C++-Style Casts

C++ is a statically typed language - the types of objects (variables, constants) are determined at compile time and they cannot be changed during runtime. Recall that a *type* is a set of values and the set of operations that can be applied on these values. Every object can then be thought of as having memory storage associated with it and these storage bits having values determined by the object's type. During execution, operations are performed on the object that change the values of these storage bits but not their meanings. The use of cast operators specifies a brute force approach to looking differently at the objects' storage bits: either with a different interpretation of the bits or with access rights such as changing the `const`ness of objects.

Old-style

From C, C++ inherited the notation `(T)e`, which performs the necessary conversion to make a value of type `T` from expression `e`. In turn, C++ bequeathed the construction of a value of type `T` from an expression `e` using the function cast notation `T(e)`. This function style cast expression `T(e)` is exactly equivalent to the C-style cast `(T)e`. Certain conversions are not possible with function style casts:

```
int i = 10;
int *pi = &i;
unsigned int ui = 2;

// with function style casts, the type name must be a single word
ui = unsigned int(i); // ERROR - however certain compilers may compile this
// with function style casts, the type name cannot be a pointer
pi = int*(&ui); // ERROR
```

Old-style casts have a number of shortcomings.

1. The notation is hard to spot in a large program. Since casts are deviations from the normal type checking implemented by the language's standard, a useful debugging technique relies on the identification and evaluation of old-style casts to ensure the language's typing mechanism is not subverted. Syntactically, casts consist of little more than a pair of parentheses and an identifier and these symbols are used everywhere in C++. This makes it tough to answer the most basic cast-related questions': "Are any casts used in this program?" That's because human readers are likely to overlook casts because they look like function calls or constructors, and tools like `grep` cannot distinguish them from non-cast constructs that are syntactically similar.
2. The second problem with old-style casts is that they let you cast pretty much any type to pretty much any other type. The kind of conversion intended by the program is not explicit. That is `(T)e` might be doing a portable conversion between related types, a non-portable conversion between unrelated types, or removing the `const` modifier from a pointer or reference type. Without knowing the exact types of `T` and `e`, nobody can tell.

   It would be nice to be able to specify more precisely the purpose of each cast. There is a great difference, for example, between a cast that changes a pointer-to-`const` object into a pointer-to-non`const`-object (that is, a cast that changes only the `const`ness of an object) and a cast that changes a pointer-to-base-class-object into a pointer-to-derived-class-object (that is, a cast that completely changes an object's

type). Old-style casts make no such distinctions. However, you can't cast a `struct` into an `int` or a `double` into a pointer or a function pointer to an object pointer or vice versa. The following table summarizes the permissible cases.

Destination (cast) type	Permitted source types
any arithmetic type	any arithmetic type
any integer type	any pointer type
pointer to (object) `T`, or `(void*)`	a. any integer type b. `(void*)` c. pointer to (object) `Q`, for any `Q`
pointer to (function) `T`	a. any integer type b. pointer to (function) `Q`, for any `Q`
structure or union	none; not a permitted cast
array of `T`, or function returning `T`	none; not a permitted cast
`void`	any type

C++-style casts

C++ addresses the shortcomings of old-style casts by introducing four new cast operators: `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. For most purposes, all you need to know about these operators is that what you're accustomed to writing like this `(type)expression` you would now generally writes like this: `static_cast<type>(expression)`:

```
// Using old-style casts
int num, den;

/* other code here */

// recall that in general, unary operator () has higher precedence
// than arithmetic operator
double result = (double)num/den;

// Using cast operators
double result = static_cast<float>(num)/den;
// named cast operators have higher precedence than arithmetic operators
```

`static_cast`

`static_cast` is a compile-time conversion with basically the same power and meaning as the general purpose C-style cast operator. `static_cast` converts between related types such as one pointer type to another in the same class hierarchy, an integral type to an enumeration, or a floating-point type to an integral type. It also does conversions defined by constructors and conversion operators.

