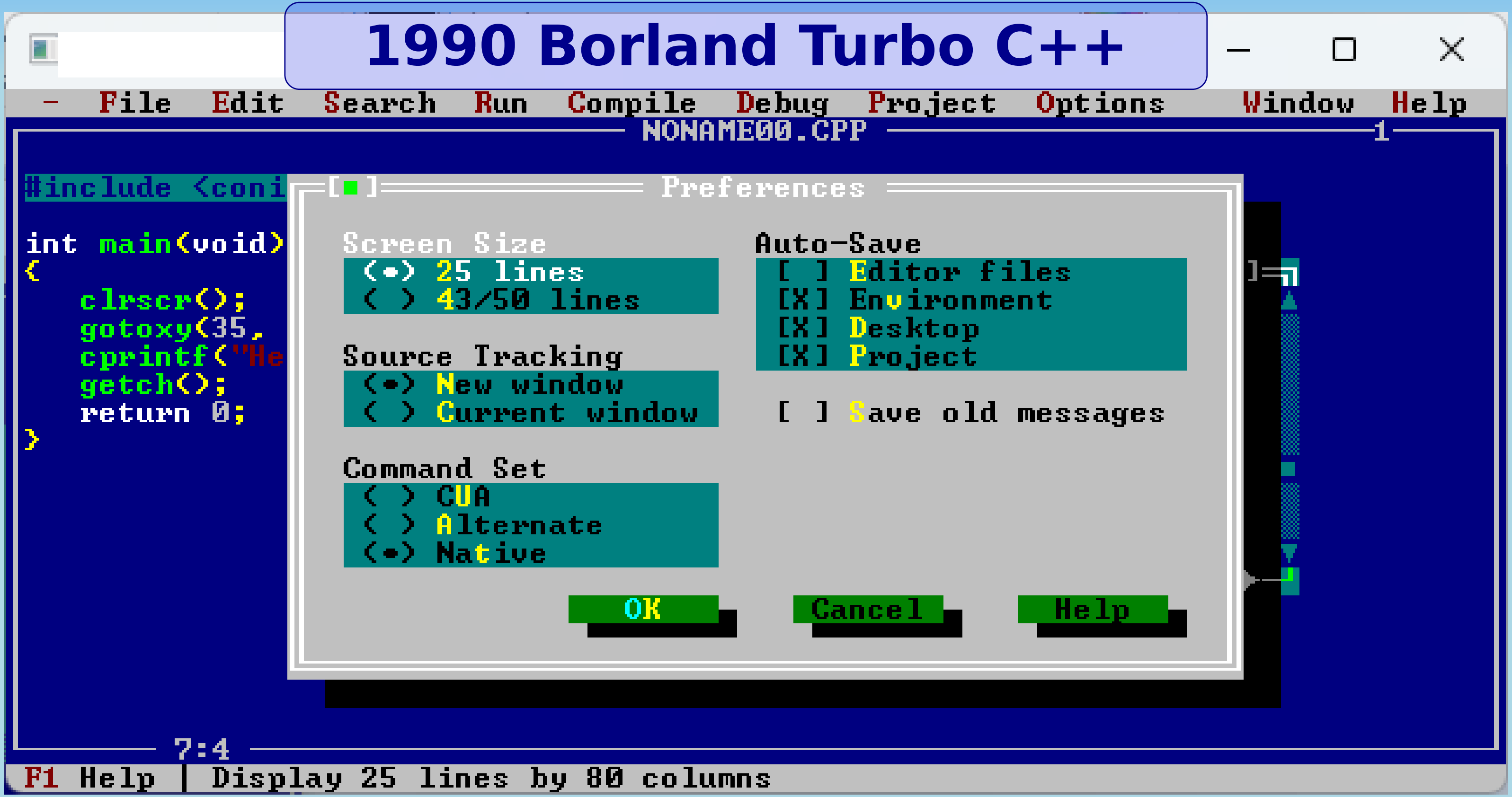
A Journey Into Non-Virtual Polymorphism

RUDYARD MERRIAM

Cppcon
The C++ Conference
2023
October 01 - 06

Who **Doesn't** Recognize This??



