Advanced Programming Tutorial
Worksheet 8: Templates

Keywords: templates, RAII, iterators, concepts

Prepare!

Before coming to the tutorial, and so that the time is enough to solve the exercise, revise the slides of the lecture unit "Templates" and answer the following questions:

- ☐ How do we define a function template? How do we use it?
- ☐ What is the difference between classes inherited from the same base class and different instantiations of a class template?
- ☐ What is the two-phase lookup? What are example first- and second-phase errors?
- ☐ What is a variadic template? How do we define and use one?
- ☐ What is a `concept` and what problem does it solve?
- ☐ Look into the skeleton files: What does each `main` function do? What is the structure of each involved class?

Exercise 1: Making our own vector

C++ templates are often used when we want to implement generic data types or containers. By this point, you have seen and used `std::vector` a fair number of times and some of you may already have some idea how this works. In this exercise, we will partially implement the functionality provided by `std::vector`. In `Vec_double.h` and `Vec_double.cpp`, you will find an *incomplete* implementation of a vector for a particular type: `double`. Our goal is to templatize this class so that it works for a generic type `T`.

Note: `Vec` has member variables `_size` and `_capacity`. `_capacity` keeps track of the size of underlying array allocated on the heap / free store, while `_size` counts how many actual elements are currently stored in `Vec`. If at any point `_size == _capacity`, and we want to add an element to `Vec`, we would need to reallocate the underlying heap array so that it can fit more elements.

1. In `Vec.h`, create a templatized version of `Vec`. Can we split the templatized version into header (`.h`) and implementation (`.cpp`) files as well?
2. Implement the `push()` method. Follow the comments in the skeleton and make sure that every time you need to reallocate the underlying array, you increase the capacity by a factor of 2. This strategy for increasing capacity is similar to the ones employed by some implementations of the standard library. Can you imagine why this may be helpful?
3. Implement `operator[]` for `Vec` such that it returns a reference to the element at index `i` for `myvec[i]`. Do you understand the rationale for returning a reference to the element rather than returning the element by value, i.e., returning a copy?

4. **Idea:** Implement the rest of the basic operators for `Vec` (copy assignment etc).

5. **Idea:** Remember the concept of iterators: an iterator is an object that allows us to traverse a container. To itearate over our `Vec` type, we can just use pointers, since a pointer is also an iterator (do you understand why?). Implement the methods `begin()` and `end()` that return pointers to the first and to the one past the last element of the vector respectively. This will allow you to use `Vec` with ranged-based for loops. In general, specifying iterators for containers requires a bit more work.

6. **Idea:** Compile your code with the address sanitizer enabled (`-fsanitize=address`, if available) to check if your code has any memory leaks. If you previously had no issues with getting address sanitizer to work with your compiler / platform then you can just uncomment the relevant line in the included `CMakeLists.txt` file.

7. **Idea:** Use clang-tidy to analyze and fix common issues in your code. Start from the `cppcoreguidelines-prefer-member-initializer` check (use an assignment instead of initializer list to trigger it). Then, specify all `cppcoreguidelines-*` checks. Find more interesting checks in the list. Read the clang-format starter clues.

Exercise 2: Concepts

Sometimes, we want to restrict template parameters to types with certain properties. For example, in a function that calls `operator+` on a type `T`, we may want to express the constraint that `T` defines `operator+` in code, rather than in a comment. C++20 allows us to express such constraints using concepts and additionally, this allows the compiler to give us better error messages if a type does not satisfy the given constraints.

In `sortable.cpp`, you will find a skeleton code that defines a templated function `animal_sort` and a class `Penguin`:

1. Implement the `concept Sortable`. What makes a class sortable?

2. Call the `animal_sort` function on the vector `penguins`. Does this work? If not, why?

3. Modify the class `Penguin` so that it satisfies the constraints defined by the `Sortable` concept. You may need to choose a particular criterion, e.g., height, to sort penguins.

4. Make the `animal_sort` function only work for animals that are both `Cute` and `Sortable`. You can chain multiple concepts together by adding additional `requires concept_name<Type>` clauses after the `template` declaration.

Important: You will need to compile this exercise with `-std=c++20`. If you do not have a recent enough compiler, just use the Compiler Explorer.

Some new concepts

A `concept` specifies a set of named requirements on template parameters. These requirements can appear within a `requires` expression:

```
template<typename T>
concept HasSize = requires(T m)
{
        m.size();
};
```

This concept simply says that for objects `m` of type `T`, the expression `m.size()` is valid, i.e., it will compile. We can have multiple such requirements for a single `concept`.

Having defined a concept, we can use it to constrain particular types like so:

```
template<typename T>
requires HasSize<T> && Sortable<T>
void only_sorts_items_with_size(std::vector<T>& items);
```

Here, the `requires` keyword is used to introduce constraints that type `T` must satisfy. Concepts offer us a fair bit more than what we have explored here. For details, see cppreference.

C++ Core Guidelines

ES.31 Don't use macros for constants or "functions"

T.1 Use templates to raise the level of abstraction of code

T.2 Use templates to express algorithms that apply to many argument types

T.3 Use templates to express containers and ranges

T.5 Combine generic and OO techniques to amplify their strengths, not their costs

T.61 Do not over-parameterize members (SCARY)

T.62 Place non-dependent class template members in a non-templated base class

T.81 Do not mix hierarchies and arrays

Epilogue

General-purpose PDE frameworks, such as OpenFOAM, extensively use templates, e.g., to read different data types from a configuration file or to maintain and pass around lists of points, faces, or cells. While concepts make templated code easier to maintain, they are not yet that common. Since you are the C++ developers of the future, however, start using them already!

Chair of Scientific Computing in Computer Science – Technical University of Munich
Last updated on November 2, 2023