parameterized declarations and template parameters

1. Declaration form: template<parameters here> + normal class, func, var or alias declaration
   called parameterization clause

2. Member templates defined outside of template class:
```
template <typename T>          -> temp class param
  template <typename U>        -> temp mem class param
class List<T>::Handle{};
```
   "class" here is for Handle

★ Constructor template deletes default ctor.

★ Default function parameter can be templated, but not considered by SFINAE, if provided in func call, not instantiated:
```
template <typename T>
void f(int i = T{}){...};
```

★ Template class, non-template member outside def: eg: static double member
```
template <int I>
double CupBoard<I>::weight = 0.0;
```

★ Class temp cannot share name with a different entity.
```
int X; class X;        (✓ fine)
int Y; template <typename T>
       class Y;        (✗ error)
```

★ Linkage: external unless ~~namespace~~ static func.
   namespace scope (which include global namespace scope)

★ The linkage of an instance of temp has the linkage of that temp. Interesting example:
```
int const zero_int = int{};                  zero_int has internal linkage since it's const (Cpp rule)
template <typename T> T zero = T{};          zero forever has ext linkage no matter what (even if T = const int)
template <typename T> int const volume = 11; forever external linkage !!!
```

★ template parameters names can be omitted: template<typename, int> class X {};

★ non-type template parameters: int, enum, ptr, ptr to member, lval ref, std::nullptr_t, auto, decltype(auto)

★ func and array in template param will decay:
```
template<int func()> struct FuncWrap;
template<int(*func)()> struct FuncWrap;   > equivalent.
```

★ non-ref, non-type params, when appears in expressions, are treated PRValues.
eg:
```
template<unsigned I>
class Foo
{ Foo(){unsigned t = I;} }
```
I → PRvalue

eg:
```
template<int I>
class Foo
{ Foo(){auto&& i = I;} }
```
i has type int&&

★ The parameters of template template parameters can have default template args:
```
template<template<typename T, typename A = MyAllocator> class Container>
class Adaptation {
    Container<int> storage // implicitly equivalent to Container<int, MyAllocator>
    ...
};
```

★ The parameters of template template parameters can be use only in the declaration of other parameters of that ~~parameter parameter~~ template template parameter:
```
template<template<typename T, T*> class Buf>   // ok
class Lexer
{ static T* storage; //error!!! a template template param's param cannot be used here.
};
```

eg: template<typename... T> class Tuple

Title: template parameter pack
: three dots before template param name

template<typename T, unsigned... Dimensions> struct MultiArray;

template<typename... Types> void f(Types... args);

template<typename T, template<typename, typename>... Contains>
void testContainers();

★ Primary class templates, primary variable templates, and alias template can have at most one parameter pack at end

eg: template<typename... Types, typename Last>
class LastType; // error, template parameter pack not at last

★ Function templates can have multiple parameter pack, as long as each template parameter after a parameter pack can be deducted or has a default value:

The similar rule applies to partial specialization of class temp or var temp.

```
template<typename... Types, typename T>
void runTests(T value);    // T is deductable
```

```
template<unsigned...> struct Tensor;
template<unsigned... Dims1, unsigned... Dims2>
auto compose(Tensor<Dims1...>, Tensor<Dims2...>); // ok, Dims2 can be deducted
```

```
template<typename...> TypeList;
template<typename X, typename Y> struct Zip;
template<typename... Xs, typename... Ys>
    struct Zip<TypeList<Xs...>, TypeList<Ys...>>; // partial specialization deducts Xs and Ys
```

★ Conditions default template argument is not permitted:

1. Partial specializations: template<typename T=int> class<T*>; // error
2. Parameter packs: template<typename... Ts=int> struct X; // error
3. Out-of class definition of a member of a class template:
   template<typename T=int> T X<T>::f() { ... }; // Error
4. A friend class template: struct S { template<typename=void> friend struct F; } // Error
5. A friend function template declaration unless it is a def and no declaration of it appears anywhere else in the translation unit.

```
struct S { template<typename=void> friend void f();
    template<typename=void> friend void g() { }; // ok so far
};
template<typename> void g(); // Error, g() was given a default template argument when defined
                             // no other declaration may exist here.
```