



# Programmazione

Prof. Marco Bertini

[marco.bertini@unifi.it](mailto:marco.bertini@unifi.it)

<http://www.micc.unifi.it/bertini/>

---



# C++ | I language extensions



# General features

---



# auto type specifier

- To store the result of an expression in a variable we need to know the type of the expression...
- ...sometimes it's very verbose or hard to guess !
- just let the compiler deduce the type with the auto keyword:

```
auto x = expression;
```

e.g.:

```
auto y = val1 + val2;  
auto z = doSomething();
```

---



# auto type specifier

- To store the result of an expression in a variable we need to know the type of the expression...  
E.g. when dealing with templates, like STL classes
- ...sometimes it's very verbose or hard to guess !
- just let the compiler deduce the type with the auto keyword:  

```
auto x = expression;
```

e.g.:

```
auto y = val1 + val2;  
auto z = doSomething();
```



# auto type specifier

Auto forces the initialization of a variable (otherwise it wouldn't be able to guess the type...):

```
auto x1; // does not compile
int x1; // OK for the compiler
```

- just let the compiler deduce the type with the auto keyword:

```
auto x = expression;
```

e.g.:

```
auto y = val1 + val2;
auto z = doSomething();
```

---



# auto type - cont.

- auto ignores `const`-ness of types (but not the `const`-ness of pointed types, i.e. a pointer to `const`):

```
const int ci = i
auto b = ci; // b is an int
// (top-level const in ci is dropped)
```

- If we want to keep the `const`-ness ask for it:

```
const auto f = ci;
// deduced type of ci is int;
// f has type const int
```

---



# auto type - cont.

- We can also ask for a auto reference:

```
auto& g=ci; // g is a const int&  
           // that is bound to ci
```

- As with any other type specifier, we can define multiple variables using auto. Because a declaration can involve only a single base type, the initializers for all the variables in the declaration must have types that are consistent with each other
-





# auto and complex declarations

- auto simplifies complex declarations such as iterators, STL containers, smart pointers:

```
std::shared_ptr<some_type_t> mySmartPointer =  
    std::make_shared<some_type_t>();  
auto mySmartPointer = std::make_shared<some_type_t>();
```

```
for (std::map<std::string, std::map<std::string,  
int>>::iterator mapIter = myContainer.begin();  
mapIter != myContainer.end(); mapIter++)  
for (auto mapIter = myContainer.begin(); mapIter !=  
myContainer.end(); mapIter++)  
for (auto const &iter : myContainer)
```



# auto and return types

- Function declarations may be hard to read:  
`int (*func(int i))[10];`
- Under the new standard, another way to simplify the declaration of func is by using a **trailing return type**:

```
// func takes an int argument  
// and returns a pointer to an  
// array of ten ints  
auto func(int i) -> int(*)[10];
```

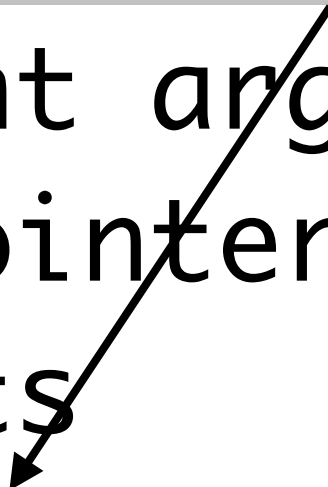


# auto and return types

- Function declarations may be hard to read:  
`int (*func(int i))[10];`
- Under the new standard, another way to simplify the declaration of func is by using a **trailing return type**:

Lambda functions use this syntax

```
// func takes an int argument  
// and returns a pointer to an  
// array of ten ints  
auto func(int i) -> int(*)[10];
```





# Auto and Lambda

- C++ 11 lets you store lambda expressions in named variables in the same manner you name ordinary variables and functions. This enables you to use the lambda expression multiple times in different places without having to copy the code all the time.

```
auto func_mult = [](int a, int b) ->
                  int { return a * b; };
std::cout << func_mult(2, 3) << std::endl;
for_each( container.begin(),
          container.end(), func_mult);
```

- It's an alternative to using a function object...



