Every-day idiomatic C++ 11

Passing data in and out of functions

- Good old C++ 98
- Move semantics quick recap
- Cheap returns in C++11
- Want speed? Pass by value!

Implementing regular types

- What's a regular type?
- Implementing move semantics
- Corner case: move assignment to self

Wil Evers
Core Systems Developer, Optiver, Amsterdam
Moderator, `comp.lang.c++.moderated`
`w.j.evers@versatel.nl`

Passing data in and out of functions: C++98 style

```
bool is_prime(int number);
bool is_valid(const std::string& name);

void trim(std::string& line);
    // modifies line in-place
std::string normalize(const std::string& input);
    // returns normalized copy of input

std::string email(const std::string& user,
    const std::string& domain);
void build_address_line(std::string& result,
    const std::string& street_name,
    const std::string& house_number);
```

- **In**: pass by value (small things), or pass by reference to const (big things)
- **In and out:** pass by reference to non-const, or separate output from input
- **Out:** use return by value or pass by reference to non-const

C++11: adding move semantics to the picture

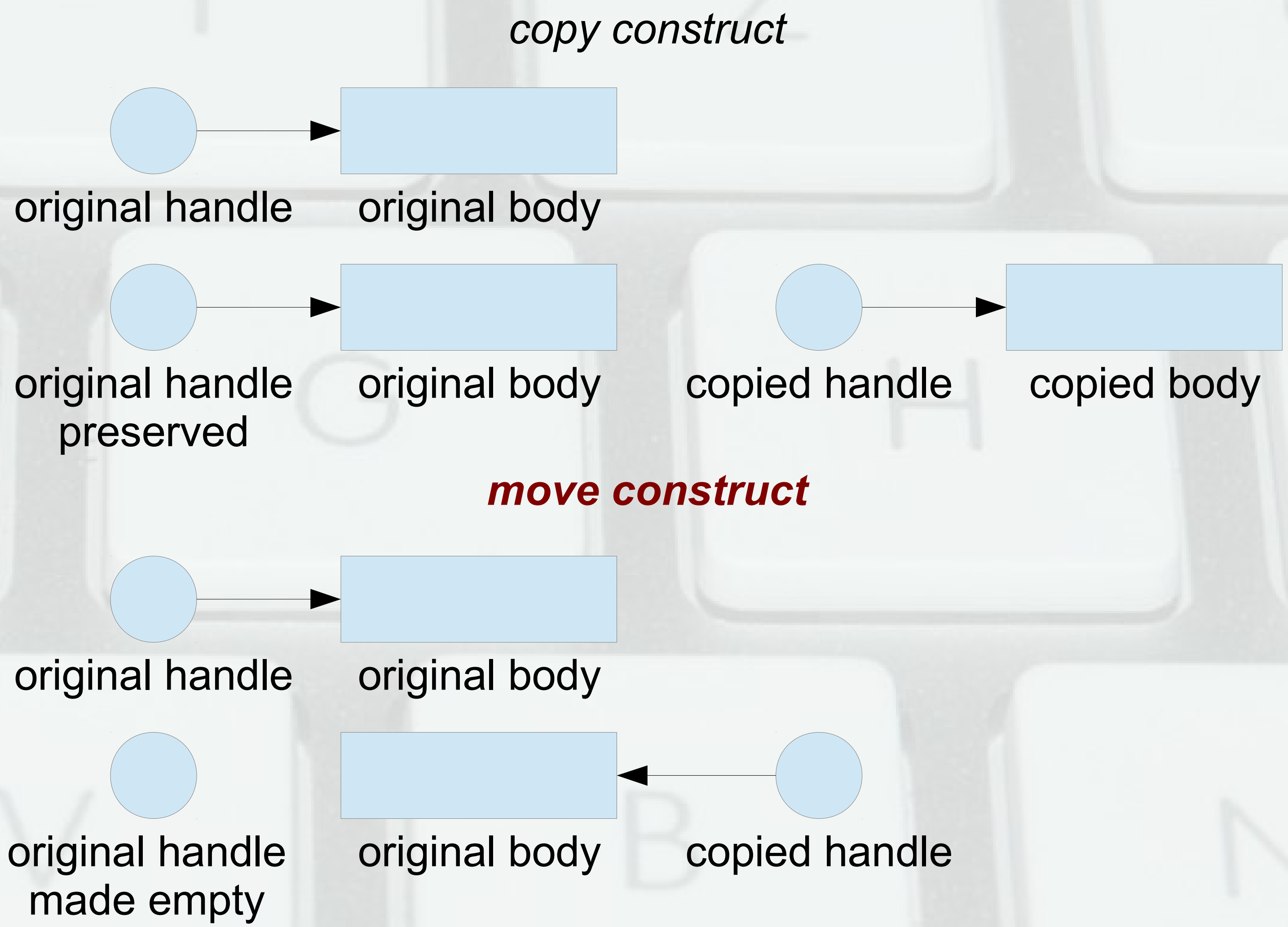
```
class C {
public :
    // other members omitted

    C(const C& rhs);                          // copy construct
    C(C&& rhs) noexcept;                      // move construct

    C& operator=(const C& rhs);               // copy assign
    C& operator=(C&& rhs) noexcept;           // move assign
};
```

