TP C++ - 3

Thierry Géraud, Edwin Carlinet

Contents

1 Functions and overloads

In C++, one can have multiple functions (or methods) with the same name, but with different argument types. This is a form of *polymorphism* called *oveloading*.

```
class MessageSender
{
  public:
    int send(const struct in_addr* target , const std:: string data);
    int send(const std:: string data);
  private:
    const struct in_addr* last_target_;
};
```

Here, the MessageSender class exports two send methods: one that takes a target and a message to send, and another that just takes a message to send and sends it to the last used target. The use of method overloading makes the code cleaner and more meaningful. After all, we are sending a message, be it with or without specifying the target.

1.1 Default parameters

In C++, you can assign a default value to a parameter. This allows to define function overloads in a simple way.

```
# include <iostream >

void foo(int x, int y = 7)
{
  std::cout << x << " " << y << std::endl;
}

int main ()
{
  foo(1);
  foo(1,2);
}
```

We have two overloads for `foo`: `foo(int)` and `foo(int,int)`. This is equivalent to the following definitions:

```
void foo(int x, int y)
{
  std::cout << x << " " << y << std::endl;
}

void foo(int x)
{
  foo(x, 7);
}
```

1.2 Interaction with inclusion polymorphism

Consider the following piece of code:

```
struct A {
  virtual void foo(float);
};

struct B : A {
  virtual void foo(int);
};
```

This is a common pitfall, here `B::foo(int)` does not define an overload for `foo`, but hides `A::foo(float)`. Let b be an instance of `B`, then `b.foo(42.0f)` actually calls `B::boo(int)` and not `A::foo(float)`.

To be able to add see the parent method from the child class, you must *import* it as shown below:

```
struct B : A {
  using A::foo;
  virtual void foo(int);
};
```

Note also that overloads defined in a child class cannot be seen by the base class as the overload selection is static (compile-time) and the selection of virtual methods is dynamic (run-time). Thus, in the code below:

```
struct B : A {
  virtual void foo(float) override;
  virtual void foo(int);
};

int main()
{
  A* a = new B;
  a->foo(42);
}
```

`a->foo(42)` actually calls `B::foo(float)` because at compile-time there is only a single candidate method: `A::foo(float)` which will dispatch to `B::foo(float)` at run-time.

1.3 Operator overloading

You can also treat operators as polymorphic functions and give them different behaviors depending on its arguments:

```
class myInteger
{
  public:
  myInteger(int num)
  {
    this->num = num;
  }

  virtual ~myInteger ()
  {
  }

  bool operator == (const myInteger& num2) const
  {
    return num == num2.num;
  }
  bool operator == ( const float& num2) const
  {
    return num == static_cast<int> (num2);
  }

  private :
  int num;
};
```

2 Parametric Polymorphism

Templates are one of the C++ killer features. they allow to write generic functions and classes, parametrized by values or even by types. us, a single declaration can generate multiple classes or functions.

2.1 Template functions

The syntax for templating a function is:

```
template <type identifier , ...>
return_type function_name(args)
{
}
```

For instance, a declaration of su a function could be:

```
template <int add , int mult >
int compose(int x)
{
  return x * mult + add;
}
```

To parametrize with a type, you must specify `typename` instead of `int`, `bool`, etc. For example, let us declare a generic function which tests if its two arguments are equal. If those arguments were integers, this function would simply be:

```
bool equal(int a, int b)
{
  return a == b;
}
```

To make it more generic, we parametrize it by the arguments type:

```
template <typename T>
bool equal(T a, T b)
{
  return a == b;
}
```

`equal` is now a generic function parametrized by `T`. us, it can generate the right code for any type that can be compared with `==`; for instance, `equal<float>` can compare floats:

```
equal <int >(42, 42);
equal <bool >(true, false);
```

These generated functions act exactly as if we had written a version of `equal` dedicated to integer values and another one dedicated to booleans, but thats the compiler that achieved this ungrateful work for us, from the template function we wrote previously. When calling a template function, template parameters are optional if they can be deduced from the function arguments (this is call *template type deduction*).

```
equal (42, 42); // OK, T is int
equal (true, false); // OK T is bool
```

But, if there is an ambiguity in the deduction, compilation fails. For instance:

```
equal(42, 42.0); // Error int vs float
equal<float>(42, 42.0); // OK int will be casted to T=float
```

2.2 Overload resolution

So what is the difference between templated functions and overloading ? Well, if you try to call the name of an overloaded function, the compiler will check if the template arguments match exactly the type of the arguments. If it does, it will instantiate a function template specialization and add it to the set of the candidates in the overload resolution. If it doesn't, it will choose the most appropriate one from the set of the candidate functions.

```cpp
# include <iostream >

template < typename T>
void f(T a, T b) // (1)
{
}

template < typename T>
void f(T* a, T* b) // (2)
{
}

void f(int a, int b) // (3)
{
}

int main ()
{
f(1, 2);      // Candidates: f(int, int) and f(T,T) with T=int,
              // it calls f(int, int) which is better because not template.
f(1, 'b');    // Candidates: f(int, int) so it calls f(int, int)
f("a", "b");  // Candidates: f(T,T) with T=const char* and f(T*,T*) with T = const char
              // f(T*,T*) is better because more specialized
f('a', 'b');  // Candidates: f(int, int) and f(T,T) with T=char
              // f(T,T) is an exact match
}
```

