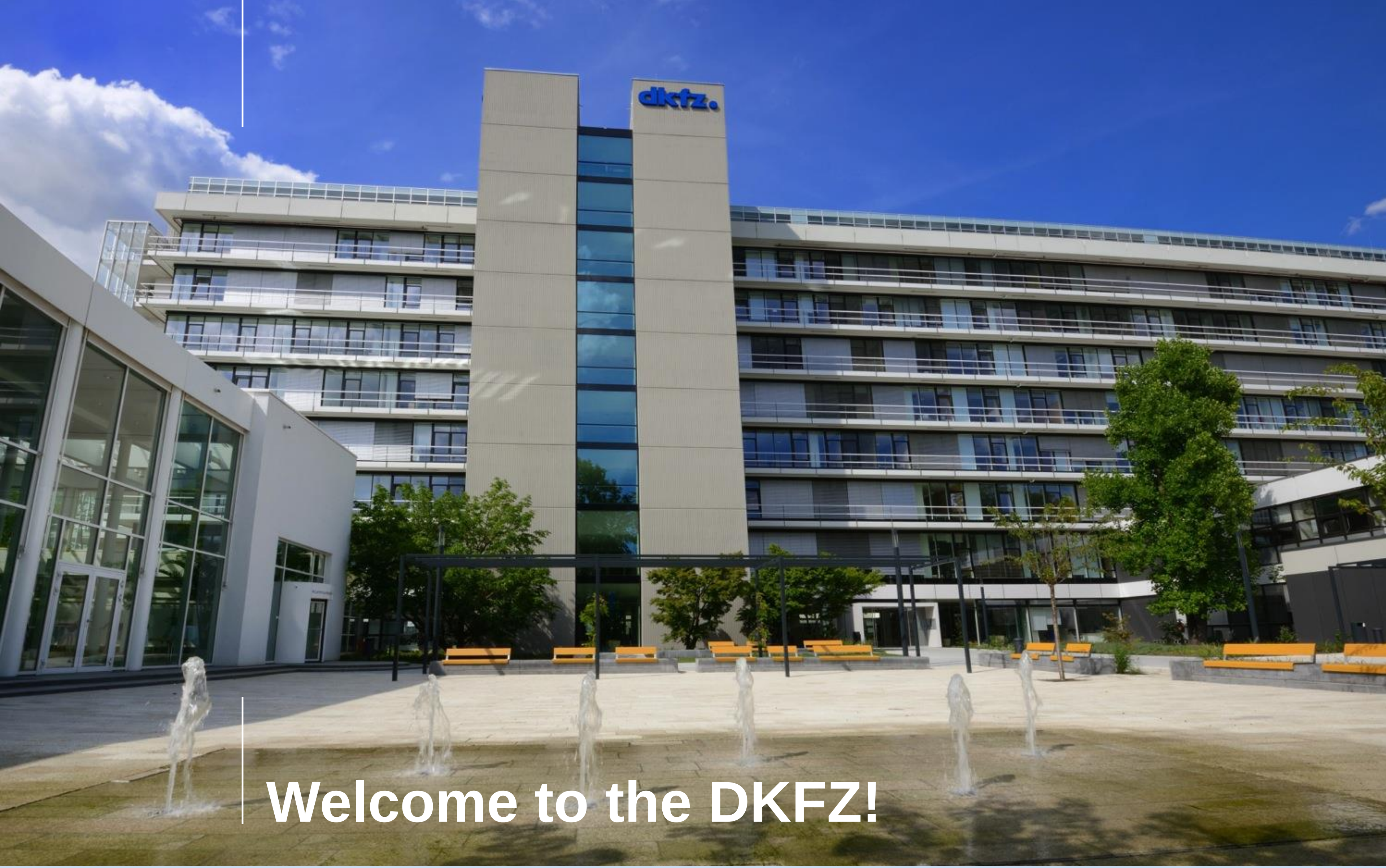
5/10/2016

# C++ 11: Move Semantics

Sandy Engelhardt
MITK Bugsquashing Seminar, March 2016

GERMAN CANCER RESEARCH CENTER IN THE HELMHOLTZ ASSOCIATION

50 Years – Research for A Life Without Cancer




# Welcome to the DKFZ!

dkfz. GERMAN CANCER RESEARCH CENTER IN THE HELMHOLTZ ASSOCIATION

50 Years – Research for
A Life Without Cancer

# Rvalue References I



- ***Rvalue references*** is a small technical extension to the C++ language
- allow programmers to **avoid logically unnecessary copying** and to provide **perfect forwarding functions**
- primarily meant to aid in the design of **higher performance** and more robust libraries.
- a new category of reference variables for **unnamed objects** (temporaries)
- typical examples of *unnamed objects* are **return values of functions** or **type-casts**

# Rvalue References II



```
// standard C++ lvalue reference variable
std::string& ref;

// C++11 rvalue reference variable
std::string&& rrstr;
```

- **lvalue** – may appear on the left hand side of an assignment, represents storage region locator value
- all the rest is **non-lvalue** or **rvalue** (because can appear on the right hand side of assignment only)

An rvalue reference behaves just like an lvalue reference except that it *can* bind to a temporary (an rvalue), whereas you can not bind a (non const) lvalue reference to an rvalue.

# Copying Objects



- Hitherto, **copying** has been the only means for transferring a state from one object to another

(* state of object = collective set of its non-static data members' values)

- Formally, copying causes a **target object t** to end up with the same state as the **source s**, without modifying s.

- Using the value of a temporary object e.g. these to **initialize another object** or to **assign its value**, does not require a copy:

the object is never going to be used for anything else, and thus, its value can be ***moved into*** the destination object.

# Example: Copy

```
string func()
{
  string s;
  //do something with s here
  return s;
}
string mystr=func();
```

1. When `func()` returns, C++ constructs a temporary copy of `s` on the caller's stack memory.
