[l, gl, x, r, pr]values

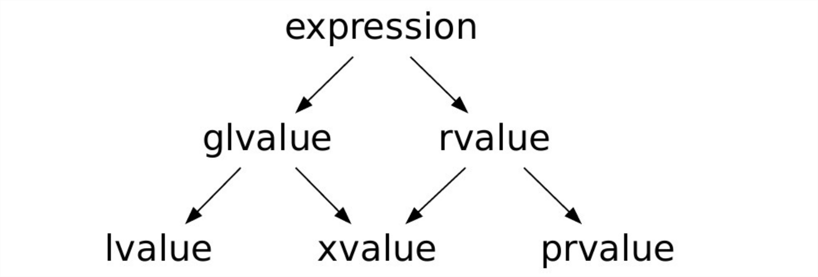
Value categories

Dawid Pilarski

dawid.pilarski@tomtom.com

Introduction

How are expressions categorized?



How to understand fundamental classifications?

- lvalue - T&

How to understand fundamental classifications?

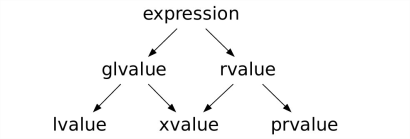
- lvalue - T&
- xvalue - T&&

How to understand fundamental classifications?

- lvalue - T&
- xvalue - T&&
- prvalue - T

The common mistake

Usually people think about expression categories:



As categories of references, which is wrong

Getting it right

category $<=>$ *expression*

reference $=>$ *category*

category $\neq>$ *reference*

[Note: there is no reference of type prvalue]

prvalue vs glvalues

glvalues

Generalized lvalues. It's everything that references the *object*

prvalues

Pure rvalues. It's a value.

Values vs Objects

Objects

- many object with same value
- object can be changed
- many references to the same object

Values

- value is unique
- value cannot be changed
- value

Into the details - glvalues

Xvalues

xvalues mean:
eXpiring values

Xvalues are such kind of expressions, that its' results point to the object, which will soon expire.

Xvalues examples

There are fixed number of ways we can get xvalues:

- function call which result type is rvalue reference (T&&).
- explicit cast to rvalue reference.
- subscript operator call on the xvalue arrays.
- non reference member access to the xvalue objects (also through pointer to member).
- temporary materialization conversion.

function call which result type is rvalue reference

```
struct Foo{};
Foo&& bar();

int main(){
  bar(); // "bar()" is the xvalue expression
}
```

explicit cast to rvalue reference

```
struct Foo{/* definition */};

int main() {
  Foo a;
  std::move(a); // "std::move(a)" casts a to Foo&&
  static_cast<Foo&&>(a); // does same thing as std::move
}
```

subscript operator call on the xvalue arrays

```
int main(){
  Foo arr[10] = {};
  std::move(arr)[0]; // xvalue ref to the first arr element
}
```

non reference member access to the xvalue objects

```
template <typename T>
struct Foo{
T member;
};

int main(){
  Foo<int> a{};
  std::move(a).member; //xvalue

  Foo<int&> a{.member = a.member};
  std::move(a).member; // lvalue
                       // due to reference collapsing
}
```

non reference member access to the xvalue objects II

```cpp
int main(){
  int Foo<int>::* pointer = &Foo<int>::member;
  Foo<int> foo{};
  std::move(foo).*pointer; //xvalue expression
  return 0;
}
```

temporary materialization conversion

```cpp
struct Foo{int member;};
Foo().member; // member access requires glvalue
              // tmc converts the prvalue to xvalue
```

Complete type requirements

glvalue expressions can operate on non-complete type

```
struct Foo{};

Foo& first_foo();
Foo& second_foo();

Foo& first_of_two(Foo& first, Foo& second){return first;}

int main(){
  auto& result = first_of_two(second_foo(), first_foo());
  if(&result == &second_foo())
    std::cout << "result is second" << std::endl;
}
```

glvalue and void

expression, which result is of type void cannot be glvalue expression.

- It's impossible to create object of type void
- It's impossible to have a reference to void

into the details - prvalues

What are prvalues expressions

Those are expression which results are the values.

prvalues examples

```
struct Foo{};
Foo(); // returns value of type Foo.

Foo bar();
bar(); // prvalue returns type Foo
```

prvalues and void

Prvalues expressions can return void type.

Type completeness requirements

Prvalues expressions that yield type T needs this type to be complete.

```
Foo first_copy_of_two(Foo& first, Foo& second){return first;}

int main(){
  // call to first_of_two is now prvalue expression
  // the program will not compile
  const auto& result = first_of_two(second_foo(),
                                    first_foo());
  if(&result == &second_foo())
    std::cout << "result is second" << std::endl;
}
```

Expression categories conversion

Types of categories conversions

glvalue to prvalue	prvalue to glvalue
• array to pointer conversion • function to pointer conversion • lvalue to rvalue	• temporary materialization conversion

array to pointer conversion

```
void printme(const char* str);
int main(){
  char str[] = {'a', 'b', 'c', 'd', '\0'};
  printme(str);
}
```

function to pointer

```
void foo(){}
void foo2(void(*)());
void foo3(void(*)()&);

void main(){
  foo; // type void(&)()
  foo2(foo); // void(&)() -> void(*)()
  foo3(foo); // also fine
};
```

lvalue to rvalue conversion

Does not take place for:

- arrays
- funtions

For not-complete type conversion is ill-formed.

lvalue to rvalue

- for non class types the cv qualifiers are discarded
- for class types the cv qualifiers are preserved

lvalue to rvalue conversion

```
void foo(Bar value);
Bar bar;
foo(bar);
```