Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 9

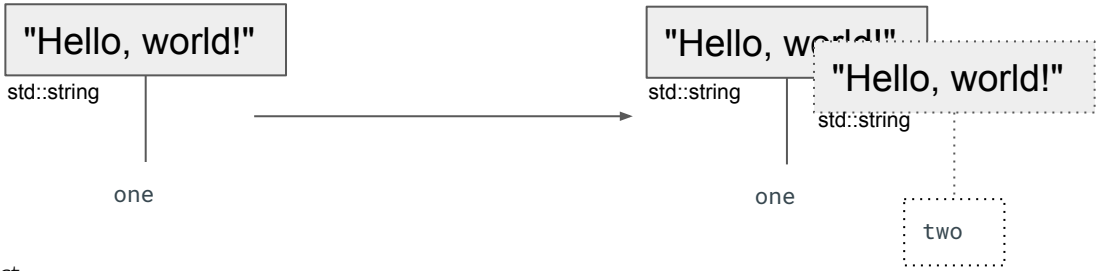
C++
Copying & moving

Copying

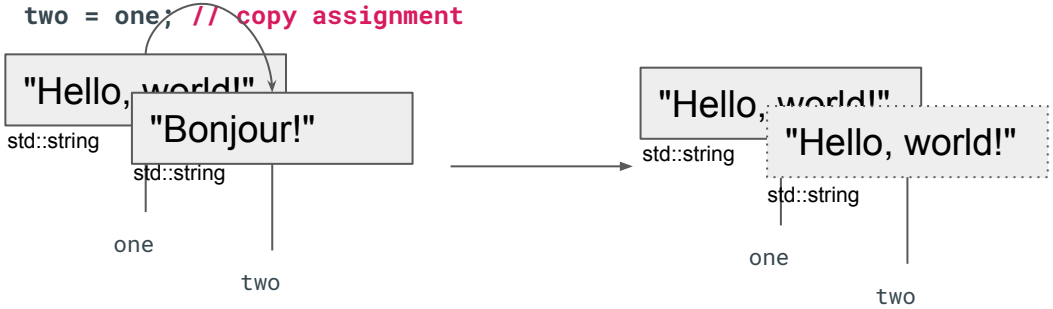
A **copy** of an object can happen in two ways:

- **Copy construction**, by making a new object from an existing object.
- **Copy assignment**, by filling an existing object with another object.

```
std::string one {"Hello, world!"};
std::string two = str; // copy construction
```




```
std::string one {"Hello, world!"};
std::string two {"Bonjour!"};
two = one; // copy assignment
```



Moving

A **move** is like a copy, *but leaves the original object in an empty state*. It's like stealing!

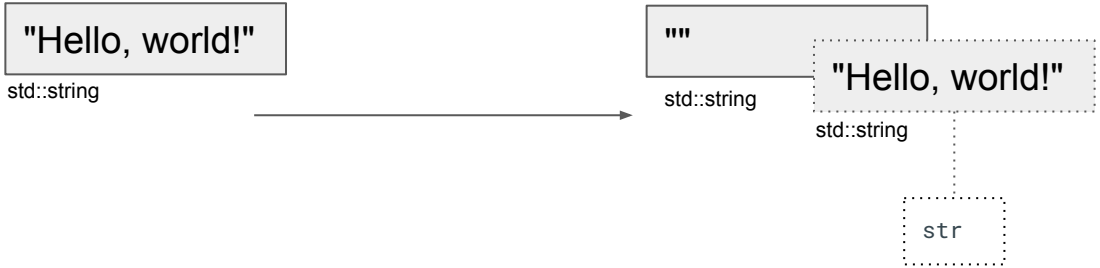
A move is executed when:

- If it's an object, it must support moving
- the to-be-stolen-from object is an rvalue

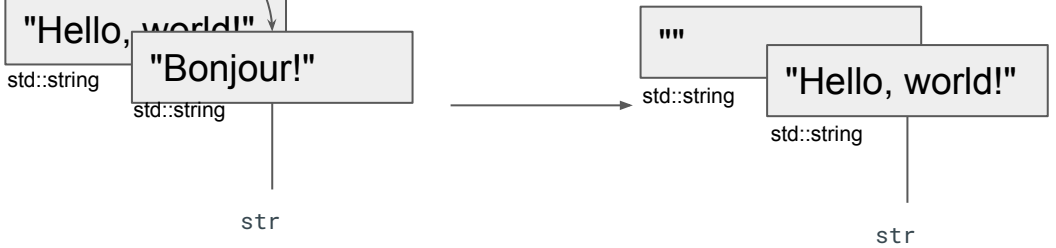
A *move* can happen in the same way as a *copy*:

- **Move construction**
- **Move assignment**

```
std::string str = std::string("Hello, world!"); // move construction
```




```
std::string str {"Bonjour!"};
str = std::string("Hello, world!"); // move assignment
```

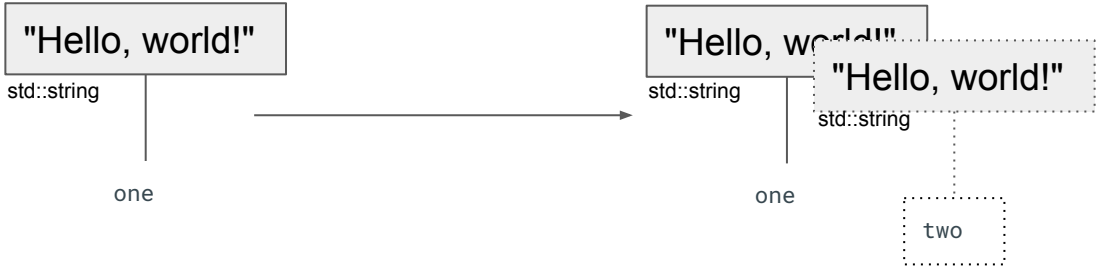


Copy vs. move

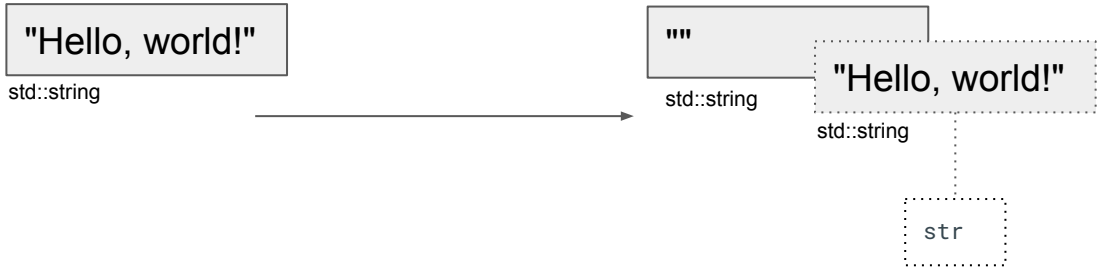
Copy construction requires an lvalue expression.

Move constructor requires an rvalue expression.

```
std::string one {"Hello, world!"};
std::string two = str; // copy construction
```



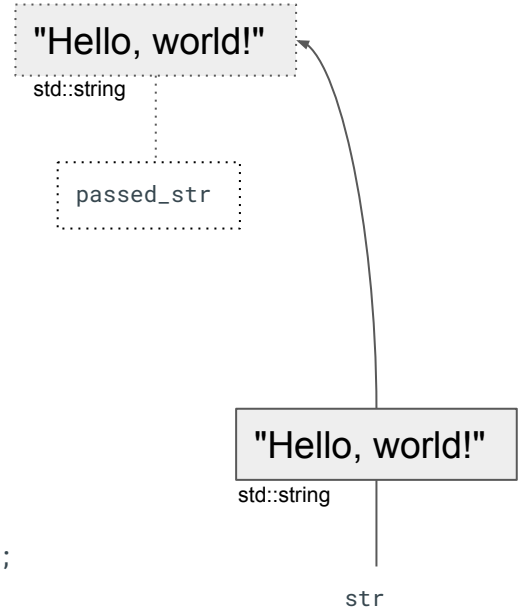

```
std::string str = std::string("Hello, world!"); // move construction
```



Passing-by-lvalue: copy

We already knew this! That's why we liked references.

```
void f(std::string passed_str) {
    // ...
}
```

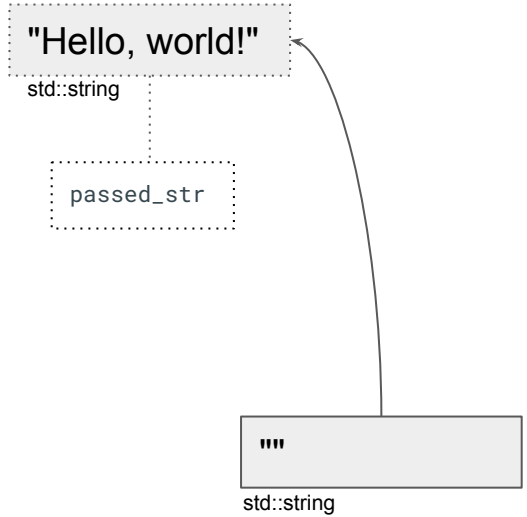



```
std::string str {"Hello, world!"};
f(str); // pass-by-lvalue => copy
```

Passing-by-rvalue: move

We probably thought of this intuitively: it is not necessary to copy “Hello, world”.

```
void f(std::string passed_str) {
    // ...
}
```




```
f(std::string("Hello, world!")); // pass-by-rvalue => move
```

C++
rvalue references

What are *rvalue references*?

