C++ Move Semantics

Mikel Irazabal

Meetup C++ Barcelona

mikel@irazabal.eu

February 14, 2019

Overview

Genesis

- In 1995 the STL is borned, developed by Meng Lee and Alexander Stepanov

Figure: Alexander Stepanov

Genesis

- Fundamentals of Generic Programming is published [DS00]
  - "We call the set of axioms satisfied by a data type and a set of operations on it a **concept**".
  - *Regular types are presented*
    - $T\ a = b;\ assert(a{=}{=}b);$
    - $T\ a;\ a = b;\ \iff\ T\ a = b;$
    - $T\ a = c;\ T\ b = c;\ a = d;\ assert(b{=}{=}c);$
    - $T\ a = c;\ T\ b = c;\ zap(a);\ assert(b{=}{=}c\ \ a!{=}b);$
  - *And equality of Regular types is also presented...*

Genesis

- How would you define equality for built-in types? eg. int

Genesis

- How would you define equality for built-in types? eg. int
- How would you define equality for reference types? eg. char*

Genesis

- How would you define equality for built-in types? eg. int
- How would you define equality for reference types? eg. char*
- How would you define equality for user defined types? eg. std::shared_ptr

Genesis

- Logicians might define equality via the following equivalence:
- $\text{x} == \text{y} \iff \forall\ \textit{predicate}\ P, P(x) == P(y)$
- **Definition**: Two objects are equal if their corresponding parts are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

String example

Example (string class)

```
template<typename T>
class my_basic_string{
T* data_;
size_t size_;
public:
my_basic_string(T* data, size_t size) {
    ?????
}
my_basic_string(const my_basic_string& s {
    ?????
}
my_basic_string& operator=(const my_basic_string& s){
    ?????
}
```

String example

Example (string class)

```
template<typename T>
class my_basic_string{
T* data_;
size_t size_;
public:
my_basic_string(T* data, size_t size) : data_(new T[size])
                                       , size_{size}
{
    std::copy(data, data + size, data_ );
};
```

String example

Example (string class)

```
template <typename T>
class basic_string
{
my_basic_string(const my_basic_string& s):
                        data_{new T[s.size_]},
                        size_{s.size_}
{
    std::copy(s.data_, s.data_ + s.size_, data_ );
}
};
```

String example

Example (string class)

```
template <typename T>
class basic_string
{
my_basic_string& operator=(const my_basic_string& s)
{
    delete[] data_;
    data_ = new T[s.size_];
    size_ = s.size_;
    std::copy(s.data_, s.data_ + s.size_, data_ );
}
};
```

String example

Example (string class)

```
template <typename CharT>
bool operator==(const basic_string<CharT>& lhs,
                const basic_string<CharT>& rhs )
                 {
                   return !strcmp(lhs.val_,rhs.val_);
                 };
```

String example

Example (string class)

```
using my_str = my_basic_string<char>;
my_str str1("Hi");
my_str str2 = str1;
assert(str1 == str2);
str1 = "Hello";
assert(str1 != str2);
```

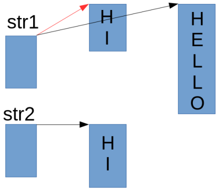

Figure: String layout

Regular types limits

- With big power comes big responsibility
- First big fiasco in C++ with `auto_ptr` in 1998
  - *From Scott Meyers* : Conceptually, `auto_ptr` is extremely simple: An `auto_ptr` is an object that acts just like a pointer, except it automatically deletes what it points to when it (the `auto_ptr`) is destroyed.
- There was still no ownership concept.
- It was extremely hard to use it in the STL because it was not a regular type.

Performance

- Howard Hinnant, was working in his free time building the STL from scratch

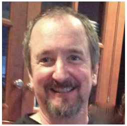

Figure: Howard Hinnant

- He wanted to build the fastest vector class
- "A Proposal to Add Move Semantics Support to the C++ Language" [HEHA02]
- semantics already exists in the current language and library to a certain extent:
  - copy constructor elision in some contexts
  - `auto_ptr` "copy"
  - list::splice
  - swap on containers

Vector simplified

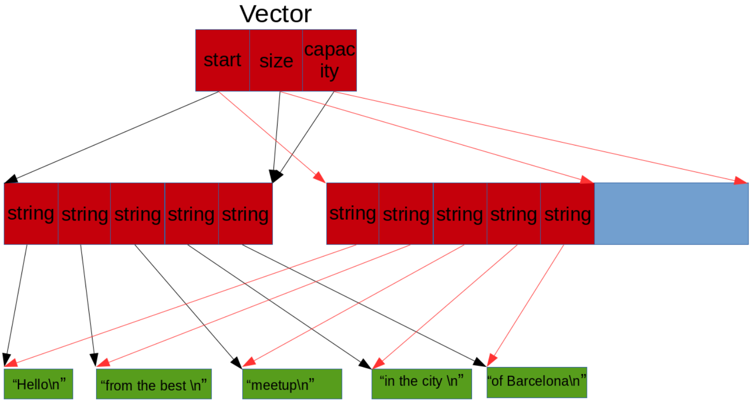

Figure: Vector Resize

Questions?