# Lambda

- A lambda is just an object and, like other objects it may be copied, passed as a parameter, stored in a container, or an `auto` variable.
- The lambda object has its own scope and lifetime which may, in some circumstances, be different to those objects it has “captured” (the parameters within `[]` ). Be very careful when capturing local objects by reference because a lambda’s lifetime may exceed the lifetime of its capture list.  
I.e. the lambda may have a reference to an object no longer in scope; in this case capture by value (`[=]` captures everything by value, `[i]` captures `i` by value)



# Lambda

In practice the compiler creates a functor

- A lambda is just an object and, like other objects it may be copied, passed as a parameter, stored in a container, or an `auto` variable.
- The lambda object has its own scope and lifetime which may, in some circumstances, be different to those objects it has “captured” (the parameters within `[]` ). Be very careful when capturing local objects by reference because a lambda’s lifetime may exceed the lifetime of its capture list.  
I.e. the lambda may have a reference to an object no longer in scope; in this case capture by value (`[=]` captures everything by value, `[i]` captures `i` by value)



# Lambda - cont.

- Captures (that basically provide the context of the Lambda, like data members in a class) can be by value or reference, with defaults:
    - `[&]() { i = 0; j = 0; }` is a lambda that captures `i` and `j` by reference. `[&]` means ‘capture by-reference all variables that are in use in the function’
    - `[=]() { cout << k; }` is a lambda that captures `k` by value. Similarly, `[=]` means ‘capture by-value all variables that are in use in the function’
    - You can also mix and match: `[&, i, j]() {}` captures all variables by reference except for `i` and `j` which are captures by value. And of-course the opposite is also possible: `[=, &i, &j]() {}`.
-



# Lambda - cont.

- lambda's operator() (i.e. the code of the Lambda) is `const` by-default, meaning it can't modify the variables it captured by-value (which are analogous to class members). To change this default add `mutable`:

```
int i = 1;
```

```
[&i]() { i = 1; }; // ok, 'i' is captured  
// by-reference.
```

```
[i]() { i = 1; }; // ERROR: assignment of  
// read-only variable 'i'.
```

```
[i]() mutable { i = 1; }; // ok.
```





# Lambda - cont.

- A Lambda defined within a class method can access all the class data members if it captures the pointer to the class (`this`)
  - Otherwise it is simply another (separate) class and does not have any access to the members of the including class
-



# decltype type specifier

- Sometimes we want to define a variable with a type that the compiler deduces from an expression but do not want to use that expression to initialize the variable.
  - For such cases use `decltype`, which returns the type of its operand.
  - The compiler analyzes the expression to determine its type but does not evaluate the expression.
-



# decltype type - cont.

- `decltype(f()) sum = x;`  
`// sum has whatever type f returns`
- Differently from `auto`, when the expression to which we apply `decltype` is a variable, `decltype` returns the type of that variable, including top-level `const` and references:

```
const int ci = 0, &cj = ci;  
decltype(ci) x = 0; // x has type const int  
decltype(cj) y = x; // y has type const int&  
                  // and is bound to x
```



# decltype type - cont.

- When we apply decltype to an expression that is not a variable, we get the type that that expression yields.
  - some expressions will cause decltype to yield a reference type.
  - Practically, decltype returns a reference type for expressions that yield objects that can stand on the left-hand side of the assignment
-



# decltype type - cont.

- The dereference operator `*` is an example of an expression for which `decltype` returns a reference:
  - when we dereference a pointer, we get the object to which the pointer points.  
Moreover, we can assign to that object.
- ```
int* p;  
decltype(*p) j; // j is int&  
                // not plain int
```



# decltype and return types

- `int odd[] = {1,3,5,7,9};`  
`// returns a pointer to an`  
`// array of five int elements`  
`decltype(odd) *arrPtr(int i)`
  - The type returned by `decltype` is an array type, to which we must add a `*` to indicate that `arrPtr` returns a pointer.
-



# decltype and return types

- The trailing return type syntax is really about scope:

```
auto mul(int x, int y) -> decltype(x*y)
{
    return x*y;
}
```



# decltype and return types

We use the notation `auto` to mean “return type to be deduced or specified later.”

```
auto mul(int x, int y) -> decltype(x*y)
{
    return x*y;
}
```

---





# decltype and return types

We use the notation `auto` to mean “return type to be deduced or specified later.”

```
auto mul(int x, int y) -> decltype(x*y)
{
    return x*y;
}
```

x and y are in scope only after their declaration



# decltype and return types

We use the notation `auto` to mean “return type to be deduced or specified later.”