2. `s` is destroyed and the temporary is used for **copy-constructing** `mystr`
3. the temporary itself is destroyed

# Copy vs. Move



- **Copying a string** requires the allocation of dynamic memory and copying the characters from the source.
- **Moving** (new!) achieves the same effect without so many copies and destructor calls along the way.
- **Moving a string** is almost free; it merely assigns the values of the source's data members to the corresponding data members of the target.
- **Move operations** *tend to be faster* than copying because they transfer an existing resource to a new destination, whereas copying requires the creation of a new resource from scratch.

# An addition to the fabulous 4…

- Default constructor
- Copy constructor
- Copy assignment operator
- Destructor

- C++11 introduces two new special member functions: the ***move constructor*** and the ***move assignment operator***.

- If a class doesn't have any user-declared special member functions, C++ declares its remaining five (or six) special member functions implicitly, e.g.

```
class S{};
```

# Move Constructor



- A move constructor looks like this:

```
//C++11 move constructor
MyClass::MyClass(MyClass&& other);
```

- It doesn't allocate new resources. Instead, it *pilfers* other's resources and then sets other to its default-constructed state.

- The move constructor ***is much faster than*** a copy constructor because:
  - it doesn't allocate memory
  - It doesn't copy memory buffers

# Example Move Constructor

```
class MemoryPage
{
size_t size;
char * buf;

public:

//constructor
explicit MemoryPage(int sz=512):
size(sz), buf(new char [size]) {}

//destructor
~MemoryPage(){delete[] buf;}

//C++03 copy constructor
MemoryPage(const MemoryPage&);

//C++03 assignment operator
MemoryPage& operator=(const MemoryPage&);
};
```

```
//C++11 move constructor
MemoryPage(MemoryPage&& other):
size(0), buf(nullptr)
{
  // pilfer other's resource
  size=other.size;
  buf=other.buf;

  // reset other
  other.size=0;
  other.buf=nullptr;
}

//C++11 move assignment operator
MemoryPage&
MemoryPage::operator=(MemoryPage&&
other)
{…}
```

