To lie...

... and hopefully, to lie usefully

Patrice Roy, Patrice.Roy@clg.qc.ca, Patrice.Roy@USherbrooke.ca

To lie…

- We lie. We all lie
- Hopefully, these lies are “little white lies”
  - For example, we might have prepared a birthday gift for someone and want to hide that fact to surprise a loved one…
  - … or we might want to preserve some magic about teeth being traded for money
- Sometimes, they are… not as nice, and might even bring some discomfort
  - Telling the truth tends to be the right thing to do, as a rule of thumb

To lie...

- When programming, we have a cute name for lies: **casts**, a short name for type casts
- Casts are those things we do when the compiler's view on the types in our program does not suit our needs
  - They are tools to lie to the compiler, purposefully
- So let's talk about casts
  - What they are, what they do...
  - ...why they have this peculiar syntax

To lie…

- When programming, we have a cute name for lies: **casts**, a short name for type casts
- Casts are those things we do when the compiler's view on the types in our program does not suit our needs
  - They are tools to **lie to the compiler**, purposefully
- So let's talk about casts
  - W
  - …

Yes, I know that on occasion, we're informing the compiler of some hidden truth it cannot see without our help. Oh well…

To lie...

- When programming, we have a cute name for lies: **casts**, a short name for type casts
- Casts are those things we do when the compiler's view on the types in our program does not suit our needs
  - They are tools to lie to the compiler, purposefully
- So let's talk about casts
  - What they are, what they do...
  - ...why they have this peculiar syntax

Note to self, and to others: when programming as when dealing with others, the less lies we tell the better

To lie…

- In this talk, we will look at the various casts that C++ provides, including the «C cast», to know what they do and how they can be used efficiently
- Then, we will see how one can write casts (yes, we can!) when these can clarify the intent in a program
- Finally, we will reflect on the reasons why programmers sometimes have to lie to their compiler.

To lie...

```
// warns (probably) as conversion is lossy
const float pi_might_warn = 3.14159;
```

To lie...

```
// warns (probably) as conversion is lossy
const float pi_might_warn = 3.14159;
// probably harmless
const float pi_lies_ish =
    static_cast<float>(3.14159);
```

To lie...

```
// warns (probably) as conversion is lossy
const float pi_might_warn = 3.14159;
// probably harmless
const float pi_lies_ish =
   static_cast<float>(3.14159);
// of course, telling the truth is better
const float pi_clearer = 3.14159f;
```

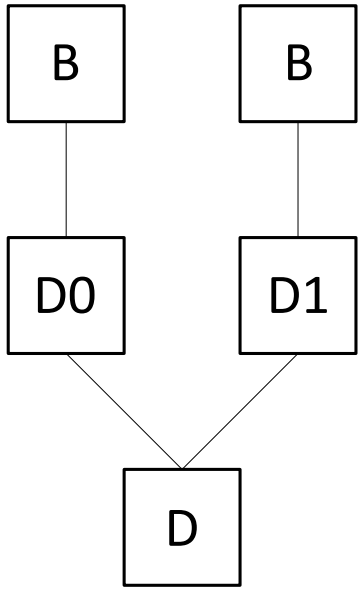
To lie...

```
struct B {};
struct D : B {};
int main() {
    D d;
    B & b0 = static_cast<B&>(d); // fine
    B & b1 = d; // fine too
}
```

To lie...

```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    B & b1 = static_cast<D0&>(d); // Ok
    B & b2 = static_cast<D1&>(d); // Ok, different B
}
```

To lie...




```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    B & b1 = static_cast<D0&>(d); // Ok
    B & b2 = static_cast<D1&>(d); // Ok, different B
}
```

To lie...

```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    B & b1 = static_cast<D0&>(d); // Ok
    B & b2 = static_cast<D1&>(d); // Ok, different B
}
```

Beginners sometimes confuse casts with mutating conversions, thinking casts change the nature of the « casted-from » object. They are not, obviously, as they leave the «casted-from» object as is

What a cast does is change the type of an expression from its « natural » type to a (potentially) different type. It influences how a compiler perceives that expression

To lie...

```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    B & b1 = static_cast<D0&>(d); // Ok
    B & b2 = static_cast<D1&>(d); // Ok, different B
}
```

Here, we have a case where treating a D& as a B& is ambiguous since there are two B base subojbects in a D object (one in its D0 base subobject, another in its D1 base subobject)

To lie...

```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    B & b1 = static_cast<D0&>(d); // Ok
    B & b2 = static_cast<D1&>(d); // Ok, different B
}
```

In this case, a cast lets us remove that ambiguity by choosing whether we want to use the B in the D0 base subobject or the one in the D1 base subobject

To lie...

```
struct B {};
struct D0 : B {};
struct D1 : B {};
struct D : D0, D1 {};
int main() {
    D d;
    // B & b = d; // nope (which B?)
    D0 &d0 = d;  // Ok
    B  &b0 = d0; // Ok
    D1 &d1 = d;  // Ok
    B  &b1 = d1; // Ok, different B
}
```

The cast we used in this case is morally equivalent to introducing a temporary reference to an intermediate base class subojbect, albeit an anonymous one

To lie…

- There are many reasons to lie when programming
- Different kinds of lies have different impacts, different effects, and different reasons for being
- Let us begin our examination of lies… er, casts in C++

The traditional casts

The traditional casts

- We will start our exploration with what might call the « traditional » casts and what each one means
  - static_cast
  - dynamic_cast
  - const_cast
  - reinterpret_cast
  - The « C cast » and the « function-style cast »

static_cast<T>(expr)

In an ideal world…

[expr.static.cast]

static_cast<T>(expr)

- For most programmers, particularly performance-oriented ones, static_cast is the cast of choice
  - When there are many options and static_cast is one of them, static_cast is *usually* (not always) the right choice
  - Because it’s our « go-to little lie », we will spend more time looking at this one than at the others

static_cast<T>(expr)

- If one needs a cast involving only fundamental types (int to float, double to short, long to int, etc.), static_cast is the tool for the job
  - In most cases, these conversions will work without a cast
  - … but the implicit behavior might warn if it is narrowing (might lose information)

static_cast<T>(expr)

```
int main() {
    double x = 3.14159;
    const int n0 = x; // might warn
    [[maybe_unused]]
    const int n1 = static_cast<int>(x);
    x = n0; // or n1, same thing
}
```

static_cast<T>(expr)

« Might » is important, as some compilers will not warn:
https://wandbox.org/permlink/ZyaPtlsOkJTS9Dmb

```
int main() {
    double x = 3.14159;
    const int n0 = x; // might warn
    [[maybe_unused]]
    const int n1 = static_cast<int>(x);
    x = n0; // or n1, same thing
}
```

static_cast<T>(expr)

```
int main() {
    double x = 3.14159;
    const int n0 = x; // might warn
    [[maybe_unused]]
    const int n1 = static_cast<int>(x);
    x = n0; // or n1, same thing
}
```

This one can be expected not to warn as the programmer (through a cast) took responsibility for the lossy conversion

static_cast<T>(expr)

```
int main() {
    double x = 3.14159;
    const int n0 = x; // might warn
    [[maybe_unused]]
    const int n1 = static_cast<int>(x);
    x = n0; // or n1, same thing
}
```

This conversion is not considered « lossy ». It's perhaps debatable, but that's how it is

static_cast<T>(expr)

https://wandbox.org/permlink/r8d4Ew0mjfBu5mH4

```
#include <print>
using std::println;
struct Noisy {
   Noisy() { println("Noisy()"); }
   Noisy(const Noisy&) { println("Noisy(const Noisy&)"); }
   Noisy(Noisy&&) { println("Noisy(Noisy&&)"); }
   // ~Noisy(), operator=(const Noisy&), operator=(Noisy&&)
};
void f(Noisy&) { println("f(Noisy&)"); }
void f(Noisy&&) { println("f(Noisy&&)"); }
int main() {
   Noisy n0;
   f(n0); // f(Noisy&)
   f(static_cast<Noisy&&>(n0)); // f(Noisy&&), no move occurred
   f(Noisy{ }); // f(Noisy&&)
}
```

static_cast<T>(expr)

https://wandbox.org/permlink/3hiUq8jsKhShO1Oj

```
#include <print>
using std::println;
struct Noisy {
   Noisy() { println("Noisy()"); }
   Noisy(const Noisy&) { println("Noisy(const Noisy&)"); }
   Noisy(Noisy&&) { println("Noisy(Noisy&&)"); }
   // ~Noisy(), operator=(const Noisy&), operator=(Noisy&&)
};
void f(Noisy) { println("f(Noisy)"); }
int main() {
   Noisy n0;
   f(n0); // f(Noisy), copy occurred
   f(static_cast<Noisy&&>(n0)); // f(Noisy), move occurred
}
```

static_cast<T>(expr)

```
struct B { /* ... */ };
struct D0 : B { /* ... */ }; // note: public inheritance
struct D1 : B { /* ... */ }; // note: public inheritance
int main() {
    D0 d0a;
    B & b = d0a; // Ok, static_cast would work too
    // D0 &d0b = b; // illegal: could be dangerous
    D0 & d0b = static_cast<D0&>(b); // Ok, if you say so
    D1 & d1 = static_cast<D1&>(b); // compiles but wrong
}
```

static_cast<T>(expr)