Background

- Experience with many langagues
  - FORTRAN IV (1968!), PL/M, assembly, C, Turbo Pascal, Forth
- Taught *Introduction to C++* - U of Houston / Clear Lake
- C++ in NASA / NIST Robotic Competitions after retired
- Writing
  - Magazines in 90s - Embedded Systems, Software Development, Programmers Journal, PC Magazine
  - Hackaday.com: C++ for embedded systems (Arduino, Raspberry Pi)
  - Medium.com (https://medium.com/@rudmerriam)

Polymorphism

- Greek: “many forms” (Obligatory statement)
- *The use of a single symbol to represent multiple different types...*
  - Polymorphic Variable

The challenge is the variable

- *...or the provision of a single interface to entities of different types*
  - Polymorphic Invokable
- *Polymorphism* is *type based dispatch*

Polymorphic Variables

- Base class pointer to derived class
  - Virtual functions are not **bad!**
- Standard Template Library
  - **`std::any`**
  - **`std::variant`**
  - **`std::tuple`**

Polymorphic Invokables

- Overloaded functions and operators

```
int plus(int, int);
string plus(string, string);
```

- Auto Parameters and Templates

```
auto plus = [](auto, auto)
auto plus = []<typename T>(T, T)

template <typename T> plus(T, T);
template <typename T, template U> plus(T, U);
```

- Curiously Recurring Template Pattern (CRTP)

Substandard C++ Warning

Conceptware Ahead!

Works with GCC 13.2

C++17 C++20 C++23

```
struct DigitalPin {
   DigitalPin(int const p, bool const v): mPin{p}, mValue{v} {}
   void set() const { .... }
   const uint8_t mPin;
   bool mValue{};
};
```

Remember this lambda in about 20 minutes

```
auto digi_out=[](const DigitalPin* const pin){ pin→set();};
```

```
struct AnalogPin {
  AnalogPin(uint8_t const pin,int const value):
        mPin{pin}, mValue{value} {
  void write() const { .... }
  const uint8_t mPin;
  int mValue;
};
```

```
struct SerialPort {
  SerialPort(int const p) : mPortNum{p} {}
    void send() const {...}
    const int mPortNum;
    std::string mMsg{};
};

DigitalPin digi{13, false};
DigitalPin digi2{14, true};
AnalogPin anl{15, 0};
SerialPort serial{23};
```

The Big Questions

```
std::vector< ??? *> outputs{&digi, &anl, &serial};

for (auto o: outputs) {
    ???
}
```

What is the polymorphic type for the vector?

What is the polymorphic invocation?

std::any

A Type That Can Contain Any Type

Challenging as a Polymorphic Variable

`std::any`

- **`std::any`** is allowed to dynamically allocate memory
  - It may use Small Buffer Optimization (SBO)
- Use **`std::any_cast<type>`** to access value
  - No easy way to determine **`type`** in the variable
  - No easy way to invoke function with **type**
  - Users must **know** what types might be used

```
void out_any(std::any const& val) {

 static auto digi_type{std::type_index(typeid(DigitalPin))};
 static auto anl_type{std::type_index(typeid(AnalogPin))};
 static auto
     serial_type{std::type_index(typeid(SerialPort))};

 const auto val_type{std::type_index(val.type())  };

 if (val_type == digi_type) {
    std::any_cast<DigitalPin>(val).set(); }
 else if (val_type == anl_type){
    std::any_cast<AnalogPin>(val).write(); }
 else if (val_type == serial_type) {
    std::any_cast<SerialPort>(v
```

Returns
type_info&

std::variant

Type Safe Union

std::variant< class... Types >

- Types enumerated in template parameter list
- Values are stored in the variant
  - No dynamic memory allocation
  - Types may dynamically allocate memory
- Straightforward access to values
  - Uses ***data type*** or ***position*** in template pack

```
using var_def =
    std::variant<*DigitalPin,*AnalogPin, *SerialPort>;
var_def var{&digi};
std::get<0>(var)->set();
var = &serial;
std::get<SerialPort*>(var)→send();
std::get<AnalogPin*>(var)→send();
```