- In C++11, we can add extra overloads to the copy constructor and copy assignment operator: the **move constructor** and **move assignment operator**.
- In general, these are called when preserving the value of the source argument (`rhs`) is not required.
- The compiler uses these overloads when the source argument is an rvalue, in a return statement, or by explicit user request (`std::move`).

Moving is often cheaper than copying



Return by value in C++11

```
bool is_prime(int number);
bool is_valid(const std::string& name);

void trim(std::string& line);
    // modifies line in-place
std::string normalize(const std::string& input);
    // returns normalized copy of input

std::string email(const std::string& user,
    const std::string& domain);
std::string build_address_line(
    const std::string& street_name,
    const std::string& house_number);
```

In C++11, returning by value uses the move constructor, so:

- `normalize()` and `email_address()` perform better
- `build_address_line()` can afford to return by value

A closer look at `normalize()`

```
std::string normalize(const std::string& input)
{
    std::string result = input;
    // ...manipulate result...
    return result;
}
```

As a first step, `normalize()` always copies its input, so we might as well take the input by value. `normalize()`'s private copy is now supplied at the point where it is called.

```
std::string normalize(std::string input)
{
    // ...manipulate input...
    return input;
}
```

Even for C++98, performance will be roughly the same.

Want speed? Pass by value!

```
std::string read_line();
std::string normalize_98(const std::string& input);
std::string normalize_11(std::string input);

void user()
{
    std::string line1 = normalize_98(read_line());
    std::string line2 = normalize_11(read_line());
    // ...
}
```

- Because `normalize_98()` takes its input by const reference, it must first make a copy before starting to manipulate it.
- In contrast, `normalize_11()` leaves it to the caller to provide the input object it will manipulate.
- If constructed from an anonymous, temporary rvalue, this object is move-constructed. It is never copied.
- Guideline: in C++11, functions that *copy their input arguments* should take these arguments *by value*.

Passing data in and out of functions: C++11 style

```
bool is_prime(int number);
bool is_valid(const std::string& name);

void trim(std::string& line);
    // modifies line in-place
std::string normalize(std::string input);
    // returns normalized from

std::string email(std::string user,
    const std::string& domain);
std::string build_address_line(
    std::string street_name,
    const std::string& house_number);
```

- **In**: pass by value if small **or copied**, otherwise pass by reference to const
- **In and out:** pass by reference to non-const, **prefer** to separate output from input
- **Out:** **prefer return by value** over pass by reference to non-const

Simple-minded email address class, C++11 style

```
class email_address {
public :
    email_address(std::string user, std::string domain)
    :  usr(std::move(user)), domn(std::move(domain))
    {  }
    const std::string& get_user() const
    {  return usr; }
    const std::string& get_domain() const
    {  return domn; }
    std::string full_text() const
    {  return usr + "@" + domn; }
    void set_user(std::string user)
    {  usr = std::move(user); }
    void set_domain(std::string domain)
    {  domn = std::move(domain); }
private :
    std::string usr;
    std::string domn;
};
```

What is a regular type?