- We already have seen (*lvalue) references* (T&)?
  - An lvalue reference would **T&** bind to an lvalue of (sub)type **T**.
    i. *Remember the exception*: a **const T&** would bind rvalues.

- An **rvalue reference**, is… a reference to an rvalue.
  - Binds rvalues, but does not bind (const) lvalues!
  - Notation: **T&&**
    i. Not a reference to a reference!

- Rvalue references extend rvalue lifetime, and facilitate moves.

Which of the following are valid?

```
// source: https://www.learncpp.com/cpp-tutorial/rvalue-references/
int x = 0;

int & ref1 = x;         // A
int & ref2 = 5;         // B

const int & ref3 = x;   // C
const int & ref4 = 5;   // D

int && ref5 = x;        // E
int && ref6 = 5;        // F

const int && ref7 = x; // G
const int && ref8 = 5; // H
```

Which of the following are valid?

```
// source: https://www.learncpp.com/cpp-tutorial/rvalue-references/
int x = 0;

int & ref1 = x;         // A - valid: lvalue -> T&
int & ref2 = 5;         // B - invalid: rvalues don't bind

const int & ref3 = x;   // C - valid: lvalue -> const T&
const int & ref4 = 5;   // D - valid: rvalue -> const T&

int && ref5 = x;        // E - invalid: lvalues don't bind
int && ref6 = 5;        // F - valid: rvalue -> T&&

const int && ref7 = x; // G - invalid: lvalues don't bind
const int && ref8 = 5; // H - valid: rvalue -> const T&&
```

All ideas for passing variables

rvalue references message "The referenced value is going to be thrown away soon."

```
/* 1: I'll copy that what you pass to me.
      - if const: As a programmer, I have no intentions to change it. */
void f([const] std::string foo) {
    ...
}

/* 2: I'll borrow your object.
 *    - if const: I'll promise not to change it! */
void f([const] std::string & foo) {
    ...
}

/* 3: Since you'll throw it away, I might steal from it!
 *    - const rarely useful */
void f([const] std::string && foo) {
    ...
}
```

C++

std::move & xvalues

Motivation

Sometimes, we make a copy, even when we don't need the variable after.

```
void print(std::string str) { ... }

void foo() {
    // 1. Suppose we have an lvalue, somehow.
    std::string text {"Hello, world!"};

    // 2. We have to give it away
    //    and this triggers a copy :(
    print(text);

    // 3. Too bad! We even didn't need `text`.
    return;
} // `text` deleted
```

Solution: std::move

std::move turns an lvalue into an rvalue reference.

```
void print(std::string str) { ... }

void foo() {
    // 1. Suppose we have an lvalue, somehow.
    std::string text {"Hello, world!"};

    // 2. Turning it into an rvalue reference,
    //    causes move construction when passing by value.
    print(std::move(text));

    // 3. `text` is still an object, but its contents are stolen
    return;
} // `text` deleted
```

Which f is called?

```
// https://en.cppreference.com/w/cpp/language/reference
void f(int & x) { std::cout << "lvalue reference"; }
void f(const int & x) { std::cout << "const lvalue reference”; }
void f(int && x) { std::cout << "rvalue reference"; }

int main() {
    f(3);           // A


    int i = 1;
    f(i);           // B
    f(std::move(i)); // C

    const int ci = 2;
    f(ci);          // D

    int && x = 1;
    f(x);           // E
    f(std::move(x)); // F
}
```

Which f is called?

```
// https://en.cppreference.com/w/cpp/language/reference
void f(int & x) { std::cout << "lvalue reference"; }
void f(const int & x) { std::cout << "const lvalue reference"; }
void f(int && x) { std::cout << "rvalue reference"; }

int main() {
    f(3);  // “rvalue reference”
           // would call f(const int &) if f(int &&) wasn't there

    int i = 1;
    f(i);            // “lvalue reference”
    f(std::move(i)); // “rvalue reference”

    const int ci = 2;
    f(ci);           // “const lvalue reference”

    int && x = 1;
    f(x);            // “lvalue reference”
    f(std::move(x)); // “rvalue reference”
}
```

