Don't use C-style casts

Pieter P

What is a C-style cast?

A C-style cast is an explicit type conversion of the form `(type)expression` or `type(expression)`. For example:

```
int i = 42;
float f = (float)i; // C-style cast
char c = char(i);   // functional cast, equivalent to C-style cast
```

Why are C-style casts an issue?

C-style casts allow you to perform dangerous conversions, while suppressing any warnings or errors from the compiler, and with complete disregard for the C++ type system.
Additionally, they do not clearly show the programmer's intent, and are hard to search for in a code base.

Some concrete issues:

1. They allow you to perform dubious casts between integers and pointers:

```
int value = 42;
int *very_bad = (int *)value; // accidental int-to-pointer conversion

const char *message = "12345";
int bad             = (int)message; // accidental pointer-to-int conversion
int also_bad        = int(message); // idem
```

2. They allow you to cast away `const` and `volatile` qualifiers:

```
const char *message = "12345";
char *very_bad = (char *)message; // casts away const

volatile uint8_t buffer[8];
uint8_t *also_very_bad = (uint8_t *)buffer; // casts away volatile
```

3. They allow you to cast between pointers to unrelated types:

```
struct Pineapple { /* ... */ };
class Bulldozer { /* ... */ };

Pineapple p;
Bulldozer *b = (Bulldozer *)&p; // b points to a pineapple, not a bulldozer
```

All of these casts undermine the C++ type system and prevent the compiler from catching common bugs.

What to use instead

Consider one of the following safer alternatives to C-style casts.

No cast

Sometimes you don't need an explicit cast. Just let the type system do its thing.

In the case of literals, you can use a literal suffix to avoid a cast:

```
float bad  = (float)123;
float good = 123.f;
```

Use braces

When you do need an explicit cast to a specific type, for example to select a specific function overload, use curly braces instead of using parentheses. This creates a temporary of the given type. In cases where such a conversion is valid, the effect is the same as a C-style cast, but invalid or narrowing type conversions are rightly rejected:

```
const char *message = "12345";
int better          = int {message}; // compile-time error (as it should)
```

```
error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
    2 | int better          = int {message}; // compile-time error (as it should)
      |                             ^~~~~~~
      |                             |
      |                             const char*
```

Sensible conversions are allowed, for example, casting an integer to a wider integer, or explicitly converting an integer to milliseconds:

```
auto i = long {42}; // okay
auto ms = std::chrono::milliseconds {i}; // okay
```

Use named casts

When you need to force a narrowing conversion, use `static_cast<type>(expression)`.
The advantage of `static_cast` is that it disallows many questionable casts that would violate the rules of the type system, such as casting away qualifiers or converting between unrelated types;

```
const char *message = "12345";
char *little_better = static_cast<char *>(message); // compile-time error (as it should)
```

```
error: invalid ‘static_cast’ from type ‘const char*’ to type ‘char*’
    2 | char *little_better = static_cast<char *>(message); // compile-time error (as it should)
      |                       ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
Pineapple p;
Bulldozer *b = static_cast<Bulldozer *>(&p); // compile-time error (as it should)
```

```
error: invalid ‘static_cast’ from type ‘Pineapple*’ to type ‘Bulldozer*’
    5 | Bulldozer *b = static_cast<Bulldozer *>(&p); // compile-time error (as it should)
      |                ^~~~~~~~~~~~~~~~~~~~~~~~~~~
```

For safely casting polymorphic types, e.g. converting a pointer-to-base to a pointer-to-derived in an inheritance hierarchy, use `dynamic_cast`.

If you *really* need a more powerful (read: dangerous) cast, you might need a `reinterpret_cast`, e.g. to convert between integers and pointers or to convert pointers to objects to pointers to arrays of bytes.
If you need to cast away `const` or `volatile` qualifiers, you can use `const_cast`.

Both `reinterpret_cast` and `const_cast` come with huge caveats, making it very easy to shoot yourself in the foot and invoke Undefined Behavior. They should generally only be used in low-level code or when dealing with old C APIs, and demand good encapsulation and an even better justification. Keep in mind that `reinterpret_cast` cannot be used for type punning: Despite its name, you cannot use it to interpret a variable of one type as a different type (except in some very limited cases, see the cppreference link above for details).

C++ Core Guidelines

If you're unconvinced by these arguments, you might want to have a look at what the official C++ Core Guidelines have to say about casting:

- ES.48: Avoid casts
- ES.49: If you must use a cast, use a named cast (as opposed to a C-style cast)
- ES.50: Don't cast away `const`
- ES.64: Use the `T{e}` notation for construction
- Pro.safety: Type-safety profile