ***specific*** for type `int`

```
1  class FooInt
2  {
3  public:
4      FooInt() : i_(0) {}
5      FooInt(int i) : i_(i) {}
6      int get() const { return i_; }
7      void set(int i) { i_ = i; }
8
9  private:
10     int i_;
11 };
```

This class is ***generic*** for type `T`

```
1  template <typename T>
2  class Foo
3  {
4  public:
5      Foo() : i_(0) {}
6      Foo(T i) : i_(i) {}
7      T get() const { return i_; }
8      void set(T i) { i_ = i; }
9
10 private:
11     T i_;
12 };
```

```
1  int main(void)
2  {
3      FooInt x(1); // limited to int only
4      Foo<int> xi(1); // the same as above
5      std::cout << xi.get() << std::endl;
6
7      // easy to get an instance for double
8      Foo<double> xd(1.1);
9      std::cout << xd.get() << std::endl;
10     return 0;
11 }
```

- Use the template type `T` anywhere needed in the class definition
- You do not need to write `Foo<T>` for constructors, destructors, etc. inside the class. The compiler is aware of it.

CLASS TEMPLATES AND INHERITANCE

- You can inherit from class templates in the same way as with specific classes:

```
template <typename T>
class Base
{
public:
    Base(T d) : data_(d) {}
    const T &get_data() const { return data_; }

protected:
    T data_;
};

template <typename T>
class Derived : public Base<T>
{
public:
    Derived(T d) : Base<T>(d) {}
    void set_data(const T &d) { this->data_ = d; }
};
```

- Note that you must inherit from a specific *type* therefore you must explicitly specify the type of any base classes you use.

CLASS TEMPLATES AND INHERITANCE

- You can inherit from class templates in the same way as with specific classes:

```
1 template <typename T>
2 class Derived : public Base<T>
3 {
4 public:
5     Derived(T d) : Base<T>(d) {}
6     void set_data(const T &d) { this->data_ = d; }
7 };
```

- In class templates, the `this` pointer becomes a *dependent expression*. Inherited attributes or methods must be referenced *explicitly* with `this->`.
- Alternatively, you may use a `using` statement to explicitly define all the attributes you may need:

```
 1 template <typename T>
 2 class Derived : public Base<T>
 3 {
 4     using Base<T>::data_;
 5     using Base<T>::get_data;
 6 public:
 7     Derived(T d) : Base<T>(d) {}
 8     void set_data(const T &d) { data_ = d; }
 9     const T &another_get() const { return get_data(); }
10 };
```

CLASS TEMPLATES AND NESTED TYPES

- It may be that the type `T` defines another type. In order to use such *nested* types you must use the `typename` keyword:

```
 1 class SomeClass
 2 {
 3 public:
 4     using Real = double; // type definition
 5 };
 6
 7 template <typename T>
 8 class OtherClassTemplate
 9 {
10 public:
11     void set_data(const typename T::Real d) { data_ = d; }
12
13 private:
14     typename T::Real data_;
15 };
16
17 int main(void)
18 {
19     SomeClass::Real d = 1.0;
20     OtherClassTemplate<SomeClass> t;
21     t.set_data(d);
22     return 0;
23 }
```

TEMPLATE TEMPLATE PARAMETER

- Template parameter can be templates themselves. In this case it often makes sense to use both the `typename` and `class` keywords in the template parameter list:

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

template <typename T, template <typename> class C>
class SomethingMoreComplex
{
public:
    SomethingMoreComplex(const int n) : data_(n) {}
    T get_index(const int i) { return data_[i]; }

private:
    C<T> data_; // C is a template template parameter
};

int main(void)
{
    SomethingMoreComplex<double, std::vector> c(10);
    std::cout << c.get_index(0) << std::endl;
    return 0;
}
```

GENERIC PROGRAMMING: PROS AND CONS

- Generic programming (templated code) allows for great flexibility and effective reuse of code.
- The compiler will only generate template instantiations that are *actually used* in your code.
- Compilation time of heavily templated code can be very high. In addition, compiling such code will require *a lot* of memory (at least 1GB).
- Compiler errors related to templates can be very cryptic and are hard to comprehend.
- Writing templated code is an advanced feature of `C++`.

IMPLEMENTATION OF TEMPLATED CODE

- The compiler must know the ***definition*** of a template whenever it encounters its use during the compilation phase.
- We can therefore not just simply link the missing pieces together in the last stage of compilation.
- The easiest way to solve this problem is to define templates in *headers only*. This is how it is done for most open-source projects and is called the *inclusion model*. Examples are the Eigen library, `pybind11` or the Standard Template Library (STL).
- Header-only implementations expose the code and is not preferred by companies that sell proprietary software. A way to circumvent this problem are *explicit instantiations* or sometimes precompiled headers. We are not going deeper into this topic here.

HANDS-ON: TEMPLATED CONTAINER CLASS

Hands-On: Write a vector class template (see for example the STL vector: https://en.cppreference.com/w/cpp/container/vector). It should provide the following minimum interface:

1. Constructor that takes the number of elements in the vector.
2. A `resize` method that takes a new number of elements as an argument. Preserve as much data as possible in this operation. Initialize other elements to zero.
3. Indexing operator that checks for out of bounds indices when compiled with debug flags. It must be possible to use the operator on both sides of an assignment.
4. A `size()` method that returns the number of elements in the vector.