Software Development

Part E - Polymorphism

Templates

Design polymorphic objects to amplify the reusability of code
Introduce function and class templates
Introduce constrained casting to improve type safety

"Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. " Wikipedia (2013).

Polymorphism is not restricted to related types in object-oriented languages. Many languages also support selection across unrelated types. This polymorphism, which perfects the separation of interfaces from implementations, is called *parametric* or *generic* polymorphism. In parametric polymorphism the type and the logic executed on that type are independent of one another. Different clients can access the same logic using different totally unrelated types.

The C++ language implements parametric polymorphism using template syntax. The compiler generates the implementation for each client type at compile-time from the template defined by the developer.

This chapter describes how to implement parametric polymorphism using template syntax with reference to functions and classes. This chapter also describes the templated keywords available for casting values from one type to another.

FUNCTION TEMPLATE

Template Syntax

A template definition resembles that of a global function with the parentheses replaced by angle brackets. A template header takes the form

```
template<Type identifier[, ...]>
```

The keyword **template** identifies the subsequent code block as a template. The less-than greater-than angle bracket pair (**< >**) encloses the parameter definitions for the template. The ellipsis denotes more comma-separated parameters. ***identifier*** is a placeholder for the argument specified by the client.

Each parameter declaration consists of a type and an identifier. ***Type*** may be any of

- **typename** - to identify a type (fundamental or compound)
- **class** - to identify a type (fundamental or compound)
- **int**, **long**, **short**, **char** - to identify a non-floating-point fundamental type
- a template parameter

The following examples are equivalent to one another:

```
template <typename T>
// ... template body follows here
    T value; // value is of type T
```

```
template <class T>
// ... template body follows here
    T value; // value is of type T
```

The compiler replaces **T** with the argument specified by the client code.

Complete Definition

Consider the following function that swaps values in two different memory locations. This code is defined using references to two `int` variables:

```
void swap(int& a, int& b) {
    int c;
    c = a;
    a = b;
    b = c;
}
```

The template for all functions that swap values in this way follows from replacing the specific type `int` with the type variable `T` and inserting the template header:

```
// Template for swap
// swap.h

template<typename T>
void swap(T& a, T& b) {
    T c;
    c = a;
    a = b;
    b = c;
}
```

We place template definitions in header files; in this case, in `swap.h`.

Calling a Templated Function

A call to a templated function determines the specialization that the compiler generates. The compiler binds the call to that specialization.

For example, to call the `swap()` function for two `double`s and two `long`s, we write the following and leave the remaining work to the compiler:

```
// Calling a Templated Function
// swap.cpp

#include <iostream>
#include "swap.h" // template definition

int main() {
    double a = 2.3;
    double b = 4.5;
    long   d = 78;
    long   e = 567;

    swap(a, b); // compiler generates
                // swap(double, double)

    std::cout << "Swapped values are " <<     Swapped values are 4.5 and 2.3
         a << " and " << b << std::endl;

    swap(d, e); // compiler generates
                // swap(long, long)

    std::cout << "Swapped values are " <<     Swapped values are 567 and 78
         d << " and " << e << std::endl;
}
```

If the arguments in each call are unambiguous in their type, the compiler can specialize the template appropriately. If the arguments are ambiguous, the compiler reports an error.

CLASS TEMPLATE

The syntax for class templates is similar to that for function templates.

The following template defines **Array** classes of specified size in static memory. The template parameters are the type (**T**) of each element in the array and the number of elements in the array (**N**):

```
// Template for Array Classes
// Array.h

template <class T, int N>
class Array {
    T a[N];
public:
    T& operator[](int i) { return a[i]; }
};
```

For the following code, the compiler generates the class definition for an array of element type **int** and size **5** from the **Array** template definition. The output from executing this client program is shown on the right:

```
// Class Template
// Template.cpp

#include <iostream>
#include "Array.h"

int main() {
    Array<int, 5> 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] << ' ';                    0 1 4 9 16
    std::cout << std::endl;
}
```

CONSTRAINED CASTS

Constrained casts improve type safety. Type safety is an important feature of any strongly typed language. Bypassing the type system introduces ambiguity to the language itself and is best avoided. Casting a value from one type to another type circumvents the type system's type checking facilities. It is good programming practice to implement casts only where absolutely unavoidable and localize them as much as possible.