Unambiguous conversion from derived class to (accessible) base class does not require a cast. Some will still want a cast thinking it makes the intent clearer: in that case, static_cast is the right tool

```
struct B { /* ... */ };
struct D0 : B { /* ... */ }; // n
struct D1 : B { /* ... */ }; // note: public inheritance
int main() {
    D0 d0a;
    B & b = d0a; // Ok, static_cast would work too
    // D0 &d0b = b; // illegal: could be dangerous
    D0 & d0b = static_cast<D0&>(b); // Ok, if you say so
    D1 & d1 = static_cast<D1&>(b); // compiles but wrong
}
```

static_cast<T>(expr)

```
struct B { /* ... */ };
struct D0 : B { /* ... */ }; /
struct D1 : B { /* ... */ }; /
int main() {
    D0 d0a;
    B & b = d0a; // Ok, static_cast would work too
    // D0 &d0b = b; // illegal: could be dangerous
    D0 & d0b = static_cast<D0&>(b); // Ok, if you say so
    D1 & d1 = static_cast<D1&>(b); // compiles but wrong
}
```

Conversion from base to derived without a cast does not compile as this could be an erroneous conversion. Requiring a cast is appropriate: the programmer doing something potentially dangerous should take responsibility and make the intent clear in the source code

static_cast<T>(expr)

```
struct B { /* ... */ }
struct D0 : B { /* ...
struct D1 : B { /* ...
int main() {
    D0 d0a;
    B & b = d0a; // Ok, static_cast would work too
    // D0 &d0b = b; // illegal: could be dangerous
    D0 & d0b = static_cast<D0&>(b); // Ok, if you say so
    D1 & d1 = static_cast<D1&>(b); // compiles but wrong
}
```

With a cast, of course, conversion from base to derived works fine. In this case, static_cast is a tool that lets a programmer express « this cast is Ok and I take responsibility for it » and get the conversion essentially for free. In this case, the standard guarantees the second cast gives us a reference the same object as the original one:
https://wandbox.org/permlink/6Kt0Xm3e5BGAFxK4

static_cast<T>(expr)

```
struct B { /* ... */ };
struct D0 : B { /* ... */
struct D1 : B { /* ... */
int main() {
    D0 d0a;
    B & b = d0a; // Ok, static_cast would work too
    // D0 &d0b = b; // illegal: could be dangerous
    D0 & d0b = static_cast<D0&>(b); // Ok, if you say so
    D1 & d1 = static_cast<D1&>(b); // compiles but wrong
}
```

It goes without saying that if you tell the compiler « believe me, I know what I'm doing » and you make a mistake, well… it's your fault and there might be nasty consequences. If you use a static_cast to perform a « downcast », you better know what you're doing!

static_cast<T>(expr)

```
class B {};
class D0 : public B {}; // note: public inheritance
class D1 : B { // note: private inheritance
   friend B* f(D1*);
};
B* f(D1 *p) { return static_cast<B*>(p); }
int main() {
   D0 d0;
   B * b0 = static_cast<B*>(&d0);
   D1 d1;
   // B * b1 = static_cast<B*>(&d1); // no, not an accessible base class
   B *b2 = f(&d1);
}
```

static_cast<T>(expr)

```
class B {};
class D0 : public B {}; // note: public inheritance
class D1 : B { // note: private inheritance
    friend B* f(D1*);
};
B* f(D1 *p) { return static_cast<B*>(p); }
int main() {
    D0 d0;
    B * b0 = static_cast<B*>(&d0);
    D1 d1;
    // B * b1 = static_cast<B*>(&d1); // no, not an accessible base class
    B *b2 = f(&d1);
}
```

static_cast is respectful of access qualifiers. Here, since B is a private base class of D1, this conversion cannot be performed by main()

static_cast<T>(expr)

```
class B {};
class D0 : public B {}; // note: public inheritance
class D1 : B { // note: private inheritance
    friend B* f(D1*);
};
B* f(D1 *p) { return static_cast<B*>(p); }
int main() {
    D0 d0;
    B * b0 = static_cast<B*>(&d0);
    D1 d1;
    // B * b1 = static_cast<B*>(&d1); // no, not an accessible base class
    B *b2 = f(&d1);
}
```

Of course, the conversion from D1 to B can be performed by B (not shown here) as well as by others (like f()) who can access and use that inheritance relationship

static_cast<T>(expr)

- Conceptually, any pointer-to-some-object can be considered to have void* as « parent »
  - This statement supposes similar cv-qualifiers, e.g.: *const* T* can be seen as being a « specialization » of *const* void*
- Casting to and from void* can thus be performed with static_cast
  - Casting *to* void* is usually implicit
  - Casting *from* void* requires explicit action in C++ as it circumvents the protections of the type system

static_cast<T>(expr)

```
struct some_operating_system_thingy {
   // ...
};
int operating_system_call(void *);
int main() {
   some_operating_system_thingy thing { /* ... */ };
   operating_system_call(&thing); // Ok, implicit conversion
   operating_system_call(static_cast<void*>(&thing)); // Ok
   void *p= &thing; // fine
   // some_operating_system_thingy *q = p; // no, dangerous
   auto q= static_cast<some_operating_system_thingy *>(p); // Ok
}
```

static_cast<T>(expr)

```
struct some_operating_system_thingy {
    // ...
};
int operating_system_call(void *);
int main() {
    some_operating_system_thingy thing { /* ... */ };
    operating_system_call(&thing); // Ok, implicit conversion
    operating_system_call(static_cast<void*>(&thing)); // Ok
    void *p= &thing; // fine
    // some_operating_system_thingy *q = p; // no, dangerous
    auto q= static_cast<some_operating_system_thingy *>(p); // Ok
}
```

It's common for operating system calls to accept void* arguments since these functions seek to be language-agnostic (but require some binary layout compatibility, as they need to make sense of these arguments)

static_cast<T>(expr)

Casting from some T* to void*is implicitly fine (all pointers represent addresses, informally), but static_cast can be used to make the intent clearer

```
struct some_operating_system_thingy {
    // ...
};
int operating_system_call(void *);
int main() {
    some_operating_system_thingy thing { /* ... */ };
    operating_system_call(&thing); // Ok, implicit conversion
    operating_system_call(static_cast<void*>(&thing)); // Ok
    void *p= &thing; // fine
    // some_operating_system_thingy *q = p; // no, dangerous
    auto q= static_cast<some_operating_system_thingy *>(p); // Ok
}
```

static_cast<T>(expr)

```
struct some_operating_system_thingy {
   // ...
};
int operating_system_call(void *);
int main() {
   some_operating_system_thingy thing { /* ... */ };
   operating_system_call(&thing); // Ok, implicit conversion
   operating_system_call(static_cast<void*>(&thing)); // Ok
   void *p= &thing; // fine
   // some_operating_system_thingy *q = p; // no, dangerous
   auto q= static_cast<some_operating_system_thingy *>(p); // Ok
}
```

Casting from void* to some T* requires an explicit cast. This is a dangerous operation where compilers cannot help you (passing through void* induces a form of type erasure, obviously)

static_cast<T>(expr)

https://wandbox.org/permlink/V6mqS2GVV95nEklr

```
struct some_operating_system_thingy {
    // ...
};
int operating_system_call(void *);
int main() {
    some_operating_system_thingy thing { /* ... */ };
    operating_system_call(&thing); // Ok, implicit conversion
    operating_system_call(static_cast<void*>(&thing)); // Ok
    void *p= &thing; // fine
    // some_operating_system_thingy *q = p; // no, dangerous
    auto q= static_cast<some_operating_system_thingy *>(p); // Ok
}
```

static_cast<T>(expr)

