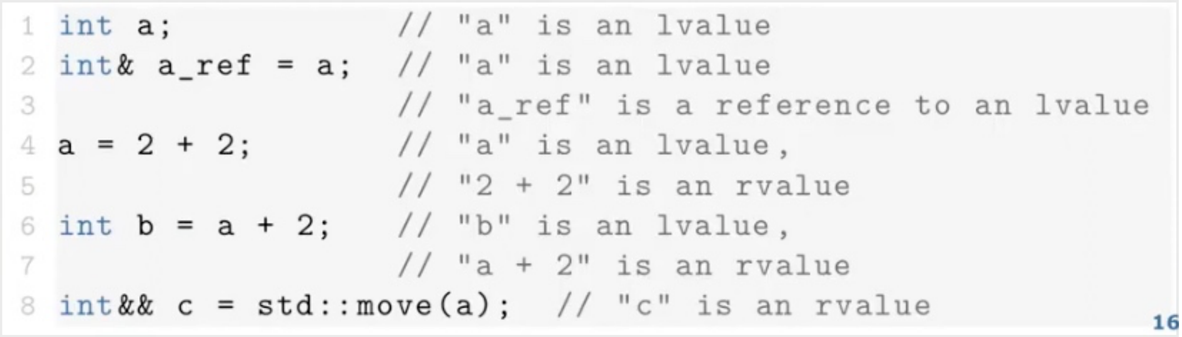
Move Semantics

Intuition lvalues and rvalues

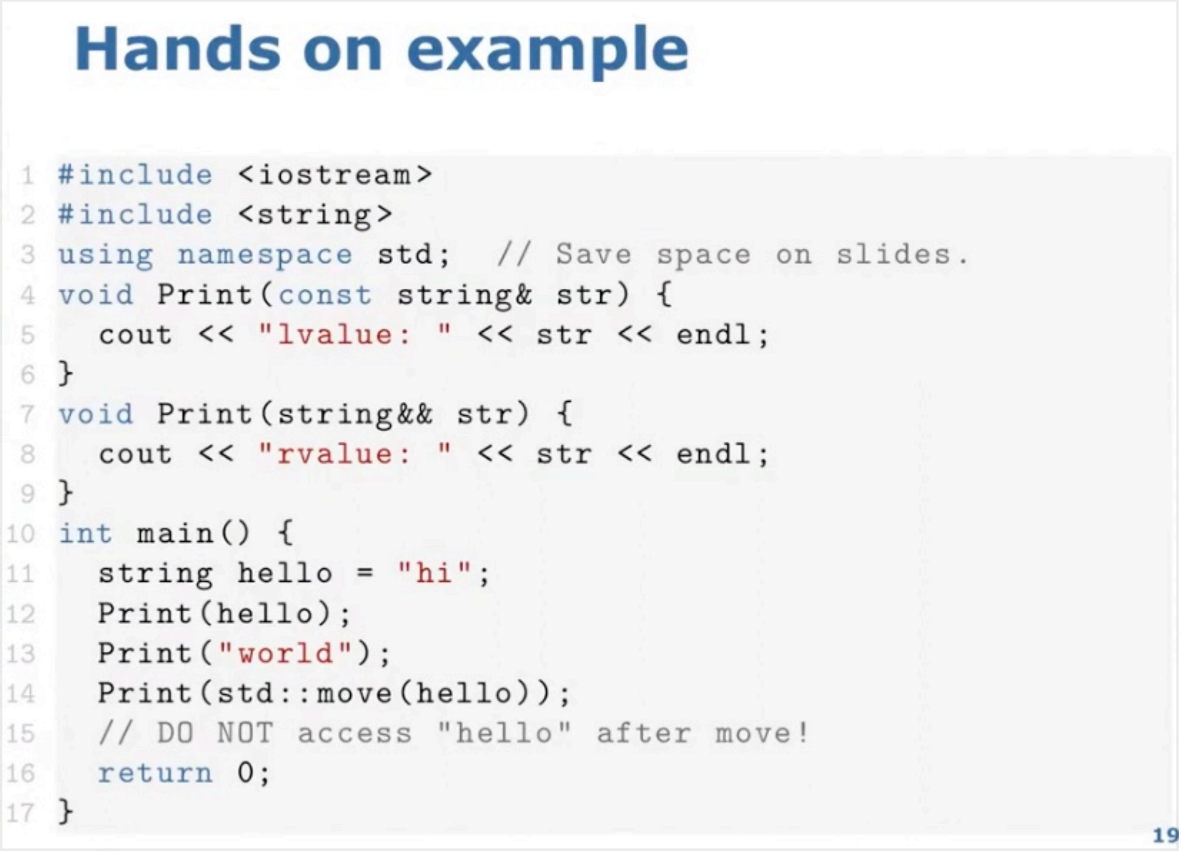
- Every expression is an lvalue or an rvalue.
- lvalues can be written on the left of assignment operator (=).
- rvalues are all the other expressions.
- Explicit rvalue can be defined using &&.
- Use std::move() to explicitly convert an lvalue to rvalue.



```
int a;                 // "a" is an lvalue
int& a_ref = a;        // "a" is an lvalue
                       // "a_ref" is a reference to an lvalue
a = 2 + 2;             // "a" is an lvalue,
                       // "2 + 2" is an rvalue
int b = a + 2;         // "b" is an lvalue,
                       // "a + 2" is an rvalue
int&& c = std::move(a);  // "c" is an rvalue
```