C++ supports constrained type casting through template syntax using one of the following keywords:

- **static_cast<*Type*>(*expression*)**
- **reinterpret_cast<*Type*>(*expression*)**
- **const_cast<*Type*>(*expression*)**
- **dynamic_cast<*Type*>(*expression*)**

Type specifies the destination type. ***expression*** refers to the value to be cast to the destination type.

Related Types

The **static_cast<*Type*>(*expression*)** keyword converts the expression from its evaluated type to the specified type. By far, this is the most common form of constrained cast.

For example, to cast **minutes** to a **float** type, we write:

```
// Cast to a Related Type
// static_cast.cpp

#include <iostream>

int main() {
    double hours;
```

```
    int minutes;

    std::cout << "Enter minutes : ";
    std::cin >> minutes;
    hours = static_cast<double>(minutes)/ 60;  // int and float are related
    std::cout << "In hours, this is " << hours;
}
```

`static_cast<Type>(expression)` performs limited type checking. It rejects conversions between pointer and non-pointer types.

For example, the following constrained cast generates a compile-time error:

```
#include <iostream>

int main() {
    int x = 2;
    int* p;

    p = static_cast<int*>(x);  // FAILS: unrelated types

    std::cout << p;
}
```

Some static casts are portable across different platforms.

Unrelated Types

The `reinterpret_cast<Type>(expression)` keyword converts the expression from its evaluated type to an unrelated type. This cast may produce a value that has the same bit pattern as the evaluated expression.

For example, to cast an `int` type to a pointer to an `int` type, we write:

```
// Cast to an Unrelated Type
// reinterpret_cast.cpp

#include <iostream>

int main( ) {
    int x = 2;
    int* p;

    p = reinterpret_cast<int*>(x);  // int and int* are unrelated

    std::cout << p;
}
```

`reinterpret_cast<Type>(expression)` performs minimal type checking. It rejects conversions between related types.

For example, the following constrained cast generates a compile-time error:

```
#include <iostream>

int main( ) {
    int x = 2;
    double y;

    y = reinterpret_cast<double>(x);  // FAILS types are related

    std::cout << y;
}
```

Few reinterpret casts are portable. Uses include

- evaluating raw data
- recovering data where types are unknown
- quick and messy calculations

Unmodifiable Types

The **const_cast<*Type*>(*expression*)** keyword removes the **const** status from an expression.

A common use case for this constrained cast is a function written by another programmer that does not receive a **const** parameter but should receive one. If we cannot call the function with a **const** argument, we temporarily remove the **const** status and hope that the function is truly read only.

```
// Strip const status from an Expression
// const_cast.cpp

#include <iostream>

void foo(int* p) {
    std::cout << *p << std::endl;
}

int main( ) {
    const int x = 3;
    const int* a = &x;
    int* b;

    // foo expects int* and not const int*
    b = const_cast<int*>(a);  // remove const status
    foo(b);
}
```

const_cast<*Type*>(*expression*) performs minimal type checking. It rejects conversions between different types.

For example, the following code generates a compile-time error:

```
#include <iostream>

int main( ) {
    const int x = 2;
    double y;

    y = const_cast<double>(x); // FAILS

    std::cout << y;
}
```

Inherited Types

The **dynamic_cast<Type>(expression)** keyword converts the value of an expression from its type to another type within the same class hierarchy and performs some type checking.

Downcasts

dynamic_cast<Type>(expression) rejects a downcast from a base class pointer to a derived class pointer if a mismatch occurs or the object cannot be derived from the expression type. The result of a dynamic cast must be tested to ensure that the conversion was successful.