- In cases such as casting to and from void*, reinterpret_cast works as well as static_cast
- However, static_cast can be used in a constexpr context, giving it a serious advantage
  - It is also sometimes possible to use reinterpret_cast in constexpr contexts, but there are more restrictions

static_cast<T>(expr)

https://godbolt.org/z/86e86Tc89

```
constexpr int compile_time_maybe(const void *p){
    return *static_cast<const int*>(p);
}
int main() {
    constexpr int n = 3;
    static_assert(compile_time_maybe(&n) == 3);
}
```

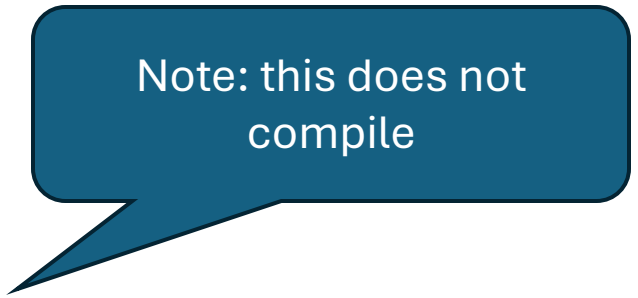
This example requires C++26

static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
   D d;
   B & b = reinterpret_cast<B&>(d);
   int &n = reinterpret_cast<int&>(b);
   return reinterpret_cast<void*>(&b) ==
          reinterpret_cast<void*>(&n);
}
int main() {
   static_assert(f());
}
```

static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
    D d;
    B & b = reinterpret_cast<B&>(d);
    int &n = reinterpret_cast<int&>(b);
    return reinterpret_cast<void*>(&b) ==
           reinterpret_cast<void*>(&n);
}
int main() {
    static_assert(f());
}
```



static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
    D d;
    B & b = reinterpret_cast<B&>(d);
    int &n = reinterpret_cast<int&>(b);
    return reinterpret_cast<void*>(&b) ==
           reinterpret_cast<void*>(&n);
}
int main() {
    static_assert(f());
}
```

Up until this point, compilers will let you get by with either static_cast or reinterpret_cast. This is defined as a « pair of static_cast » (through void* then through T*)

static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
    D d;
    B & b = reinterpret_cast<B&>(d);
    int &n = reinterpret_cast<int&>(b);
    return reinterpret_cast<void*>(&b) ==
           reinterpret_cast<void*>(&n);
}
int main() {
    static_assert(f());
}
```

This part can be done at run-time but not at compile-time

static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
    D d;
    B & b = reinterpret_cast<B&>(d);
    int &n = reinterpret_cast<int&>(b);
    return reinterpret_cast<void*>(&b) ==
           reinterpret_cast<void*>(&n);
}
int main() {
    static_assert(f());
}
```

https://wandbox.org/permlink/fRajVngWfZJW67ct

static_cast<T>(expr)

```
struct B { int n = 3; };
struct D : B {};
constexpr bool f() {
    D d;
    B & b = reinterpret_cast<B&>(d);
    int &n = reinterpret_cast<int&>(b);
    return static_cast<void*>(&b) ==
           static_cast<void*>(&n);
}
int main() {
    static_assert(f());
}
```

With static_cast, you're fine:
https://wandbox.org/permlink/fRajVngWfZJW67ct

static_cast<T>(expr)

- One interesting aspect of constexpr contexts is that they do not allow undefined behavior
  - Implementations are required to diagnose UB in constexpr contexts
  - We can use this to our advantage (and it’s another good reason to prefer static_cast to alternatives when it makes sense to do so)

static_cast<T>(expr)

https://wandbox.org/permlink/EzK1s0o8fda5ikBE

```
struct B {};
struct D0 : B { float f = 3.0f; };
struct D1 : B { int n = 3; };
constexpr int f() {
    D0 d0;
    B & b = static_cast<B&>(d0); // fine, cast not needed
    return static_cast<D1&>(b).n; // UB
}
int main() {
    constexpr auto res = f();
}
```

static_cast<T>(expr)

https://wandbox.org/permlink/HdHB9f527ZDNtCpH

```
#include <limits>
constexpr bool f() {
    int n = std::numeric_limits<int>::max();
    // UB (signed integral overflow)
    return static_cast<bool>(n + 1 > n);
}
int main() {
    constexpr auto res = f();
}
```

static_cast<T>(expr)

https://wandbox.org/permlink/NfkeleOsUH0vfH2j

```
constexpr int f() {
    union {
        int n;
        float f;
    } u { 3 }; // initializes u.n
    return static_cast<int>(u.f); // UB (reading from non-
                                  // active union member)
}
int main() {
    constexpr auto res = f();
}
```

static_cast<T>(expr)

- One cannot use static_cast to « cast away const »
  - Adding const is of course easier as it’s implicitly correct
  - The same rules apply to volatile, the ‘v’ in « cv-qualifier », but volatile is a lot less common than const in practice…

static_cast<T>(expr)

```
int main() {
    int n = 3;
    // Ok, but prefer const_cast for this
    const int & cn = static_cast<const int&>(n);
    // int &r = static_cast<int&>(cn); // No
}
```

https://wandbox.org/permlink/7QJwJoAemYW2B6tN

static_cast<T>(expr)

https://wandbox.org/permlink/sn5OfeaQfVuyPygE

```
struct X {
   // bad idea
   int f() { return 3; }
   int f() const { return 4; }
};
void f(const X &x) {
   x.f(); // 4
   // static_cast<X&>(x).f(); // No, you want const_cast here
}
int main() {
   X x;
   x.f(); // 3
   f(x); // Ok, could use static_cast or const_cast too
}
```

static_cast<T>(expr)

- So if static_cast is dangerous for a downcast, why is it actually used in such contexts?
  - Because sometimes, you know what you’re doing, by design
  - If you know what you’re doing, you might not want to pay for protection
- The following example is inspired by some standard library facilities

static_cast<T>(expr)

```
struct Facet {
   Facet() = default;
   Facet(const Facet&) = delete;
   Facet& operator=(const Facet&) = delete;
   struct id final {
   private:
      static inline long cur{};
      long val;
   public:
      id() noexcept : val{ cur++ } {
      }
      bool operator<(const id &other) const noexcept {
         return val < other.val;
      }
   };
   virtual ~Facet() = default;
};
```

Facets are a type numbering system. Each type derived from Facet is expected to expose a static data member named id of type Facet::id

Class Facet has a virtual destructor but no other virtual member function. This is unusual and often wrong, but in this case there is method to the madness

static_cast<T>(expr)

```
struct Facet {
   Facet() = default;
   Facet(const Facet&) = delete;
   Facet& operator=(const Facet&) = delete;
   struct id final {
   private:
      static inline long cur{};
      long val;
   public:
      id() noexcept : val{ cur++ } {
      }
      bool operator<(const id &other) const noexcept {
         return val < other.val;
      }
   };
   virtual ~Facet() = default;
};
```

Type Facet::id is comparable with operator<() and each object of that type has a different val data member

static_cast<T>(expr)

```
template <class F>
    struct FacetWrapper : Facet {
        F facet;
        static const inline Facet::id id;
        FacetWrapper(const F &facet)
            : facet{ facet } {
        }
    };
```

A FacetWrapper<F> wraps an object of type F and associates a Facet::id with type FacetWrapper<F>, thus realizing the aforementioned type numbering scheme

Technically speaking, FacetWrapper<F> is a Facet but F is not, but in practice client code will know F, not FacetWrapper<F> (FacetWrapper<F> will be an implementation detail)

static_cast<T>(expr)

```
// include <map>, <memory>, <utility>
class facet_not_installed{};
class FacetServer {
   std::map<Facet::id, std::unique_ptr<Facet>> facets;
public:
   FacetServer(const FacetServer&) = delete;
   FacetServer& operator=(const FacetServer&) = delete;
   static auto &get() {
      static FacetServer singleton;
      return singleton;
   }
// ...
```

In our design, a FacetServer will hold Facets (for us: FacetWrapper<F> objects) and associate them with their Facet::id values in order to retreive them efficiently

static_cast<T>(expr)

```
// ...
private:
   FacetServer() {
      // install default facets, if any
   }
public:
   template<class F>
      void install(F &&f) {
         using namespace std;
         unique_ptr<Facet> p { new FacetWrapper<F>{ std::move(f) } }; // important: not auto!
         if (auto it = facets.find(FacetWrapper<F>::id); it != end(facets))
            swap(it->second, p);
         else
            facets[FacetWrapper<F>::id] = std::move(p);
      }
// ...
```

A FacetServer holds facets, but these facets have to be installed and there will never be more than one facet object of each type in the facet server

static_cast<T>(expr)

To get a specific facet, one uses the FacetServer's facet() member function