It also has the same kind of restrictions. For example, you can't cast a `struct` into an `int` or a `double` into a pointer using `static_cast` any more than you can with a C-style cast. Furthermore, `static_cast` can't remove `const`ness from an expression, because another new cast, `const_cast`, is designed specifically to do that. Finally, the compiler will not perform two or more user-defined conversions.

```
double d{3.14567};
int i1 = d;  // OK: implicit conversion but hard to find
int i2 = static_cast<int>(d); // same as above but makes the conversion more
                              // obvious to both humans and to computer programs

char x = 'a';
int *p1 = &x; // ERROR: no implicit char* to int* conversion
int *p2 = static_cast<int*>(&x); // ERROR: no implicit char* to int* conversion

void *pv = &n; // OK: implicit int* to void*
int *p3 = static_cast<int*>(pv); // OK: implicit int* to void*

// an enumeration class ...
enum class Flag : char{x = 1, y = 2, z = 4, e = 8};
// plain enumeration
enum Warning { green, yellow, orange, red};

// there is no implicit conversion from an integer to an enumeration since
// most integer values don't have a representation in a particular enumeration
Flag f1 = 4; // ERROR: no implicit Flag to int conversion
Flag f2 = static_cast<Flag>(5); // OK: brute force conversion
Flag f3 = static_cast<Flag>(350); // ERROR: 300 is not a char value
int i3 = f2; // ERROR: no implicit Flag to int conversion
int i4 = static_cast<int>(f2); // OK: brute force conversion

Warning w1 = 5; // ERROR: no implicit conversion from int to Warning
Warning w2 = static_cast<Warning>(w1); // OK: brute force conversion
Warning w3 = static_cast<Warning>(f2); // OK: brute force conversion
Flag f4 = static_cast<Flag>(w2); // OKL brute force conversion

// plain-old-struct with conversion operator
struct C {
public:
  // lots of other stuff here
  operator int (); // convert C to int
private:
  // some attributes here
};

// base class with conversion operator
class B {
public:
  // lots of other stuff here
  operator C (); // convert B to C
private:
  // some attributes here
};
class D : public B { ... }; // derived class
```

```
49
50 D ad[5]; // array of D objects
51 B *pb1 = ad; // OK: implicit conversion for upcast from D* to B*
52 B *pb2 = static_cast<B*>(ad); // OK: brute force conversion
53 D *pd1 = pb2; // ERROR: no implicit downcast conversion from B* to D*
54 D *pd2 = static_cast<D*>(pb2); // OK: brute force conversion
55
56 // compiler will not do two or more user-defined conversions:
57 // notice that we have a way to convert a B to a C to an int
58 B b;
59 int i5 = b; // ERROR: cannot perform two implicit casts
60 int i6 = static_cast<int>(b); // ERROR: still doesn't compile!!!
61 int i7 = static_cast<int>(static_cast<C>(b)); // OK: brute force
62 i7 = static_cast<C>(b); // OK: one explicit cast and one implicit conversion
```

const_cast

`const_cast` is used to cast away or add `const`ness or `volatile`ness of an expression. By using a `const_cast`, you emphasize to both humans and compilers that the only thing you want to change through the cast is the `const`ness or `volatile`ness of something. This meaning is enforced by compilers. If you try to employ `const_cat` for anything other than modifying the `const`ness or `volatile`ness of an expression, your cast will be rejected. The keyword `volatile` informs the compiler that a variable can be modified from somewhere else. For instance, certain memory entries are set by hardware, and we must be aware of this when we write drivers for this hardware. Those memory entries cannot be cached or held in registers and must be read each time from main memory.

```
1  class Widget { ... };
2  class SpecialWidget : public Widget { ... };
3
4  void update(SpecialWidget *psw);
5
6  SpecialWidget sw;              // sw is a non-const object
7  SpecialWidget const &csw {sw}; // but csw is a reference to it as a const object
8
9  update(&sw);  // OK: &sw is SpecialWidget*
10 update(&csw); // ERROR: can't pass a SpecialWidget const* to a function taking
11              // a SpecialWidget*
12 update(const_cast<SpecialWidget*>(&csw)); // OK: constness of &csw is explicitly
13                                           // cast away and csw may now be changed
14                                           // inside update()
15 update((SpecialWidget*)&csw); // OK: same as above, but using a
16                              // harder-to-recognize C-style cast
17 Widget *pw { new SpecialWidget };
18 update(pw); // ERROR: pw's type is Widget* but update takes a SpecialWidget*
19 update(const_cast<SpecialWidget*>(pw)); // ERROR: const_cast can be used only to
20                                         // affect constness or volatileness, never
21                                         // to cast down inheritance hierarchy
```

