UNDERSTANDING AND USING CAST OPERATIONS IN C AND C++

Farrokh Attarzadeh
Department of Engineering Technology
University of Houston

Eric Nagler
Lawrence Technological University
Southfield, Michigan

ABSTRACT

The cast operators in C++ are varied in representation and usage. They support an old-style cast operator that is carried over from the C programming language and still recognized for backward compatibility with legacy code. The C++ language adds the function-style cast operator and, more importantly, introduces four new cast operators. These operators constitute a very important addition to the C++ programming language.

There have been many papers written on these new cast operators, but over the years of teaching and training in the industry, the authors have observed that there is still a noticeable lack of knowledge and proper usage of these operators in academia and the development community. Other observations include the lack of adequate coverage of implicit and explicit conversions in both C and C++, and particularly the new cast operators and the function-style cast in C++. In addition, the issue of writing clear and efficient code is rarely addressed in the many textbooks used in schools and colleges. Because the use of the cast operators is almost inevitable in writing production type code, the authors encourage you to introduce them early in the C++ programming courses and teach their proper use.

This paper will first introduce the concept of conversions and castings in general, and then present the traditional C language casting followed by the new ways to perform casting in C++. The strategy for converting the old-style C cast to new C++ cast operators will also be discussed. In addition, the authors will address the proper compiler settings to gain even more benefit from the compiler-generated messages. Some efficiency considerations will be addressed as well. Finally, several examples will be provided such that educators and technical trainers engaged in teaching C++ programming courses can put them to an immediate use.

INTRODUCTION

The understanding and proper usage of casting operations and conversions is one of the important indicators of robust code. There have been many articles written on the subject of casting and conversions during and after the release of the ISO C++ Standard [1-8]. There are books available that provide a good coverage of the casting and conversions as well [9-15]. It is a well-known fact that casts and conversions are an integral part of both the C and C++ programming languages, but many programmers confuse the terms “cast” and “conversion,” so this paper will provide precise definitions of these two terms.

The C language supports only one style of casting, whereas C++ adds two more styles: (1) the function-style cast (sometimes called a C++ style cast), and (2) the four new-style cast operators. The legacy C-style cast is still supported for purposes of backward compatibility, and at one time it was even targeted for deprecation by the standardization committee. Consequently, its use in C++ programs is problematic and highly discouraged due to its obscureness and inherent unsafe nature, both of which can lead to subtle program bugs. Fortunately, the four new-style C++ cast operators solve this problem with their clear

visibility in the code and built-in safety checks that ensure the proper type of casting is being done. If not, the compiler will emit appropriate diagnostics. Each of these four new-style cast operators will be discussed in depth. Finally, the C++ function-style cast gives the programmer the capability to instantiate a temporary, unnamed instance of a class on the stack and thereby greatly enhance the efficiency of the executable code. This too will be illustrated in great detail.

The authors firmly believe that training developers and teaching students entails more than just describing a particular programming language and at best deciphering the syntax of the language. The authors emphasize the need to instill good programming practices that are not only good now, but good for the future as well. This will encourage educators and trainers to nurture those practices that will last a lifetime. Today it is C++, Java, and C#, but who knows what other programming languages will surface in the future and what the compiler requirements will be. The authors also feel that the habits of most programmers are formed in the introductory programming classes. If and when damage is done, it will be very difficult to fix.

Good programming practices in regard to conversions and casting will be presented in this paper. All illustrations (or sample codes) were tested with three compilers in the Windows operating system: (1) Borland CBuilderX, compiler version 5.6.4; (2) Microsoft .NET 2002; (3) Comeau 4.3.0 front-end compiler. Any differences in behavior of these three compilers will be addressed as well.

STANDARD CONVERSIONS

The standard conversions, also known as built-in or implicit conversions, are conversions performed by the compiler. A conversion can be viewed as a process during which an object of some source type is transformed into an object of some destination type, as shown in Figure 1. The programmer has no control over the conversion other than understanding the rules of the language. All compiled languages, such as C and C++, generally perform such conversions.

C LANGUAGE CONVERSIONS

When looking at the C language, which supports primitive (built-in) types only, there are a variety of possible conversions that can be performed, such as:

- Trivial conversions
- Conversions to integer types
- Conversions to floating-point types
- Conversions to structure and union types
- Conversions to enumeration types
- Conversions to pointer types
- Conversions to array and function types
- Conversions to void type

In general, one need not be too concerned about these different types. It's usually enough to be aware that the compiler will implicitly convert an object or expression of any primitive type into an object of any other primitive type. If the destination primitive type is "wider" (more significant) than the source type, then there is no possibility of losing any accuracy,

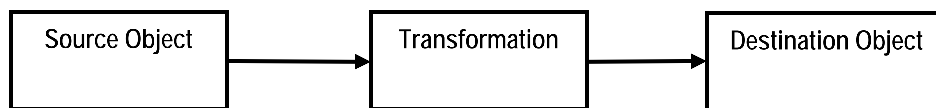

Figure 1. Standard Conversions in C++

and a warning message will never be emitted. For example, the conversion of an *int* into a *double* poses no danger:

```
int i = 5;
double d = i; // OK; d contains 5.0
```