2.3 Template Class

Similarly to functions, it is possible to define a templated class, which will generate classes for many kinds of parameters. This is heavily used by the Standard Template Library to provide generic containers.

```cpp
template <...>
class MyClass
{
  // ...
};
```

2.4 Template Class Specialization

One can specialize a template to give it a different behaviour when its parameters have a particular value:

```cpp
// For class
template <class T>
struct MyClass
{
};

template <>
struct MyClass<int> // Specialization for int
{
};
```

```
// For function
template <class T>
void foo(T x)
{
}

template <>
void foo<int>(int x)
{
}
```

2.4.1 Total Specialization

Thee equal function (wrien previously) will work with any type which implements the comparison with `==`... well, almost!

```
const char x[] = "toto";
const char y[] = "toto";
bool res = equal(x, y) // Call equal(T,T) with T = const char*
```

But `res` is `False`. Indeed, we have been comparing addresses, while we would like to compare the actual string. Then, we should specialized our function for pointers.

```
#include <cstring>

template <>
bool
equal<const char*>(const char* x, const char* y)
{
  return strcmp(x,y) == 0;
}
```

Exercice. Try removing the `const` before `x` and `y`. Why does it not work anymore ?

2.4.2 Partial specialization

It is also possible to specialize only some parameters of a template.

```
template <int i, int j>
class Foo
{
// Generic implementation of class Foo.
};

template <int j>
class Foo <51, j>
{
  /*
  ** Specialized implementation of class Foo when the first
  ** template parameter is 51.
  */
};
```

Note that we can partially specialize a class template, but not a function.

2.5 Accessing types from template types

As seen above, the typename keyword can be used to represent a generic type in template declarations. It can also be used to introduce unknown type identifiers, as in:

```
struct A
{
  typedef int value_type;
};


template <class X>
struct GetValueType
{
  typedef typename X::value_type type;
};
```

Here, `GetValueType<A>::type` is `int`. Why did we add `typename` here? Because although value_type is a type, it could as well be in this very case a method or a field of the parameter `T`. The compiler cannot guess it is a type, so we must explicitly express it. That is, when the scope resolution operator `::` is used, we may try to access to one of there but the compiler can't know for sure which one it is. The `typename` keyword allows you to tell the compiler you are trying to access to a type.

3 Compilation

It is important to understand how templates work to master their compilation. A template is a code generator, not a simple piece of code: it is therefore not possible to compile it in an object file (`*.o`) for later use. When it reaches a template instantiation, the compiler needs the full definition to be able to generate the corresponding piece of code. Template class/function code must then be included wherever the corresponding classes/functions are used. The usual procedure is to use classical header files (`*.hh`) that include `*.hxx` files containing the templated code. Thus, any file including the header will also include its implementation. Beware: if a method is implemented in a `*.hxx` file, it must be `inline` or `template`, or you will get errors like *multiple definitions of symbol foo*. On the other total specialization (which are not really template anymore) should be implemented in `cc` files.

For example, with a template class TheClass, files are as follow:

```
// File TheClass.hh
#ifndef THE_CLASS_HH
# define THE_CLASS_HH

struct BaseClass {
  virtual void a_method() const = 0;
  void b_method();
};

template <typename T>
struct TheClass : BaseClass {
  virtual void a_method() const override;
};

template <>
struct TheClass<char*> : BaseClass {
  virtual void a_method() const override;
```

```
};

template <typename T>
void call_a(const TheClass<T>& x);

# include "TheClass.hxx"
#endif /* !THE_CLASS_HH */
```

```
// File TheClass.hxx
template <typename T>
void TheClass <T>::a_method () const {
}

template <typename T>
void call_a(const TheClass<T>& x) {
  x.a_method();
}
```

```
// File TheClass.cc
#include "TheClass.hh"

void BaseClass::b_method () {
}

template <>
void TheClass<char*>::a_method() const {
}
```

4 Exercices

4.1 Stack

Take the `MyStack` class written during your first C++ day and make it generic a generic class `MyStack<T>` where T can be any arbitrary type.

Test it with `int`, `float`, and `int&`.

