Generic Programming

Basic of Generic Programming - Template

A template is a **construct** that generates an ordinary type or function **at compile time** based on arguments the user supplies for the template parameters.

For example:

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

- A template start with keyword `template`
- A template for a generic function with a single type parameter `T`
    - the `typename` keyword says that this parameter is a placeholder for a type.
- the compiler will replace every instance of `T` with the concrete type argument that is either specified by the user or deduced by the compiler.
- The process in which the compiler generates a class or function from a template is referred to as ***template instantiation***

Template Instantiation

- The parameter `T` in the template definition is called the formal parameter or formal argument of the template.
- When the compiler instantiates a template, it tries to determine the actual type of the template parameter by looking at the types of the actual arguments in a function call.
- there is no automatic type conversion for template arguments

Type Parameters

There is no practical limit to the number of type parameters. Separate multiple parameters by commas:

```
template <typename T, typename U, typename V> class Foo{};
```

The keyword `class` is equivalent to `typename` in this context. You can express the previous example as:

```
template <class T, class U, class V> class Foo{};
```

Argument:

Any **built-in** or **user-defined type** can be used as a type argument.

you can use std::vector in the Standard Library to store variables of type `int`, `double`, std::string, `MyClass`, `const MyClass*`, `MyClass&`, and so on.

Restriction: when using templates is that a type argument must support any operations that are applied to the type parameters.

Non-type Parameters

non-type parameters, also called value parameters.

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};

```

The `size_t` value is passed in as a template argument **at compile time** and must be `const` or a `constexpr` expression.

Above template will be use like:

```
MyArray<MyClass*, 10> arr;
```

空