If the opposite is true, then the worst that could happen is that the compiler will emit a warning (not fatal) message, such as the conversion of a *double* to an *int*. Note that in this case truncation, not rounding, occurs.

```
double d = 5.6;
int i = d;  // OK; i contains 5,
            // but the fractional part is lost
```

Note that the Java language, while eliminating some C language primitive types (e.g., *long double*, all *unsigned* integral types, all enumerated types) tightens up on its use of implicit type conversion. That is, a "widening" conversion is still permitted to be performed implicitly by the compiler:

```
double d = 1;      // OK in C and in Java
```

but the opposite (a "narrowing" conversion) is no longer permitted, and instead requires a cast to be performed:

```
int x = 34.56;         // OK in C, error in Java
int x = (int)34.56;    // OK in C and in Java,
                       // but the fraction is lost
```

Although the compiler does these implicit conversions, the authors prefer the use of a cast. This serves the purpose of conveying to the compiler and to the reader of the code that the programmer is aware of the possible loss of significance. This also encourages good programming practices. For example, consider this call to the math library's square root function:

```
int value = 2;
int squareRoot = sqrt(value);
```

There are two implicit conversions occurring. First, since *sqrt()* is declared in *math.h* to receive type *double*, the compiler will implicitly convert the integer 2 into a temporary object on the stack of type *double*, giving it the value 2.0. Second, the function returns its answer as type *double*, in this case 1.414. Since the variable *squareRoot* is declared as type *int*, once again the compiler will implicitly convert this value to type *int* by truncating the fractional portion, thus resulting in a loss of precision. Consequently, the integer value 1 gets stored. If this is really the intent of the programmer, then it would be better to write:

```
int value = 2;
int squareRoot = (int)sqrt((double)value);
```

Now any and all warning messages have been eliminated, and there is no doubt as to the programmer's intent.

Implicit type conversion excludes the commingling of pointer and non-pointer types:

```
int *ptr = 1; // Error; no conversion from int to int*
// ...
int x = 0;
int y = &x;  // Error; no conversion from int* to int
```

The only exception here is that an integral type can be converted to a pointer type if the value of the integral type is a constant zero. For example:

```
int *ptrX = 0;     // OK
```

Any pointer type may implicitly be converted to a *void** type:

```
int x = 0;
void *ptrVoid = &x;      // OK
```

And C supports the implicit conversion of a *void** into some other pointer type:

```
void *ptrVoid = 0;
int *ptrInt = ptrVoid;     // OK
```

A non-*void** pointer type may not be converted to another non-*void** pointer type:

```
int x = 0;
//All compilers yield a fatal error
char *ptrChar = &x;    // Error; no conversion from
                       //   int* to char*
```

C LANGUAGE CASTS

By definition, a *cast* is an explicit conversion. This is the opposite of an implicit conversion in which the compiler needs no permission from the programmer to perform the requisite conversion. The programmer gives the compiler "permission" to perform an explicit conversion by writing a cast. In C a cast is performed by writing the destination type between parentheses, immediately to the left of the object or expression to be cast. It is a unary operator with right-associativity. For example:

```
int i = 5;
double d = (double)i;
// ...
double d = 5.6;
int i = (int)d;
int j = (int)(d + 1.0);    // Writing 1.0 eliminates
                           //   an implicit conversion
                           // from int to double
```

Another way to look at a cast is that it instructs the compiler to avoid any inherent safety checks on the conversion and accept at face value what the programmer is doing. Sometimes a cast is optional, e.g., to clearly show the conversion that is being performed, sometimes it is necessary to achieve the desired result, and sometimes it is needed to eliminate the compiler fatal error and achieve the desired result.

```
// Here the cast is optional
int i = 5;
double d = (double)i;

// Here the cast is mandatory to perform floating
//  point arithmetic
int dividend = 5;
int divisor = 2;
double quotient = (double)dividend / divisor;

// Here the cast is mandatory to avoid a fatal error
unsigned *port = (unsigned*)0x84;
```

C++ LANGUAGE CONVERSIONS (INHERITED)

In C++ all of the standard conversions inherent in the C language are valid, with the following exceptions. First, the implicit conversion from a *void** type to another pointer type no longer works:

```
int x = 0;
void *ptrVoid = &x; // OK
int *ptrX = ptrVoid; // Error
```

This is a good improvement over the C language because it prevents the following (very dangerous) code from compiling, the result of which would be a pointer-to-double now points to an array of integers. Good luck when trying to perform pointer arithmetic.

```
int array[] = { 1, 2, 3 };
void *ptrArray = array;             // OK
double *ptrDouble = ptrArray;  // Error
```

Also, the implicit conversion from an integral type into an enumerated type, while valid in C, is no longer valid in C++:

```
enum colors { red, white, blue };
colors c = red;          // OK
colors c = 9;            // Error
colors c = red + 1;    // Error
```

The C++ standard [16] classifies all of the inherited C language conversions into several groups:

- Lvalue-to-rvalue conversion
- Array-to-pointer conversion
- Function-to-pointer conversion
- Qualification conversions
- Integral promotions
- Floating-point promotion
- Integral conversions
- Floating-point conversions