The term regular type was introduced by Alexander Stepanov, the designer of the STL. We informally say: a regular type is a type that works well with the containers and algorithms in the standard library. In particular, objects of a regular type:

- Can be default-constructed.
- Can be copied.
  The copy gets the *same observable state* as the original, and the original's observable state is not changed. However, the observable state is *not shared*: the original and the copy can be manipulated independently.
- Can be assigned to.
  The object assigned to *gets the same observable state* as the object assigned from, and the observable state of the object assigned from is not changed. Again, the observable state *does not become shared*.
- Can be compared.
  Many STL algorithms assume a total ordering on the values of the objects. Usually, we can supply a separately defined comparator without changing definition of the type itself.
- Can be destructed.

XPensive: C++98-style regular type

```
class XPensive {
public :
    explicit XPensive(int size = 0);
    XPensive(const XPensive& rhs);
    XPensive& operator=(const XPensive& rhs);
    int size() const { return sz; }
    const char& at(int idx) const { return buf[idx]; }
    char& at(int idx) { return buf[idx]; }
    ~XPensive();

private :
    int sz;
    char *buf;
};
```

- C++98 *rule of three* for value classes that own resources: copy constructor, copy assignment operator and destructor all provided

XPensive implementation: constructors and destructor

```
XPensive::XPensive(int size)
:   sz(size),
    buf(sz ? new char[sz] : 0)
{
    std::fill(buf, buf + sz, 0);
}

XPensive::XPensive(const XPensive& rhs)
:   sz(rhs.sz),
    buf(sz ? new char[sz] : 0)
{
    std::copy(rhs.buf, rhs.buf + sz, buf);
}

XPensive::~XPensive()
{
    delete[] buf;
}
```

XPensive implementation: copy assignment operator

```
XPensive& XPensive::operator=(const XPensive& rhs)
{
    // prepare phase:
    // (throwing is fine, modification is bad)
    char *new_buf = new char[rhs.sz];
    std::copy(rhs.buf, rhs.buf + rhs.sz, new_buf);

    // commit phase:
    // (throwing is bad, modification is fine)
    delete[] buf;
    buf = new_buf;
    sz = rhs.sz;
    return *this;
}
```

- Please note that *self-assignment* is properly handled...
- ...and that we provide *the strong exception guarantee*

C++11: Adding move semantics to XPensive

```
class XPensive {
public :
    explicit XPensive(int size = 0);
    XPensive(const XPensive& rhs);
    XPensive(XPensive&& rhs) noexcept;
    XPensive& operator=(const XPensive& rhs);
    XPensive& operator=(XPensive&& rhs) noexcept;
    int size() const { return sz; }
    const char& at(int idx) const { return buf[idx]; }
    char& at(int idx) { return buf[idx]; }
    ~XPensive();

private :
    int sz;
    char *buf;
};
```

- C++11 *rule of five*: copy constructor, **move constructor**, copy assignment operator, **move assignment operator** and destructor all provided

Copy construction versus move construction

```
XPensive::XPensive(const XPensive& rhs)
:   sz(rhs.sz), buf(sz ? new char[sz] : 0)
{
    std::copy(rhs.buf, rhs.buf + sz, buf);
}
```

- The copy constructor *preserves* the value stored in rhs.

```
XPensive::XPensive(XPensive&& rhs) noexcept
:   sz(rhs.sz), buf(rhs.buf)
{
    rhs.sz = 0; rhs.buf = 0;
}
```

- The **move constructor** *does not need to preserve* rhs's value. It is assumed to be a temporary that will not be observed any more.
- Thus, rhs left in a *valid*, but otherwise *unspecified*, state.