Another application of `const_cast` is to avoid code duplication in the `const` and non-`const` versions of overloaded functions.

```
class Str {
public:
  // ...
  char const& operator[](size_t pos) const { return s[pos]; }
  char& operator[](size_t pos); // just calls char const& op[]
  // ...
private:
  size_t msz;
  char   *mstr;
};

char&
Str::operator[](size_t pos) {
  return
    const_cast<char&>(               // cast away const on op[]'s return type
      static_cast<const Str&>(*this) // add const to *this's type
        [pos]                        // call const version of op[]
    );
}
```

`reinterpret_cast`

This operator is used to perform type conversions whose result is nearly always implementation-defined. When using this operator, you're telling the compiler, "I know what I'm doing, just reinterpret the bit pattern of the argument as a different type." As a result, `reinterpret_cast`s are rarely portable. Say, you're familiar with the IEEE 754 format for representing single- and double-precision floating-point numbers. Also, suppose you'd like to toggle a particular bit value of a `float` object. Using `reinterpret_cast`, you will be able to cast the address of the `float` object's address to type `uint_fast32_t` (declared since C++11 in `<cstdint>`), extract the value located at that address, apply the appropriate operations to toggle the required bit, and then write the transformed `float` object back to the correct address.

Another use of `reinterpret_cast` arises when reading from and writing to binary files:

```
int const BIG_SIZE {123456};
std::array<int, BIG_SIZE> BIG_DATA;

// fill BIG_DATA with int numbers ...

std::ofstream ofs("data.bin", ios::binary);
// assume ofs is valid
// write contents of BIG_DATA to binary file ...
ofs.write(reinterpret_cast<char const*>(BIG_DATA.data), BIG_SIZE*sizeof(int));
```

Assume - for some weird reason - you'd like to cast between function pointer types. Suppose you have an array of pointers to functions of a particular type:

```
// type name FuncPtr: pointer to function taking no args and returning void
using FuncPtr = void (*)();
FuncPtr FuncPtrArray[10]; // funcPtrArray is an array of 10 FuncPtrs

```

Let us suppose you wish (for some unfathomable reason) to place a pointer to the following function into `FuncPtrArray`:

```
int do_something();
```

You can't do what you want without a cast, because `do_something` has the wrong type for `FuncPtrArray`. The functions in `FuncPtrArray` return `void`, but `do_something` returns an `int`:

```
FuncPtrArray[0] = &do_something;  // ERROR: type mismatch
```

A `reinterpret_cast` lets you force compilers to see things your way:

```
FuncPtrArray[0] = reinterpret_cast<FuncPtr>(&do_something); // this compiles
```

A word of caution: casting function pointers is not portable - C++ offers no guarantee that all function pointers are represented the same way.

`dynamic_cast` and down-casting

Up-casting is the conversion of a derived-type pointer or reference to a base-type pointer or reference while *down-casting* is the conversion of a pointer or reference to a derived-type pointer or reference. Up-casting is implicitly implemented by the compiler since it is an inherently safe cast - inheritance ensures that a derived class object cannot exist without a base class object. However, down-casting is not a safe cast because a base class object can exist independently without derived objects. There are two ways to implement down-casts:

- use `static_cast`, which is unsafe but fast, or
- use `dynamic_cast`, which is safe but requires extra cycles and is only applicable to polymorphic types.

As the term `static` implies, `static_cast` checks compile time information to determine whether the target type is derived from the source type:

```
class Widget { ... };
class SpecialWidget : public Widget { ... };

void update(SpecialWidget *psw);

Widget *pw = new Widget;
update(static_cast<SpecialWidget*>(pw)); // RUNTIME CRASH: passes to update() an
                                         // ill-formed pointer to a SpecialWidget
                                         // object when update() requires a pointer
                                         // to an object of type SpecialWidget

Widget *pw2 = new SpecialWidget;
update(static_cast<SpecialWidget*>(pw2)); // OK: Safe down-cast
```

In the above code fragment, the compiler performs a down-cast of `pw` to generate a pointer of type `SpecialWidget*` which will be ill-formed since `pw` points to an object of type `Widget`. The `static_cast` doesn't perform any runtime checks to ensure that the object pointed to by `pw` has a runtime type `SpecialWidget`. Obviously, program behavior will be undefined and will probably result in a crash. Since no

runtime checks are performed, it is the programmers responsibility to only refer to objects of the right type. In general, it is not always possible for programmers to trace the actual type referenced by a pointer, especially in data-driven applications where the actual type is selected at runtime:

```
Widget *pw = (argc < 2) ? new SpecialWidget : new Widget;
```