```
// ...
   template <class F>
      const F& facet() const {
         return const_cast<FacetServer*>(this)->facet<F>();
      }
   template <class F>
      F& facet() {
         if (auto it = facets.find(FacetWrapper<F>::id);
             it != end(facets))
            return static_cast<FacetWrapper<F>&>(*it->second).facet;
         throw facet_not_installed{};
      }
};
```

static_cast<T>(expr)

Note the presence of two lies: a (slightly dangerous) const_cast in the const version, since using operator[] on a std::map is a non-const operation, and what is of concern to us here: a static_cast used to downcast from Facet& to FacetWrapper<F>&

```
// ...
   template <class F>
      const F& facet() const {
         return const_cast<FacetServer*>(this)->facet<F>();
      }
   template <class F>
      F& facet() {
         if (auto it = facets.find(FacetWrapper<F>::id);
             it != end(facets))
            return static_cast<FacetWrapper<F>&>(*it->second).facet;
         throw facet_not_installed{};
      }
};
```

static_cast<T>(expr)

```
// ...
   template <class F>
      const F& facet() const {
         return const_cast<Fac
      }
   template <class F>
      F& facet() {
         if (auto it = facets.find(FacetWrapper<F>::id);
             it != end(facets))
            return static_cast<FacetWrapper<F>&>(*it->second).facet;
         throw facet_not_installed{};
      }
};
```

Consider why this is a safe (and efficient!) usage of static_cast: the mapping of F (through FacetWrapper<F>) to FacetWrapper<F>::id is automated by the install() member function, so we can safely assume that if there is indeed a facet in the server with this id, it's the appropriate one (you can add validation code for uninstalled facets if you want)

static_cast<T>(expr)

```
template<class F, class S>
    const F& use_facet(const S &server) {
        return server.template facet<F>();
    }
```

It’s customary to offer a use_facet<F>() helper function to simplify writing client code

static_cast<T>(expr)

- So what's the point of this design?
- It's to put together objects unrelated to one another (say, different would-be singleton objects) and then retrieved them efficiently in order to use them (a) without going through a common interface or (b) paying for services through virtual member functions
  - One pays the lookup costs, but then uses the facets without indirection
  - It can be a trade-off that pays if you lookup once, then use many times

static_cast<T>(expr)

Suppose we want to put together objects of at least two types that are structurally unrelated in a way that will make it easy to retreive and efficient to use. First. We have a texture manager...

```
// include <iostream>, <string_view>
using namespace std;
struct Texture {
   constexpr auto getTextureName() const noexcept {
      return "I am a texture name"sv;
   }
};
class TextureManager {
public:
   Texture getTexture() const noexcept {
      return {};
   }
};
// ...
```

static_cast<T>(expr)

... then we have a sound manager. Their services are distinct from one another and non-virtual, and they do not share a common base class

```
// ...
struct Sound {
    constexpr auto getFileName() const noexcept {
        return "SomeSound.wav"sv;
    }
};
class SoundManager {
public:
    Sound getSound() const noexcept {
        return {};
    }
};
// ...
```

static_cast<T>(expr)

```
// ...
#include <print>
int main() {
    auto &server = FacetServer::get();
    server.install(TextureManager{});
    server.install(SoundManager{});
    // ...
    println("{}", use_facet<
        SoundManager
    >(server).getSound().getFileName());
    println("{}", use_facet<
        TextureManager
    >(server).getTexture().getTextureName());
}
```

https://wandbox.org/permlink/dNZTnwwa53JNre6q

Note that since C++17, we could replace this pattern with std::variant in some cases (facets are more extensible, variants are faster)

dynamic_cast<T>(expr)

Something's wrong...

[expr.dynamic.cast]

dynamic_cast<T>(expr)

- The intent of dynamic_cast is to allow safe navigation through class graphs
  - In practice, the source type has to be a polymorphic type unless we're trying to perform an unambiguous upcast

dynamic_cast<T>(expr)

```
struct B { };
struct D : B { }; // note : not polymorphic
B* f(D* d) { return dynamic_cast<B*>(d); } // equivalent to B* b = d;
//D* g(B* b) { return dynamic_cast<D*>(b); } // No, B has to be polymorphic
#include <iostream>
int main() {
   D d;
   B *b = f(&d);
   if(!b) std::cerr << "dynamic_cast<B*> from D* failed (suspicious)\n";
   //   D *dp = g(b); // would not compile (D not polymorphic)
}
```

https://wandbox.org/permlink/IV1AlY8gSleMqNQs

dynamic_cast<T>(expr)

```
struct B { };
struct D0 : B { };
struct D : D0 { }; // note : not polymorphic
B* f(D* d) { return dynamic_cast<B*>(d); } // equivalent to B* b = d;
//D* g(B* b) { return dynamic_cast<D*>(b); } // No, B has to be polymorphic
#include <iostream>
int main() {
   D d;
   B *b = f(&d);
   if(!b) std::cerr << "dynamic_cast<B*> from D* failed (suspicious)\n";
   //   D *dp = g(b); // would not compile (D not polymorphic)
}
```

https://wandbox.org/permlink/w1Vad5KuhI25tOWO

dynamic_cast<T>(expr)

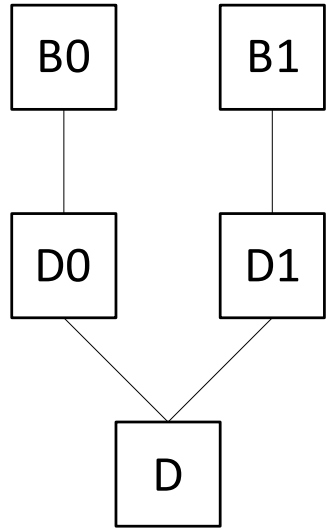
https://wandbox.org/permlink/5t4YEhck5Q7AQXAv

```
struct B { virtual ~B() = default; }; // note : polymorphic
struct D : B { };
B* f(D* d) { return dynamic_cast<B*>(d); } // equivalent to B* b = d;
D* g(B* b) { return dynamic_cast<D*>(b); } // Ok, B is polymorphic
#include <iostream>
int main() {
   D d;
   if(B *b = f(&d); !b)
      std::cerr << "dynamic_cast<B*> from D* failed (suspicious)\n";
   else
      if(D *dp = g(b); !dp)
         std::cerr << "dynamic_cast<D*> from B* that points to "
                      "a D failed (weird)";
}
```

dynamic_cast<T>(expr)

https://wandbox.org/permlink/WbW7CZUvi0AKXKAP

```
struct B0 { int f() const { return 3; } virtual ~B0() = default; };
struct B1 { int f() const { return 4; } virtual ~B1() = default; };
struct D0 : B0 {};
struct D1 : B1 {};
struct D : D0, D1 { };
int f(D0 & d0) {
    return dynamic_cast<B1&>(d0).f(); // 4
}
int main() {
    D d;
    return f(d);
}
```



dynamic_cast<T>(expr)

- As can be seen from the previous example, dynamic_cast is usually a sign that there's something wrong in our design
  - Why would someone getting a D0& argument actually want a B1&
  - But… it does happen in real code, and we need a solution for those cases
- The fact that dynamic_cast is a recognizable (and *grep*-able) token is very useful
  - We have to be pragmatic, and we have to ship code
  - Sometimes, that means fixing the weird stuff later
  - Having a recognizable marker in source code helps in that regard

dynamic_cast<T>(expr)

- Since C++20, dynamic_cast has become constexpr-friendly
  - This is probably more useful in generic code

dynamic_cast<T>(expr)

https://wandbox.org/permlink/kXgidvFEvETIVc9Q

```
struct Base {
   virtual int f() const = 0;
   virtual constexpr ~Base() = default;
};
struct D0 : Base { constexpr int f() const override { return 3; } };
struct D1 : Base { constexpr int f() const override { return 4; } };
constexpr int compute() {
   D0 d0; D1 d1; Base *v[]{ &d0, &d1 };
   int sum = 0;
   if(auto p = dynamic_cast<D0*>(v[0]); p) sum += p->f();
   if(auto p = dynamic_cast<D0*>(v[1]); p) sum += p->f();
   return sum;
}
int main() {
   static_assert(compute() == 3);
}
```

const_cast<T>(expr)

It’s a matter of security…

[expr.const.cast]

const_cast<T>(expr)

- The idea of « casting away const » is scary to many
  - Indeed, it is sometimes a dangerous thing to do and can lead to UB

const_cast<T>(expr)

