3 Generic programming using templates¶

- **You can download the template for the programming tasks of the module as a** `zip` **file from** `this link`.

Contents

- 3 Generic programming using templates
  - 3.1 Programming task

C++ Primer Chapter 16 (Templates and generic programming)

By using **templates** we can specify generic classes or functions that can handle varying types. With templates, the compiler will determine and generate the needed type-specific versions of the classes or functions. We have already used templates with containers and smart pointers, where we have specified which type of objects the container stores by giving an appropriate template argument.

To illustrate how to build a template class, let's define our own (rather dummy) container to hold at most 10 elements:

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

constexpr size_t kMaxSize = 10;

template <typename T>
class MyContainer {
public:
    MyContainer() : num(0) { }
    bool store(T item);
    T get(unsigned int i) { return data[i]; }

private:
    unsigned int num;
    T data[kMaxSize];
};

template <typename T>
bool MyContainer<T>::store(T item) {
    if (num < kMaxSize) {
        data[num] = item;
        num++;
        return true;
    } else {
        return false;
    }
}

int main() {
    MyContainer<int> holder;
    holder.store(1);
    holder.store(2);
    std::cout << "holder[0]: " << holder.get(0) << std::endl;

    MyContainer<std::string> sc;
    sc.store("Hei");
    sc.store("Moi");
    std::cout << "sc[1]: " << sc.get(1) << std::endl;
}
```

The *MyContainer* class definition uses *T* as the template parameter. This means that all places in the class definition that have *T*, will be replaced by the actual type that is given when the class is used. This can be used anywhere in the class, for example, in member variables or function arguments.

Functions outside any class can also be given template arguments. For example, below is a generic simple function that compares two values. Note that the type given to the function call must support `operator>`.

```
template <typename T>
T Largest(T a, T b) {
    if (a > b)
        return a;
    else
        return b;
}
```

We can implement **specialized templates** for certain types that need specialized handling. For example, let's assume that when we are comparing *char* type values, we want to be case-insensitive. Then we would write a specialized variant for the *char* type as follows:

```
template <>
char Largest(char a, char b) {
    if (toupper(a) > toupper(b))
        return a;
    else
        return b;
}
```

We indicate the specialization by leaving the template type list empty, and the compiler will see the specialized data types from the rest of the function interface.

Here is a short example of a *main* function that uses the above:

```
int main() {
    std::cout << "test A: " << Largest<double>(2.5, 3.5) << std::endl;
    std::cout << "test A: " << Largest<char>('a', 'B') << std::endl;
}
```

(we have omitted the needed headers in this example)

In some cases, the compiler can determine the template types automatically without the user having to list them. This happens when all the template types can be inferred from the arguments. For example, we can omit the template types in the above example as the they can be inferred from the arguments:

```
int main() {
    std::cout << "test A: " << Largest(2.5, 3.5) << std::endl;
    std::cout << "test A: " << Largest('a', 'B') << std::endl;
}
```

Before C++17, automatic template type deduction (the above) didn't work for classes, but now they can also be omitted from classes if the template types can be deduced from the constructor arguments. This is why, in module 2, we could do `std::pair("Liisa", 8)` after C++17 but had to either use `std::pair<std::string, int>("Liisa", 8)` or `std::make_pair("Liisa", 8)` (the latter uses automatic template type deduction for functions).

Templates can have multiple template parameters of different types in the following ways:

```
template <typename T1, typename T2>
class Pair {
public:
    Pair(const T1& a, const T2& b) : val_a_(a), val_b_(b) { }
private:
    T1 val_a_;
    T2 val_b_;
};
```

One can also define partial specializations for class templates. These are defined in the following way (uses the above as the template to be specialized):

```
template <typename T>
class Pair<T, T> {
public:
    Pair(const T& a, const T& b) : val_a_(a), val_b_(b) { }
    bool TypesAreSame() const { return true; };
private:
    T val_a_;
    T val_b_;
};
```

when Pair is called with two types that are the same, this specialization is used, otherwise the one first one is. We could also do

```
template <typename T>
class Pair<T, int> {
public:
    Pair(const T& a, const int& b) : val_a_(a), val_b_(b) { }
private:
    T val_a_;
    int val_b_;
};
```

which is used whenever the second type is int. Note that these two specialization examples clash: what should `Pair<int,int>` use? We would also have to define the specialization for `Pair<int,int>` or just not use `Pair<int,int>` at all, otherwise we would get a compiler error.

Note that partial specialization does not work with functions.

Because compiler will convert a template and its template parameters to a type-specific code before actually compiling it, template classes and functions are defined in a header file (only the user of the template knows with what types the code should be compiled; if the code was in a .cpp file, the compiler wouldn't know what types it should use). When something goes wrong, it is not unusual that templates result in long error messages that may be difficult to understand at first.

3.1 Programming task¶

Points **20 / 20** My submissions **11** Deadline Friday, 15 October 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 17 December 2022, 19:59).

ToString

Objective: Learn basic use of templates.

In this exercise you will need to implement different overloaded versions of function *ToString* that returns the contents of an container as a string. Any type of sequential container should be compatible, i.e. you need to use the container through template.

- One version of the function gets a container type as a parameter, in which case it will return the contents of the whole container in a string.
- Another version of the function gets beginning and ending iterators, in which case the range between iterators is printed.
- In addition, if given a string, the *ToString* should just return the string inside double quotes: `"somestring"`.
- When a string is given as two iterators (e.g., ToString(str.begin(), str.end()), it will be printed as sequence of characters: { f, o, o }

Apart from the single string case, the function should return the container items as comma separated list inside brackets. For example, in the case of string container elements, it would return something like the following (spaces are significant):

```
{ foo, bar, baz }
```

If this task feels difficult to understand at first, you could start with a version that uses only e.g. vector container. Then, modify it to use templates in place of specific container type. Note that e.g. begin() and end() functions are available for all sequential container types.

to_string.hpp

Choose File No file chosen

Submit