Efficiency

Motivation

- We've seen some speedup techniques
- Now we'll delve into some lower-level C++ details
- And discuss one commonly applied speedup pattern

Strategy

- A common strategy for data structures: Copy-on-Write (CoW)
- Idea: Whenever you assign a variable to another, the data is not copied
  - Another reference is made to same underlying data
- If one of the variables wants to alter its data, must clone underlying data

Example

- Suppose we did this:

```
string a = "foo";
string b = a;
```

- C++ string: Would make a copy at second line
  - This can be time consuming if we make lots of copies
  - How could such a thing occur?

Example

- Consider:

```
int countSpaces(string x){
    int c=0;
    for(size_t i=0;i<x.length();++i){
        if( x[i] == ' ' )
            c++;
    }
    return c;
}
```

- Calling this function results in a copy being made
  - Even though the function doesn't change the data at all

References

- Traditionally, in C++, we had one kind of reference:

```
void foo(vector<int>& foo ){
    foo.push_back(42);
}
```

- Also called an *lvalue reference*

Code

- We could eliminate the needless copy in the previous code easily:

```
int countSpaces(const string& x){
    int c=0;
    for(size_t i=0;i<x.length();++i){
        if( x[i] == ' ' )
            c++;
    }
    return c;
}
```

- Problem solved!
  - Right?

Problem

- Not everyone remembers to use the const-reference calling convention
- Sometimes it won't work!

```
//count number of spaces in the string and return it.
//also print the string, but replace tabs with spaces in
//the printed string. Don't want caller's view of
//string to change at all.
int countSpacesAndRemoveTabsAndPrint(const string& x){
    int c=0;
    for(size_t i=0;i<x.length();++i){
        if( x[i] == ' ' )
            c++;
        else if( x[i] == '\t' )
            x[i] = ' '; //need to work on local copy
    }
    cout << x;
    return c;
}
```

Oops

- Oops. Won’t work
- We’re changing x, but const won’t allow it

Problem

- We need to change parameter type to either “string” or “string&”
- Neither is ideal
  - string $\rightarrow$ Needless copying if we don’t have any tabs in string
  - string& $\rightarrow$ Forbids things like countSpacesAndRemoveTabsAndPrint(string(“foo”))
    - We’ll see why in a moment

Variables

- There are two kinds of variables: lvalues and rvalues
  - Roughly speaking, lvalues are things that can appear on lhs of assignment
  - rvalues are things that are only valid on rhs of assignment
- Ex: An ordinary variable ("x") is an lvalue
- Ex: An integer constant is an rvalue
- Ex: A function call could be either one
  - int f(){...} $\leftarrow$ rvalue
  - int& g(){...} $\leftarrow$ lvalue

Checking

- C++ compiler does some type checking to prevent errors
- Ex:

```
void foo(int& x){
    x=42;
}
void bar(){
    foo(4);
}
```

- What does this do?

Checking

- C++ forbids preceding code
- But this *is* legal:

```
void foo(const int& x){
    ...
}
void bar(){
    foo(4);
}
```

- foo() is not allowed to change x since it's const
  - x is a const lvalue reference here

Rules

- lvalues can be passed to const or non-const lvalue references
  - Or to non-references (ie, pass-by-value)
- rvalues can be passed to const lvalue references only
  - Or non-references
- const lvalue references can be passed to const lvalue references only
  - Or non-references
- Of course, non-references make copies of the data
  - Well... not really. Not all the time.

Copies?

- Consider this code. What does it output?

```
struct Foo{
    Foo(){cout << "In constructor\n";}
    Foo(Foo& f){
        cout << "In constructor 2\n";
    }
    Foo(const Foo& f){
        cout << "in constructor 3\n";
    }
};
void bar(Foo f){}
int main(int argc, char* argv[]){
    Foo f1;
    bar(f1);
    return 0;
}
```

Output

- Output:
  In constructor
  In constructor 2
- First line comes from "Foo f1"
- Second comes from function call
- Concept: *copy constructor*
- Constructor 3 isn't used here because f1 is a non-const reference

Question

- Consider this code. What is the output?

```
struct Foo{
    Foo(){
        cout << "In constructor\n";
    }
    Foo(Foo& f){
        cout << "In constructor 2\n";
    }
    Foo(const Foo& f){
        cout << "in constructor 3\n";
    }
    void operator=(const Foo& f2){
        cout << "In operator=\n";
    }
};
int main(int argc, char* argv[])
{
    Foo f1;
    Foo f2;
    f2 = f1;
    return 0;
}
```

Output

- We get:
  In constructor
  In constructor
  In operator=

Now...

- What if we change main:
  ```
  Foo f1;
  f1 = Foo();
  ```
- What output do we get?

Output

- We get:
  In constructor
  In constructor
  In operator=

Observe

- What does our operator= need to do?
  - Copy all data from f2 into *this
  - But: Suppose we have a case like previous example
  - If rhs was an rvalue, it will be going away right after the assignment
  - Wouldn’t it be better if we could “steal” rhs’s data?

