C++: to copy, ref, or move?

Matthias Kretz

Frankfurt Institute for Advanced Studies
Institute for Computer Science
Goethe University Frankfurt

March 26, 2014

HIC for FAIR
Helmholtz International Center

FIAS Frankfurt Institute for Advanced Studies

HGS-HIRe for FAIR
Helmholtz Graduate School for Hadron and Ion Research

```
void f(std::vector<int>);
```


```
void f(std::vector<int> &);
void f(std::vector<int> *);
```


```
void f(const std::vector<int> &);
void f(const std::vector<int> *);
```


```
void f(std::vector<int> &&);
```

```
template <typename T>
void f(T &&);
```

```
template <typename T>
void f(T &&x) {
  g(std::forward<T>(x));
}
```

types that fit into a register

pass by value

- copy is very cheap
- reference requires the value to be in memory
- reference requires a register to point to the object

`int`, `float`, `double`, `void *`, ...

- `struct A { int x, y, z; };`
  fits in two registers.
- `struct B { float x, y, z; };`
  fits in two registers (`%xmm0[0]`, `%xmm0[1]`, `%xmm1`).

(depends on the ABI, I'm referring to the Linux x86_64 ABI)

local copies increase optimization

- The compiler knows exactly where the variables are modified
- Non-local variables could get modified by any function call (note that most function calling conventions guarantee the contents of several registers to stay unmodified)

complex types

pass by const reference

- especially types with dynamically allocated data
- copy of `std::string` or `std::vector<T>`:
  - malloc
  - std::copy
    calls `T::T(const T &)`
- reference requires:
  - the complete object to reside at one memory location
  - a pointer to this location
  - pass this pointer via register/stack

the alternative to copy and reference is

move

On rvalues and lvalues

C++ has two categories of values:

lvalue an object with a name (a variable)
originally: *left* hand side of an assignment

rvalue a temporary object (there's no name to access it)
originally: *right* hand side of an assignment

- xvalue: (eXpiring value) a value at the end of its lifetime
- prvalue: (pure rvalue) literals & function return values that are not a reference ("any rvalue that is not an xvalue")

On rvalues and lvalues

```
int x = 1;  // '1' is a prvalue
int y = x;  // 'x' is an lvalue
int z = std::move(x);  // 'x' is casted to
                       // an xvalue

 int &   // lvalue reference
 int &&  // rvalue reference
auto &&  // "universal reference" (Scott Meyers)
   T &&
```

```
void f(const A &);
void f(A &&);

...
{
   f(A{});
}
```

Calls ?

```
void f(const A &);
void f(A &&);

...
{
  f(A{});
}
```

Calls `f(A &&)`

movable types

if applicable, move instead of copy/reference

What is a movable type?

Or: what does a move constructor do?

```
class A {
  Data *data;
public:
  A() : data(new Data) {}
  ~A() { delete data; }
  A(const A &from) : data(new Data(*from.data)) {}
  A(A &&from) : data(from.data) {
    from.data = nullptr;
  }
};
```

What is a movable type?

Or: what does a move constructor do?

```
class A {
  Data *data;
public:
  A() : data(new Data) {}
  ~A() { delete data; }
  A(const A &from) : data(new Data(*from.data)) {}
  A(A &&from) : data(from.data) {
    from.data = nullptr;
  }
};
```

You don't need to write all this, thanks to `unique_ptr`.
This is equivalent:

```
class A {
  std::unique_ptr<Data> data;
public:
  A() : data(new Data) {}
};
```


In C++11 **string** and all containers (except **array**) are moveable types.

```
class A {
  string name;
public:
  A(const string &n)
    : name(n)
  {}
};
...
A x{"Hello World"};
```


In C++11 **string** and all containers (except **array**) are moveable types.

```
class A {
  string name;
public:
  A(const string &n)
    : name(n)
  {}
};
...
A x{"Hello World"};
```


In C++11 **string** and all containers (except **array**) are moveable types.

