VALUES AND REFERENCES

Russian Roulette with Memory

Florian Warg, Max Staff

May 10, 2017

WHAT ARE VALUES?

- values are expressions in a normal form
- cannot be reduced or evaluated any further

```
1 + 2 // not a value (can be reduced)
3     // value (in normal form)
```

- variables can hold values
- value is independent from its location

```
int x = 2, y = 4, z = 4;
x + y == x + z // 2 + 4 == 2 + 4
```

LVALUES

- lvalues are expressions which can be used on the left side of an assignment operation (and also the right side)
- i.e. an lvalue has a memory address

```
int x = 1; // x is evaluated to 1
x = 2;     // x is lvalue
2 = x;     // compiler error: 2 is not an lvalue
int y = x; // both x and y are lvalues
```

- lvalues persist beyond a single expression

RVALUES

- rvalues are non-lvalues
- can only be used on the right side of an assignment operation
- e.g. temporary variables, literals, addresses are rvalues

```
/* temporaries and literals*/
”rvalue”;
2;
41 + 4;
/* not rvalues */
int x = 1, y = x; // x is lvalue, not rvalue
```

REFERENCES

- a reference is an alias for an existing value
- there are references for lvalues and rvalues
- references are not necessarily objects
- i.e. there are no pointers, arrays or references to references
- give access to a variable without copying its value
- allow modification of local variables in other scopes

LVALUE REFERENCES

- alias which refers to an lvalue

```
int x = 42;
int& lx = x; // create lvalue reference to x
++lx;        // use alias instead of x
cout << x;   // what is printed here?
```

- functions taking lvalue references can modify local variables

```
void add2(int& ref) { ref += 2; }
int x = 10;
add2(x); // ref is initialized with x
cout << x;
```

RVALUE REFERENCES

- alias which refers to an rvalue

```cpp
string&& sr = "Hello";  // create temporary
cout << sr;
```

- used to implement move semantics and perfect forwarding
- move temporaries into function instead of copying their values

```cpp
void sinkStr(string&& tmp) { cout << tmp; }
sinkStr("Hello World!");
```

POINTERS

- pointers are data types that hold addresses as their values
- can be dereferenced: interpret data at address as value
- can be used for pointer arithmetics

```
int x = 42;
int* px = &x; // px holds lvalue (address) of x
cout << px;   // print address
*px = 43;     // set content of variable at &x
cout << *px   // print 43 (value of x)
struct { int x; } t, *pt = &t;
(*pt).x = 1;  // dereference then access member
pt->x = 2;    // preferred shortcut
```

POINTER ARITHMETICS

- addresses are basically numbers
- for T* offsets are relative to sizeof(T)
- can be used for low level programming and arrays

```
int arr[12];  // &arr[0] == 0xDEAD0000
int* p = arr; // p = &arr[0]
p + 1;        // 0xDEAD004 with sizeof(int) = 4
p + 3;        // 0xDEAD00C
*p = 1;       // arr[0] = 1
*(p + 1) = 2; // arr[1] = 2
p[11] = 12;   // arr[11] = 12
```

NULLPOINTER

- a pointer that holds address 0x00 is a nullpointer
- C++ has special value called nullptr
- **never use NULL or 0 instead of nullptr**
- dereferencing nullptr is undefined behavior (usually segfault)
- testing for nullptr is important

```cpp
int x = 12;
int* p = &x;
int* np = nullptr;
if (p != nullptr) { cout << "not null"; }
if (!np) { cout << "null"; }
if (p && *p) { cout << *p; }
if (np && *np) { cout << *np; } // no error.
```

VALUE SEMANTICS

- programming style that focuses on values stored in objects
- identity of objects is irrelevant
- default behavior of C++
- mathematical approach
- easy to use
- no reference aliasing problems: cannot implicitly modify a variable
- can be highly optimized with move semantics, copy elision, etc.

VALUE SEMANTICS: EXAMPLE

```
struct Member { string s; };
struct Host { Member m; };
/* ... */
Host host;
Member member;
member.s = "Hello";
host.m = member;
cout << host.m.s; // print "Hello"
member.s = "World";
cout << host.m.s; // print "Hello"
```

REFERENCE SEMANTICS

- programming style that focuses on object identity
- can be accomplished in C++ with references and pointers
- object-oriented approach (default behavior of Java)
- allows different parties to modify objects
- reference aliasing problems can occur
- in OOP copying references is usually cheaper than copying values

REFERENCE SEMANTICS: EXAMPLE

```cpp
struct Member { string s; };
struct Host { Member* m; };
/* ... */
Host host;
Member member;
member.s = "Hello";
host.m = &member;
cout << host.m->s; // print "Hello"
member.s = "World";
cout << host.m->s; // print "World"
```

CALL BY VALUE / REFERENCE

```cpp
void val(int x)  { ++x;  }
void ref(int& x) { ++x;  }
void ptr(int* x) { ++*x; }
/* ... */
int i = 0;
cout << i; // print 0
val(i);
cout << i; // print 0
ref(i);
cout << i; // print 1
ptr(&i);
cout << i; // print 2
```

NAMESPACES

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello World!\n";
    return 0;
}
```

```cpp
using namespace std;
namespace own {
    void otherFunc() { return; }
}
own::otherFunc();
```

NAMESPACES

```cpp
void outsider() { return; }

namespace own {
    void firstFunc() { return; }

    namespace second {
        void otherFunc() {
            firstFunc(); ::outsider(); return;
        }
    }
}
own::second::otherFunc();
```

NAMESPACES

```cpp
int twice() { return 1; }

namespace own {
    int twice() { return 2; }
    void firstFunc() {
        std::cout << own::twice();
        std::cout << ::twice();
        std::cout << twice();
    }
}

inline namespace own {
    void firstFunc() { return; }
}

firstFunc();
```

NAMESPACES

```cpp
#include <iostream>
using std::cout;

int main() {
    cout << "Hello World!\n";
    return 0;
}
```

```cpp
#include <iostream>
// using namespace std;

int main() {
    std::cout << "Hello World!\n";
    return 0;
}
```