1 Introduction

In the seminars we have discussed *implicit type casting* which is a procedure for the compiler to cast data types automatically to other types without the intervention or approval of the programmer. A simple example of implicit type casting is this:

```
int foo(int x)
{
  return x;
}

int main()
{
  return foo(3.5);
}
```

Where we call `foo` with a `float` value when it expects an `int`. In this case the compiler will silently convert `3.5` into an `int` (probably the value `3`).

But sometimes this doesn't work the way we want. Take for example:

```
double f = 5 / 2;
```

Here `f` gets the value `2` instead of the expected `2.5`. This is because `5 / 2` is an operation between two integers so the program will simply perform this division in the domain of integers. Once the operation is done, the resulting value will be copied into `f` which will trigger an *integer-to-floating* conversion.

In order to get the expected result we must perform a floating-point division instead. This can easily be solved by converting the values to floating-point numbers instead:

```
double f = 5.0 / 2.0;
```

It is actually enough to just make one of the values a floating-point number, for example like this:

```
double f = 5.0 / 2;
```

Because then the compiler will have to convert `2` into a floating-point number before the division. Note, it will never convert a floating-point number to an integer during arithmetic operations, because floating-point numbers have a higher conversion rank than integers.

Now consider this case:

```
int x{5};
int y{2};

double f = x / y;
```

This will result in the same problem above. Sure, we can solve it by changing the type of either `x` or `y`, but that is not always the best solution (they are probably declared as integers for a reason).

This is a case where *explicit type conversion* comes into play. In C++ there are many different ways of performing conversions, and in this document we will outline all the ways (except `dynamic_cast` which we will talk about later), and the difference between them.

2 static_cast

The most common way to convert between two types is by using `static_cast`. The syntax looks like this:

```
T t{};
U u = static_cast<U>(t);
```

Where `T` and `U` are two data types.

It works like this:

1. If there is a way for the compiler to use the implicit conversion rules to convert `t` into an object of type `U` then it will do so.
2. If `T` has implemented the operator `T::operator U()` then use the result of that operator.
3. If `U` has a constructor that takes one parameter of type `T` then use that constructor to create a new object of type `U` where `t` is passed in as the argument.
4. Otherwise it fails (producing a compile error).

This list is simplified, for a complete list check `cppreference.com`.

Solution to our problem:

```
int x{5};
int y{2};

double f = static_cast<double>(x) / y;
```

Other examples:

```cpp
// downcasting example

class A
{ };

class B : public A
{ };

int main()
{
  {
    B* b = new B{};
    A* a = static_cast<A*>(b); // pointer
    delete b;
  }

  {
    B b{};
    A& a = static_cast<A&>(b); // reference
  }

}
```

Here `B*` and `B&` can be converted to `A*` and `A&` since `B` inherits from `A`.

The following two examples demonstrate user-defined conversions:

```cpp
struct A
{ };

struct B
{
  B(A const& a) { }
};

int main()
{
  A a{};
  B b = static_cast<B>(a);
}
```

Here `static_cast` can convert `A` into `B` by calling the `B::B(A const&)` constructor.

```
struct A
{ };

struct B
{
  operator A() { return A{}; }
};

int main()
{
  B b{};
  A a = static_cast<A>(b);
}
```

Here `B` is converted to `A` by calling `B::operator A()` (the conversion operator).

3 const_cast

It is possible to add `const` to a type with `static_cast`. This works because the compiler is able to do the same implicitly. For example:

```
int x{};

// add const
int const& y = static_cast<int const>(x);
```

But it is not valid to remove the `const`. So this is invalid:

```
int& z = static_cast<int&>(y);
```

Why? Well it is because `const` is a safety feature in the language that guarantees that specific values does not change. Take for example:

```
int const x = 5; // this should never change

int& y = static_cast<int&>(x);

y = 7;

// here x would be 7 instead of 5, meaning it has changed
```

If we then were able to cast away the `const` then we it would be possible to modify `x` even though it is `const`. *Not good*. Therefore it is forbidden to remove `const` with `static_cast`.

But in some cases it does make sense to actually remove `const` (However it is mostly bad practice to do so). Specifically it is reasonable to remove `const` from a reference to an object that was originally not `const`. Example:

```
int x{}; // not const
int const& y{static_cast<int const&>(x)}; // add const
int& z{const_cast<int&>(y)}; // remove const again
```

There is one example were `const_cast` can be good practice. Consider the following class:

```
class Cls
{
public:

  int const& do_stuff() const
  {
    // a very long and complex function
  }

};
```

Now suppose we want to make a non-`const` version of `do_stuff` that returns a non-`const` reference. I.e. we want to add the function `int& do_stuff()`. The naive way to do this is to simply copy-and-paste the code from the `const` version. But this would lead to code duplication. *Not good*.