(Value categories) Expression categories before C++11

lvalues
identity, not movable

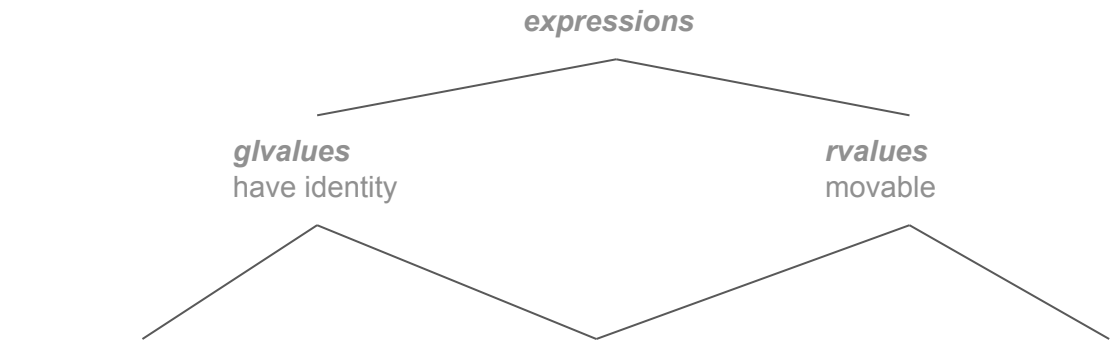
"Hello, world!"
std::string

one

rvalues
no identity, movable

"Hello, world!"
std::string

(Value categories) Expression categories after C++11



C++

Copying and moving objects

Classes: how to make your objects copyable and/or movable?

- Each class has the following implicitly defined members:
  - A default constructor `Point::Point()`
  - Copy constructor `Point::Point(const Point & p)`
  - Copy-assignment `Point & Point::operator=(const Point & p)`
  - Move constructor `Point::Point(Point && p)`
  - Move-assignment `Point & Point::operator=(const Point && p)`
  - Destructor `~Point()`

- Good news: the compiler generated these functions whenever behavior is unambiguous.
  - Not the case with *reference members*, or with *const* members.
  - **Rule of zero**: don't write custom copy/move unless your class exclusively manages a resource (memory, file, pointer, etc.).
  - **Rule of five:** if you have to overload/implement one, you have to implement all.

Classes: how to make your objects copyable and/or movable?

Source: https://blog.feabhas.com/2015/11/becoming-a-rule-of-zero-hero/

C++

Classes with value members

Classes: reference or value members?

```
class Airport {
protected:
    // values are good for "owning" or managing
    Building headquarters_; // composition

    // refs are good for borrowing and shared objects
    Airplane & plane1_;     // aggregation

    // C-style pointers can be used as either (but shouldn't)
    Airplane * plane2_;     // aggregation
}


class Car {
protected:
    Passenger driver_; // probably not a good idea
    Wheel & front_left_; // also not
}
```

Classes: reference or value members?

- Lvalue references:
  - Fastest way to initialize members (faster than move).
  - No default way of copying/moving the class anymore.
  - Class cannot guarantee that the reference is valid:
    i. Reference could go out of scope outside class
    ii. All memory has to be owned elsewhere (mostly in *main()* for now)

- Values:
  - Class “owns” the object and is responsible of its lifetime.
  - Requires using correct move semantics in order to be fast.

C++ programming

std::function

std::function

- The **std::function** is a type from the standard library that can hold *C-style functions*, *lambda functions*, as well as many other function-like types.
- C-style function:

```
float times_two(int a) {
    return a * 2.0;
}
...
int main() {
    // store `times_two` in a templated type `std::function<...>`
    auto f = std::function<float(int)>(times_two);
    integrate(std::move(f));
}
```

Lambda functions

- A **lambda function** is an alternative way of defining a function, in-line, without giving it a name.

```
#include <functional>

int main() {
    auto times_two = [](int x){ return x * 2.0; };
    auto f = std::function<float(int)>(times_two);
    integrate(f);
... }
```

Calling std::function

- A **std::function** that is received as a parameter can be called like an ordinary function:

```
double SomeClass::integrate(std::function<float(float)> f) {
    float a = f(4.0);
    ...
}

int main() {
    integrate([](float x){ return x * 9. });
}
```

This week

- Today / this week:
  - **1-Dimensional Monte Carlo methods**
    - Hit-or-miss and Simple-sampling Monte Carlo
    - More freedom for implementation
    - More standard library tools
- Next week:
  - **High-dimensional Monte Carlo**
    - Extend code to the high-dimensional case