4.2 Pair

Write the class `Pair<T1,T1>` which can hold any two value of type `T1` and `T2`. You must implement the following interface.

```
template <typename T1 , typename T2 >
class Pair
{
  public:

  Pair(); // Build a (v1,v2) where v1 and v2 are default initialized
  Pair(const T1 v1, const T2& v2); // Build the pair (v1,v2)

  T1& first (); // Return the first value
  T2& second (); // Return the second value
```

```
  const T1& first () const; // Return the first value (const version)
  const T2& second () const; // Return the second value (const version)
};
```

Example of use:

```
#include <iostream>
#include "pair.hh"

int main()
{
    Pair<int, int> point(69, 51);

    std::cout << "Coordinates: "
              << point.first() << ", "
              << point.second() << std::endl;
}
```

1. Implements the operator == for the Pair class.

   ```
   #include <iostream>
   #include "pair.hh"

   int main()
   {
       Pair<int, int> p1(69, 51);
       Pair<int, int> p2(68, 51);
       Pair<int, int> p3(69, 51);

       if (p1 == p2)
           std::cout << "p1 and p2 are equals" << std::endl;
       if (p2 != p3)
           std::cout << "p2 and p3 are not equals" << std::endl;
       if (p1 == p3)
           std::cout << "p1 and p2 are equals" << std::endl;

   }
   ```

2. Write a template `swap(T& a, T& b)` function which swaps the content of its two arguments.

   ```
   char a = 'a';
   char b = 'b';
   swap(a, b);     // a == 'b' \&\& b == 'a'

   int c = 42;
   int d = 21;
   swap(c, d);     // c == 21 \&\& d == 42

   std::string s1 = "you";
   std::string s2 = "me";
   swap(s1, s2);   // etc.
   ```

4.3 Template or not?

Add ‘typename‘ wherever it is needed:

```cpp
template <typename T>
struct A
{
  typedef T type;
};

template <typename T>
struct Container
{
  typedef T type;
};

template <typename T>
struct Any
{
  /// 1
  typedef Container<T>::type contained_t;
  Container<T>::type it;

  /// 2
  typedef Container<int>::type contained_int;
  Container<int>::type it_int;

  /// 3
  typedef A<T> alias;
  typedef alias::type alias_type;
  typedef Container<alias::type>::type contained_alias_type;
};
```

4.4 Unique pointer

We will see in the next course, the way the C++ Standard Library eases the things to handle pointers and dynamic memory management. One of these cool things is the `unique_ptr` which enables to free the memory automatically when the object gets out of the scope. For instance:

```cpp
struct MyClass{
  void foo();
};
void bar(MyClass&) {
}

int main()
{
  unique_ptr<MyClass> myptr(new MyClass);
  // Use it like a normal pointer
  myptr->foo();
  bar(*myptr);

  // Automatic memory release when myptr destructor is called.
  // no need to call delete ...
}
```

Implement the class UniquePtr¡T¿ with the following interface:

- `UniquePtr<T>::UniquePtr()` default constructor, internal pointer is NULL

- `UniquePtr<T>::UniquePtr(T* x)` constructor that takes ownership of x
- `UniquePtr<T>:: UniquePtr()` destructor that releases memory
- `T* UniquePtr<T>::get() const` returns a pointer to the owned object.
- `const UniquePtr<T>::set(T*)` sets the internal pointing object.
- `T& UniquePtr<T>::operator* () const` dereference operator that returns the owned object.
- `T* UniquePtr<T>::operator-> () const` returns a pointer to the owned object.

4.5 Singleton

Sometimes we would like a class to be instantiated only once, this is the case when a single object should exist during the program. A singleton should feature a method `getInstance()` which returns the unique instance of the class (or creates it if it does not exist). The constructor is thus set private so that it can only be instantiated by itself.

1. Implement the class `MySingleton` having the following interface:

```
class MySingleton
{
  public:
    static MySingleton& getInstance();

  protected:
    MySingleton() = default; // C++11 feature (default implementation)
    MySingleton(const MySingleton&) = default; // ignore this for the moment
    static UniquePtr<MySingleton> instance_;
};
```

   Example of use:

```
int main()
{
  MySingleton   x; // Error because constructor is private
  MySingleton&  y = MySingleton::getInstance(); // OK
  MySingleton   z = y; // Error the copy constructor is private
}
```

2. Design a class `Singletonize<T>` (based on the previous implementation) that would create a new singleton class for `T`. Example of use:

```
class MyClass {
  void foo();
};


int main()
{
  MyClass& x = Singletonize<MyClass>::getInstance();
  x.foo();
}
```

3. Now we can automatically make a class a singleton by a trick call CRTP (curiously recurring template). The idea is that the class we want to singletonize is going to inherit from `Singletonize` and passing itself as argument. Thus, its constructors will get automatically private. We just have to say that only Singletonize should be able to access them.

   And we can write cool stuff like:

```
class MyClass : public Singletonize<MyClass>
{
  friend class Singletonize<MyClass>;
  public:
  void foo();

  private:
  MyClass() = default;
  MyClass(const MyClass&) = default;
};

int main()
{
  MyClass& x = MyClass::getInstance(); // OK
  MyClass y;     // Fails, constructor is private
  MyClass z = x; // Fails too (uniqueness is ensured)
}
```