# Examples



```
// function returning a MemoryPage object
MemoryPage fn();

// default constructor
MemoryPage foo;

// copy constructor
MemoryPage bar = foo;

// move constructor
MemoryPage baz = fn();

// copy assignment
foo = bar;

// move assignment
baz = MemoryPage();
```

# STL



- The overload resolution rules of C++11 were modified to support rvalue references

- **Standard Library functions** such as `vector::push_back()` now define two overloaded versions:

| | |
|---|---|
| `const T&` | for lvalue arguments |
| `T&&` | for rvalue arguments |

- All containers are “move-aware”
- New algorithms: `move, move_forward`
- New iterator adaptor: `move_iterator`
- Optimized `swap` for movable types

# Hands-on Example



- The following program populates a vector with MemoryPage objects using two `push_back()` calls:

```
#include <vector>
using namespace std;

int main()
{
  vector<MemoryPage> vm;

  vm.push_back(MemoryPage(1024));
  vm.push_back(MemoryPage(2048));
}
```

→ arguments are rvalues: `push_back(T&&)` is called

push_back(T&&) moves the resources from the argument into vector's internal MemoryPage objects using MemoryPage's move constructor. In older versions of C++, the same program *would* generate copies of the argument since the copy constructor of MemoryPage would be called instead.

```
#include <vector>
using namespace std;

int main()
{
  vector<MemoryPage> vm;

  MemoryPage mp1(1024);
  vm.push_back(mp1);
}
```

→ arguments are lvalues: `push_back(const T&)` is called

# lvalue --- to --- rvalue



- You can enforce the selection of `push_back(T&&)` by casting an lvalue to an rvalue reference using **static_cast**:

```
// calls push_back(T&&)
vm.push_back(static_cast<MemoryPage&&>(mp));
```

- Alternatively, use the new standard function **std::move()** for the same purpose:

```
// calls push_back(T&&)
vm.push_back(std::move(mp));
```

# What happens to a *moved-from* object?




- state is unspecified, though valid !

assumption:

- object no longer owns any resources and its state is similar to that of an empty (as if default-constructed) object (though valid)

# Conclusion



- It may seem as if `push_back(T&&)` is always the best choice because it **eliminates unnecessary copies**.

> Remember that `push_back(T&&)` **empties its argument**. If you want **the argument to retain its state** after a `push_back()` call, stick to copy semantics.

- Generally speaking, don’t rush to throw away the copy constructor and the copy assignment operator !!

# References

- http://blog.smartbear.com/c-plus-plus/c11-tutorial-introducing-the-move-constructor-and-the-move-assignment-operator/
- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
- http://de.slideshare.net/oliora/hot-c-rvalue-references-and-move-semantics
- https://www.youtube.com/watch?v=IOkgBrXCtfo

  http://www.nosid.org/cxx11-about-move-semantics.html
- http://www.cplusplus.com/doc/tutorial/classes2/

# Thank you
# for your attention!

**Further information on www.dkfz.de**

# Further reading: Move Assignment Operator



- `C& C::operator=(C&& other);//C++11 move assignment operator`

A move assignment operator is similar to a copy constructor except that before pilfering the source object, it releases any resources that its object may own. The move assignment operator performs four logical steps:

1. Release any resources that (*this) currently owns.
2. Pilfer other‘s resource.
3. Set other to a default state.
4. Return (*this).

# Further reading: Example Move Assignment Operator



- ```
  //C++11 move assignment operator
  MemoryPage& MemoryPage::operator=(MemoryPage&& other)
  {
    if (this!=&other)
    {
        // release the current object's resources
        delete[] buf;
        size=0;
  ```

- ```
        // pilfer other's resource
        size=other.size;
        buf=other.buf;
        // reset other
        other.size=0;
        other.buf=nullptr;
    }
   return *this;
  }
  ```