Exception thrown when type is wrong

```
for (auto& o: outputs) {
    [](var_def const& value) {

        switch (value.index()) {
            case 0:
                std::get<DigitalPin*>(value)->set(); break;
            case 1:
                std::get<AnalogPin*>(value)->write(); break;
            case 2:
                std::get<SerialPort*>(value)->send(); break;
        }

    }(o);
}
```

index returns position of data type in template pack

Immediate call of lambda

```
[](var_def const& value) {
 if (auto res = std::get_if<DigitalPin*>(&value)) {
        (*res)->set();
 }
 else if (auto res = std::get_if<AnalogPin*>(&value)) {
     (**res).write();
 }
 else if (auto res = std::get_if<SerialPort*>(&value)){
      (*res)->send();
 }
}
```

get_if returns **value*** or **nullptr**

```
std::vector<var_def> outputs{&digi, &anl, &serial};
```

Answers what to use for type

```
for (auto o: outputs) {
    std::visit(out_overload, o /*, more variants */);
}
```

std::visit determine type of **o** and calls the function **out_overload(*cast<T>*(o))**

std::visit can process more than one variant as arguments to the invoked function

The Overload Idiom

A Polymorphic Invokable With Lambdas All The Way Down

```cpp
template<typename... Ls>
struct Overload : Ls ... {
    using Ls::operator( )...;
};
// CTAD (Class Template Argument Deduction)
// needed prior to C++ 20
template<typename... Ls> Overload(Ls...) ->
Overload<Ls...>;


auto out_overload = Overload {
   digi_out,  <<== DO YOU REMEMBER THIS LAMBDA?
   [](AnalogPin* pin) { pin->write(); },
   [](SerialPort* serial) { serial->send(); },
};
```

```
// In effect, this compiles to:

struct out_overload {
 void operator()(DigitalPin* pin) { pin->set(); },
 void operator()(AnalogPin* pin) { pin->write(); },
 void operator()(SerialPort* serial) {
     serial->send(); },
};
```

Thanks to Andreas Fertig for CppInsights

```
template<typename... Ls>
struct Overload : Ls ... {  <<== inherit all Ls
  using Ls::operator( )...; <<== use each Ls operator()
};
```

In effect, this compiles with the data types to:

```
template<>
struct Overload<l_digital, l_analog, l_serial> :
   public l_digital, public l_analog, public l_serial {
       using l_digital::operator();
       using l_analog::operator();
       using l_serial::operator();
} out_overload;
```

```cpp
auto conv_bool = [](const uint8_t* msg, bool& value) {
    value = (*msg != 0);
};

auto conv_byte = [](const uint8_t* msg, auto& value) {
    value = *msg;
};
```

```cpp
auto conv = Overload{conv_bool,conv_byte,};

uint8_t data[] {1, 'A', 0x4,};
uint8_t* msg = data;

bool b;         conv(msg, b);   msg++;
char ch{};      conv(msg, ch);  msg++;
uint8_t ui8{}; conv(msg, ui8); msg++;
```

```
struct Convert {
 void operator()(const uint8_t* msg,bool& value) {}
 void operator()(const uint8_t* msg,uint8_t& value) {}
 void operator()(const uint8_t* msg,int16_t& value) {}
} convert;


convert(msg, b);
```

Lambda capture adds flexibility

Overload Review

- Usable a polymorphic invokable
- Can return values as well as output parameters
- Calling signatures can be different
  - Different number of parameters and returns
  - Take care not to duplicate signatures

STL Variables Review

- **`std::any`** is not feasible, in my opinion
  - Too difficult to determine data types
- **`std::variant`** works well using
  - **`std::visit`**
  - The Overload Idiom

std::tuple

A *Container* ***Like*** Type

```
using tup_def = std::tuple<
        DigitalPin*,
        DigitalPin*,
        AnalogPin*,
        SerialPort*>;

tup_def tup{&digi, &digi2, &anl, &serial};

std::get<0>(tup)->set();
out_overload(std::get<1>(tup));

out_overload(std::get<AnalogPin*>(tup));
out_overload(std::get<SerialPort*>(tup));
std::get<DigitalPin*>(tup)→set();  <<== Error!!
```

```
auto tup_apply = []<typename... Ts>
 (Ts const& ... tupleArgs) { <<== Expand arg for each type
  (out_overload(tupleArgs),...); <<== Creates call for each T
 };

std::apply(tup_apply, tup);
```