- Floating-integral conversions
- Pointer conversions
- Pointer to member conversions
- Boolean conversions

These conversions are listed here just to show how expansive and varied they are. This is just too much delegation given to the compiler to do on its own. In other words, you have to be sure that your intention, and what the compiler does for you, are automatically and precisely the same. In order to see what the compiler does, you need to set the warning level appropriately to benefit from the compiler diagnostics. You will also need to disable any compiler-specific extensions to the language to minimize code portability problems. Nevertheless, all C++ compliant compilers will do them for you. These conversions are defined for built-in types (e.g., *char, int, float*) and are applied in several contexts. In this paper, the discussions are limited to the arithmetic type conversions because most introductory topics in C++ involve arithmetic operations and computations of some sort. The basic concepts of truncation, rounding, function formal arguments and return type, function overloading, and built-in library functions can also be taught in this context quite well.

C++ LANGUAGE CONVERSIONS (NEW)

In order to accommodate user-defined types (structures and classes), C++ has added two new implicit type conversions. A third conversion involving the use of an operator conversion function is not discussed here. The first one allows a primitive type to be implicitly converted to a user-defined type. In other words, given:

```
class ADT{};
void f(ADT const&);
// ...
f(1);
```

the programmer is asking the compiler to implicitly convert the integer 1 into a temporary instance of type ADT so that the one parameter being received by function f() can refer to it. Of course, the compiler has no idea whatsoever how to do this. But if the programmer gives the compiler some guidance, then the compiler will take over and perform the conversion implicitly. This guidance is achieved with the inclusion in the class of a constructor called a converting (or conversion) constructor. By definition, this is a constructor that is callable with exactly one argument. For example:

```
class ADT
{
  public:
      ADT(int arg);  // Converting constructor
      // ...
};
```

If desired, the programmer can combine a converting constructor with a default constructor:

```
class ADT
{
  public:
    ADT(int arg = 0);    // Default and
      // converting constructor
      // ...
};
```

Now the following code compiles with no problem:

```
class ADT
{
  public:
      ADT(int arg = 0);
};
void f(ADT const &);
// ...
f(1);
```

Secondly, C++ adds the conversion from a derived class into a public base class to its list of implicit type conversions. That is, given:

```
class Base{};
class Derived : public Base{};
```

then the following code is valid:

```
Derived d;
Base objectBase = d;
Base *ptrBase = &d;
Base &refBase = d;
```

The conversion from Derived to Base is sometimes called "upcasting" because in the class hierarchy the flow is going "up" from the derived class to the base class. Indeed, using UML (Unified Modeling Language) notation the arrow points up from the derived class to the base class.

C++ LANGUAGE CASTS

There are three ways to perform a cast in C++. One of them is the inherited C language cast, and C++ adds two new ways. As noted above, all casts are performed at the request of the programmer, and the compiler carries them out. The three types of casts are shown in Figure 2.

Ideally, the destination type should be the same for all three casts, given the same source type, and in most cases it is. As will be shown, there are source types that will not result in same destination types and those that are illegal in one form of cast and not in the other ones.

The idea is to demonstrate the dangers involved in continuing the practice of using the old-style C language cast and at the same time encourage you to use the new C++ casts instead, but most importantly, to discourage you from using the cast in all its flavors. Care needs to be taken because silencing the compiler is not a good practice and because there are cases where it is difficult to know what the compiler goes through.

FUNCTION-STYLE CAST FOR PRIMITIVE TYPES

The function-style cast (a.k.a. a C++-style cast) is valid for both primitive and user-defined types. Looking first at its use for primitive

types, the syntax is:

```
new-type(single-expression-to-be-cast)
```

Notice how this syntax is the "opposite" of a C-style cast in that the parentheses surround the single-expression-to-be-cast, not the new-type (although in C you still have the option to enclose the single-expression-to-be cast within parentheses). In the following example all three casts will produce the same result:

```
double d = 4.56;
int i = (int)d;      // C-style cast
int i = (int)(d);    // C-style cast
int i = int(d);      // C++-style cast
```

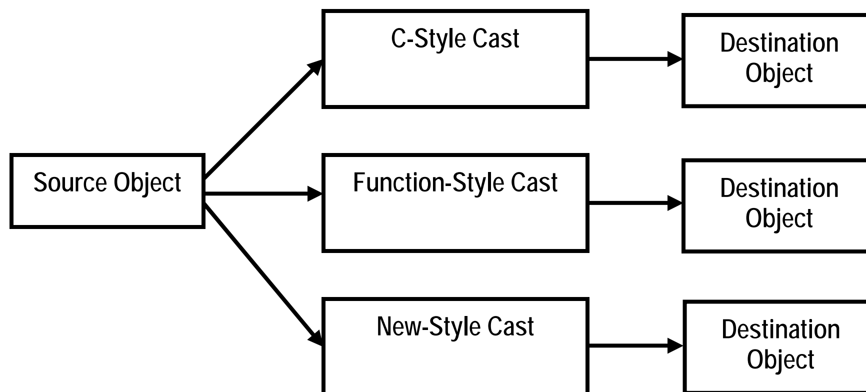

Figure 2. Standard Casts in C++

Be careful, however, when specifying the new-type using Borland CBuilderX. Some primitive types consist of two words, e.g., *unsigned int*, *unsigned long*, *long int*, or a pointer type, e.g., *int**, and these *will not fit* the *new-type*, which must be a single word. Fortunately, the problem is easily solved using a *typedef*:

```
double d = 4.56;
unsigned u = unsigned int(d);     // Error
typedef unsigned int UI;
UI u = UI(d);                     // OK
```

Note that .NET can successfully compile the error case above, even with strictest compiler settings. This implies that standard coding in C++ can be tricky at times. For cross-platform compatibility, however, the authors recommend using a typedef.

