Beyond gsl::narrow_cast

CREATING DOMAIN SPECIFIC CASTING OPERATIONS TO INDICATE INTENT AND REDUCE ERRORS.

Nick Deguillaume: nick@riskpath.co.uk

27/10/2022

ACCU - Bristol

gsl::narrow / gsl::narrow_cast

What are they?

- **gsl::narrow_cast** .... just a searchable wrapper around static_cast.

```
template <class T, class U>
constexpr T narrow_cast(U&& u) noexcept {
  return static_cast<T>(std::forward<U>(u));
}
```

- **gsl::narrow** .... similar to narrow_cast but throws an exception if the static cast would cause a truncation of the arithmetic value.

gsl::narrow / gsl::narrow_cast

Great idea BUT……..

- I am writing an application, not a library. Therefore I want an assertion on failure not an exception.
- The Microsoft implementation I found does not take advantage of C++20 concepts.
- narrow_cast and narrow do not take things far enough!

It got me thinking………

- Can I replace all instances of static_cast, reinterpret_cast, const_cast and dynamic_cast in my codebase with custom casts?
- Is this even a good idea? ….. As it turns out **YES….I THINK SO….**

Replacing dynamic_cast

We are going to need some concepts…

```
template <typename Derived, typename Base>
concept StrictlyDerivedFrom =
std::derived_from<Derived, Base> && !std::same_as<Base, Derived>;
```

```
template <typename Base, typename Derived>
concept StrictlyBaseOf =
std::derived_from<Derived, Base> && !std::same_as<Base, Derived>;
```

Replacing dynamic_cast

And a slightly more advanced one…

```
template <typename Derived, typename Base>
concept StrictlyDerivedFromStatic =
StrictlyDerivedFrom<Derived, Base> && requires(Base* b) {
  { static_cast<Derived*>(b) } -> std::same_as<Derived*>;
};
```

And its natural derivatives…

```
template <typename Derived, typename Base>
concept StrictlyDerivedFromDynamic =
StrictlyDerivedFrom<Derived, Base> &&
!StrictlyDerivedFromStatic<Derived, Base>;
```

```
template <typename Base, typename Derived>
concept StrictlyBaseOfDynamic =
StrictlyDerivedFromDynamic<Derived, Base>;
```

```
template <typename Base, typename Derived>
concept StrictlyBaseOfStatic =
StrictlyDerivedFromStatic<Derived, Base>;
```

Replacing dynamic_cast

downCast – constant pointer version

```
template <typename Derived, StrictlyBaseOfStatic<Derived> Base>
[[nodiscard]] constexpr Derived const* downCast(Base const* base) noexcept {
#ifdef NDEBUG
    return static_cast<const Derived*>(base);
#else  #ifdef NDEBUG
    Derived const* const res{dynamic_cast<Derived const*>(base)};
    RPS_ASSERT(res ≠ nullptr, u8"downCast error.");
    return res;
#endif  #ifdef NDEBUG #else
}
```

Works with **constant pointer** input and when there is **no virtual inheritance** between Base and derived.

Replacing dynamic_cast

downCast – non constant pointer version

```
template <typename Derived, StrictlyBaseOfStatic<Derived> Base>
[[nodiscard]] constexpr Derived* downCast(Base* base) noexcept {
#ifdef NDEBUG
    return static_cast<Derived*>(base);
#else  #ifdef NDEBUG
    Derived* const res{dynamic_cast<Derived*>(base)};
    RPS_ASSERT(res ≠ nullptr, u8"downCast error.");
    return res;
#endif  #ifdef NDEBUG #else
}
```

Works with **non constant pointer** input and when there is **no virtual inheritance** between Base and derived.

Replacing dynamic_cast

downCast - In action …

```
class Base {
public:
  virtual constexpr ~Base() = default;
};

class Derived final : public Base { };

static constexpr Derived derived{};

constexpr Derived const* getDerivedPtr() {
  Base const* basePtr{&derived};
  Derived const* derivedPtr{downCast<Derived>(basePtr)};
  return derivedPtr;
}

static_assert(&derived == getDerivedPtr());
```