Instead we can use the already existing `const` version, like this:

```
int& Cls::do_stuff()
{
  Cls const* self = static_cast<Cls const*>(this);
  return const_cast<int&>(self->do_stuff());
}
```

Here we add `const` to the `this` pointer and then we call `do_stuff` from that pointer. This means that we call the `const` version of `do_stuff` even though `this` is actually non-`const`.

The problem is that the `const` version of `do_stuff` returns a `const` reference which we do not want.

But, since `this` is in reality a non-`const` object it is valid to remove the `const` from the return type.

Important distinction: It is only okay to do this in this direction. The other-way around is not okay. So the following example is **invalid**:

```
class Cls
{
public:
  int& do_stuff()
  {
    // a very long and complex function
  }

  int const& do_stuff() const
  {
    // Not allowed since 'this' is of type Cls const*
    Cls* self = const_cast<Cls*>(this);
    return static_cast<int const&>(self->do_stuff());
  }

};
```

Here we are removing `const` from something that was originally `const`, which is illegal because of the reasons mentioned previously. Worth mentioning is that the compiler has a very hard time checking whether or not something was `const` originally, so it is up to *you* to make sure that this case does not occur. Otherwise you might get some very strange behaviour from your program. *So tread lighly!*

4 reinterpret_cast

`reinterpret_cast` is used to:

— Convert one type of pointer to another type of pointer,

— Convert a pointer to an integral value,

— Convert an integral value to a pointer.

The big difference from `static_cast` is that none of the types need to have a relationship. Consider the following example:

```
float x{5.0};
float* x_ptr{&x};
int* y{reinterpret_cast<int*>(x_ptr)};
std::cout << *y << std::endl;
```

Which (on my machine) prints the value `1084227584` (What?!).

What is happening here is that we are converting a `float` pointer to an `int` pointer. So when we dereference the `int` pointer the program will go to the same memory address as the `float` and read that values *as-if* it was an `int`.

If we were to try the same thing with `static_cast` it would not compile, since this conversion is (in most cases) nonsense (and undefined behaviour).

Another example:

```
long long int x{0x7ffe67a792e4}
int* y = reinterpret_cast<int*>(x);
```

Here we are converting an integral value to a pointer, meaning we *reinterpret* the value as a memory address.

One can also use `reinterpret_cast` the other way around, i.e.:

```
int* x{new int{}};
long long y{reinterpret_cast<long long>(x)};
```

Here we are reinterpreting the address as an integral value.

One last example of `reinterpret_cast`:

```
struct A
{
  int const x{5};
};

struct B
{
  int const y{7};
};

int main()
{
  A* a{new A{}};
  B* b{new B{}};

  // will print 5
  std::cout << a->x << std::endl;

  // will print 7
  std::cout << b->y << std::endl;

  // will print 5
  std::cout << reinterpret_cast<B*>(a)->y << std::endl;

  // will print 7
  std::cout << reinterpret_cast<A*>(b)->x << std::endl;

}
```

`A` and `B` have no relationship, so using `static_cast` to cast pointers of one type to the other will result in an error (since no downcasting can occur).

But with `reinterpret_cast` we will simply change the pointers to the other type without any problems (but it will most likely result in strange and undefined behaviour).

Final notes; there are *extremely* few cases where `reinterpret_cast` is a good idea. In almost all cases it would likely lead to undefined behaviour. If you can't think of an example, don't worry about it. `reinterpret_cast` is mostly used for low-level programming. So as a general rule-of-thumb: avoid `reinterpret_cast` whenever possible.

5 C-style casts

There is another way of casting values, the so called *C-style casts*. They look like this:

```
int x{5};
float f = (float)x; // Syntax #1
double d = double(x); // Syntax #2
```

Syntax #1 is how you cast values in the programming language C. Syntax #2 is equivalent to syntax #1, but is a side-effect of how direct initialization works in C++; we are initializing a new `double` with the value of `x`.

In C++ there are better alternatives (the ones described above) than C-style casts. The reason for C-style casts being a bad idea is due to safety.

C-style casts will try the following conversions until one of the succeeds:

1. `const_cast`
2. `static_cast`
3. `static_cast` followed by a `const_cast`
4. `reinterpret_cast`
5. `reinterpret_cast` followed by a `const_cast`

Can you spot any problems here?

It will first try to use `const_cast` to remove `const` from a type. This means that:

```
int const x{};
int& y{(int&)x};
```

Is totally valid, which it really shouldn't be since this is probably not what we meant.

Another issue is that C-style casts can use `reinterpret_cast` to make it work. Can you think of any other weirdness that can occur due to this?

Another good reason for using C++-style casting is that one can easily search for places in the code where a conversion occurs. You can for example search for `static_cast` to find all places where such a cast occurs. This is not possible with C-style casts.

Rule-of-thumb: Never, *ever* use C-style casts. Communicate your intent directly with either `const_cast`, `static_cast` or `reinterpret_cast`, your peers will thank you!