The same situation arises when you want to cast into a pointer type. For example:

```
char const *ptrConst = "C++";
char *ptrNonConst = (char *)ptrConst;       //C-style cast
char *ptrNonConst = char *(ptrConst);       // Error
typedef char *PNC;
PNC ptrNonConst = PNC(ptrConst);            // OK
```

Even .NET cannot compile the error case here. Later on you will see how a function-style cast involving a primitive type is better handled by one of the new-style cast operators, and therefore should be written only for user-defined types.

FUNCTION-STYLE CAST FOR USER-DEFINED TYPES

In the world of C++, the function-style cast allows zero or more arguments to be enclosed within parentheses, preceded by a class (or structure) name. This syntax causes an unnamed temporary instance to be created on the stack and a constructor to be called.

```
class-name( /* List of zero or more arguments */ )
```

When is necessary to use this style of casting? Consider a complex number class called Complex that encapsulates both real and imaginary parts as type *int*, with a default/converting constructor, a copy constructor, and a destructor. Each function definition displays a message when invoked.

```
#include <iostream>
class Complex
{
public:
        Complex(int r = 0, int i = 0);
        Complex(Complex const &obj);
        ~Complex();
   private:
        int real, imag;
};
Complex::Complex(int r, int i)
   : real(r), imag(i)
{
std::cout << "Default/converting constructor\n";
}
Complex::Complex(Complex const &obj)
  : real(obj.real), imag(obj.imag)
{
  std::cout << "Copy constructor\n";
}
Complex::~Complex()
{
  std::cout << "Destructor\n";
}
```

Next, suppose that a non-member function called getComplexNumber() is designed to prompt the user for two *int* values to be used to instantiate a Complex number. This number is then returned from the function (by value, of course, since it lives on the stack and will be destroyed). The function can be written in two ways. The first will instantiate the Complex class as a named object on the stack, and then return it. The second will instantiate using a function-style cast. The appropriate code will be executed according to the macro FUNCTION_STYLE_CAST.

```
#define FUNCTION_STYLE_CAST
Complex getComplexNumber()
{
  std::cout << "Enter two integer values: ";
  int real, imag;
  std::cin >> real >> imag;
  #ifndef FUNCTION_STYLE_CAST
    Complex number(real, imag);
    return number;
  #else
    return Complex(real, imag);
  #endif
}
```

Then you might call the function like this:

```
int main()
{
   Complex
complexNumber(getComplexNumber());
   // ...
   return 0;
}
```

Without including the #define FUNCTION_STYLE_CAST statement, a named object on the stack will be created. In this case CBuilderX produces (excluding the prompt):

```
Default/converting constructor
Copy constructor
Destructor
Copy constructor
Destructor
Destructor
```

The creation of number causes the first call to the default/converting constructor. Next, when number is returned from the function by value, a copy must be made, so the copy constructor gets called. Then the destructor for number is called. Back in main(), since complexNumber is being created from an existing Complex object, the copy constructor is called again. Finally, the destructor for the temporary object and for complexNumber is called.

Note, however, that .NET and Comeau both produce:

```
Default/converting constructor
Copy constructor
Destructor
Destructor
```

These compilers thus optimized away one call to the copy constructor.

After reinstating the #define statement, CBuilderX and Comeau produce:

```
Default/converting constructor
Copy constructor
Destructor
Destructor
```

The function-style cast must still invoke the default/converting constructor, but the call to the copy constructor for this temporary unnamed object has been optimized out of existence by the compilers before they invoke the copy constructor to create complexNumber. Therefore, two function calls have been eliminated. As for .NET, it now produces:

```
Default/converting constructor
Destructor
```

This is really good news since the compiler has optimized away both calls to the copy constructor, leaving just one constructor and one destructor call. You cannot do much better than that.

It is interesting to observe what happens when the main() function is changed to use C-style initialization of complexNumber instead of C++ style. That is:

```
int main()
{
  Complex complexNumber = getComplexNumber();
   // ...
   return 0;
}
```

The C++ Standard deems C++-style initialization as possibly being more efficient in terms of code generation. Without using the function-style cast, CBuilderX and .NET now produce:

```
Default/converting constructor
Copy constructor
Destructor
Destructor
```

and Comeau yields:

```
Default/converting constructor
Destructor
```

With the function-style cast, all three compilers produce:

```
Default/converting constructor
Destructor
```

It should thus be clear that the use of a function-style cast will, in most cases, produce more efficient code. But you should also be observant of the style of your instantiations, i.e., C-style using the '=' sign, and C++-style using parentheses. Careful experimentation similar to what was done here will show what options work best in your environment.