For example:

```
// Downcast within the Hierarchy
// downcast.cpp

#include <iostream>

class Base {
  public:
    virtual void display() const { std::cout << "Base\n"; }
};
class Derived : public Base {
  public:
    void display() const { std::cout << "Derived\n"; }
};
```

```
int main( ) {
    Base* b1 = new Base;
    Base* b2 = new Derived;
    Derived* d1 = dynamic_cast<Derived*>(b1);
    Derived* d2 = dynamic_cast<Derived*>(b2);

    if (d1 != nullptr)
        d1->display();
    else                                                    d1 is not derived
        std::cerr << "d1 is not derived" << std::endl;      Derived

    if (d2 != nullptr)
        d2->display();
    else
        std::cerr << "d2 is not derived" << std::endl;

    delete b1
    delete d2;
}
```

Upcasts

dynamic_cast<Type>(expression) rejects an upcast from a derived class pointer to a base class pointer if a mismatch occurs or the object is not derived from the expression type. The result of a dynamic cast must be tested to ensure that the conversion was successful.

For example, to cast a derived class pointer to a base object **d** to a pointer to its base class part, we write:

```
// Upcast within the Hierarchy
// upcast.cpp

#include <iostream>

class Base {
  public:
    void display() const { std::cout << "Base\n"; }
};
class Derived : public Base {
  public:
    void display() const { std::cout << "Derived\n"; }
};

int main( ) {
    Base* b;
    Derived* d = new Derived;

    b = dynamic_cast<Base*>(d);  // in the same hierarchy
    if (b != nullptr)
        b->display();
    else
        std::cerr << "Mismatch" << std::endl;              Base
    d->display();                                         Derived
    delete d;
}
```

Note that here the **display()** member function is not **virtual**. If it were, both calls to it would produce the same result.

Compile-Time Checking

dynamic_cast<Type>(expression) performs some compile-time type checking. It rejects conversions from a base class pointer to a derived class pointer if the object is monomorphic; that is, if the base class is not a polymorphic type.

For example, the following constrained cast generates a compile-time error:

```
// Dynamic Cast - Compile Time Checking
// dynamic_cast.cpp

#include <iostream>

class Base {
  public:
```

```
    void display() const { std::cout << "Base\n"; }
};
class Derived : public Base {
  public:
    void display() const { std::cout << "Derived\n"; }
};

int main( ) {
    Base* b = new Base;
    Derived* d;

    d = dynamic_cast<Derived*>(b);  // FAILS
    b->display();
    d->display();
    delete d;
}
```

Note that a **static_cast** works here and may produce the result shown on the right. However, the **Derived** part of the object would then be incomplete. **static_cast** does not check if the object is complete, leaving the responsibility to the programmer.

```
// Static Cast - Compile Time Checking
// static_cast.cpp

#include <iostream>

class Base {
  public:
    void display() const { std::cout << "Base\n"; }
};
class Derived : public Base {
  public:
    void display() const { std::cout << "Derived\n"; }
};

int main( ) {
    Base* b = new Base;
    Derived* d;

    d = static_cast<Derived*>(b);  // OK
    b->display();                                   Base
    d->display();                                   Derived
    delete d;
}
```

Note that if **display()** is declared **virtual** the output may be the same for both calls to **display()**.

SUMMARY

- a template header consists of the keyword **template** followed by the template parameters
- the compiler generates the template specialization based on the argument types in the function call
- avoid type casting that completely bypasses the language's type-checking facilities
- if type casting is necessary, use one of the four type cast keywords (usually **static_cast**)

EXERCISES

- Complete the Handout on Function Templates.
- Complete the Workshop on Function Templates.
- Read the Wikipedia article on Templates

print this page Top

Previous: Abstract Base Classes Next: Overview of Polymorphism

Designed by Chris Szalwinski

Last Modified: 05/20/2017 11:50