```
#include <print>
int main() {
    const float pi = 3.14159f;
    // what are you trying to do???
    const_cast<float&>(pi) = 3.0f;
    print("{}", pi); // who knows? It's UB!
}
```

https://wandbox.org/permlink/cdReEAZqdluwj6tw but results are not guaranteed

const_cast<T>(expr)

- The idea of « casting away const » is scary to many
  - Indeed, it is sometimes a dangerous thing to do and can lead to UB
- There are however valid reasons to cast away const
  - For example, when integrating third party code over which you have no control and which might not be const-correct

const_cast<T>(expr)

```
// argument should be const but is not, for reasons out of our control
void log_message(string&);
// ...
void use(const string&);
void use_then_log(const string &s) {
   use(s); // Ok, const-correct
   // log_message(s); // does not compile (not const-correct)
   // ---
   // string str = s;
   // log_message(str); // compiles but costs us a temporary
   // ---
   log_message(const_cast<string&>(s)); // Ok if log_message() does
                                        // not mutate s
}
```

const_cast<T>(expr)

- A const_cast is not limited to casting away const
  - This lie allows us to add or remove const and volatile
  - Thus, we can (for example) use it to go from a volatile T& object to a const T& object
  - Essentially, const_cast lets us lie about security qualifiers in the type system

const_cast<T>(expr)

```
// suppose process(T&) and process(const T&)
// do slightly different things
template <class T>
   T& process(T &arg) { /* ... */ }
template <class T>
   const T &process(const T &arg) { /* ... */ }
#include <vector>
int main() {
   std::vector v{ 2,3,5,7,11 }; // note: non-const
   process(v); // calls process(T&)
   // calls process(const T&) (we are adding const)
   process(const_cast<const std::vector<int>&>(v));
}
```

const_cast<T>(expr)

- There are some who have suggested we could use volatile-qualified member functions in multithreaded programs to write classes that would have both thread-safe and thread-unsafe versions of a given member function
  - This is interesting, but has not seen wide adoption
  - Still, we can do it
- As an example, suppose our company prefers its own wrapper class over operating system services to standard threading
  - It happens, even today… sometimes for good reasons

const_cast<T>(expr)

```
// pImpl-style mutex class
#include <memory>
class Mutex {
   struct Impl;
   std::unique_ptr<Impl> p; // implicitly uncopyable
public:
   Mutex();
   ~Mutex(); // will be defaulted in the .cpp file
   Mutex(Mutex&&); // will be defaulted in the .cpp file
   Mutex& operator=(Mutex&&); // will be defaulted in the .cpp file
   void lock() const volatile;
   void unlock() const volatile;
};
```

const_cast<T>(expr)

The programmer's intent is that Mutex is an indirection over operating system services and as such, lock() and unlock() do not mutate p. The volatile qualifier is meant to express « can be used safely in multithreaded contexts »

```
// pImpl-style mutex class
#include <memory>
class Mutex {
   struct Impl;
   std::unique_ptr<Impl> p; // implicitly uncopyable
public:
   Mutex();
   ~Mutex(); // will be defaulted in the .cpp file
   Mutex(Mutex&&); // will be defaulted in the .cpp file
   Mutex& operator=(Mutex&&); // will be defaulted in the .cpp file
   void lock() const volatile;
   void unlock() const volatile;
};
```

const_cast<T>(expr)

```
// pImpl-style mutex class
#include <memory>
class Mutex {
   struct Impl;
   std::unique_ptr<Impl> p; // implicitly uncopyable
public:
   Mutex();
   ~Mutex(); // will be defaulted in the .cpp file
   Mutex(Mutex&&); // will be defaulted in the .cpp file
   Mutex& operator=(Mutex&&); // will be defaulted in the .cpp file
   void lock() const volatile;
   void unlock() const volatile;
};
```

The programmer's intent is that Mutex is an indirection over operating system services and as such, lock() and unlock() do not mutate p. The volatile qualifier is meant to express « can be used safely in multithreaded contexts »

Remember that not everyone agrees with this use of volatile-qualified member functions

const_cast<T>(expr)

```
template <class M>
class LockGuard {
   const volatile M &m;
public:
   LockGuard(const volatile M &m) : m{ m } {
      m.lock();
   }
   ~LockGuard() {
      m.unlock();
   }
   LockGuard(const LockGuard&) = delete;
   LockGuard& operator=(const LockGuard&) = delete;
};
```

const_cast<T>(expr)

```
template <class M>
class LockGuard {
   const volatile M &m;
public:
   LockGuard(const volatile M &m) : m{ m } {
      m.lock();
   }
   ~LockGuard() {
      m.unlock();
   }
   LockGuard(const LockGuard&) = delete;
   LockGuard& operator=(const LockGuard&) = delete;
};
```

That argument m is const and volatile is a local property. One could pass a non-const and non-volatile object as argument to this constructor

const_cast<T>(expr)

```
template <class M>
class LockGuard {
   const volatile M &m;
public:
   LockGuard(const volatile M &m) : m{ m } {
      m.lock();
   }
   ~LockGuard() {
      m.unlock();
   }
   LockGuard(const LockGuard&) = delete;
   LockGuard& operator=(const LockGuard&) = delete;
};
```

That argument m is const and volatile is a local property. One could pass a non-const and non-volatile object as argument to this constructor

We do require for lock() and unlock() to be both const-qualified and volatile-qualified

const_cast<T>(expr)

```
// include "Mutex.h", <string>, <string_view>
class TransitChannel {
   std::string data;
   Mutex m;
public:
   // fast but not thread-safe
   void add(std::string_view s) {
      data.append(s.begin(), s.end());
   }
   std::string extract() {
      std::string s;
      s.swap(data);
      return s;
   }
   // ...
```

const_cast<T>(expr)

```
// class TransitChannel (continued)
   // slower but thread-safe
   void add(std::string_view s) volatile {
      LockGuard _ { m }; // Ok, m.lock() and m.unlock()
                         // are volatile-qualified
      // delegate to non-volatile implementation
      const_cast<TransitChannel*>(this)->append(s);
   }
   std::string extract() volatile {
      LockGuard _ { m }; // Ok, m.lock() and m.unlock()
                         // are volatile-qualified
      return const_cast<TransitChannel*>(this)->extract();
   }
};
```

const_cast<T>(expr)

In essence, the volatile-qualified versions ensure synchronization, then delegate to the simpler and faster version of the member function

```
// class TransitChannel (continued)
   // slower but thread-safe
   void add(std::string_view s) volatile {
      LockGuard _ { m }; // Ok, m.lock() and m.unlock()
                         // are volatile-qualified
      // delegate to non-volatile implementation
      const_cast<TransitChannel*>(this)->append(s);
   }
   std::string extract() volatile {
      LockGuard _ { m }; // Ok, m.lock() and m.unlock()
                         // are volatile-qualified
      return const_cast<TransitChannel*>(this)->extract();
   }
};
```

reinterpret_cast<T>(expr)

“You are a chicken!”

[expr.reinterpret.cast]

reinterpret_cast<T>(expr)

- Sometimes, we need to perform casts between unrelated types
  - An int and an array of sizeof(int) bytes
  - An int* and a float*
  - A char* and a signed char* (or an unsigned char*)
  - Two class types that are unrelated but have a common initial sequence of data members
  - etc.
- Even when these « kind of » make sense, they tend to lead to unportable results
  - For example, the order in which bytes appear in an integer varies between implementations

reinterpret_cast<T>(expr)

```
#include <iostream>
#include <iomanip>
int main() {
   unsigned int n = 0xaabbccdd;
   auto p = reinterpret_cast<unsigned char*>(&n);
   for(auto q = p; q != p + sizeof n; ++q)
      std::cout << std::hex
                << static_cast<int>(*q) << std::dec;
}
```

reinterpret_cast<T>(expr)

```
#include <iostream>
#include <iomanip>
int main() {
    unsigned int n = 0xaabbccdd;
    auto p = reinterpret_cast<unsigned char*>(&n);
    for(auto q = p; q != p + sizeof n; ++q)
        std::cout << std::hex
                  << static_cast<int>(*q) << std::dec;
}
```

https://wandbox.org/permlink/9cGMv5gARBUEhHk1 but output is not guaranteed (might be aabbccdd or ddccbbaa most probably)

reinterpret_cast<T>(expr)

- Informally, reinterpret_cast allows one to
  - Convert from a null pointer to some integral type
  - Convert from an integral type or an enum to a pointer
  - Convert between function pointer types (yish!)
  - Convert between object pointer types
    - Equivalent to a cast to cv-void* followed by a cast to cv-destination type
  - Convert between pointer-to-member types
- Note that reinterpret_cast cannot « cast away const »

reinterpret_cast<T>(expr)