You can observe similar results whenever an exception needs to be thrown. Where possible, always use a function-style cast. For example, if you want to throw an exception of type std::string, then avoid the creation of a named object:

```
// Poor style
std::string exception("Out of range");
throw exception;

// Better style
throw std::string("Out of range");
```

In this manner you give the compiler a much better chance of optimizing away the call to the class's copy constructor. The conclusion should be obvious: the use of a function-style cast, where appropriate, can be a more efficient implementation than creating a named object.

NEW-STYLE CASTS

The four new-style casts in C++ (all of which are operators) constitute a replacement for the inherently unsafe and obscure C-style cast. Furthermore, the new-cast operators are a subset of the casts allowed by the old-cast style. When you use them, you automatically make your code more type-safe and easier to read. You can also search for them much easier using a text editor and searching for the expression "_cast" (which identifies a new-style cast), whereas searching for the old-style cast requires some syntactic analysis because the operator (open/close parentheses) is used in many other contexts, e.g., grouping arithmetic expressions, function declarations, and function calls. Additionally, you will make your intention very explicit to someone reading your code.

The generic format of all of the new-style cast operators is:

```
type-of-cast<destination-type>(expression-to-be-cast)
```

where the type-of-cast is one of the following C++ keywords (and a new operator):

- *static_cast*
- *reinterpret_cast*
- *const_cast*
- *dynamic_cast*

Note the angle brackets surrounding the destination-type and the mandatory parentheses around *expression-to-be-cast*. One thing is for sure – that is a cast! Let us look at each one in detail.

static_cast

The *static_cast* operator is used to perform a conversion that requests an "equivalent" value

in a different representation. Since no run-time check is performed in this type of casting, the safety of the conversion is not ensured. For example, you would use a *static_cast* to find the "equivalent" type *int* value of a *double* value. Any implicit type conversion including standard conversions and user-defined conversions, that the compiler would normally do can be written using a *static_cast*. While it is possible to use a function-style cast in some situations, and achieve the correct result, a *static_cast* is the preferred style. Also note that using a *static_cast* eliminates any warning message that the compiler might generate, e.g., when significance would be lost. The *static_cast* operator cannot cast away the *const* or *volatile* attributes. Instead, you must use the *const_cast* operator for that purpose, discussed later in this paper. Additionally, *static_cast* will never silently reinterpret bits as a different type. For that you would use the *reinterpet_cast* operator, discussed next.

Here is an example of a *static_cast*:

```
double d = 4.56;
int x = d;   // OK; may produce a warning message
int y = static_cast<int>(d); // Never produces a
                             // warning
```

Suppose that you are doing integral arithmetic with type *int*. If you fear the possibility of overflow, it is incumbent upon you to use greater precision, e.g., arithmetic involving a long value. Thus, to ensure that long arithmetic is being used, you can cast one of your values to type *long* using a *static_cast*:

```
long add(int x, int y)
{
   return static_cast<long>(x) + y;
}
```

If you are doing a division using integral types, and you want a fractional quotient, then one of the operands must be converted into a floating-point type using a *static_cast*:

```
double divide(int x, int y)
{
   return static_cast<double>(x) / y;
}
```

Note that in both of these functions, writing the return statements as

```
   return static_cast<long>(x + y);
and
   return static_cast<double>(x / y);
```

respectively, even though they are syntactically correct, logically are still wrong. The computations (addition and division) are being performed before the cast is made, and this is wrong. The authors have seen such utilization by students and developers quite often. The range of values used for x and y many times contribute to misunderstanding of the reason for casting. Whereas

```
   return static_cast<long>(x) +
static_cast<long>(y);
 and
   return static_cast<double>(x) /
static_cast<double>(y);
```

are perfectly valid, even though an extra cast is being made in both cases. These are some of the areas that the teachers and professional trainers need to emphasize the most.

While a *static_cast* can be used where an implicit type conversion would normally suffice, it must be used when the opposite direction is traversed, even if a cast is required to make the conversion. A perfect example occurs when using enumerated types; going from type *enum* into some other type is an example of implicit type conversion, and a cast is optional. However, going from some other type into type *enum* is considered very unsafe, and a cast is required.

```
enum color {red, white, blue};
void someFunction(color c)
{
   int x = c;                              // Implicit type
                                           // conversion
   int y = static_cast<int>(c);        // Cast optional
// If the int value is not in the range of enumeration
//  values, the result in c is undefined
   c = static_cast<color>(9);          // Cast required
  c = static_cast<color>('A');        // Cast required
// The following is an error in .NET
 c = static_cast<color>(54.631);    // Cast required
```

To avoid the .NET compilation error, you must convert the *double* to an *int*, then the *int* to a color, by writing:

```
c = static_cast<color>(static_cast<int>
(54.631));
```

