**Part C - Class Relationships**  
  
**Class Templates**  
  
Model polymorphic behavior using templates (generics)  
Specialize a template for a particular type

*"Templates supports generic programming, template metaprogramming, etc. through a combination of features such as integer template arguments, specialization, and uniform treatment of built-in and user-defined types. The result is flexibility, generality, and performance unmatched by "generics". The STL is the prime example. " Stroustrup (2014).*

[Template Syntax](https://ict.senecacollege.ca/~oop345/pages/content/templ.html#tem) | [Function Templates](https://ict.senecacollege.ca/~oop345/pages/content/templ.html#fun) | [Class Templates](https://ict.senecacollege.ca/~oop345/pages/content/templ.html#cla) | [Variadic Templates](https://ict.senecacollege.ca/~oop345/pages/content/templ.html#var) | [Exercises](https://ict.senecacollege.ca/~oop345/pages/content/templ.html#exe)

The relationship that describes an identical structure shared by classes or functions is called *parametric polymorphism*.  Each class or function definition is the implementation of the shared structure for a specific type.  The structure is independent of the type involved.  By defining that structure in generic form we reduce code duplication.  Class and function templates serve this purpose.  The compiler generates the class and function definitions from our templates for those types that we specify explicitly.

This chapter describes template syntax for both function and class definitions, reviews function and class templates and introduces template specialization.  This chapter also describes templates that can take a variable number of arguments.

**TEMPLATE SYNTAX**

A template declaration begins with the keyword **template** and does one of the following

* declares or defines a function or class
* defines a member function, a member class, a member enumeration, or static data member of a class template or a class nested within a class template
* defines a member template of a class or class template
* declares an alias

A template declaration is a definition if it defines a function, a class, or a static data member of a class template.  The compiler uses the code immediately following the template header to define the function, class, or static data member.  The template's body does not extend beyond this definition.  A template declaration takes either of the following forms

|  |
| --- |
| **template < *template-parameter-list* > // a function template header**  ***return-type function-name( ... ) { // a function template body***  ***// ...***  ***}***  **template < *template-parameter-list* > // a class template header**  ***class-key Class-name { // a class template body***  ***// ...***  ***};*** |

The less than (**<**), greater than (**>**) pair in the template header encloses the template's parameter list.

The parameter list consists of a comma-separated set of parameters.

**Template Parameters**

A template parameter may be any combination of:

* a type template parameter
* a non-type template parameter
* a template template parameter

**Type Template Parameter**

The admissible type parameters are

* **typename**
* **class**

The **typename** keyword identifies a *template type*.  The **class** keyword is an alternative identifier for a *template type*.

**Non-Type Template Parameter**

A non-type template parameter may be

* an integral or enumeration type - a non-floating-point fundamental type
* a pointer to a object or a function
* an lvalue reference to an object or a function
* a pointer to a member
* **std::nullptr**
* **auto**

Note that a non-type template parameter may not be a floating-point type.

**Template Template Parameter**

A template template parameter may be

* **template< *parameter-list* > typename *name***
* **template< *parameter-list* > class *name***

**Template Body**

A template body accepts externally specified types through the parameters in the template header.  Parameters serve as placeholders throughout the template body for the arguments specified in the template call.  For example,

|  |
| --- |
| **template <typename T> // template header**  **T value;**  **foo<int>(); // template call** |

**T** is the sole template parameter.  The compiler replaces **T** with the implicit or explicitly specified type (here, **int**).

A template name has linkage.  A non-member function template can have internal linkage.  Any other template name has external linkage.