```
#include <print>
struct A { int n; };
struct B { int m = 4; char c = 'A'; };
struct C { float f = 3.0f; };
// include <cassert>, <concepts>, <bitset>
template <class T>
   void print_bits(const T &arg) {
      static_assert(std::integral<T>);
      std::bitset<sizeof(T)> bs = arg;
      for(int i = static_cast<int>(sizeof arg); i >= 0; --i)
         std::print("{}", static_cast<int>(bs[i]));
   }
// ...
```

reinterpret_cast<T>(expr)

```
// ...
int main() {
    A a{ 3 };
    B * b = reinterpret_cast<B*>(&a);
    assert(b->m == 3);
    // bad (not guaranteed to compile)
    // float & f = reinterpret_cast<float&>(a.n);
    C *c = reinterpret_cast<C*>(&a);
    std::println("{}", c->f); // UB
    print_bits(reinterpret_cast<B*>(c)->m);
}
```

reinterpret_cast<T>(expr)

```
// ...
int main() {
    A a{ 3 };
    B * b = reinterpret_cast<B*>(&a);
    assert(b->m == 3);
    // bad (not guaranteed to compile)
    // float & f = reinterpret_cast<float&>(a.n);
    C *c = reinterpret_cast<C*>(&a);
    std::println("{}", c->f); // UB
    print_bits(reinterpret_cast<B*>(c)->m);
}
```

This should work: neither A nor B is polymorphic, and b->m is at the same position in a B object as a.n is in a A object

reinterpret_cast<T>(expr)

```
// ...
int main() {
    A a{ 3 };
    B * b = reinterpret_cast<B*>(&a);
    assert(b->m == 3);
    // bad (not guaranteed to compile)
    // float & f = reinterpret_cast<float&>(a.n);
    C *c = reinterpret_cast<C*>(&a);
    std::println("{}", c->f); // UB
    print_bits(reinterpret_cast<B*>(c)->m);
}
```

This is not technically acceptable, but compilers will tend to let it pass (results are nasty: this breaks strict aliasing rules)

reinterpret_cast<T>(expr)

```
// ...
int main() {
    A a{ 3 };
    B * b = reinterpret_cast<B*>(&a);
    assert(b->m == 3);
    // bad (not guaranteed to compile)
    // float & f = reinterpret_cast<float&>(a.n);
    C *c = reinterpret_cast<C*>(&a);
    std::println("{}", c->f); // UB
    print_bits(reinterpret_cast<B*>(c)->m);
}
```

This might work (compilers tend to be lenient here) but bit_cast (later) is probably your friend here

reinterpret_cast<T>(expr)

```
// ...
int main() {
    A a{ 3 };
    B * b = reinterpret_cast<B*>(&a);
    assert(b->m == 3);
    // bad (not guaranteed to compile)
    // float & f = reinterpret_cast<float&>(a.n);
    C *c = reinterpret_cast<C*>(&a);
    std::println("{}", c->f); // UB
    print_bits(reinterpret_cast<B*>(c)->m);
}
```

This should work and print 0011 in practice

reinterpret_cast<T>(expr)

- C++20 introduces a number of traits that can help programmers validate if a conversion between class members is reinterpret_cast-friendly
  - is_layout_compatible<T,U>
  - is_pointer_interconvertible_base_of<B,D>
  - is_pointer_interconvertible_with_class<T,M>
  - is_corresponding_member<T0,T1,M0,M1>

reinterpret_cast<T>(expr)

- C++20 introduces a number of trai
  validate if a conversion between
  reinterpret_cast-friendly
  - **is_layout_compatible<T,U>**
  - is_pointer_interconvertible_base_of
  - is_pointer_interconvertible_with_cla
  - is_corresponding_member<T0,T1,M0,M1>

The idea is to know if the layout of data members of two types is such that one can be used instead of the other for low-level programming tricks

For example, to access a non-active member of a union, that member has to be layout-compatible with the active member of that union

reinterpret_cast<T>(expr)

- C++20 introduces a number of trai
  validate if a conversion between
  reinterpret_cast-friendly
  - **is_layout_compatible<T,U>**
  - is_pointer_interconvertible_base_of<B
  - is_pointer_interconvertible_with_class
  - is_corresponding_member<T0,T1,M0,M

Some subtleties

- Type T is layout-compatible with its const or volatile counterparts
- A signed integral type is not layout-compatible with its unsigned counterpart
- Types char, signed char and unsigned char are not layout-compatible
- Two enumerated types are layout-compatible if they have the same underlying type
- However, an enumerated type is not layout-compatible with its underlying type
- Two arrays of different but layout-compatible types are not layout-compatible, even if they have the same number of elements

reinterpret_cast<T>(expr)

- C++20 introduces a number of traits that ca
  validate if a conversion between class mem
  reinterpret_cast-friendly
  - is_layout_compatible<T,U>
  - **is_pointer_interconvertible_base_of<B,D>**
  - is_pointer_interconvertible_with_class<T,M>
  - is_corresponding_member<T0,T1,M0,M1>

Lets us detect if a pointer to a D and a pointer to a B are interchangeable (e.g.: through a reinterpret_cast)

Conditions have to be met:

- That B and D are the same type (const or volatile conditions notwithstanding, e.g.: volatile int and int are a good fit), or
- That B is an unambiguous ancestor of D
- That D is a complete type
- An empty class is pointer-interconvertible with its base classes

reinterpret_cast<T>(expr)

- C++20 introduces a number of traits that can help programmers validate if a conversion between class members is reinterpret_cast-friendly
  - is_layout_compatible<T,U>
  - is_pointer_interconvertible_base_of<B,D>
  - **is_pointer_interconvertible_with_class<T,M>**
  - is_corresponding_member<T0,T1,M0,M1>

Lets us detect if a pointer to a data member of some type M can be converted (e.g.: through reinterpret_cast) as a pointer to T

reinterpret_cast<T>(expr)

- C++20 introduces a number of traits that can help programmers validate if a conversion between class members is reinterpret_cast-friendly
  - is_layout_compatible<T,U>
  - is_pointer_interconvertible_base_of<B,D>
  - is_pointer_interconvertible_with_class<T,M>
  - **is_corresponding_member<T0,T1,M0,M1>**

Lets us detect, if M0 is a pointer to a member of T0 and M1 is a member to a pointer of T1, if M0 and M1 are at corresponding addresses in a common initial sequence of T0 and T1

reinterpret_cast<T>(expr)

- C++20 introduces a number of traits that can he
  validate if a conversion between class members
  reinterpret_cast-friendly
  - is_layout_compatible<T,U>
  - is_pointer_interconvertible_base_of<B,D>
  - is_pointer_interconvertible_with_class<T,M>
  - **is_corresponding_member<T0,T1,M0,M1>**

Essentially, the common initial sequence of two types is the (potentially empty) part for which objects of these types have equivalent layouts, irrespective of const and volatile qualifications

Lets us detect, if M0 is a pointer to a member of T0 and M1 is a member to a pointer of T1, if M0 and M1 are at corresponding addresses in a common initial sequence of T0 and T1

reinterpret_cast<T>(expr)

```
struct A { int n; };
struct B { const int n = 3; };
struct C { int m; char c; };
struct D : A {};
#include <type_traits>
int main() {
   using namespace std;
   static_assert(is_layout_compatible_v<A,B>);
   static_assert(!is_layout_compatible_v<A,C>);
   static_assert(is_layout_compatible_v<A,D>);
   static_assert(is_pointer_interconvertible_base_of_v<A,D>);
   static_assert(is_pointer_interconvertible_with_class_v<A,&A::n>);
   static_assert(is_corresponding_member_v<A, C, &A::n, &C::m>);
}
```

https://godbolt.org/z/dzc3baTzd but note that implementation support is «shaky» as of today

reinterpret_cast<T>(expr)

- A reinterpret_cast is always at no run-time cost
  - It’s just a change of perspective
  - « It’s not an int, it’s a chicken! » (and then, the programmer accepts the consequences of that *very* explicit lie)
- It’s tempting to replace static_cast with reinterpret_cast when upcasting or downcasting in a class hierarchy to save time, but be careful!
  - The (very small) address adjustment incurred from a static_cast is actually necessary to get correct code

reinterpret_cast<T>(expr)

```
struct B0 { int n = 3; };
struct B1 { int m = 4; };
struct D : B0, B1 {};
#include <cassert>
int main() {
   D d;
   void* p = static_cast<void*>(&d);
   assert(d.m != d.n);
   assert(p == static_cast<B0*>(&d));
   assert(p != static_cast<B1*>(&d));
   assert(p == reinterpret_cast<B0*>(&d));
   assert(p == reinterpret_cast<B1*>(&d)); // BAD!
}
```

reinterpret_cast<T>(expr)

```
struct B0 { int n = 3; };
struct B1 { int m = 4; };
struct D : B0, B1 {};
#include <cassert>
int main() {
    D d;
    void* p = static_cast<void*>(&d);
    assert(d.m != d.n);
    assert(p == static_cast<B0*>(&d));
    assert(p != static_cast<B1*>(&d));
    assert(p == reinterpret_cast<B0*>(&d));
    assert(p == reinterpret_cast<B1*>(&d)); // BAD!
}
```

In a given D object, the B0 and B1 base subobjects are at different locations. A static_cast shows this reality… but a reinterpret_cast just means « take this address and consider it to point to another type than the one you see »

(T) expr and T(init)

The “C cast” and the “function-style cast”

https://en.cppreference.com/w/cpp/language/explicit_cast.html

(T) expr

```
struct B { int m = 3; };
struct D : B { int n = 4; };
#include <print>
int main() {
    int n = 3;
    float f = n; // Ok
    n = (int) f; // to avoid a warning
    D d;
    B &b = (B&) d; // cast derived-to-unambiguous-base
    b.m++;
    std::print("{} {}", b.m, d.n); // 4 4
}
```

(T) expr

https://wandbox.org/permlink/920LydhNswFIYS6g

What's there not to like?

