Compile-time and Runtime Polymorphism in C++

Yunwei Ren

2018.12.15

What is polymorphism?

- Single Interface
- Multiple behaviours
- Runtime
  - Inheritance and virtual functions
  - Dynamic cast
- Compile-time
  - Function overload
  - Template and specialization
  - ……

TypeList

- To store a list of types
- How to "store" a type in C++?
    - `typedef int T;`
    - `using T = int;  // double val = 1.0;`
    - `using Ts = int, bool, char; // ???`
- `template<class... Ts> struct TypeList {};`
- `using IntBoolChar = TypeList<int, bool, char>`
- Variadic template
    - `template<class... Ts> void foo(Ts... args);`
    - `goo(args...);`
- How to recover the content of a TypeList?

Template (Partial) Specialization

- The most basic CTP: function overload

```
void print(const int &x) {
  std::cout << "int: " << x;
}
void print(const double &x) {
 std::cout << "double: " << x;
}
template <class T> void print(const T &x) {
  std::cout << "unknown: " << x;
}
print(x);  // the actual behavior dependents on
           // the type of x
```

- Template (partial) specialization
    - Class overload
- e.g. remove the const qualifier

The template

```
template <class T>
struct RemoveConst { using type = T; };

template <class U>
struct RemoveConst<const U> { using type = U; };
```

The partial specialization

- Both `RemoveConst<int>::type` and `RemoveConst<const int>::type` are `int`
- `RemoveConst<const int>::type a = 1;`

TypeList (Basic Type Computation)

- How to recover the content of a TypeList?
  - e.g. Append a type into a TypeList

```
template <class TL, class T> struct Append;
// not defined!

template <class T, class... Ts>
struct Append<TypeList<Ts...>, T> {
  using type = TypeList<Ts..., T>;
};

template <class TL, class T>
using Append_t = typename Append<TL, T>::type;
// a helper "function"

using IntChar = Append_t<TypeList<int>, char>;
```

TypeList (Basic CT Value Computation)

static_assert(sizeof...(Ts) > 0,
"The parameter pack should be nonempty")

- **A compile-time boolean value**
- `constexpr bool Foo = sizeof...(Ts) > 0;`
- **Get the size of a TypeList**

```
template <class TL> struct Size;

template <class TL>
constexpr std::size_t Size_v = Size<TL>::value;

template <class... Ts> struct Size<TypeList<Ts...>> {
  static constexpr std::size_t value = sizeof...(Ts);
};
```

Convert Values to Types

- Unify the computation of values and types

```
template <class T, T v> struct IntegralConstant {
  using type = T;
  static constexpr T value = v;
};

using IntOne = IntegralConstant<int, 1>;
using IntTwo = IntegralConstant<int, 2>;
// IntOne and IntTwo are different types

using TrueType = IntegralConstant<bool, true>;
using FalseType = IntegralConstant<bool, false>;
```

TypeList (Computation)

- Get the n-th type in the TypeList
  - Compile-time loop ?
  - Recurrence (a functional approach)

```
// the pseudo code
type get(TypeList tl, size_t n) {
  if (n == 0)
    return tl.head;
  return get(tl.tail, n - 1);
}

// the declaration (C++)
template <class TL, std::size_t n> struct Get;
```

```
// the implementation

template <class Head, class... Tail>
struct Get<TypeList<Head, Tail...>, 0> {
  // boundary value
  using type = Head;
};

template <std::size_t n, class Head, class... Tail>
struct Get<TypeList<Head, Tail...>, n> {
  using type =
      typename Get<TypeList<Tail...>, n - 1>::type;
};
```

TypeList

- Append (we won't use it)
- Get the size of a TypeList
- Get the n-th element in a TypeList
- Check whether a TypeList contains a certain type (IsIn)

Variant (A type-safe union)

- `union IntChar { int a; char b; };`
- `union IntString { int a; std::string b; };`
- Why this is invalid?
  - Non-trivial destructor!
- The `void*` approach?
  - Non-trivial destructor!
- Type erasure

Variant (Content)

```
struct Visitable {
  virtual ~Visitable() = default;
};

template <class V>
struct Value : Visitable {
  V val;
};

std::shared_ptr<Visitable> p;

template <class... Args>
Value(Args &&... args)
  : val(std::forward<Args>(args)...) {}
```

```
// detail::Content
template <class... Ts> struct Content {

  struct Visitable {…};

  template <class V> struct Value : Visitable {…};

  std::shared_ptr<Visitable> p;

};
```