**FUNCTION TEMPLATES**

The following function template swaps two values of type **T**.

|  |
| --- |
| **// Function Template**  **// swap.h**  **template <typename T>**  **void swap(T& a, T& b) {**  **T c;**  **c = a;**  **a = b;**  **b = c;**  **}** |

The following program uses this template to swap two **double**s and two **long**s:

|  |  |
| --- | --- |
| **// Function Template**  **// swap.cpp**  **#include <iostream>**  **#include "swap.h"**  **int main(int argc, char\* argv[]) {**  **if (argc > 4) {**  **double a = atof(argv[1]);**  **double b = atof(argv[2]);**  **long d = atol(argv[3]);**  **long e = atol(argv[4]);**  **swap(a, b);**  **std::cout << "Swapped values are " <<**  **a << " and " << b << std::endl;**  **swap(d, e);**  **std::cout << "Swapped values are " <<**  **d << " and " << e << std::endl;**  **}**  **}** | **>swap 2.3 4.5 78 567**  **Swapped values are 4.5 and 2.3**  **Swapped values are 567 and 78** |

The compiler uses the argument types in the function calls to determine the function definitions to generate.

**Specialization**

A template *specialization* defines an exception to a template definition.

Consider a function template for returning the maximum value of two arguments:

|  |
| --- |
| **// Template Specialization**  **// maximum.h**  **template <typename T>**  **T maximum(T a, T b) {**  **return a > b ? a : b;**  **}** |

This definition applies to all fundamental types, but not to pointers to those types; for instance, not to the **char\*** type.  To create an exception for the **char\*** type, we define the following template specialization:

|  |
| --- |
| **// Template Specialization**  **// maximum.h**  **#include <cstring>**  **template <class T>**  **T maximum(T a, T b) {**  **return a > b ? a : b;**  **}**  **// specialization for char\* types**  **//**  **template <> // denotes specialization**  **char\* maximum<char\*>(char\* a, char\* b) {**  **return std::strcmp(a, b) > 0 ? a : b;**  **}** |

The empty parameter list identifies a specialization.  A specialization does not use template parameters, but declares specific types explicitly.

The following example determines the maximum of two **double**s and two C-style strings:

|  |  |
| --- | --- |
| **// Template Specialization**  **// maximum.cpp**  **#include <iostream>**  **#include "maximum.h"**  **int main(int argc, char\* argv[]) {**  **if (argc > 4) {**  **double a = atof(argv[1]);**  **double b = atof(argv[2]);**  **const char\* d = argv[3];**  **const char\* e = argv[4];**  **double c = maximum(a, b);**  **std::cout << "Greater of " <<**  **a << ", " << b <<**  **" is " << c << std::endl;**  **const char\* f = maximum(d, e);**  **std::cout << "Greater of " <<**  **d << ", " << e <<**  **" is " << f << std::endl;**  **}**  **}** | **>maximum 2.3 4.5 abc def**  **Greater of 2.3, 4.5 is 4.5**  **Greater of abc, def is def** |

**Resolving Ambiguities**

If the arguments in a function call are ambiguous in type, the compiler requires an explicit specification of the type for which to generate the definition.  We specify the type explicitly before the opening parentheis of the function call:

|  |  |
| --- | --- |
| **// Resolving Ambiguities**  **// ambiguities.cpp**  **#include <iostream>**  **#include "swap.h"**  **int main(int argc, char\* argv[]) {**  **if (argc > 4) {**  **double a = atof(argv[1]);**  **double b = atof(argv[2]);**  **float d = atof(argv[3]);**  **float e = atof(argv[4]);**  **double c = maximum<double>(a, d);**  **std::cout << "Greater of " <<**  **a << ", " << d <<**  **" is " << c << std::endl;**  **float f = maximum<float>(b, e);**  **std::cout << "Greater of " <<**  **b << ", " << e <<**  **" is " << f << std::endl;**  **}**  **}** | **>ambiguities 2.3 4.5 3.4 2.1**  **Greater of 2.3, 3.4 is 3.4**  **Greater of 4.5, 2.1 is 4.5** |

**CLASS TEMPLATES**

A template declaration for a class follows the same rules as a template declaration for a function.  Consider a class that contains an array of up to 50 **int**s:

|  |
| --- |
| **// Array**  **// array.h**  **class Array {**  **int a[50];**  **unsigned n;**  **int dummy;**  **public:**  **Array() : n{0u}, dummy{0} {}**  **int& operator[](unsigned i) {**  **return i < 50u ? a[i] : dummy;**  **}**  **};** |

