Cast Operations

This chapter discusses the new cast operators in the C++ standard: `const_cast`, `reinterpret_cast`, `static_cast` and `dynamic_cast`. A cast converts an object or value from one type to another.

7.1 New Cast Operations

The C++ standard defines new cast operations that provide finer control than previous cast operations. The `dynamic_cast<>` operator provides a way to check the actual type of a pointer to a polymorphic class. You can search with a text editor for all new-style casts (search for `_cast`), whereas finding old-style casts required syntactic analysis.

Otherwise, the new casts all perform a subset of the casts allowed by the classic cast notation. For example, `const_cast<int*>(v)` could be written `(int*)v`. The new casts simply categorize the variety of operations available to express your intent more clearly and allow the compiler to provide better checking.

The cast operators are always enabled. They cannot be disabled.

7.2 `const_cast`

The expression `const_cast<T>(v)` can be used to change the `const` or `volatile` qualifiers of pointers or references. (Among new-style casts, only `const_cast<>` can remove `const` qualifiers.) `T` must be a pointer, reference, or pointer-to-member type.

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
```

```
  const A a1;
  const_cast<A&>(a1).f( );                  // remove const
ip = const_cast<int*> (cvip);    // remove const and volatile
}
```

7.3 `reinterpret_cast`

The expression `reinterpret_cast<T>(v)` changes the interpretation of the value of the expression *v*. It can be used to convert between pointer and integer types, between unrelated pointer types, between pointer-to-member types, and between pointer-to-function types.

Usage of the `reinterpret_cast` operator can have undefined or implementation-dependent results. The following points describe the only ensured behavior:

- A pointer to a data object or to a function (but not a pointer to member) can be converted to any integer type large enough to contain it. (Type `long` is always large enough to contain a pointer value on the architectures supported by Sun WorkShop C++.) When converted back to its original type, the value will be the same as it originally was.
- A pointer to a (nonmember) function can be converted to a pointer to a different (nonmember) function type. If converted back to the original type, the value will be the same as it originally was.
- A pointer to an object can be converted to a pointer to a different object type, provided that the new type has alignment requirements no stricter than the original type. If converted back to the original type, the value will be the same as it originally was.
- An lvalue of type *T1* can be converted to a type "reference to *T2*" if an expression of type "pointer to *T1*" can be converted to type "pointer to *T2*" with a reinterpret cast.
- An rvalue of type "pointer to member of X of type *T1*" can be explicitly converted to an rvalue of type "pointer to member of *Y* of type *T2*" if *T1* and *T2* are both function types or both object types.
- In all allowed cases, a null pointer of one type remains a null pointer when converted to a null pointer of a different type.

- The `reinterpret_cast` operator cannot be used to cast away `const`; use `const_cast` for that purpose.
- The `reinterpret_cast` operator should not be used to convert between pointers to different classes that are in the same class hierarchy; use a static or dynamic cast for that purpose. (`reinterpret_cast` does not perform the adjustments that might be needed.) This is illustrated in the following example:

```
class A { int a; public: A(); };
class B : public A { int b, c; };
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);      // safe
     B* bp = reinterpret_cast<B*>(&a1);    // unsafe
     const A a2;
     ap = reinterpret_cast<A*>(&a2);  // error, const removed
}
```

7.4 `static_cast`

The expression `static_cast<T>(v)` converts the value of the expression *v* to type *T*. It can be used for any type conversion that is allowed implicitly. In addition, any value can be cast to `void`, and any implicit conversion can be reversed if that cast would be legal as an old-style cast.

```
class B             { ... };
class C : public B { ... };
enum E { first=1, second=2, third=3 };
void use_of_static_cast(C* c1 )
{
  B* bp = c1;                  // implicit conversion