Another good example would be the conversion to and from a *void** or *void const** type:

```
char const *ptrChar = "Some string";
// Cast optional
void const *ptrVoid = static_cast<void const
*>(ptrChar);
// Cast required
double const *ptrDouble = static_cast<double
const *>(ptrVoid);
```

You can use *static_cast* to cast *down* a hierarchy (from a base to a derived pointer or reference), but since the conversion is not checked; the result might not be useable.

```
class Base{};
class Derived : public Base{};
void test_static_cast1(Base *bp);
int main()
{
   Base *ptrBase = new Base;
   test_static_cast1(ptrBase);
   delete ptrBase;
   return 0;
}
void test_static_cast1(Base *bp)
{
   Base *ptrB = bp;
   Derived *ptrD = static_cast<Derived*>
(bp);   // Explicit conversion
 }
```

Finally, a *static_cast* cannot be used to cast down from a virtual base class.

reinterpret_cast

The reinterpret_cast operator is resolved at compile time and is used to cast a value so that the bits, while not changing, take on a totally different meaning. Without a cast this conversion is always a compile time error because, in essence, it is an unsafe cast. For example, the conversion of type *int** to a *char** is something that will never be done implicitly by the compiler, and therefore requires a

reinterpret_cast.

```
char const *ptrChar = "C++";
   // Error
int const *ptrInt = ptrChar;
   // OK
int const *ptrInt = reinterpret_cast<int const
*>(ptrChar);
```

Use the *reinterpret_cast* to convert an integral argument to a pointer type and convert a pointer type explicitly to an integral type large enough to contain it. Although not an introductory topic, nonetheless, a pointer to function can be converted explicitly to a pointer to an object type.

```
void test_reinterpret_cast_1(void *v);
int main()
{
   // Integral type to a pointer type using cast
   test_reinterpret_cast_1(reinterpret_cast<voi
d *>(20));

   // Pointer to function of one type cast to
   // Pointer to function of another type.
   typedef void (* ptrF)();
   ptrF pf =
reinterpret_cast<ptrF>(test_reinterpret_cast_1);
   pf();
```

```
    return 0;
  }
  void test_reinterpret_cast_1(void *v)
  {
    // Pointer type to integral type using cast.
    int i = reinterpret_cast<int>(v);
  }
```

Additional examples that show some bizarre consequences of using *reinterpret_cast* are reported[11]. The *reinterpret_cast* is considered the most dangerous of all the new-cast operators. For all practical purposes, you will almost never have occasion to perform a *reinterpret_cast*. Furthermore, the result of the *reinterpret_cast* is implementation dependent and for all likelihood not portable, you should use it only when absolutely necessary.

const_cast

The *const_cast* operator is how C++ explicitly casts away the const-ness or volatile-ness of an object. It can also be used to add const-ness or volatile-ness to an object, even though this is an implicit type conversion and a cast is not needed. As a matter of fact, C++ even allows the use of a *static_cast* or *reinterpret_cast* to do this. When using a *const_cast*, the destination-type and expression-to-be-cast must be the same type except for *const* and *volatile* modifiers.

```
  int x = 0;
  int *ptr1 = &x;
    // Cast optional
  int const *ptr2 = const_cast<int const *>(ptr1);
    // Cast mandatory
  ptr1 = const_cast<int *>(ptr2);
```

The beauty of the new-style casts is that the compiler will not let you use the wrong one in any particular situation:

```
  // Error on both casts
  // A static_cast is needed here:
  double const value = 74.39;
  int y = reinterpret_cast<int>(value);
  int y = const_cast<int>(value);

  // Error on both casts
  // A reinterpret_cast is needed here:
  char const *ptrChar = "C++";
  int const *ptrInt = static_cast<int const
*>(ptrChar);
  int const *ptrInt = const_cast<int const
*>(ptrChar);

  // Error on both casts
  // A const_cast is needed here
  int x = 0;
  int const *ptr1 = &x;
  int *ptr2 = static_cast<int *>(ptr1);
  int *ptr2 = reinterpret_cast<int *>(ptr1);
```

Sometimes you are faced with a function that is declared with a non-const formal argument

and you need to use the function for your *const* actual argument. You have basically two choices: using the *const_cast* to remove the const-ness form your actual argument before invoking this function, or introduce an overloaded function which does exactly the same operation as the other function and is defined with a *const* formal argument.

```
  void show(int *i);
  int main()
  {
    const int value = 10;
    // const_cast needed
    show(const_cast<int*>(&value));
    return 0;
  }
  #include <iostream>
  void show(int *i)
  {
    std::cout << "value = " << *i << std::endl;
  }
```

```
  void show(int *i);
  void show(const int *i); // Overloaded
function
  int main()
  {
     const int value = 10;
     // const_cast not needed
     show(&value);                return 0;
  }
  #include <iostream>
  void show(int *i)
  {
     std::cout << "value = " << *i << std::endl;
  }
  void show(const int *i)
  {
     std::cout << "value = " << *i << std::endl;
  }
```