When used with the following application, an **Array** object yields the results on the right

|  |  |
| --- | --- |
| **// Array**  **// array.cpp**  **#include <iostream>**  **#include "array.h"**  **int main() {**  **Array a, b;**  **for (int i = 0; i < 5; i++)**  **a[i] = i \* i;**  **b = a;**  **for (int i = 0; i < 5; i++)**  **std::cout << b[i] << ' ';**  **std::cout << endl;**  **}** | **0 1 4 9 16** |

The template declaration for an **Array** class with elements of type **T** is simply:

|  |
| --- |
| **// Class Template - Array**  **// array.h**  **template <typename T>**  **class Array {**  **T a[50];**  **unsigned n;**  **T dummy;**  **public:**  **Array() : n{0u}, dummy{0} {}**  **T& operator[](unsigned i) {**  **return i < 50u ? a[i] : dummy;**  **}**  **};** |

The compiler replaces type **T** with the argument specified in the call to the **Array** template.  The following program creates an array of **long**s:

|  |  |
| --- | --- |
| **// Class Template**  **// array.cpp**  **#include <iostream>**  **#include "array.h"**  **int main() {**  **Array<long> a, b;**  **for (int i = 0; i < 5; i++)**  **a[i] = i \* i;**  **b = a;**  **for (int i = 0; i < 5; i++)**  **std::cout << b[i] << ' ';**  **std::cout << endl;**  **}** | **0 1 4 9 16** |

Here, the compiler generates a class definition for an **Array** object with elements of type **long**.

**Non-Type Template Parameters**

Non-type template parameters can receive the size of an array.  For example:

|  |
| --- |
| **// Non-Type Template Parameters**  **// array.h**  **template <class T, int size>**  **class Array {**  **T a[size];**  **unsigned n;**  **T dummy;**  **public:**  **Array() : n{0u}, dummy{0} {}**  **T& operator[](unsigned i) {**  **return i < 50u ? a[i] : dummy;**  **}**  **};** |

The argument corresponding to the non-type parameter must be a constant or a constant expression:

|  |  |
| --- | --- |
| **// Non-Type Template Parameters**  **// array.cpp**  **#include <iostream>**  **#include "array.h"**  **int main() {**  **Array <int, 50> a, b;**  **for (int i = 0; i < 5; i++)**  **a[i] = i \* i;**  **b = a;**  **for (int i = 0; i < 5; i++)**  **std::cout << b[i] << ' ';**  **std::cout << std::endl;**  **}** | **0 1 4 9 16** |

**Default Template Parameter Values**

A template declaration for a class accepts default parameter values.  We specify the default value in the same way that we specify a default value for a function parameter in a function declaration:

|  |
| --- |
| **// Default Template Parameter Values**  **// array.h**  **template <class T = int, int size = 50>**  **class Array {**  **T a[size];**  **unsigned n;**  **T dummy;**  **public:**  **Array() : n(0), dummy(0) {}**  **T& operator[](unsigned i) {**  **return i < 50u ? a[i] : dummy;**  **}**  **};** |

If all of the template parameters have default values, we can create a class without supplying any arguments:

|  |  |
| --- | --- |
| **// Default Template Parameter Values**  **// array.cpp**  **#include <iostream>**  **#include "array.h"**  **int main() {**  **Array <> a, b;**  **for (int i = 0; i < 5; i++)**  **a[i] = i \* i;**  **b = a;**  **for (int i = 0; i < 5; i++)**  **std::cout << b[i] << ' ';**  **std::cout << endl;**  **}** | **0 1 4 9 16** |

**Static Data Member Declarations in a Class Template**

A class template that includes a class variable requires a complementary template declaration that defines and initializes the class variable.

