Casting

In C/C++, casting serves several different purposes. The designers of C++ realized that and decided that there should be four kinds of cast operators.

- **static cast**

  Conventional casting of one type to another, where there is some meaning to what the cast will do. (Thus, normally you can't cast from a **Foo** to a **Bar**, unless there's a definition for how to do this).

- **dynamic cast**

  Meant for downcasting from a base class to a derived class. This check is done at runtime (as opposed to static casting which can generate the necessary code to do the casting at compile time).

- **const cast**

  To temporarily remove constness from a const variable. Mostly used to get rid of warnings about violating constness.

- **reinterpret cast**

  To allow you to cast one pointer type to another pointer type to another.

We're only interested in **static_cast** and **reinterpret_cast**.

More about `static_cast`

The main kind of static casting we're interested in falls into two categories:

- Casting from one primitive type to another (e.g., **char** to **int**, **float** to **int**).
- Casting from a shorter type to a longer type or vice versa (e.g., **short** to **int**, **long** to **int**).

Casting from one type to another is interesting, particularly, casting **float** to **int**. Like the unary minus, the casting operation does NOT change the value of the variable it is casting. Instead, like unary minus, it creates a temporary value which is the casted result.

For example, consider

```
  float f = 3.0 ;
  int val = static_cast<int>( f ) ;
```

The static cast of **f** to an int does not change the value of **f**. Instead, it creates a temporary **int**.

Casting **float** to **int** causes bits to change (from the original value to the temporary casted value). Recall that **float** are represented in IEEE 754 single precision while **int** is 2C. Those are different representations.

The semantics of casting **float** to **int** causes the value to get truncated (thus, 3.9 is truncated to 3.0), but it also changes representation.

The semantics of casting **int** to **value** causes the value to convert to float (thus, 3 is converted to 3.0), which also changes representation.

We look at this kind of casting, precisely because it makes you think about the internal data representation. 3 and 3.0 look very close when you write it in a program. However, internally, (once stored in a running program) they're represented quite differently.

Casting different sizes

Sometimes you need to cast from a shorter type to a longer type or vice versa. For example, you may need to cast from **short int** to an **int**. These *may* be the same sizes, depending on the compiler, but for now, let's assume they are different sizes.

If you go from a smaller size to a larger size, at least for an **int**, it sign extends. It zero-extends if it's an **unsigned int**. This preserves the value as you increase the number of bytes.

If you go from a larger size to a smaller size (in both **int** and **unsigned int**) it chops off the uppermost bytes. This can cause the number to change values (since those uppermost bytes might be useful for the value).

Even though the representation isn't as dramatic as **float** to **int**, it still occurs.

More about `reinterpret_cast`

In C, it turns out to be useful to convert from one pointer type to another, especially if you're working on systems programming (operating systems, for example).

Since all pointers are addresses, and all addresses have the same number of bytes (at least, on a given ISA), the casting does not convert any bits.

C++ decided to call this pointer-to-pointer casting, **reinterpret_cast**. What's being reinterpreted is the meaning of the bytes.

Let's look at an example:

```
   Foo * fooPtr = new Foo( 2, 3 ) ;  // create a Foo pointer
   Bar * barPtr = reinterpret_cast<Bar *>( fooPtr ) ;
```

We had a **Foo *** pointer, which is some address in memory, that stores some number of bytes of a **Foo** object.

When reinterpeting that pointer to a **Bar *** pointer, those bytes of the **Foo** object are *unchanged*.

If you dereference **barPtr**, the runtime system will consider the bytes stored at the addreess of **barPtr** (which is the same as the address of **fooPtr**) to be a **Bar** object.

If you're thinking "there's no way that those bytes, which are currently meaningful for a **Foo** object, could also be meaningful for a **Bar** object", then you're right. More likely than not, **barPtr** points to a **Bar** object with garbage bytes.

Worse, if **Bar** is sufficiently larger than **Foo**, then you might cause a core dump when accessing data members of **Bar** (because they may fall outside the valid rance of memory used for the **Foo** object).

You may think that it's completely useless to reinterpret cast. For the most part, it is useless. However, the most common use of reinterpret cast is to look at the individual bytes of some structure.