- Question: which is the advantage of "moving" an int64_t?
- Question: which is the advantage of "moving" a std::array?
- Question: which is the advantage of "moving" a std::string?
- Question: which is the advantage of "moving" a std::vector?
- Question: which is the advantage of "moving" a std::unique_ptr?
- Question: which is the advantage of "moving" a std::mutex?

Move possible implementation

Example (std::move)

```
template<typename T>
decltype(auto) move(T&& param)
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

Rvalue References

Example (Binding temporaries to references)

```
void incr(int& r){
    ++r;
}

void bar(){
    double ss = 1;
    incr(ss);
    std::cout << ss << '\n';
}
```

Expression Category [Mil10]

- The idea of identity and move capability (from New Value Terminology by Bjarne Stroustrup)
- A **glvalue** is an expression whose evaluation determines the identity of an object, bit-field, or function.
- A **prvalue** is an expression whose evaluation initializes an object or a bit-field, or computes the value of an operand of an operator, as specified by the context in which it appears, or an expression that has type cv void.
- An **xvalue** is a **glvalue** that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime).
- An **lvalue** is a **glvalue** that is not an **xvalue**.
- An **rvalue** is a **prvalue** or an **xvalue**.

Value Category

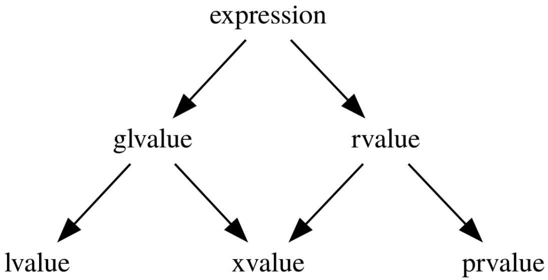

Figure: Expression category taxonomy

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
template <class T>
struct value_category{static char* value = "prvalue";};
template <class T>
struct value_category<T&> {static char* value = "lvalue";};
template <class T>
struct value_category<T&&>{static char* value = "xvalue";};

#define PRINT_VALUE_CAT(expr) std::cout << #expr << " is a "
<< value_category<decltype((expr))>::value << '\n'

 int main() {
   PRINT_VALUE_CAT(4);
   PRINT_VALUE_CAT('a');
  }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
template <class T>
struct value_category{static char* value = "prvalue";};
template <class T>
struct value_category<T&> {static char* value = "lvalue";};
template <class T>
struct value_category<T&&>{static char* value = "xvalue";};

#define PRINT_VALUE_CAT(expr) std::cout << #expr << " is a "
<< value_category<decltype((expr))>::value << '\n'

 int main() {
   PRINT_VALUE_CAT(4); // prvalue
   PRINT_VALUE_CAT('a'); // prvalue
  }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
template <class T>
struct value_category{static char* value = "prvalue";};
template <class T>
struct value_category<T&> {static char* value = "lvalue";};
template <class T>
struct value_category<T&&>{static char* value = "xvalue";};

#define PRINT_VALUE_CAT(expr) std::cout << #expr << " is a "
<< value_category<decltype((expr))>::value << '\n'

 int main() {
   const char* str = "Hello";
   PRINT_VALUE_CAT(str);
   int32_t a{0};
   PRINT_VALUE_CAT(a);
 }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
template <class T>
struct value_category{static char* value = "prvalue";};
template <class T>
struct value_category<T&> {static char* value = "lvalue";};
template <class T>
struct value_category<T&&>{static char* value = "xvalue";};

#define PRINT_VALUE_CAT(expr) std::cout << #expr << " is a "
<< value_category<decltype((expr))>::value << '\n'

 int main() {
   const char* str = "Hello";
   PRINT_VALUE_CAT(str); // lvalue
   int32_t a{0};
   PRINT_VALUE_CAT(a); // lvalue
 }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
int foo(){return 10;};
int&& foo_ref(){return 10;};

 int main() {
   int&& r = 42;
   PRINT_VALUE_CAT(r);
   PRINT_VALUE_CAT(std::move(r));
   PRINT_VALUE_CAT(foo());
   PRINT_VALUE_CAT(foo_ref());
  }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
int foo(){return 10;};
int&& foo_ref(){return 10;};

 int main() {
   int&& r = 42;
   PRINT_VALUE_CAT(r); // lvalue
   PRINT_VALUE_CAT(std::move(r)); // xvalue
   PRINT_VALUE_CAT(foo()); // prvalue
   PRINT_VALUE_CAT(foo_ref()); // xvalue
  }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
int main() {
 int32_t i{0};
 PRINT_VALUE_CAT(++i);
 PRINT_VALUE_CAT(i++);

 }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
int main() {
 int32_t i{0};
 PRINT_VALUE_CAT(++i); // lvalue
 PRINT_VALUE_CAT(i++); // prvalue
 }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
struct S { int i; };

int main() {
 PRINT_VALUE_CAT(S{0});
 PRINT_VALUE_CAT(S{0}.i);

 }
```