Variant (Basic)

```
template <class... Ts>
class Variant : protected detail::Content<Ts...> {
public:
  template <class T> const T &as() const;
  template <class T> T &as();
};


template <class T> const T &as() const {
  static_assert(tl::IsIn_v<TL, T>,
      "No alternative is of type T");
  assert(std::dynamic_pointer_cast<Value<T>>(this->p));
  auto q =
    std::static_pointer_cast<Value<T>>(this->p);
  return q->val;
}


 template <class T> T &as() {
   return const_cast<T &>(
     static_cast<const Variant &>(*this).as<T>());
 }
```

Variant (Constructors)

- Goal: For `Variant<int, std::string>`
  - `explicit Variant(const int &);`
  - `explicit Variant(const std::string &);`
  - ……
- How to generate an unknown number of constructors?
- Expanding a … function package ?
  - `Variant(const Ts & val)...;  // ???`
  - Inheritance and recurrence!

Recursive Inheritance

```
template <class... Ts> struct Impl;

template <class T> struct Impl<T> {
   void foo(T) {…}
};

template <class Head, class... Tail>
struct Impl<Head, Tail...> : Impl<Tail...> {
   void foo(Head) {…}
   using Impl<Tail…>::foo;
};
```

- Declare the `foo` for each `Head`
- And inherit the other functions
  - Expose them

Variant (Constructors)

```
// declaration
template <class Content, class... Ts> class Impl;

// boundary
template <class Content>
class Impl<Content> : protected Content {};
```

- For convenience
- So that we can access the content, i.e. `p`

```
template <class Content, class Head, class... Tail>
class Impl<Content, Head, Tail...>
    : public Impl<Content, Tail...> {
  using Value = typename Content::template Value<Head>;

public:
  Impl() = default;

  explicit Impl(const Head &v) {
    this->p = std::make_shared<Value>(v);
  }

  using Impl<Content, Tail...>::Impl;
};


template <class... Ts>
class Variant
  : public detail::Impl<detail::Content<Ts...>, Ts…>
```

- **Default constructor ?**
  - **Let's take a rest…**

Variant (Copy Constructor)

- `Variant(const Variant & other) {`
  ```
    this->p = std::make_shard<Value<?>>(?);
  }
  ```
- How to get the value in `other` at runtime ??
- Brute force via `dynamic_cast` ?
  - `Variant<A, B>` where `A : B`
- `this->p = other.p->copy();`

```
// in Visitable
virtual std::shared_ptr<Visitable> copy() const = 0;

// in Value<V>
std::shared_ptr<Visitable> copy() const override {
  return std::make_shared<Value<V>>(val);
}
```

Variant (Visitor)

- Generalization of `copy()`
  - Dispatch according to the runtime "type" of the variant

```
class VisitIntStr
    : public Visitor<int, std::string> {
  void operator()(int &) const {…}
  void operator()(std::string) const {…}
};
```

Visitor (Variant side)

```
// in Visitable
virtual void accept(
const detail::VisitorBase &visitor) = 0;

// in Content<Ts…>::Value<V>
void accept(
  const detail::VisitorBase &visBase) override {

  const auto &vis =
      static_cast<const detail::VisitorImpl<V> &>(
          visBase);
  vis(val);
}

// in Variant<Ts…>
template <class Visitor>
void accept(const Visitor &visitor) {
  this->p->accept(visitor);
}
```

Visitor (Visitor Side)

```
template <class Head, class... Tail>
class VisitorImpl<Head, Tail...>
  : public VisitorImpl<Tail...> {
public:
  virtual void operator()(Head &) const = 0;
};
```

- Ensure that the user will override the functions

Variant (Default Constructor)

- Only if the type of the first alternative is default constructible

```
// the basic implementation
Variant() {
  using Head = tl::Get_t<TL, 0>;
  this->p = std::make_shared<Value<Head>>(Head());
}
// which works pretty well actually
```

- But how to check whether a type is default constructible?

IsConstructible

```
template <class...> using Dummy = void;

template <class, class T, class... Args>
struct IsConstructibleImpl : std::false_type {};

template <class T, class... Args>
struct IsConstructibleImpl<
  Dummy<decltype(T(std::declval<Args>()...))>,
  T, Args...> : std::true_type {};

template <class T, class... Args>
struct IsConstructible
    : detail::IsConstructibleImpl<
        detail::Dummy<>, T, Args...> {};
```

- Thx
- 2018.12.15