In effect, `tup_apply` becomes:

```
void tup_apply(DigitalPin* digi, DigitalPin* digi2,
               AnalogPin* anl, SerialPort* serial) {
   out_overload.operator()(digi),
   out_overload.operator()(digi2),
   out_overload.operator()(anl),
   out_overload.operator()(serial);
}
```

```cpp
auto tup = std::make_tuple(
     DigitalPin{1, false}, DigitalPin{2, true},
     AnalogPin{3, 42}, SerialPort{5});

auto& [digi1, digi2, anl, serial] = tup;

anl.mValue = 21;

std::apply(tup_apply, tup);

==>> AnalogPin 3  21
```

```
std::array ary{1, 2};  <<== tuple-like

auto i_out = [](auto i) { std::cout << i << '\t'; };

auto ary_apply = []<typename... Ts>
        (Ts const& ... tupleArgs) {
    (i_out(tupleArgs), ...);
};

std::apply(ary_apply, ary); ==>> 1   2


auto cat_ary = tuple_cat(ary, ary);
std::apply(ary_apply, cat_ary);
```

Review

- **`std::tuple`** is *container-like*
- **`std::apply`** is *loop-like*
- Overload Idiom works with **`std::apply`**

Curiously Recurring Template Pattern (CRTP)

```cpp
template<typename D>
struct Shape {
    void draw() {

        auto& derived{static_cast<D&>(*this)};
        derived.draw_impl();

    }
};

struct Rectangle : public Shape<Rectangle> {
    void draw_impl() const { std::cout << "Rectangle\n"; }
};
struct Square : public Shape<Square> {
    void draw_impl() const { std::cout << "Square\n"; }
};
struct Triangle : public Shape<Triangle> {
    void draw_impl() const { std::cout << "Triangle\n"; }
};
```

```cpp
Rectangle rect;
Square sqr;
Triangle tri;

rect.draw();
sqr.draw();
tri.draw();
```

```
template<typename D>
struct Shape {
    void draw() const { derived().draw_impl(); }
    void erase() const { derived().draw_impl(true); }
    D const& derived() const {
      return static_cast<D const&>(*this); };
};


struct Rectangle : public Shape<Rectangle> {
private:
   friend Shape;
   void draw_impl( bool const erase = false) const {
       std::cout << "Rectangle\n";
   }
};
```

```
std::vector<Shape<????>*> shapes{&rect, &sqr, &tri};
```

Shape requires a template argument!

```
using var_def = std::variant<Rectangle*,
                                         Square*, Triangle*>;
std::vector<var_def> shapes{&rect, &sqr, &tri};


for (auto& s: shapes) {
    std::visit([](auto* v) { v->draw(); }, s);
}
```

```
template <typename DerivedT>
struct WaitFor {
  void wait_for(auto member, int16_t timeout = 100) {
    auto& derived = *static_cast <DerivedT*>(this);
      while((derived.*member)().invalid()...

class Power :  public WaitFor<Power> {...

Power pow;
pow.wait_for( Power::isAwake);
```

C++23: Explicit Object Parameter

AKA, Deducing This

```
struct Shape {
    template<typename T>
    void draw(this T&& self) { self.draw_impl();}
};


struct Rectangle : public Shape {
 void draw_impl() const { std::cout << "Rectangle\n"; }
};
```

Review

- The CRTP is an abstraction
- CRTP defines an interface for related types
- Remember CRTP *does not* directly-provide compile time polymorphism
- The addition of *concepts* will change CRTP

Now go learn more and use in code

A Journey Into Non-Virtual Polymorphism

Hope you enjoy CppCon

RUDYARD MERRIAM

Cppcon The C++ Conference
2023 October 01 - 06