Expression category quiz

Example (constexpr and const avoided due to space constraints)

```
struct S { int i; };

int main() {
 PRINT_VALUE_CAT(S{0}); // prvalue
 PRINT_VALUE_CAT(S{0}.i); // gcc prvalue clang xvalue ???

 }
```

Ownership

- Rvalue References also solves the ownership problem
- std::unique_ptr, std::thread, std::mutex are not copyable but are movable
- In C++ after moving an object you can reuse it as long as you do not trust in state.
- Rust has another approach. All the heap allocated objects are moved when copied. If reused, the compiler will give an error.

Example (constexpr and const avoided due to space constraints)

```
std::unique_ptr<int> ptr_a = std::make_shared(50);
std::unique_ptr<int> ptr_b = std::move(ptr_a);
XXXXXXXXX here, you can reuse ptr_a XXXXXXXXXXXXXX
```

Perfect Forwarding [HEH06]

- A prvalue can be passed to a const, but not a lvalue

Example (Factory Method)

```
template <class T, class A1>
std::shared_ptr<T>
factory(const A1& a1)    // one argument version
{
    return std::shared_ptr<T>(new T(a1));
}

 int main() {
     std::shared_ptr<A> p = factory<A>(5);
  }
```

Perfect Forwarding [HEH06]

- If a const-qualified type is passed to the factory, the const will be deduced into the template parameter (A1 for example) and then properly forwarded to T's constructor. Similarly, if a non-const argument is given to factory, it will be correctly forwarded to T's constructor as a non-const.

Example (Factory Method)

```
template <class T, class A1>
std::shared_ptr<T> factory(A1& a1){
    return std::shared_ptr<T>(new T(a1));
}
 int main() {
    int a = 5;
    std::shared_ptr<A> p = factory<A>(a); // OK
    std::shared_ptr<A> p = factory<A>(5); // error
  }
```

Perfect Forwarding [HEH06]

- Rvalue references offer a simple, scalable solution to this problem:
- Now rvalue arguments can bind to the factory parameters. If the argument is const, that fact gets deduced into the factory template parameter type.

Example (Factory Method)

```
template <class T, class A1>
std::shared_ptr<T> factory(A1&& a1){ // also known as forward
  return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
}
int main() {
    int a = 5;
    std::shared_ptr<A> p = factory<A>(a); // OK
    std::shared_ptr<A> p = factory<A>(5); // OK
  }
```

Perfect Forwarding [HEH06]

- Like move, forward is a simple standard library function used to express our intent directly and explicitly, rather than through potentially cryptic uses of references. We want to forward the argument a1, so we simply say so.
- Forward preserves the lvalue/rvalue-ness of the argument that was passed to factory. If an rvalue is passed to factory, then an rvalue will be passed to T's constructor with the help of the forward function. Similarly, if an lvalue is passed to factory, it is forwarded to T's constructor as an lvalue.

P0898R3

- Regular types
  - DefaultConstructible
  - CopyConstructible, CopyAssignable
  - MoveConstructible, MoveAssignable
  - Destructible
  - Swappable
  - EqualityComparable

Bibliography

James C Dehnert and Alexander Stepanov.
Fundamentals of generic programming.
In *Generic Programming*, pages 1–11. Springer, 2000.

Bronek Kozicki Howard E. Hinnant, Bjarne Stroustrup.
Perfect forwarding.
WG21 N2027, 06 2006.

Peter Dimov Howard E. Hinnant and Dave Abrahams.
A proposal to add move semantics support to the c++ language.
JTC1-SC22/WG21 paper N1377, 09 2002.

William M. Miller.
A taxonomy of expression value categories.
WG21 N3055, 03 2010.

Questions?