Sometimes you have a class that declares a member function that is *const* and provides a *const_cast* to allow for changes in the data member.

```
  class A
  {
     public:
          A(int i = 0);
          int g() const;
     private:
          int value;
  };
  #include <iostream>
  int main()
  {
     A a;
     std::cout << a.g() << std::endl;
     return 0;
  }
  A :: A(int i) : value(i){}
  int A :: g() const
  {
     return ++*(const_cast<int*>(&value));
     // or...
    A ref = *const_cast<A* const>(this);
    return ++ref.value;
  }
```

Obviously, the class is poorly designed. You can improve the class design by declaring the data member as *mutable* (a C++ keyword) and change the code in the member function by eliminating the need for a *const_cast*. A *mutable* non-static data member is one that can legally be modified by a constant member function.

```
  class A
  {
     public:
          A(int i = 0);
          int g() const;
     private:
          mutable int value;
  };
  #include <iostream>
  int main()
  {
     A a;
     std::cout << a.g() << std::endl;
     return 0;
  }
  A :: A(int i) : value(i){}
  int A :: g() const
  {
     return ++value;
  }
```

Using the *mutable* in the declaration results in a much better and cleaner solution.

dynamic_cast

The *dynamic_cast* operator is used to navigate safely down a class hierarchy. It is part of the runtime type information (RTTI) mechanism by which the type of an object can be determined at execution time as opposed to compilation time. The *dynamic_cast* operator is by far the most difficult one to understand, to use properly, and know when it is needed. You have to make sure that the RTTI switch in the compiler environment is enabled prior to any compilation of your code.

To see the need for a *dynamic_cast*, suppose you have a pointer of some base class type. It

may, of course, be pointing to either a base class object or a derived class object:

```
Base *ptr;
// ...
ptr = new Base;        // OK
delete ptr;
ptr = new Derived;   // OK; same pointer used
delete ptr;
```

If you wish to invoke a certain member function with this pointer, and this function has been declared virtual in the base class, then there is absolutely no problem since polymorphism will automatically take effect. This process is referred to upcast and it is an implicit conversion where the compiler performs the static type-check to ensure the validity of the conversion.

```
class Base
{
   public:
        virtual void func();
};
class Derived : public Base
{
   public:
        virtual void func();
};
void test(Base *ptr)
{
   ptr->func();          // Call Base::func or
                         // Derived::func
}
```

But suppose that this function is first declared in the derived class so that the base class knows nothing about it. In this case, the compiler does not allow you to call the function when using a base class pointer:

```
class Base{};
class Derived : public Base
{
   public:
        void func();
};
void test(Base *ptr)
{
   ptr -> func();        // Compiler error; no
                         // Base::func
}
```

Instead, you must use a pointer of the specific derived class type in order to invoke the function:

```
class Base{};
class Derived : public Base
{
   public:
        void func();
};
void test(Derived *ptr)
{
   ptr -> func();        // OK
}
```

But if all you have to work with is a base class pointer, then what are you going to do? That is, how do you know if this pointer does, in fact, point to an instance of the base class or the derived class? If it is the derived class, then you can safely cast the base class pointer to a derived class pointer (called a downcast). To answer this critical question, you need to use the *dynamic_cast* operator. It works like this: If a base class pointer is used in the downcast, and the pointer does indeed point to a derived class object, the resultant derived class pointer will be non-zero. If the base class pointer does not point to a derived class object, then the resultant derived class pointer will be zero.

The same principle applies when a base class reference is used. In this case, if the reference does indeed refer to a derived class object, then the resultant derived class reference is created successfully. However, if the reference does not refer to a derived class object, it is impossible to create a “null” reference, so an exception of type std::bad_cast (declared in the header file typeinfo) will be thrown.

Note that because the *dynamic_cast* mechanism uses the compiler’s table of virtual functions, the base class must contain at least one virtual function.

Here is a simple example to illustrate its use:

```
class Base
{
   public:
       virtual ~Base() { }
};
class Derived : public Base{};

#include <iostream>
#include <typeinfo>
using std::cout;

void func(Base *ptrBase, Base &refBase)
{
   Derived *ptrDerived =
dynamic_cast<Derived *>(ptrBase);
   if(ptrDerived)
       cout << "Pointing to a Derived\n";
   else
       cout << "Pointing to a Base\n";
   try
   {
       Derived &refDerived =

   dynamic_cast<Derived &>(refBase);
       cout << "Referring to a Derived\n";
   }
   catch(std::bad_cast const &)
   {
       cout << "Referring to a Base\n";
   }
}
int main()
{
   Base *ptrBase1 = new Base;
   Base &refBase1 = *ptrBase1;
   func(ptrBase1, refBase1);
   delete ptrBase1;

   Base *ptrBase2 = new Derived;
   Base &refBase2 = *ptrBase2;
   func(ptrBase2, refBase2);
   delete ptrBase2;
   return 0;
}
```

All three compilers produced the following output:

```
Pointing to a Base
Referring to a Base
Pointing to a Derived
Referring to a Derived
```

Use of the *dynamic_cast* has a small run-time overhead and you should use it when it is absolutely essential. For non-polymorphic classes you can use the *static_cast* to perform the conversions between classes. Just be careful when performing a downcast; otherwise, you might inadvertently create a derived class pointer that points to a base class instance, which can lead to disaster.