Also for templates:

```
template<typename T, typename U>  
auto add(T x, U y) -> decltype(x+y) {  
    return x+y;  
}
```



# Uniform initialization

- Before C++11 there were different ways to initialize objects, and some syntaxes that looked like initializations were declarations...
- ... easy to misuse, resulting in error messages:

```
string a[] = { "foo", " bar" };  
// ok: initialize array variable  
void f(string a[]);  
f( { "foo", " bar" } );  
// syntax error: block as argument  
int a(1); // variable definition  
int b(); // function declaration  
int b(foo); // variable definition or  
// function declaration
```



# Uniform initialization

- The C++11 solution is to allow `{}`-initializer lists for all initialization:

```
X x1 = X{1,2};
```

```
X x2 = {1,2}; // the = is optional
```

```
X x3{1,2};
```

```
X* p = new X{1,2};
```

```
class D : public X {  
    D(int x, int y):X{x,y} { /*...*/ };  
};
```

---



# Uniform initialization

Moreover:

```
{ } does not allow narrowing conversions:  
long double ld = 3.1415926536;  
int c(ld), d = ld;  
// ok: but value will be truncated  
int a{ld}, b = {ld};  
// error: narrowing conversion required
```

Prefer initializing using `{}`, including especially everywhere that you would have used `()` parentheses when constructing an object, prefer using `{ }` braces instead.



# In-class member initializers

- Java programmers have always used it, at last it is possible to initialize data members within a class declaration in C++11:

```
class A {  
public:  
    A() {}  
    A(int value) : a(value) {}  
private:  
    int a = 4; // alternatively: int a {4};  
    float b = 3.14;  
    std::string s = "hello";  
};
```

---



# Strongly-typed enums

- “Traditional” enums in C++ have some drawbacks:
  - they export their enumerators in the surrounding scope (which can lead to name collisions, if two different enums in the same scope define enumerators with the same name),  

```
enum Alert {green, yellow, red};  
enum Color {red, green, blue}; // error: redefinitions
```
  - they are implicitly converted to integral types (e.g., `int a = red;`) and
  - cannot have a user-specified underlying type.
- C++11 strongly-typed enums are specified with the `enum class` keywords.
  - They no longer export their enumerators in the surrounding scope,
  - They are no longer implicitly converted to integral types and
  - can have a user-specified underlying type (a feature also added for traditional enums).

```
enum class Options {None, One, All};  
Options o = Options::All;
```

---



# std::function

- Callable object is a generic name for any object that can be called like a function:
    - A member function (pointer)
    - A free function (pointer) - also in C language, e.g. pointer-to-function used in qsort
    - A functor
    - A lambda
  - All these objects have different signatures. A way to have a uniform syntax to use them, is to wrap them within the `std::function` template function
-





# std::function

- Callable object is a generic name for any object that can be called like a function:
  - A member function (pointer)

```
std::function< Return Type(ParameterList) >
```

The callable object must have the signatures that says that it returns a `Return Type` and gets as parameters the types of the `ParameterList`

- All these objects have different signatures. A way to have a uniform syntax to use them, is to wrap them within the `std::function` template function
-



# std::function

```
#include <functional>

class SimpleCallback {
public:
    SimpleCallback(
        std::function<void(void)> f
    ) : callback(f) {}
    void execute();

private:
    std::function<void(void)>
        callback;
};

void SimpleCallback::execute() {
    if (callback != nullptr)
        callback(); // like a function
}
```

```
void func() {
    // free function
}

SimpleCallback cb1(func);
cb1.execute();

SimpleCallback cb2(
    []() { /* lambda */ }
);
cb2.execute();
```



