C++ Templates

Overview There are two types of C++ templates: **function templates** and **class templates**. A *function template* is a parameterized function over type parameters while a *class template* is a parameterized class over type parameters.

Compile-time vs runtime polymorphism Template is for *compile-time polymorphism* while dynamic object binding is for *runtime polymorphism*.

Typical use in STL

```
template<typename C>
C& printLast(const C& container) {
  typename C::const_iterator iter(container.begin());
  while (iter != container.end())
    ++iter;
  return *iter;
}
```

Case #1: Function Templates *Function templates* are special functions that can operate with **generic types**.

```
template <class T>
inline T const& maximum(T const& a, T const& b) {
  return a < b ? b : a;
}

template <typename T> function_decl;
```

When the compiler sees a call to a function `maximum`, it **instantiates** the function template by replacing type parameters with *concrete types*.

```
double f1, f2;
std::cout << maximum(f1, f2) << std::endl;
```

Case #2: Class Templates

```
template<typename T>
class Stack {
  std::vector<T> elems;

public:
  Stack() { }
  ~Stack() { }

  /* copy constructor */
  Stack(T const& s);

  /* copy assignment operator */
  T& operator=(T const& rhs);

  void push(T const& t);
  void pop();
  T const& top() const;
  bool empty() const;
};
```

```
template<typename T>
void Stack<T>::push(T const& t)
{
  elems.push_back(t);
}

template<typename T>
void Stack<T>::pop()
{
  if (!elems.empty())
    elems.pop_back();
}

template<typename T>
T const& Stack<T>::top() const
{
  return elems.back();
}

template<typename T>
bool Stack<T>::empty() const
{
  return elems.empty();
}
```

Template specialization

```
template <typename T1, typename T2>
class Pair {
  ...
};

/* partial specialization: both have same type */
template <typename T>
class Pair<T, T> { ... };
};

/* second type is int */
template <typename T>
class Pair<T, int> { ... };

/* both are pointers */
template <typename T1, typename T2>
class Pair<T1 *, T2 *> { ... };
```

C++ traits class A *traits* class is a class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that *extra level of indirection* that solves all software problems.

Traits are basically *compile-time else-if-then*s. The unspecizliaed template is the else clause, the specializations are the if clauses. Essentially, traits allow you to make *compile-time decisions based on **types***, much as you would make *run-time decisions based on **values***.

Concretely, a **C++ traits class** is a *class template* used to associate information or behavior to a compile-time entity, typically a datatype or a constant, without modifying the existing entity.

A **generic template** is defined that implements the default behavior. In this case, all but one type is void, so

`is_void::value` should be `false`, so we start with:

```
template <typename T>
struct is_void {
  static const bool value = false;
};
```

Next, we add a **specialization** for `void`:

```
template <>
struct is_void<void> {
  static const bool value = true;
};
```

Now, we have a complete traits type that can be used to detect if any give type, passed in as a template parameter, is void.

C++ traits: Example Let's consider a class which works on `float` and `double` datatypes. Each datatype has a size in number of bits. We want to create a traits class which contains type-specific speciazliation for the datatype size.

```
// general template
template <class T>
struct FP_traits { };

// specialiation for float
struct FP_traits<float> {
  typedef float FP_type;
  static inline FP_type width() { return 32; }
};

// speciailization for double
struct FP_traits<double> {
  typedef double FP_type;
  static inline FP_type width() { return 64; }
};
```

Next, when we want to create a class

```
template <class T>
class matrix {
  typedef T number_type;
  typedef FP_traits<number_type> traits_type;

  inline number_type width() {
    return traits_type::width();
  }
}
```