

VALUES AND REFERENCES

Russian Roulette with Memory

Florian Warg, Max Staff

April 19, 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 are expressions which can be used on the left side of an assignment operation
- i.e. an lvalue has a memory address

```
int x = 1; // x holds value 1
x = 2;     // x is lvalue
&x;       // might give us the lvalue 0xFA02E1
2 = x;     // compiler error: 2 is not an lvalue
```

- rvalues are non-lvalues or temporary lvalues
- can only be used on the right side of an assignment operation
- temporary variables and literals are rvalues
- non-temporary variables hold lvalues and rvalues

```
int x = 2; // non-temporary variable x  
&x;      // address of x is its lvalue  
x;       // 2 is rvalue of x
```

- 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

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

- functions taking lvalue references can modify local variables

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

- alias which refers to an rvalue

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

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

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

- 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
-

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

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

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

- 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
-

```
1 int x = 12;
2 int* p = &x;
3 int* np = nullptr;
4 if (p != nullptr) { cout << "not null"; }
5 if (!np) { cout << "null"; }
6 if (p && *p) { cout << *p; }
7 if (np && *np) { cout << *np; } // no error.
```

- 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

```
1 struct Member { string s; };
2 struct Host { Member m; };
3 /* ... */
4 Host host;
5 Member member;
6 member.s = "Hello";
7 host.m = member;
8 cout << host.m.s; // print "Hello"
9 member.s = "World";
10 cout << host.m.s; // print "Hello"
```

- 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

```
1 struct Member { string s; };
2 struct Host { Member* m; };
3 /* ... */
4 Host host;
5 Member member;
6 member.s = "Hello";
7 host.m = &member;
8 cout << host.m->s; // print "Hello"
9 member.s = "World";
10 cout << host.m->s; // print "World"
```

```
1 void val(int x) { ++x; }
2 void ref(int& x) { ++x; }
3 void ptr(int* x) { ++*x; }
4 /* ... */
5 int i = 0;
6 cout << i; // print 0
7 val(i);
8 cout << i; // print 0
9 ref(i);
10 cout << i; // print 1
11 ptr(&i);
12 cout << i; // print 2
```