  C* c2 = static_cast<C*>(bp); // reverse implicit conversion

  int i = second;              // implicit conversion

  E e = static_cast<E>(i);    // reverse implicit conversion
}
```

The `static_cast` operator cannot be used to cast away `const`. You can use `static_cast` to cast "down" a hierarchy (from a base to a derived pointer or reference), but the conversion is not checked; the result might not be usable. A `static_cast` cannot be used to cast down from a virtual base class.

7.5 Dynamic Casts

A pointer (or reference) to a class can actually point (refer) to any class derived from that class. Occasionally, it may be desirable to obtain a pointer to the fully derived class, or to some other subobject of the complete object. The dynamic cast provides this facility.

Note – When compiling in compatibility mode (`-compat[=4]`), you must compile with `-features=rtti` if your program uses dynamic casts.

The dynamic type cast converts a pointer (or reference) to one class *T1* into a pointer (reference) to another class *T2*. *T1* and *T2* must be part of the same hierarchy, the classes must be accessible (via public derivation), and the conversion must not be ambiguous. In addition, unless the conversion is from a derived class to one of its base classes, the smallest part of the hierarchy enclosing both *T1* and *T2* must be polymorphic (have at least one virtual function).

In the expression *dynamic_cast*<*T*>(*v*), *v* is the expression to be cast, and *T* is the type to which it should be cast. *T* must be a pointer or reference to a complete class type (one for which a definition is visible), or a pointer to *cv* `void`, where *cv* is an empty string, `const`, `volatile`, or `const volatile`.

7.5.1 Casting Up the Hierarchy

When casting up the hierarchy, if *T* points (or refers) to a base class of the type pointed (referred) to by *v*, the conversion is equivalent to `static_cast<T>(v)`.

7.5.2 Casting to `void*`

If *T* is `void*`, the result is a pointer to the complete object. That is, *v* might point to one of the base classes of some complete object. In that case, the result of `dynamic_cast<void*>(v)` is the same as if you converted *v* down the hierarchy to the type of the complete object (whatever that is) and then to `void*`.

When casting to `void*`, the hierarchy must be polymorphic (have virtual functions). The result is checked at runtime.

7.5.3 Casting Down or Across the Hierarchy

When casting down or across the hierarchy, the hierarchy must be polymorphic (have virtual functions). The result is checked at runtime.

The conversion from *v* to *T* is not always possible when casting down or across a hierarchy. For example, the attempted conversion might be ambiguous, *T* might be inaccessible, or *v* might not point (or refer) to an object of the necessary type. If the runtime check fails and *T* is a pointer type, the value of the cast expression is a null pointer of type *T*. If *T* is a reference type, nothing is returned (there are no null references in C++), and the standard exception `std::bad_cast` is thrown.

For example, this example of public derivation succeeds:

```
class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, public B { };
void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = &ab;        // no casts needed
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // succeeds
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

whereas this example fails because base class `B` is inaccessible.

```
class A { public: virtual void f(); };
```

```
class B { public: virtual void g(); };
class AB : public virtual A, private B { };
void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab;    // C-style cast needed to break protection
  A* ap  = dynamic_cast<A*>(bp); // fails, B is inaccessible
  assert(ap == NULL);
  AB& abr = dynamic_cast<AB&>(*bp);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // fails, B is inaccessible
  }
  catch(const bad_cast&) {
    return; // failed reference cast caught here
  }
  assert(0); // should not get here
}
```

In the presence of virtual inheritance and multiple inheritance of a single base class, the actual dynamic cast must be able to identify a unique match. If the match is not unique, the cast fails. For example, given the additional class definitions:

```
class AB_B :     public AB,       public B  { };
class AB_B__AB : public AB_B,     public AB { };
```

Example:

```
void complex_dynamic_casts( )
{
  AB_B__AB ab_b__ab;
  A*ap = &ab_b__ab;
                    // okay: finds unique A statically
  AB*abp = dynamic_cast<AB*>(ap);
                    // fails: ambiguous
  assert( abp == NULL );
                    // STATIC ERROR: AB_B* ab_bp = (AB_B*)ap;
                    // not a dynamic cast
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // dynamic one is okay
  assert( ab_bp != NULL );
}
```

The null-pointer error return of `dynamic_cast` is useful as a condition between two bodies of code--one to handle the cast if the type guess is correct, and one if it is not.

```
void using_dynamic_cast( A* ap )
{
  if ( AB *abp = dynamic_cast<AB*>(ap) )
```

```
    {            // abp is non-null,
                 // so ap was a pointer to an AB object
                 // go ahead and use abp
      process_AB( abp ); }
  else
    {          // abp is null,
               // so ap was NOT a pointer to an AB object
               // do not use abp
      process_not_AB( ap );
    }
}
```