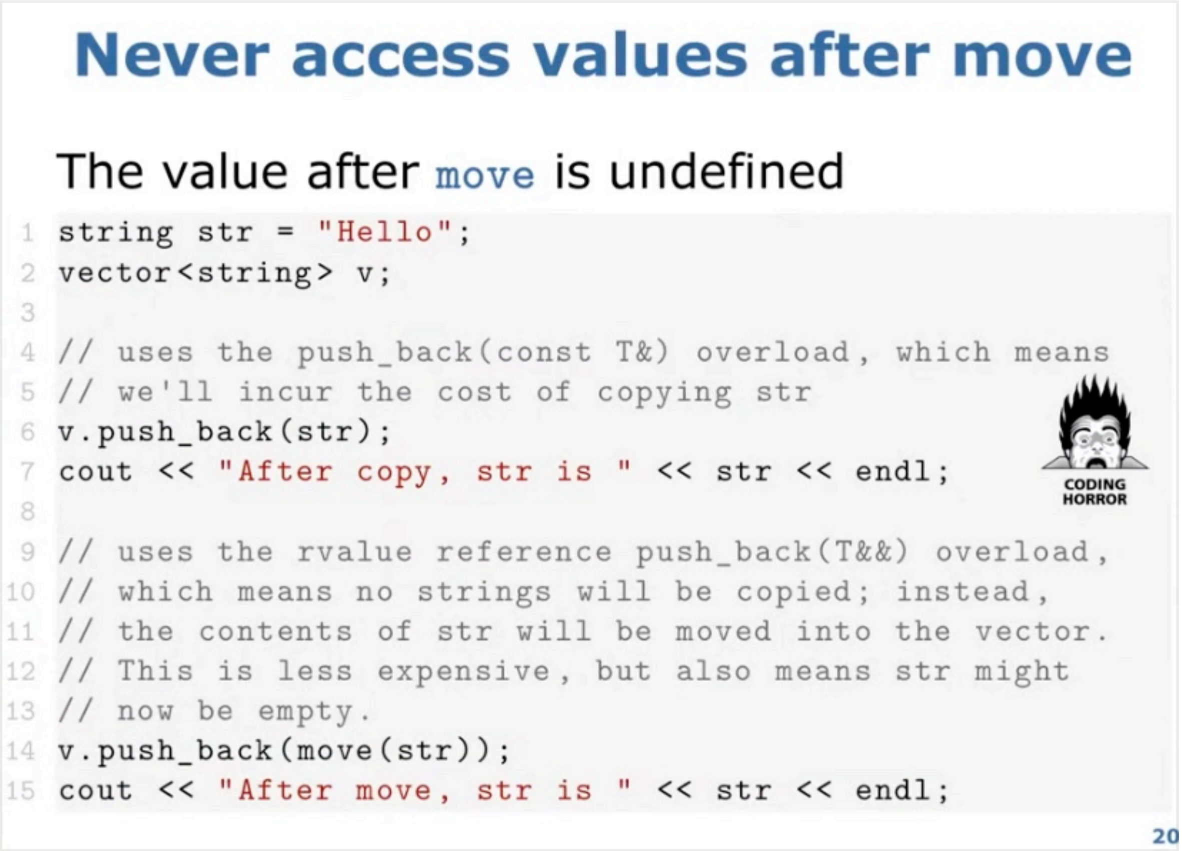
std::move

- The std::move() is a standard library function returning an rvalue reference to its argument.
- Std::move(x) means “give me an rvalue reference to x”.
- That is, std::move(x) does not move anything, instead, it allows a user to move x.



Hands on example

```cpp
#include <iostream>
#include <string>
using namespace std;  // Save space on slides.
void Print(const string& str) {
  cout << "lvalue: " << str << endl;
}
void Print(string&& str) {
  cout << "rvalue: " << str << endl;
}
int main() {
  string hello = "hi";
  Print(hello);
  Print("world");
  Print(std::move(hello));
  // DO NOT access "hello" after move!
  return 0;
}
```



Never access values after move

The value after `move` is undefined

```cpp
string str = "Hello";
vector<string> v;

// uses the push_back(const T&) overload, which means
// we'll incur the cost of copying str
v.push_back(str);
cout << "After copy, str is " << str << endl;

// uses the rvalue reference push_back(T&&) overload,
// which means no strings will be copied; instead,