Example

- Add data field to Foo to make it more obvious what we want:

```
struct Foo{
    vector<int> x;
    Foo(){
        cout << "In constructor\n";
    }
    Foo(Foo& f){
        cout << "In constructor 2\n";
    }
    Foo(const Foo& f){
        cout << "in constructor 3\n";
    }
    void operator=(const Foo& f2){
        cout << "In operator=\n";
        x = f2.x;
    }
    void operator=(Foo& f2){
        cout << "In operator= (swap)\n";
        x.swap(f2.x);
    }
};
int main(int argc, char* argv[])
{
    Foo f1;
    f1 = Foo();
    return 0;
}
```

Output

- This won’t do what we want!
- Output:
  In constructor
  In constructor
  In operator=
- Why?

Remember

- C++ won't pass an rvalue to a non-const lvalue reference
- So C++ chooses the non-swapping operator=
- What we have here is wrong anyway:

```
Foo f1;
Foo f2;
f1.x.push_back(42);
f2 = f1;
cout << f1.x.size() << "\n";
```

- Outputs:

  In constructor

  In constructor

  In operator= (swap)

  0
- So f1 lost its data!

Solution

- C++ 11 introduced the *rvalue reference:* This can be used to implement *move semantics*

```
struct Foo{
    vector<int> x;
    Foo()               { cout << "In constructor\n";   }
    Foo(Foo& f)         { cout << "In constructor 2\n"; }
    Foo(const Foo& f)   { cout << "in constructor 3\n"; }
    void operator=(const Foo& f2){
        cout << "In operator=\n";
        x = f2.x;
    }
    void operator=(Foo&& f2){
        cout << "In operator= (swap)\n";
        x.swap(f2.x);
    }
};
int main(int argc, char* argv[])
{
    Foo f1;    Foo f2;
    f1.x.push_back(42);
    f2 = f1;
    cout << f1.x.size() << "\n-----------------\n";
    Foo f3;
    f3 = Foo();
    return 0;
}
```

Output

- Result:
  In constructor
  In constructor
  In operator=
  1

  In constructor
  In constructor
  In operator= (swap)

Note

- If we *only* define the rvalue reference version of operator= then this line:
  f2=f1
  is an error
  - f1 is an lvalue reference, and it can't be passed to an rvalue reference parameter

Also

- A proper move assignment (or move constructor) should also be tagged "noexcept"
- What does this output?

```
struct Foo{
    vector<int> x;
    Foo()                { cout << "In constructor\n";                }
    Foo(Foo& f)          { cout << "In constructor 2\n";              }
    Foo(const Foo& f)    { cout << "in constructor 3\n";              }
    Foo(Foo&& f) noexcept { cout << "In constructor 4\n"; x.swap(f.x); }
    void operator=(const Foo& f2){
        cout << "In operator=\n";
        x = f2.x;
    }
    void operator=(Foo&& f2) noexcept{
        cout << "In operator= (swap)\n";
        x.swap(f2.x);
    }
};
int main(int argc, char* argv[])
{
    Foo f4(Foo());
    return 0;
}
```

Surprise!

- It outputs *nothing*
- Why?
  - C++'s "most vexing parse" rule: Anything that might be a function declaration *is* a function declaration
  - This line: Foo f4(Foo()); is parsed as a function prototype:
  - Function name is f4
  - Function return type is Foo
  - Function has one parameter
    - Type of parameter is a function pointer
    - That function has no arguments
    - That function returns a Foo object
- So main() doesn't do anything. Hence, no output.

Fix

- In C++ 11 we can fix this with *uniform initialization syntax*

  ```
  int main(int argc, char* argv[])
  {
      Foo f4(Foo{});
      return 0;
  }
  ```

- This outputs:

  In constructor

- Surprise! No move constructor
  - C++ compiler constructs the temporary "in-place" where f4 will live
  - Avoids either a copy or a move!

Move Constructor

- Move constructors are used less often than one might expect
- Ex: Is move constructor called here?

```
struct Foo{
    vector<int> x;
    Foo()                 { cout << "In constructor\n";                    }
    Foo(Foo& f)           { cout << "In constructor 2\n";                  }
    Foo(const Foo& f)     { cout << "in constructor 3\n";                  }
    Foo(Foo&& f) noexcept { cout << "In constructor 4\n"; x.swap(f.x);     }
    void operator=(const Foo& f2){
        cout << "In operator=\n";
        x = f2.x;
    }
    void operator=(Foo&& f2) noexcept{
        cout << "In operator= (swap)\n";
        x.swap(f2.x);
    }
};
Foo baz(){
    Foo rv;
    return rv;
}
int main(int argc, char* argv[]){
    Foo f1 = baz();
    return 0;
}
```

Nope!

- No move constructor call here
- C++ compilers use *return value optimization* here
  - Returned item is constructed in-place where it will finally live

Example

- This function will use the move constructor for the returned value:

```
Foo bar(int x){
    if(x){
        Foo rv;
        return rv;
    } else{
        Foo rv2;
        return rv2;
    }
}
```

- Compiler doesn't know which `Foo` will be returned
  - So it can't construct it in-place

COW

- Now we can show our copy-on-write string + a small test harness
- cowstring.h cowtest.cpp

Sources

- Thomas Becker. C++ Rvalue References Explained. http://thbecker.net/articles/rvalue_references/section_01.html
- Jonathan Boccara. The Most Vexing Parse: How to Spot It and Fix It Quickly. https://www.fluentcpp.com/2018/01/30/most-vexing-parse/
- Kurt Guntheroth. Optimized C++. O'Reilly Media.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org