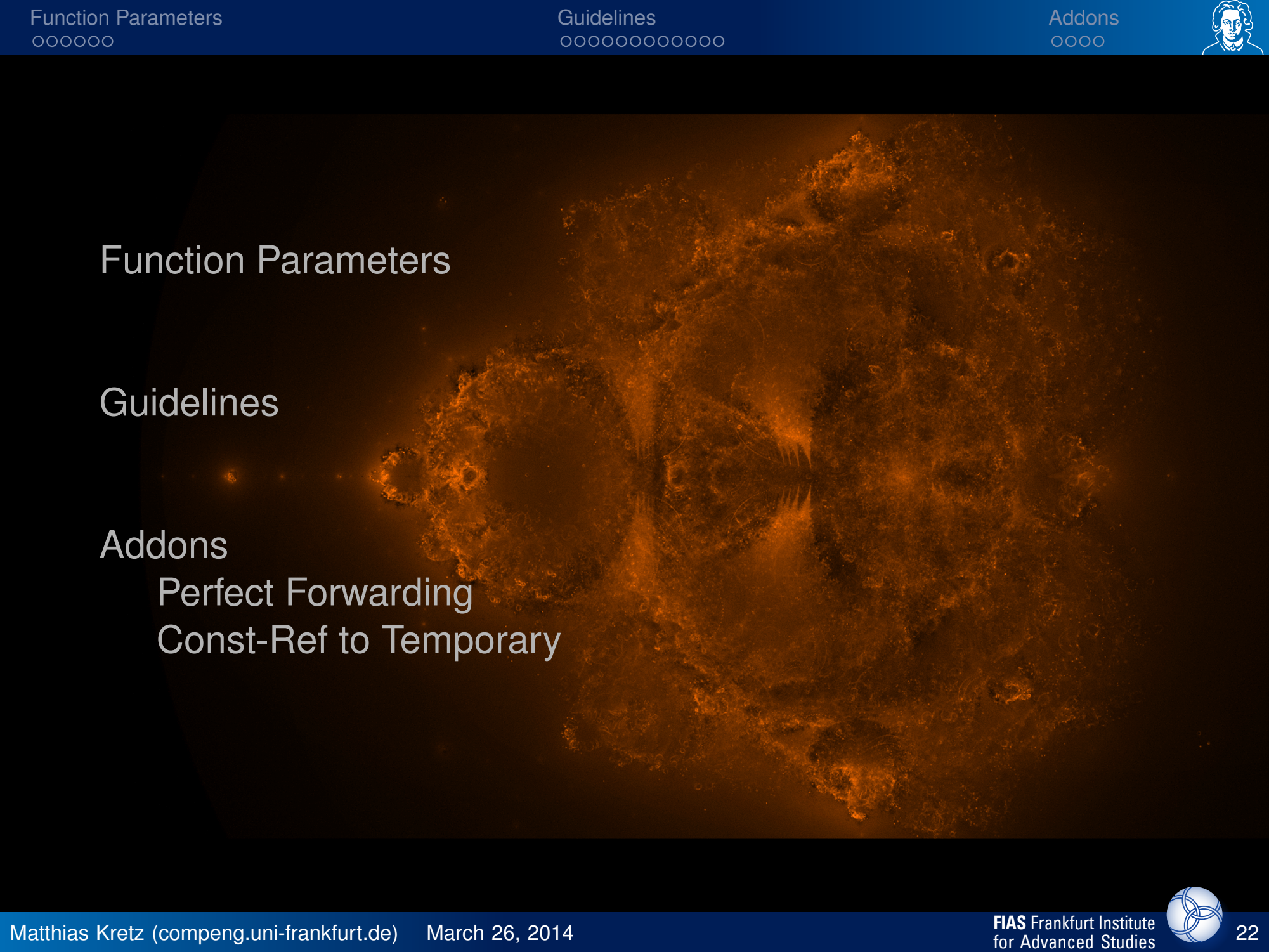
```
class A {
  const string &name;
public:
  A(const string &n)
    : name(n)
  {}
};
...
A x{"Hello World"};
```


In C++11 **string** and all containers (except **array**) are moveable types.

```
class A {
   string name;
public:
   A(string n)
      : name(n)
   {}
};
...
A x{"Hello World"};
```


In C++11 **`string`** and all containers (except **`array`**) are moveable types.

```
class A {
   string name;
public:
   A(string n)
      : name(std::move(n))
   {}
};
...
A x{"Hello World"};
```

Function Parameters

Guidelines

Addons
- Perfect Forwarding
- Const-Ref to Temporary

Perfect Forwarding

```
template <typename T> void f(T    x);
template <typename T> void f(T &  x);
template <typename T> void f(T &&x);

void h() {
  int x = 1;
  const int y = 1;
  f(x); // (1) lvalue
  f(y); // (2) const lvalue
  f(1); // (3) rvalue
}
```

Perfect Forwarding

```
template <typename T> void f(T    x);
template <typename T> void f(T &  x);
template <typename T> void f(T &&x);

void h() {
  int x = 1;
  const int y = 1;
  f(x); // (1) lvalue
  f(y); // (2) const lvalue
  f(1); // (3) rvalue
}
```

Perfect Forwarding

```
template <typename T> void f(T    x);
template <typename T> void f(T &  x);
template <typename T> void f(T &&x);

void h() {
  int x = 1;
  const int y = 1;
  f(x); // (1) lvalue
  f(y); // (2) const lvalue
  f(1); // (3) rvalue
}
```

std::forward

```
template <typename T> void f(T &&x) {
  g(x);
}
```

std::forward

```
template <typename T> void f(T &&x) {
  g(std::move(x));
}
```


std::forward

```
template <typename T> void f(T &&x) {
  g(std::forward<T>(x));
}
```


```
void f(const A &);

...
{
   f(A{});
}
```

Does this work?
And why?

```
void f(const A &);

...
{
   f(A{});
}
```

works: temporary exists until the end of the statement
(also: const-ref to temporary extends lifetime)

```
void f(A &);

...
{
   f(A{});
}
```

error: invalid initialization of non-const reference of type ’A &’ from an rvalue of type ’A’