On the other hand, `dynamic_cast` performs safe down-casts or casts across an inheritance hierarchy. That is, you use `dynamic_cast` at runtime to cast pointers or references to base class objects into pointers or references to derived or sibling base class objects in such a way that you can determine whether the casts succeeded. Failed casts are indicated by a `nullptr` when casting pointers or an exception of type `bad_cast` when casting references.

```
class Widget { ... }; // assume Widget is polymorphic type
class SpecialWidget : public Widget { ... };

void update(SpecialWidget *psw);

Widget *pw;
// some other code here ...
update(dynamic_cast<SpecialWidget*>(pw)); // OK: passes to update() a pointer to the
                                          // SpecialWidget that pw points to if pw
                                          // really points to one, otherwise passes
                                          // nullptr

void updateViaRef(SpecialWidget& rsw);
updateViaRef(dynamic_cast<SpecialWidget*>(*pw)); // OK: passes to updateViaRef the
                                                 // SpecialWidget that pw points to
                                                 // if pw really points to one,
                                                 // otherwise throws an exception
                                                 // of type bad_cast
```

The `dynamic_cast` operator therefore performs two operations at once. It begins by verifying that the requested cast is valid. Only if the cast is valid does the operator actually do the cast. In general, the type of the object to which the reference or pointer is bound isn't known at compile-time. A pointer-to-base can be assigned to point to a derived object. Similarly, a reference-to-base can be initialized by a derived object. As a result, the verification that the `dynamic_cast` operator performs must be done at runtime.

```
if (Derived *derived_ptr = dynamic_cast<Derived*>(base_ptr)) {
  // use the Derived object to which derived_ptr points
} else {
  // use the Base object to which base_ptr points
}
```

Because there is no such thing as a null reference, it is not possible to use the same checking strategy for references that is used for pointer casts. Instead, when a cast fails, it throws a `std::bad_cast` exception that is presented in `<typeinfo>`.

```
void f(Base const& rb) {
  try {
    Derived const& rd = dynamic_cast<Derived const&>(rb);
    // use the Derived object that rd refers to
  } catch (std::bad_cast e) {
    // handle the fact that the cast failed
  }
}
```

`dynamic_cast` is implemented under the hood as a `virtual` function. Therefore, it is only applicable for polymorphic classes that define or inherit one or more `virtual` functions. Otherwise, all classes would incur the cost of a `vtable`. Polymorphic functions have these anyway, so the cost of the `dynamic_cast` is one extra pointer in the `vtable`.

`dynamic_cast` and cross-casting

An interesting use of `dynamic_cast` arises in multiple inheritance. Consider the following diamond hierarchy:

```
class A { ... }; // A is polymorphic type
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

We can cast across the diamond by taking a source pointer to `B` to target pointer to `C` as long as the addressed object's type is a derived class of both types:

```
B *pbb = new B; // static type of pbb is B* and dynamic type is B*
B *pbd = new D; // static type of pbd is B* and dynamic type is D*

C *pcd = dynamic_cast<C*>(pbd); // OK: B* -> C* with D object being addressed
C *pcb = dynamic_cast<C*>(pbb);
```

We cannot directly use `static_cast` to cross-cast from `B` to `C` since neither is a base nor derived class of the other. However, with some trickery, it is possible to subvert the compiler's type checking. As with other `static_cast`s, it is the programmer's responsibility to determine whether the addressed object can really be casted in this manner.

```
C *pcd2 = static_cast<C*>(pbd); // ERROR: B and C are siblings
C *pcd3 = static_cast<C*>(static_cast<D*>(pbd)); // OK: B* -> D* -> C*
```

Conclusion

The new casts are ugly and hard to type. However, what the new casts lack in beauty they make up for in precision of meaning and easy recognizability. Programs that use the new casts are easier to parse both for humans and for tools. For example, the term `_cast` can be easily `grep`ed to find out which portions of the code are using cast operators to subvert C++'s typing mechanism. Similarly, they allow compilers to diagnose casting errors that would otherwise go undetected. These are powerful arguments for abandoning old-style casts.