Copy assignment versus move assignment

```
XPensive& XPensive::operator=(const XPensive& rhs)
{
    char *new_buf = new char[rhs.sz];
    std::copy(rhs.buf, rhs.buf + rhs.sz, new_buf);
    sz = rhs.sz;
    delete[] buf;
    buf = new_buf;
    return *this;
}

XPensive& XPensive::operator=(XPensive&& rhs) noexcept
{
    sz = rhs.sz;
    delete[] buf;
    buf = rhs.buf;
    rhs.sz = 0;
    rhs.buf = 0;
    return *this;
}
```

A closer look: what about move assignment to self?

```
XPensive& XPensive::operator=(XPensive&& rhs) noexcept
{
    sz = rhs.sz;          // integer self-assigned
    delete[] buf;         // buf becomes dangling pointer
    buf = rhs.buf;        // dangling pointer self-assigned
    rhs.sz = 0;           // sz set 0
    rhs.buf = 0;          // buf set to 0
    return *this;
}
```

- As implemented here, self-move-assigning an XPensive invokes undefined behavior, because it reads a dangling pointer. (Most current architectures tolerate this.)
- Furthermore, the object assigned to (which is *not* a temporary) does not obtain rhs's (that is, its own) observable state. Instead, it is left in the empty state.
- Therefore, as implemented here, XPensive is not a regular type.

Self move-assignment: why bother?

- Normally, rvalue reference parameters refer to temporary objects that will not be observed after the function returns. However, the function itself *is* allowed to observe the temporary:

```
void f(XPensive& dst, XPensive&& src)
{
    // ...observe src here...
    dst = std::move(src);
    // ...observe dst here...
}
```

- After the move, we assume:
  - *Nothing* about `src`. It is in an unspecified, but valid state.
  - That `dst` now has the observable state `src` had *before the move*.

- This should be true, *even if* `src` and `dst` refer to the same object.

XPensive's move assignment operator: second attempt

```
XPensive& XPensive::operator=(XPensive&& rhs) noexcept
{
    // phase 1
    int new_sz = rhs.sz;
    char *new_buf = rhs.buf;
    rhs.sz = 0;
    rhs.buf = 0;

    // phase 2
    sz = new_sz;
    delete[] buf;
    buf = new_buf;

    return *this;
}
```

- First, we obtain our new values, pilfering `rhs`, and leaving it in a valid state.
- We only commit our new values to `*this` when we're done with `rhs`.
- As a result, self-move-assignment just works!

Constructor and destructor reuse in the assignment operators

Suppose we had:

```
void XPensive::swap(XPensive& other)
{   std::swap(sz, other.sz); std::swap(buf, other.buf); }
```

Then our assigment operators could be simplified to:

```
XPensive& operator=(const XPensive& rhs)
{
    XPensive tmp(rhs);              // reuse copy ctor
    this->swap(tmp);
    return *this;
}

XPensive& operator=(XPensive&& rhs)
{
    XPensive tmp(std::move(rhs)); // reuse move ctor
    this->swap(tmp);
    return *this;
}
```

A unified assignment operator?

Remember pass by value? Here we go:

```
XPensive& operator=(XPensive rhs)
{
    this->swap(rhs);
    return *this;
}
```

This would be perfect: a single implementation that serves as *both* a copy- and a move assignment operator.

Unfortunately, while the standard recognizes this as a copy assignment operator, it doesn't recognize it as a move assignment operator. (Of course, it is both).

That means that classes embedding an XPensive as a data member will not obtain a default-generated move assignment operator. To obtain that, we need to provide two overloaded assignment operators.