```
struct B { int m = 3; };
struct D : B { int n = 4; };
#include <print>
int main() {
    int n = 3;
    float f = n; // Ok
    n = (int) f; // to avoid a warning
    D d;
    B &b = (B&) d; // cast derived-to-unambiguous-base
    b.m++;
    std::print("{} {}", b.m, d.n); // 4 4
}
```

(T) expr

- The C language has a much simpler type system than C++
- In C++, when writing

```
(T) expr
```

- … one might be trying to
  - Perform a “safe” conversion between types such as int or float
  - Perform a conversion between base and derived types (and vice versa)
  - Perform a conversion that suppresses or adds const or volatile qualifiers
  - Perform a conversion between pointers to unrelated types
- That makes it hard to understand “what the plan is”

(T) expr

- In practice, the «C cast» tries the following casts, in order
  - A const_cast
  - A static_cast, with the peculiarity that the «C cast» does not need to respect access qualifiers and can (for example) convert a pointer-to-derived to a pointer-to-private-base
  - A static_cast followed by a const_cast
  - A reinterpret_cast
  - A reinterpret_cast followed by a const_cast
- Note that the first cast that satisfies its requirements is selected, even if the result ends up being ill-formed

(T) expr

- The «C cast» can also perform casts between incomplete types
  - If the source and destination types are both incomplete, you might get a static_cast or you might get a reinterpret_cast
  - (it’s unspecified which one you get)

(T) expr

- One useful and frequently encountered use of the «C cast» in C++ is casting an expression to void

(T) expr

- One useful and frequently encountered use of the «C cast» in C++ is casting an expression to void

```
template <class IIt, class OIt, class F>
    void transform(IIt bs, IIt es, OIt bd, F f) {
        for(; bs != es; ++bs, (void) ++bd)
            *bd = *bs;
    }
```

Casting to void in this case avoids a hostile user hijacking our algorithm through malicious use of «operator comma»

T (init)

- The «function-style cast» is akin to a constructor call from an «initializer», which can be an expression (for T(expr)) or an initializer list (for T{ args })
  - In the case of a single expression with parentheses, T(expr) is equivalent to (T) expr

T (init)

https://wandbox.org/permlink/dtrnMsC57BIMmlYL

```
struct X { int a, b; };
int main() {
    auto x = X{ 2, 3 };
    auto n = int(3);
    auto d = auto(3.0);
    int m = int(3.0);
}
```

(T) expr and T (init)

- What’s there not to like?
- Well…
  - The actual conversion being performed is not always self-evident from the source code
  - The syntax is not very much «grep-able» (these casts look like function calls)
- If you can avoid lies, do so
  - If you cannot, prefer to lie in a way that is easy to « grep » for and that conveys the intent unambiguously

The more recent casts

New additions to the family

The more recent casts

- « Recent » versions of C++ have added new casts to our set of tools
  - duration_cast, associated with the <chrono> library
  - bit_cast, which makes some low-level conversions safer and constexpr-friendly
  - any_cast, to safely recover a value that has been type-erased through std::any
  - *_pointer_cast, to circumvent bugs that result from using “naïvely” the traditional casts on shared_ptr objects

duration_cast<T>(expr)

Now, this is the time unit I want

duration_cast<T>(expr)

- duration_cast is a duration conversion facility from the <chrono> library
- Like almost all of chrono, duration_cast is constexpr-friendly

duration_cast<T>(expr)

https://wandbox.org/permlink/BIUgi2jOOeBy9zC1

```
#include <chrono>
using namespace std::chrono;
int main() {
  constexpr auto dt = 3h + 25min + 7s;
  constexpr auto secs = duration_cast<seconds>(dt);
  static_assert(secs == seconds{ 3 * 60 * 60 }+
                        seconds{ 25 * 60 } +
                        seconds{ 7 });
}
```

duration_cast<T>(expr)

```
#include <print>
#include <chrono>
using namespace std;
using namespace std::chrono;
int f(int);
int main() {
    auto pre = system_clock::now();
    int res = f(3);
    auto post = system_clock::now();
    print("{} in {}", res,
          duration_cast<microseconds>(post - pre));
}
```

bit_cast<T>(expr)

Here is how I want to see those bits

bit_cast<T>(expr)

- C++ traditionally offers some mechanisms to perform « bitwise conversions » between two objects of the same size
  - union
  - memcpy()
  - reinterpret_cast…

bit_cast<T>(expr)

- C++ traditionally offers some mechanisms to perform « bitwise conversions » between two objects of the same size
  - **union**
  - memcpy()
  - reinterpret_cast…

« In a union, at most one of the non-static data members can be active at any time, that is, the value of at most one of the non-static data members can be stored in a union at any time »

https://eel.is/c++draft/class.union#general-2

bit_cast<T>(expr)

- C++ traditionally offers some mechanisms to perform « bitwise conversions » between two objects of the same size
  - union
  - **memcpy()**
  - reinterpret_cast…

« For any trivially copyable type T, if two pointers to T point to distinct T objects obj1 and obj2, where neither obj1 nor obj2 is a base-class subobject, if the underlying bytes […] making up obj1 are copied into obj2, obj2 shall subsequently hold the same value as obj1 »

https://eel.is/c++draft/basic.types#general-3

bit_cast<T>(expr)

- C++ traditionally offers some mechanisms to perform « bitwise conversions » between two objects of the same size
  - union
  - memcpy()
  - **reinterpret_cast**…

https://eel.is/c++draft/expr.reinterpret.cast (the rules are complicated)

bit_cast<T>(expr)

- C++ traditionally offers some mechanisms to perform « bitwise conversions » between two objects of the same size
  - union
  - memcpy()
  - reinterpret_cast…
- These operations can be useful when writing low-level data manipulations
  - Transferring UDP packets on the network
  - Converting a double to an int64_t in order to examine individual bits

bit_cast<T>(expr)

- The std::bit_cast<D>(const S&) function fills this niche
  - It takes as argument a sequence of bits of type S and informs the compiler that these bits (taken as the return value from the function) as an object of type D
  - There are, of course, rules
    - S and D are trivially copyable
    - sizeof(S)==sizeof(D)
  - bit_cast is constexpr if
    - S and D are not union types, pointer types, pointer to member functions, volatile-qualified, or type that hold data members of reference types

bit_cast<T>(expr)

- The std::bit_cast<D>(const S&) function fills this niche
  - It takes as argument a sequence of bits of type S and informs the compiler that these bits (taken as the return value from the function) as an object of type D
  - There are, of course, rules
    - **S and D are trivially copyable**
    - **sizeof(S)==sizeof(D)**
  - bit_cast is constexpr if
    - S and D are not union types, pointer types, pointer to member functions, volatile-qualified, or type that hold data members of reference types

This avoids the kind of « dumb errors » one could make with reinterpret_cast or memcpy()

bit_cast<T>(expr)

- The std::bit_cast<D>(const S&) function fills this niche
  - It takes as argument a sequence of bits of type S and informs the compiler that these bits (taken as the return value from the function) as an object of type D
  - There are, of course, rules
    - S and D are trivially copyable
    - sizeof(S)==sizeof(D)
  - **bit_cast is constexpr if**
    - S and D are not union types, pointer types, pointer to member functions, volatile-qualified, or type that hold data members of reference types

This requires some measure of « compiler magic »

bit_cast<T>(expr)