// the contents of str will be moved into the vector.
// This is less expensive, but also means str might
// now be empty.
v.push_back(move(str));
cout << "After move, str is " << str << endl;
```

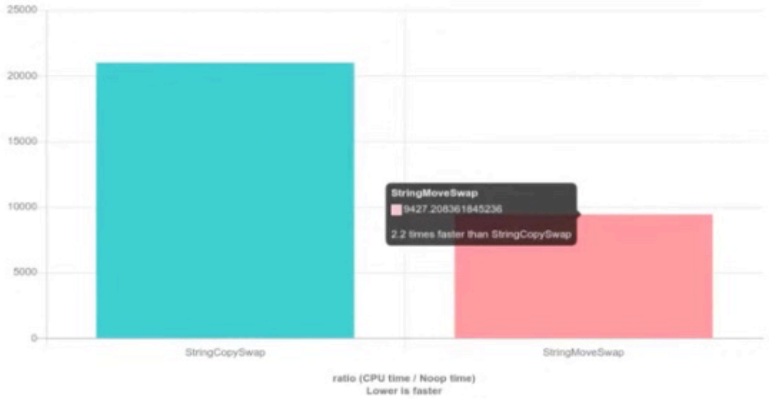
CODING HORROR

std::move performance

```cpp
void copy_swap(MyClass& obj1, MyClass& obj2) {
  MyClass tmp = obj1;   // copy obj1 to tmp
  obj1 = obj2;          // copy obj2 to obj1
  obj2 = tmp;           // copy tmp  to obj1
}

void move_swap(MyClass& obj1, MyClass& obj2) {
  MyClass tmp = std::move(obj1);  // move obj1 to tmp
  obj1 = std::move(obj2);         // move obj2 to obj1
  obj2 = std::move(tmp);          // move tmp  to obj1
}
```

std::move performance



Quick Benchmark available to play:
https://bit.ly/2DFfhko

How to think about std::move

- Think about ownership.
- Entity owns a variable if it deletes it, e.g.
  - A function scope owns a variable defined in it.
  - An object of a class owns its data members.
- Moving a variable transfers ownership of its resources to another variable.
- Runtime: better than copying, worse than passing by reference.