Lecture 3a: Const and References

CS 70: Data Structures and Program Development

Tuesday, February 4, 2020

Learning Targets

1. I can use references in my code
2. I can visualize references in the CS70 memory model
3. I know what the const keyword means
4. I can use the const keyword in my programs
5. I can track which variables are constant in the memory model

Motivation

```
int triple(int value) {
  int tripled = value * value * value;
  return tripled;
}

int main() {
  int n = 10;
  n = triple(n);
  return 0;
}
```

Wouldn't it be nice to be able to change `n` inside `triple()`?

Intro to Reference Types

```
int a = 10;
int b = a;    // b is an int
              // whose value is the value of a
```

vs.

```
int  a = 10;
int& b = a;   // b is a reference to an int
              // that refers to the variable a
```

Example memory diagram

- Think of b as an alias, or another name for, a

```
int  a = 10;
int& b = a;
```

(summary of new diagram rules at end)

What will print?

```
int  m = 10;  // m is an int whose value is 10
int& n = m;   // n is a reference to an int
              //      that refers to m
int  p = 20;  // p is an int whose value is 20

m = 5;
cout << "m = " << m << ", n = " << n << endl;

n = p;
cout << "m = " << m << ", n = " << n << endl;
```


Exercise: what will print?

```
int  a = 10;
int& b = a;
int& d = b;

d = 20;
cout << "a = " << a << ", b = " << b << endl;
```

Exercise: What does this code print?

```
void spotTransplant(Cow donor, Cow recipient) {
    size_t donorSpots = donor.getNumSpots();
    size_t recipientSpots = recipient.getNumSpots();

    recipient.setNumSpots(donorSpots + recipientSpots);
    donor.setNumSpots(0);
}

int main() {
    Cow buttercup{5, 3};
    Cow flopsy{3, 13};
    spotTransplant(flopsy, buttercup);

    cout << "Buttercup: " << buttercup.getNumSpots() <<
          " spots." << endl;
    cout << "Flopsy: " << flopsy.getNumSpots() <<
          " spots." << endl;
}
```


What does *this* code print?

```
void spotTransplant(Cow& donor, Cow& recipient) {
    size_t donorSpots = donor.getNumSpots();
    size_t recipientSpots = recipient.getNumSpots();

    recipient.setNumSpots(donorSpots + recipientSpots);
    donor.setNumSpots(0);
}

// same main() as previous slide
int main() {
    Cow buttercup{5, 3};
    Cow flopsy{3, 13};
    spotTransplant(flopsy, buttercup);

    cout << "Buttercup: " << buttercup.getNumSpots() <<
          " spots." << endl;
    cout << "Flopsy: " << flopsy.getNumSpots() <<
          " spots." << endl;
}
```

Exercise: what does this code print?

```
void div(int a, int b, int& quo, int& rem) {
  rem = a;
  quo = 0;
  while(rem >= b) {
    rem = rem - b;
   ++quo;
  }
}

int main() {
  int x = 24;
  int y = 7;
  int quotient;
  int remainder;
  div(x, y, quotient, remainder);
  cout << "x = " << x << ", y = " << y;
  cout << " quotient = " << quotient <<
          ", remainder = " << remainder << endl;
}
```


Warning: C++ References aren't Java References

A C++ reference is never "null"

A C++ reference cannot be "moved" or "redirected"

Defining Constants

```
const int PENNIES_PER_DOLLAR = 100;
const string WELCOME_MSG = "Greetings!";
const double SQRT2 = sqrt(2);
```

Examples: `const` and References

```
int a = 3;
const int& b = a;

++a; //Okay!
++b; //Causes a compiler error!
```


Examples: `const` and References

```
const int c = 3;
int& d = c; //Causes a compiler error!
```

```
const int e = 3;
const int& f = e; //Okay!
```


Exercise: How many errors in this code?

```
int a = 3;
const int b = a;
int& c = a;
const int& d = a;
int& e = b;

++a;
++b;
++c;
++d;
```

Easy way to make the function more efficient?

```
size_t getSize(vector<int> v) {
  return v.size();
}

int main() {
  vector<int> w(100000);
  cout << getSize(w) << "\n";
  return 0;
}
```


Summary of New Diagram Rules

- **References**
  - When a function is called, allocate space for all local variables ***except*** **references**.
  - When a reference is initialized, write its name next to its referent.
  - When the referent is in a different stack frame, note that on the right side of the diagram.
  - When a reference is destroyed, cross out its name (not necessarily the referent's name!).
- **`Const`**
  - Draw a padlock on names that are labeled as `const` (meainging "can't use this name to change this value").

Static Const Class Members

- **We can declare a member of a class as `static const`.**
  - No matter how many objects of the class are created, there is only one copy of the `static const` member.
  - A `static const` member is shared by all objects of the class.

In cow.hpp:

```
class Cow{
    ...
    static const string TAXONOMIC_FAMILY = "bovidae";
    ...
};
```

In main.cpp:

```
cout << Cow::TAXONOMIC_FAMILY << endl;
```


Makefiles

- **`make`: A helpful tool to automate compiling**
- ***Makefile* has *rules***

```
target: dependency_1 dependency_2 ...
<tab>   commands_to_run
```

- **Example**

Suppose farm.cpp includes cow.hpp. We can write a `make` rule:

```
farm.o: farm.cpp cow.hpp
    clang++ -o farm.o -c farm.cpp
```

The program `make` detects if `farm.cpp` or `cow.hpp` have changed since the last time `farm.o` was generated, and then (re)compiles into `farm.o` if necessary.