Function templates

```
template<typename T>
T sum(T x, T y)
{
    cout << "The input type is " << typeid(T).name() << endl;
    return x + y;
}
```

template functions

```
// instantiates sum<double>(double, double)
template double sum<double>(double, double);
// instantiates sum<char>(char, char), template argument deduced
template char sum<>(char, char);
// instantiates sum<int>(int, int), template argument deduced
template int sum(int, int);
```

- A function template is not a type, or a function, or any other entity.
- No code is generated from a source file that contains only template definitions.
- The template arguments must be determined, then the compiler can generate an actual function

Class templates

- A class template defines a family of classes.
- Class template instantiation.

```
// Explicitly instantiate
template class Mat<int>;
```

```
template<typename T>
class Mat
{
    size_t rows;
    size_t cols;
    T * data;
  public:
    Mat(size_t rows, size_t cols): rows(rows), cols(cols)
    {
        data = new T[rows * cols * sizeof(T)]{};
    }
    ~Mat()
    {
        delete [] data;
    }
    T getElement(size_t r, size_t c);
    bool setElement(size_t r, size_t c, T value);
};
```

mattemplate.cpp

模板类的函数实现与声明都必须在同一个.hpp/.h/.cpp/.c文件中，因为在调用某种特定类后编译器会自动生成一串代码

Non-Type Paramaters

- To declare a template

  `template < parameter-list > declaration`

- The parameters can be
  - type template parameters
  - template template parameters
  - non-type template parameters
    - integral types
    - floating-point type
    - pointer types
    - lvalue reference types
    - ...

在编译时定下.

```
template<typename T, size_t rows, size_t cols>
class Mat
{
    T data[rows][cols];
  public:
    Mat(){}
    Mat(const Mat&) = delete;
    Mat& operator=(const Mat&) = delete;
    T getElement(size_t r, size_t c);
    bool setElement(size_t r, size_t c, T value);
};
```

静态数组，能全部拷贝、赋值

Class Template Specialization

一般类模板能适应大部分情况，但若想对bool类型减少内存开销，需特化.

```
template<typename T>
class MyVector
{
    size_t length;
    T * data;
  public:
    MyVector(size_t length): length(length)
    { data = new T[length * sizeof(T)]{}; }
    ~MyVector()
    { delete [] data; }
    MyVector(const MyVector&) = delete;
    MyVector& operator=(const MyVector&) = delete;
    T getElement(size_t index);
    bool setElement(size_t index, T value);
};
```

- Specialize MyVector for bool

```
template<>
class MyVector<bool>
{
    size_t length;
    unsigned char * data;
  public:
    MyVector(size_t length): length(length)
    {
        int num_bytes = (length - 1) / 8 + 1;
        data = new unsigned char[num_bytes]{};
    }
    ~MyVector()
    {
        delete [] data;
    }
    MyVector(const MyVector&) = delete;
    MyVector& operator=(const MyVector&) = delete;
    bool getElement(size_t index);
    bool setElement(size_t index, bool value);
};
```

specialization.cpp

Std classes

std::basic_string

Defined in header <string>

Type	Definition
std::string	std::basic_string<char>
std::wstring	std::basic_string<wchar_t>
std::u8string (C++20)	std::basic_string<char8_t>
std::u16string (C++11)	std::basic_string<char16_t>
std::u32string (C++11)	std::basic_string<char32_t>
std::pmr::string (C++17)	std::pmr::basic_string<char>
std::pmr::wstring (C++17)	std::pmr::basic_string<wchar_t>
std::pmr::u8string (C++20)	std::pmr::basic_string<char8_t>
std::pmr::u16string (C++17)	std::pmr::basic_string<char16_t>
std::pmr::u32string (C++17)	std::pmr::basic_string<char32_t>

Std::array

- a container that encapsulates fixed size arrays.

```
template<
    class T,
    std::size_t N
> struct array;
```

```
std::array<int, 3>a2 ={1, 2, 3};
```

*Keyword: typename/class, class/struct

Some other templates

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<
Key>
> class set;
```

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

```
template<
    class T,
    class Container = std::deque
<T>
> class stack;
```