SUMMARY AND CONCLUSIONS

This paper presented the conversions and casts in C and C++. The authors defined and provided clear distinctions between the available styles of casting. The examples shown demonstrated that using the different casting mechanisms under different compliant compilers can result in different behaviors. Good programming practices were spread throughout the paper. It is up to the educators and professional trainers to incorporate the authors’ suggestions. Some of these rules and guidelines are trivial, but the authors’ experience show that even most seasoned programmers do not follow them. Most of the topics covered in this paper can be presented in introductory courses in C++. The authors have avoided the use of the casting in

more complicated cases such as cross-casts and some other slick solutions that exist when dealing with the concept of casts. Since there are other good compliant C++ compilers, and if you are using a compiler other than the three compilers used here, check the examples provided here and their results against the results from your compiler, and note any differences. It is quite understandable that when a language is new, there may be several differences in the implementations of that language. By now, all C++ compliant compilers should behave, for most part, as described in this paper. For comparison purposes with other compliers, we suggest that you turn off any language extensions that a particular compiler may have provided. This not only eliminates many of the surprises when you do plan to port your code to other implementation, but it also improves the readability of your code for those who are familiar with the language's formal definitions.

Whether you use implicit or explicit conversions, you still bear a great responsibility for the accuracy of the results. Implicit conversions pretty much follow the language specification without any intervention from you. They have been, and continue to be, a source of confusion for many programmers, and this is yet another reason for introducing conversions and casts early in any C++ programming course.

REFERENCES

1. Bjarne Stroustrup, New Casts Revisited, AT&T Bell Laboratories Murray Hill, New Jersey, 07974 http://anubis.dkung.dk/jtc1/sc22/wg21/docs/papers/1993/N0349a.pdf

2. Sean Batten, Casting in C++, C/C++ Users Journal, April 1997.

3. Chuck Allison, Coding Capsule: Conversions and Casts, C/C++ Users Journal, September 1994.

4. Bobby Schmidt, Learning C/C++ Curve: Controlling Silent Conversions, C/C++ Users Journal, April 1996.

5. Pete Becker, The Journeyman's Shop: Casts and Conversions, C/C++ Users Journal, April 2000.

6. Steve Dewhurst, Common Knowledge: A Question of Respect, C/C++ Users Journal, April 2001.

7. Farrokh Attarzadeh, Cast Operators in C++, Technical Presentation at Intertrust Technologies, Santa Clara, CA, January 2001.

8. New Cast Operators, http://www.informit.com/guide/printerFriendly.sap?guide id={F26EC407-2D4B-4910-BF997CAAA8C44C06} &element id={32771716-7F84-4235-BAA0-B9FDD3431806}.

9. Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994.

10. Bjarne Stroustrup, C++ Programming Language, Third Edition, Addison-Wesley, 1997.

11. Stanley B Lippman & Josee Lajoie, C++ Primer, Third Edition, Addison-Wesley, 1998.

12. Scott Meyers, Effective C++, Second Edition, Addison-Wesley, 1998.

13. Scott Meyers, More Effective C++, Addison-Wesley, 1996.

14. Mats Henricson and Erik Nyquist, Industrial Strength C++: Rules and Recommendations, Prentice- Hall, 1997.

15. Eric Nagler, Learning C++, A Hands-On Approach, Third Edition, International Thomson Publishing, 2004.

16. C++ standard, ISO/IEC 14882 Programming Languages - C++

BIOGRAPHICAL INFORMATION

Farrokh Attarzadeh currently serves as an Associate Professor of Computer Engineering Technology at the University of Houston. He obtained his B.S. in Electronics Engineering from the California State Polytechnic University, Pomona, the M.S.E.E. from the California State University, Long Beach, and the Ph.D. in Electrical Engineering from the University of Houston. His research interests include software design and code efficiency, programming languages, microprocessor-based control, web-based usability, and electromechanical folk arts. He is a member of ASEE. You can contact him at Fattarzadeh@uh.edu.

Eric Nagler received his B.A. in mathematics from the University of Michigan and immediately went to work for the federal government as a computer programmer and systems analyst. He has been working in the data processing field ever since. After coming to California in 1972, he first started teaching computer languages in 1980 at a local community college, and taught his first C++ course in 1990. In addition to his community college experience, Eric has taught C, C++ and Java for the University of California Santa Cruz Extension and at numerous companies in the San Jose, California Bay Area and around the country. Moreover, he has spoken at several C++ conferences on technical issues. In 2002 he moved back to his home state of Michigan and is currently on the staff of Lawrence Technological University. He is available for on-site training, and can be reached at epn@eric-nagler.com.

ASEE MEMBERS

How To Join Computers in Education Division

(CoED)

1) **Check ASEE annual dues statement for CoED** Membership and add $7.00 to ASEE dues **payment.**

2) **Complete this form and send to American Society for Engineering Education, 1818 N. Street, N.W., Suite 600, Washington, DC 20036.**

I wish to join CoED. Enclosed is my check for $7.00 for annual membership (make check payable to ASEE).

PLEASE PRINT

NAME: ____________________

MAILING ADDRESS: ____________________

CITY: ____________________

STATE: ____________________

ZIP CODE: ____________________