Replacing dynamic_cast

downCast - Don’t forget the reference versions …

```
template <typename Derived, StrictlyBaseOfStatic<Derived> Base>
[[nodiscard]] constexpr Derived const& downCast(Base const& base) noexcept {
#ifdef NDEBUG
    return static_cast<const Derived&>(base);
#else  #ifdef NDEBUG
    Derived const* const res{dynamic_cast<Derived const*>(&base)};
    RPS_ASSERT(res != nullptr, u8"downCast error.");
    return *res;
#endif  #ifdef NDEBUG #else
}
```

```
template <typename Derived, StrictlyBaseOfStatic<Derived> Base>
[[nodiscard]] constexpr Derived& downCast(Base& base) noexcept {
#ifdef NDEBUG
    return static_cast<Derived&>(base);
#else  #ifdef NDEBUG
    Derived* const res{dynamic_cast<Derived*>(&base)};
    RPS_ASSERT(res != nullptr, u8"downCast error.");
    return *res;
#endif  #ifdef NDEBUG #else
}
```

Replacing dynamic_cast

virtualDownCast

```
template <typename Derived, StrictlyBaseOfDynamic<Derived> Base>
[[nodiscard]] constexpr Derived const* virtualDownCast(
    Base const* base) noexcept {
#ifdef NDEBUG
    return dynamic_cast<const Derived*>(base);
#else  #ifdef NDEBUG
    Derived const* const res{dynamic_cast<Derived const*>(base)};
    RPS_ASSERT(res != nullptr, u8"downCast error.");
    return res;
#endif  #ifdef NDEBUG #else
}
```

- You are going to need this version if virtual inheritance makes a static cast impossible.
- It is nice to have a different (and longer) name since this cast is more costly.
- This slide shows one of four overloads (const | non-const) X (pointer |reference)

Casting To and From void*


```
#include <cassert>
#include <concepts>

template <typename T>
concept NonVoid = !std::same_as<T, void>;

template <NonVoid T>
[[nodiscard]] constexpr void* toVoidPointer(T* ptr) noexcept {
  return static_cast<void*>(ptr);
}

template <NonVoid U, std::same_as<void> T>
[[nodiscard]] U* voidPointerTo(T* ptr) noexcept {
  return static_cast<U*>(ptr);
}

void test() {
  class X { };
  X x{};
  assert(voidPointerTo<X>(toVoidPointer(&x)) == &x);
}
```

- **toVoidPointer** only accepts non void pointers
- **voidPointerTo** only accepts **void** pointers. No implicit conversions!!!!!!!!!!
- Really useful when inter-operating with C code

Some example numeric casts


```
template <IntegralIncByte In>
[[nodiscard]] constexpr size_t to_size_t(In n) noexcept {
  if constexpr (std::same_as<In, b8_t>)
    return std::to_integer<size_t>(n);
  else
    return internal::toUnsignedSpecified<size_t>(n);
}

template <IntegralIncByte Out, std::same_as<size_t> In>
[[nodiscard]] constexpr Out size_t_to(In n) noexcept {
  return internal::unsignedTo<Out>(n);
}
```

Alias of std::byte

```
template <IntegralIncByte Out, Enum EnumType>
[[nodiscard]] constexpr Out enumTo(EnumType e) noexcept {
  return anySzCast<Out>(
    static_cast<std::underlying_type_t<EnumType>>(e));
}
```

- I use a lot of these. This is a small sample.
- Note that **size_t_to** will only accept **size_t**. Really useful when you need to compile in both 32 and 64 bit.

Some other conversions


```
template <AnyChar OutChar, AnyChar InChar>
  requires (sizeof(InChar) == sizeof(OutChar))
[[nodiscard]] OutChar const* sameSzCharPtrCast(
  InChar const* ptr) noexcept {
  return reinterpret_cast<OutChar const*>(ptr);
}
```

```
template <typename T>
[[nodiscard]] T constexpr atomicCast(
  std::atomic<T> const& x) noexcept {

  return static_cast<T>(x);
}
```

- **Warning!!! sameSzCharPtrCast** can lead to undefined behaviour!
- For example, when converting from

  const char* to char8_t*

Some tips…

- Make each cast as narrow as possible. (Concepts are helpful)
- Name your casts well!
- Keep all your casts together so they can be found by your co-workers.
- Avoid implicit conversions on the inputs. (Use std::same_as<> and other concepts to enforce this).
- Use plenty of static_asserts and runtime asserts.
- Use constexpr and constexpr if, where possible. (reinterpret_cast will spoil this)
- If you come across a new situation, you will probably need a new cast.
- Ban the use of const_cast, reinterpret_cast, static_cast and dynamic_cast, unless they are inside one of the custom cast functions.

Some benefits I have seen…

- Catching more bugs at compile time and run time.
- Less noisy and more concise code.
- More readable code.
- Less noise from the linter.
- Forces me to really think about what I am doing. Considering what asserts I can and should use makes my code more secure.