For example, suppose you're interested in finding out what the individual bytes of a **long int** look like (you can pick any type instead of **long int**).

```
   long val = 0x01234567cafebabe ;
   char * charPtr = reinterpret_cast<char *>( & val ) ;
   for ( int i = 0 ; i < sizeof( long ) ; i++ )
     {
       cout << byteToHex( *charPtr ) << endl ;
       charPtr ++ ;
     }
```

This puts a pointer at the beginning of **val** and prints out the bytes of **val** in hex format (assume **byteToHex()** returns a string that represents the hex value of the **char** passed as argument).

Why would you do this? Functions like **read()** and **write()** (which are part of the **istream** and **ostream** methods), often take a **char *** pointer, which is used as an address of any object/data type, and the prints out or reads in the contents of some object byte by byte. Again, it's useful to think of **char** as a byte type, in this case, instead of thinking of it as a character, just like it's useful to think of **char *** as an address in memory of some object, rather than an address of a character.

Accessing the individual bytes can be useful when sending structures across a network, or when saving a structure to a file.

The only problem with this sending objects or structures in this is potential incompatibility with the host machine (which sends the data) and the remote machine (which receives the data). If you send out an arbitrary object byte-by-byte over the network, the remote machine that receives this data must represent the object in exactly the same way that the host machine represents it (the same endianness, same order of data members, etc).

If the remote machine has a different representation, it may be worth encoding the structure of the object using some convention both sides agree on (say, XML). This will, more than likely, take more space and more processing, but should be more compatible for a wider variety of machines. In any case, coming up with a representation makes you think of a representation rather than assuming any object sent over the network will automatically be put together properly on the other end.

Another reason for using reinterpret cast is to allow you to edit individual bytes in memory by using bitwise/bitshift operators. This may seem like a very odd reason, but if you're writing linkers or loaders, it may be necessary to edit the instructions (this is called **binary rewriting**) as you link or load.

Converting float to int while preserving bits

Casting from **float** to **int** or vice-versa causes the representation of the casted result to change from the original representation.

Suppose a **float** and **int** both have 4 bytes. You've written a number in **float** but want to know its corresponding **int** value. That is, in both cases, the bits are exactly the same, but it has one meaning, if you think of those bits as a **float** and another meaning if you think of those bits as an **int**.

How do you convert between the two types without changing the bits (which occurs when you static cast)? The answer? Reinterpret cast. Of course, reinterpret cast works with pointers, so we'll need pointers.

The code is amazingly short:

```
   float val = 3.0 ;
   int num = * reinterpret_cast<int *>( & val ) ;
```

In this code, we create a pointer to **val** using the & operator. This is reinterpreted as a **int *** pointer. Recall that reinterpreting pointers does not change the bytes in memory.

Then, you dereference it to get the **int**.

Of course, this code assumes that **float** and **int** have the same number of bytes. On most machines/compilers, this is true, but it's possible that the sizes may be different. Languages like Java avoid this problem by mandating the sizes of various types. Languages like C/C++ leave the number of bytes of each type up to the compiler writer. The assumption is, for C/C++, that the compiler writer will pick the size that's the most "natural" for the CPU it runs on (thus, on old PCs, **int** was 2 bytes, and on recent PCs, they are 4 bytes).

Summary

The reason we look at static casting is to note that casting often causes a change of representation. Either the representation can completely change (say, from 2C to IEEE 754), or it changes in the number of bytes (say, from int to long). The bits change when casting is performed.

On the other hand, when you reinterpret cast (which is casting from one pointer type to another) the bits do not change. That's because all pointers on a given machine have the same number of bytes (usually 4 or 8 bytes for a 32 bit and 64 bit machine, respectively). Reinterpret casting preserves the address.

The difference is when you dereference the pointer. Suppose the new pointer type requires N bytes of memory. The runtime system will use the N bytes at the address, and consider those bytes as the object. Usually, this will cause you to have a nonsense object.

Thus, the most common use of reinterpret casting is to cast to a **char *** where you can then look at the individual bytes of an object (since **char** holds one byte), possibly so you can save it to a file or send the bytes over a network. Occasionally, you can edit bytes by casting to a **char *** (and dereferencing).