```
#include <bit>
int main() {
  using namespace std;
  static_assert(sizeof(float)==sizeof(unsigned));
  constexpr auto f = 1234.5678f;
  constexpr auto u = bit_cast<unsigned>(f);
  // popcount() returns the number of bits set to 1
  // in an unsigned integer
  static_assert(popcount(u) == 13);
}
```

bit_cast<T>(expr)

Here, 13 bits of f are set and popcount() is performed on f but considered as a 32 bit unsigned integer

```
#include <bit>
int main() {
  using namespace std;
  static_assert(sizeof(float)==sizeof(unsigned));
  constexpr auto f = 1234.5678f;
  constexpr auto u = bit_cast<unsigned>(f);
  // popcount() returns the number of bits set to 1
  // in an unsigned integer
  static_assert(popcount(u) == 13);
}
```

any_cast<T>(expr)

I think I know what it is

any_cast<T>(expr)

- Type std::any from C++17 performs type erasure to let us store an object of essentially any type
  - It’s a non-pointer type that covers an analoguous to void* for pointers

any_cast<T>(expr)

https://wandbox.org/permlink/FjoFX56DrGIQ3Ptk

```
#include <any>
#include <string>
#include <vector>
int main() {
    using namespace std;
    any a = 3; // stores an int
    a = "hello"s; // now stores a std::string
    // now stores a std::vector<std::string>
    a = vector{ "hi"s, "there"s };
}
```

any_cast<T>(expr)

- There are many ways one could implement such a type
  - Some involve RTTI, for example: https://wandbox.org/permlink/GZm9iibgpEbp0V98
  - Some do not, for example: https://wandbox.org/permlink/PBWPtR9LfJ8f0VmI

any_cast<T>(expr)

- The point of storing an object into a type-erased entity like a std::any object is to be able to retrieve it later in a type-safe manner
  - For this, we have any_cast

any_cast<T>(expr)

https://wandbox.org/permlink/5ZYpTRljB97VEV12

```
#include <any>
#include <string>
#include <iostream>
int main() {
   using namespace std;
   any a = 3; // stores an int
   a = "hello"s; // now stores a std::string
   try {
      int n = any_cast<int>(a); // will fail
   } catch(bad_any_cast&) {
      cerr << "oops!\n";
   }
   cout << any_cast<string>(a) << '\n'; // will succeed
}
```

*_pointer_cast<T>(expr)

[util.smartptr.shared.cast] etc.

*_pointer_cast<T>(expr)

- The C++ standard exposes four smart pointer-related casts that play the role of the « traditional » casts but for shared_ptr objects
  - static_pointer_cast
  - dynamic_pointer_cast
  - const_pointer_cast
  - reinterpret_pointer_cast

*_pointer_cast<T>(expr)

- All four are of the same « shape », so let's take one as an example

```
template<class T, class U>
  shared_ptr<T>
    static_pointer_cast(const shared_ptr<U>&)
noexcept;
template<class T, class U>
  shared_ptr<T>
    static_pointer_cast(shared_ptr<U>&&) noexcept;
```

- This cast is the moral equivalent of a static_cast<T> applied to some object of type U, and will only work if that static_cast would also work

*_pointer_cast<T>(expr)

- All four are of the same « shape », so le

```
template<class T, class U>
  shared_ptr<T>
    static_pointer_cast(const
noexcept;
template<class T, class U>
  shared_ptr<T>
    static_pointer_cast(shared_ptr<U>&&) noexcept;
```

- This cast is the moral equivalent of a static_cast<T> applied to some object of type U, and will only work if that static_cast would also work

This does the equivalent of:

```
shared_ptr<T>(
  R , static_cast<
    typename shared_ptr<T>::element_type*
  >(r.get())
)
```

… where *R* is the argument being used as a lvalue-reference or as a rvalue-reference depending on the overload

*_pointer_cast<T>(expr)

- All four are of the same « shape », so le

```
template<class T, class U>
   shared_ptr<T>
     static_pointer_cast(const
noexcept;
template<cl
   shared_p
     static
```

- This cast is t
  object of type U, and will only work if that static_cast would also work

This does the equivalent of:

```
shared_ptr<T>(
  R , static_cast<
    typename shared_ptr<T>::element_type*
  >(r.get())
)
```

… where *R* is the argument being used as a lvalue-reference or as a rvalue-reference depending on the overload

There is an accompanying note explaining why this cast exists:

« The seemingly equivalent expression **shared_ptr<T>(static_cast<T*>(r.get()))** will eventually result in undefined behavior, attempting to delete the same object twice »

Writing our own casts

Yes, we can

Writing our own casts

- As shown with the « library casts » like duration_cast, we can write functions that respect the cast-like notation
  - This can lead to a familiar, user-controlled conversion mechanism

narrow_cast<T>(expr)

A small but useful example

narrow_cast<T>(expr)

- I took this one from Bjarne (Stroustrup) himself (thanks!)
- This cast targets the problem of « can the value of expr be cast to a T without losing expressiveness? »
  - It's a cute solution to a thorny problem
  - The strategy is « if I can cast the value to the destination type, then cast back to the original type and retrieve a value equivalent to the original one, then the conversion is not lossy »

narrow_cast<T>(expr)

```
class narrowing_exception{};
template <class D, class S>
    D narrow_cast(const S &src) {
        // might lose information
        D dest = static_cast<D>(src);
        return static_cast<S>(dest) == src ?
            dest : throw narrowing_exception{};
    }
// ...
```

Note that for ergonomics, the order of template parameters is important (we want to use the fact that S will be deduced)

narrow_cast<T>(expr)

https://wandbox.org/permlink/kFGyvAqVPdWrJgLL

```
// ...
#include <iostream>
int main() {
    int n = 2;
    float f = narrow_cast<float>(n); // fine
    try {
        n = narrow_cast<int>(f + 0.5f); // throws
    } catch(narrowing_exception) {
        std::cerr << "oops\n";
    }
}
```

lexical_cast<T>(expr)

Not the one from Boost, but I like the name!

lexical_cast<T>(expr)

- We can convert from some type to an object of type string easily in C++

```
int n = 3;
string s = to_string(n);
float f = 3.14159f;
s = to_string(f);
```

- It's a simple, homogeneous and extensible way to convert … something to a string

lexical_cast<T>(expr)

- We can also convert from a string to an object of some type easily in C++

```
string s = "3";
int n = stoi(s);
s += ".14159";
float f = stof(s);
```

- The fact that each name is different is annoying when writing generic code

lexical_cast<T>(expr)

- Using a cast-like syntax, we can achieve a more generic-friendly format

```
int n = 3;
string s = lexical_cast<string>(n);
float f = 3.14159f;
s = lexical_cast<string>(f);
double x = lexical_cast<double>(s);
```

- The syntax is the same regardless of the destination type

lexical_cast<T>(expr)

- Using a cast-like syntax, we can achieve a more generic-friendly format

```
int n = 3;
string s = lexical_cast<string>(n);
float f = 3.14159f;
s = lexical_cast<string>(f);
double x = lexical_cast<double>(s);
```

- The syntax is the same regardless of the destination type

Complete example at https://wandbox.org/permlink/LJG7PFKevcreqeFP

Why we lie

Yes, there are tools, but why do we use them?

Why we lie

- So casts exist, in most programming languages
- We can lie to our compilers
  - We try not to, but we do

Why we lie

- Why do we lie?
  - Sometimes, we have to maintain code that shows its age
  - Sometimes, we have to interact with non-idiomatic code
  - Sometimes, we are trying to be more efficient (and sometimes, we realize we should have told the truth anyway!)
  - Sometimes, we do so to hide something we're not proud of
  - Sometimes, we have a design that keeps the lies under careful control and that brings rewards
  - Sometimes, we know things the compiler cannot know (e.g.: about the shape and form of our data)
- We need to be pragmatic
- We need to ship code!

Why we lie

- There are other lies than casts
  - Things like start_lifetime_as<T>() for example
  - Inline assembly code
  - Hiding type-related details through a void*
  - Trying to express a data-invariant function without the optimizer getting in the way
- Anything that prevents the type system from doing its job, really

Why we lie

- Lying is a part of a programmer's life
  - It's a part we want to keep under control
  - Yet, it's a sometimes necessary part
- If you lie, make it visible
  - Use named casts that are easy to look for through lexical lookup
- If you lie, make the intent clear
  - Use the right cast for the job you want to do
- Make your own job easier if, in the future, you want to express the ideas in a more truthful manner

Thank you

Try to lie less, and when you do, lie usefully

Patrice Roy, Patrice.Roy@clg.qc.ca, Patrice.Roy@USherbrooke.ca