Let us introduce a class variable named **count** to count the number of objects of an **Array** class that currently exist.  We add the template declaration for the definition of the class variable to the header file that contains the template declaration for the class:

|  |
| --- |
| **// Static Data Member Declaration**  **// array.h**  **template <typename T = int, int size = 50>**  **class Array {**  **T a[size];**  **unsigned n;**  **T dummy;**  **static unsigned count;**  **public:**  **Array() : n{0}, dummy{0} { ++count; }**  **T& operator[](unsigned i) {**  **return i < 50u ? a[i] : dummy;**  **}**  **static unsigned cnt() { return count; }**  **~Array() { --count; }**  **};**  **template <typename T = int, int size = 50>**  **unsigned Array<T>::count = 0u;** |

Note that the parameter list following the **template** keyword for the static data member is identical to the list for the class definition.

The following example displays the number of objects of each type that have been instantiated:

* objects of the default type for an array of default size
* objects of the **double** type for an array with the default size
* objects of the **int** type for an array with a size of **40**

|  |  |
| --- | --- |
| **// Static Data Member Declaration**  **// array.cpp**  **#include <iostream>**  **#include "array.h"**  **int main() {**  **Array<> s, t;**  **Array<double> u;**  **Array<int, 40> v;**  **std::cout << Array<>::cnt() << std::endl;**  **std::cout << Array<double>::cnt() << std::endl;**  **std::cout << Array<int, 40>::cnt() << std::endl;**  **}** | **2**  **1**  **1** |

Note that

* **s** and **t** are instances of class **Array<>**
* **u** is an instance of the separate class **Array<double>**
* **v** is an instance of the separate class **Array<int, 40>**

Each class is a separate instance of template **Array**.  The first and third classes differ only in the number of elements allocated.

**VARIADIC TEMPLATES (OPTIONAL)**

A template that accepts an arbitrary number of arguments is called a *variadic* template.  A declaration of a variadic template includes a *parameter-pack* as one of its parameters.  An ellipsis to the left of the parameter name identifies a parameter-pack.  A parameter-pack is either a *template parameter-pack* or a *function parameter-pack*.  For example, the following declares a class named **Variadic** that takes at least one argument:

|  |
| --- |
| **template <typename T, typename... parameter-pack>**  **class Variadic;** |

All of the following are valid instantiations of this class template:

|  |
| --- |
| **Variadic<double> a; // 1 argument (minimum)**  **Variadic<double, int> b; // 2 arguments**  **Variadic<double, int, int> c; // 3 arguments**  **Variadic<double, double, int, double> d; // 4 arguments** |

Variadic templates are used with inheritance hierarchies

|  |
| --- |
| **template <typename... BaseClasses>**  **class Variadic : public BaseClasses... {**  **};** |

and with initialization lists

|  |
| --- |
| **template <typename... TT>**  **void foo(TT... args) {**  **const int size = sizeof...(args) + 1; // number of arguments + 1**  **int x[size] = {args..., 0};**  **// ...**  **};** |

An ellipsis to the right of the parameter name identifies a *pack-expansion*.  A *pack-expansion* consists of a *pattern* and an ellipsis.  Note the parameter-pack and the pack-expansion in the following program:

|  |  |
| --- | --- |
| **// Variadic Templates**  **// variadic.cpp**  **#include <iostream>**  **template <typename T>**  **void print(const T& t) {**  **std::cout << t << std::endl;**  **}**  **template <typename T, typename... etc>**  **void print(const T& t,const etc&... pp) {**  **std::cout << t << " | ";**  **print(pp...);**  **}**  **int main() {**  **print(100);**  **print("abcd", 100, 34.56);**  **}** | **100**  **abcd | 100 | 34.56** |

**EXERCISES**

* Complete the practice problem in the [Handout on Templates](https://ict.senecacollege.ca/~oop345/pages/handouts/h18.html).
* Complete the Workshop on [Class Templates](https://ict.senecacollege.ca/~oop345/pages/workshops/w4.html)
* Read Wikipedia on [Templates](http://en.wikipedia.org/wiki/Template_%28programming%29)