Class Templates

In programming languages and type theory, polymorphism provides a unique interface to entities of different types.  The three categories of polymorphism are ad-hoc, inclusion and parametric. C++ implements ad hoc polymorphism by overloading a function name for different parameter types, inclusion polymorphism by using the same function signature across different classes in an inheritance hierarchy and parametric polymorphism by using the same name for classes or functions that share the same structure.  C++ implements parametric (or generic) polymorphism through templates.

The structure of each templated class or function is independent of the type(s) involved.  By defining that structure in a generic form we can reduce code duplication significantly.  The compiler generates the class and function definitions from our templates for the types that we specify explicitly.

This chapter describes template syntax for function and class definitions, introduces class template specialization, discusses inheritance in the context of templates and describes templates that can take variable numbers of arguments.

Template Syntax

A template defines one of the following

* a family of functions, classes or nested classes
* a family of member functions, member classes, member enumerations, or variables
* an alias to a family of types

A template declaration or definition begins with the keyword template is followed by a parameter list enclosed in a single chevron pair < > and then the template body.  Example forms for a template declaration are

|  |
| --- |
| template < *template-parameter-list* > // template header  *return-type function-name( ... ) { // template body for a family of functions*  *// ...*  *}*  template < *template-parameter-list* > // template header  *class-key Class-name { // template body for a family of classes*  *// ...*  *};*  template < *template-parameter-list* > // template header  *type variable\_name; // template body for a family of variables* |

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

Template Parameters

A template parameter may be:

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

Type Template Parameter

A type template parameter declaration takes the form

* typename identifier[=default]
* class identifier[=default]

The typename keyword identifies a *template type*.  The class keyword is an alternative identifier for a *template type*.  identifier is a placeholder for the parameter type throughout the template body.  default is an optional default value for the parameter type.  [ ] identifies an optional element.

An argument corresponding to a type template parameter in a template instantiation should be a recognized type.

Non-Type Template Parameter

A non-type template parameter is a template parameter that is not a placeholder for a type.  That is, its type is declared explicitly.  A non-type template parameterdeclaration takes the form

* type identifier[=default]

A non-type template parameter may be one of the following types

* 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\_t
* auto

Note that a non-type template parameter may not be a floating-point type (before C++20).

An argument corresponding to a non-type template parameter in a templated instantiation should be a constant or constant expression.

Template Template Parameter

A template template parameter is a template parameter that is a placeholder for a template.  The declaration of a template template parameter takes the form

* template< *parameter-list* > typename *identifier*[=default]
* template< *parameter-list* > class *identifier*[=default]

Template Body

We code a template body using the parameter names declared in the template header.  The identifiers serve as placeholders throughout the template body for the arguments specified in the template instantiation.  For every instantiation, the compiler generates code in which it replaces the placeholders with the arguments specified in the instantiation.  For example,

|  |  |
| --- | --- |
| #include <iostream>  template <typename T> // template header  void foo() {  T value;  value = 1.5;  std::cout << value << '\n';  }  int main() {  foo<int>(); // template instantiation  foo<double>(); // template instantiation  } | 1  1.5 |

T is a type template parameter.  The compiler replaces T with the type specified in the instantiation (here, int and then double).

A template identifier or 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 defines a family of functions that swap 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 doubles and two longs:

|  |  |
| --- | --- |
| // 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 generate the two function definitions.

Specialization

A template *specialization* of a function defines an exception to a template definition of that function.

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;  } |

Practically, this definition applies to all fundamental types, but not to pointers to those types; for instance, not to the const char\* type.  To create an exception for the const 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  const char\* maximum<const char\*>(const char\* a, const 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 the specialized type(s) explicitly.

The following example determines the maximum of two doubles 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 |

Overloading versus Specialization

As an alternative to specialization we could overload the function for the const char\* type.  Since the compiler resolves overloading before instatiating any specialization, the specialization would be redundant.  In principle, good design prefers overloading to the equivalent speciailzation, and only specializes when it is required.

|  |  |
| --- | --- |
| // Template Specialization  // maximum.h  #include <iostream>  #include <cstring>  template <class T>  T maximum(T a, T b) {  std::cout << "in template body\n";  return a > b ? a : b;  }  // specialization for char\* types  //  template <> // denotes specialization  const char\* maximum<const char\*>(const char\* a,  const char\* b) {  std::cout << "in specialization\n";  return std::strcmp(a, b) > 0 ? a : b;  } | >maximum 2.3 4.5 abc def  in template  Greater of 2.3, 4.5 is 4.5  in overloaded function  Greater of abc, def is def |
| // Template Specialization  // maximum\_overload.cpp  #include <iostream>  #include "maximum.h"  const char\* maximum(const char\* a, const char\* b) {  std::cout << "in overloaded function\n";  return std::strcmp(a, b) > 0 ? a : b;  }  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;  }  } |

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 family of classes follows the same rules as a template declaration for a family of functions.  Consider a class that contains an array of up to 50 ints:

|  |
| --- |
| // 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 client application, the Array objects yield 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 a family of Array classes 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 generates class definitions for each template instantiation and replaces type T with the argument specified in the instantiation of the Array template.  The following application generates a class named Array that holds elements of type long:

|  |  |
| --- | --- |
| // 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 |

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 family of classes 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 template declaration for a family of classes that include a class variable requires a complementary template declaration to define and initialize the family of class variables.

Into our Array example, let us insert a class variable named count to count the number of objects of the Array class that currently exist.  We add the template declaration for the family of the class variables to the header file that contains the template declaration for the class and modify the constructor and destructor definitions:

|  |
| --- |
| // 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 family of class variables is identical to the list for the family of classes.

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 the three class definitions generated from the template for their family are definitions distinct from one another

* 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>

The first and third class definitions differ only in the number of elements allocated.

Templates and Inheritance

A class can be derived directly from a templated family of classes.  All the usual rules of inheritance and polymorphism apply.  In the following example, we need to pass the template parameter T from the Derived class to the Base class template since the base template requires that type:

|  |  |
| --- | --- |
| // Templates and Inheritance  // templates\_inheritance.h  #include <iostream>  template<typename T>  class Base {  T value;  public:  void set(const T& v) { value = v; }  void display() const { std::cout << value << std::endl; }  };  template<typename T>  class Derived : public Base<T> {  public:  void set(const T& v) { Base<T>::set(v); }  }; | 12 |
| // Templates and Inheritance  // templates\_inheritance.cpp  #include "templates\_inheritance.h"  int main() {  Derived<double> d;  d.set(12);  d.display();  } |

The derived class template can include template parameters of its own.  In the following example, we add the template parameter S to the Derived family of classes:

|  |  |
| --- | --- |
| // Templates and Inheritance  // templates\_inheritance\_.h  #include <iostream>  template<typename T>  class Base {  T value;  public:  void set(const T& v) { value = v; }  void display() const { std::cout << value << std::endl; }  };  template<typename S, typename T>  class Derived : public Base<T> {  S value;  public:  Derived(const S& v) : value{v}{}  void set(const T& v) { Base<T>::set(v + value); }  }; | 16 |
| // Templates and Inheritance  // templates\_inheritance\_.cpp  #include "templates\_inheritance.h"  int main() {  Derived<int, double> d(4);  d.set(12);  d.display();  } |

Variadic Templates

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 |