# std::function

```
#include <functional>

class SimpleCallback {
public:
    SimpleCallback(
        std::function<void(void)> f
    ) : callback(f) {}
    void execute();

private:
    std::function<void(void)>
        callback;
};

void SimpleCallback::execute() {
    if (callback != nullptr)
        callback(); // like a function
}
```

```
void func() {
    // free function
}

SimpleCallback cb1(func);
cb1.execute();

SimpleCallback cb2(
    []() { /* lambda */ }
);
cb2.execute();
```

This is a way to reuse a Lambda, instead of using a `auto` variable we wrap the lambda in `std::function`



# std::function

std::function can be used to wrap also Functors

```
#include <functional>

class SimpleCallback {
public:
    SimpleCallback(
        std::function<void(void)> f
    ) : callback(f) {}
    void execute();

private:
    std::function<void(void)>
        callback;
};

void SimpleCallback::execute() {
    if (callback != nullptr)
        callback(); // like a function
}
```

```
void func() {
    // free function
}

SimpleCallback cb1(func);
cb1.execute();

SimpleCallback cb2(
    []() { /* lambda */ }
);
cb2.execute();
```

This is a way to reuse a Lambda, instead of using a auto variable we wrap the lambda in std::function



# std::function

std::function can be used to wrap also Functors

```
#include <functional>

class SimpleCallback {
public:
    void execute() {
        callback();
    }
};

void SimpleCallback::execute() {
    if (callback != nullptr)
        callback(); // like a function
}

void func() {
    // free function
}

std::function<void(void)> f;
f = []() { /* lambda */ }; // could be free
                          // function or functor

f();
```

This is a way to reuse a Lambda, instead of using a auto variable we wrap the lambda in std::function



# Move semantics / &&

---



# Ivalue

- An **Ivalue** is an expression that yields an object or function.
  - The name is an old C mnemonic that means that **Ivalues** could stand on the left-hand side of an assignment
  - In C++ not all **Ivalues** can stay on the left-hand side though: a `const` object can not...
-



# rvalue

- An **rvalue** is an expression that yields a value but not the associated location of the value.
- We can say that an **rvalue** is an unnamed value that exists only during the evaluation of an expression. E.g.:

`x+(y*z);`

- C++ creates a temporary (an **rvalue**) to store `y*z`, then adds it to `x`. The rvalue disappears when `;` is reached.
-





# rvalue

- An **rvalue** is an expression that yields a value but not the associated location of the value.
- We can say that an **rvalue** is an unnamed value that exists only during the evaluation of an expression. E.g.:

$x+(y*z);$

rvalues are objects that are about to be destroyed

- C++ creates a temporary (an **rvalue**) to store  $y*z$ , then adds it to  $x$ . The rvalue disappears when  $;$  is reached.



# lvalue and rvalue

- lvalues are locations, rvalues are actual values.  
An lvalue is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue.

```
int a = 42;
```

- a is lvalue, there's a location called a, we can get &a
  - 42 is a rvalue, there's no location for it
-



Because rvalues are short-lived, you have to capture them in lvalues if you wish to access them outside the context of their expression

any expression that evaluates to an lvalue reference (e.g., a function call, an overloaded assignment operator, etc.) is an lvalue.

Any expression that returns an object by value is an rvalue.

location and allows us to take the address of that memory location via the & operator. An rvalue is an expression that is not an lvalue.

```
int a = 42;
```

- a is lvalue, there's a location called a, we can get &a
- 42 is a rvalue, there's no location for it



# lvalue references

- C++ references are lvalue references...
- ... a reference is an alias of an object, i.e. an alternative name of an object.

```
int i = 42;  
int& ri = i;
```



# rvalue references

- C++11 has introduced rvalue references
- An rvalue reference is bound to an rvalue
- rvalue references may be bound only to an object that is about to be destroyed
- We use `&&` instead of `&`

```
int&& rr = i * 42;
```



# rvalues are ephemeral

- Because rvalue references can only be bound to temporaries, we know that
    - The referred-to object is about to be destroyed
    - There can be no other users of that object
  - These facts together mean that code that uses an rvalue reference is free to take over resources from the object to which the reference refers.
-



# rvalues are ephemeral

A variable is an lvalue; we cannot directly bind an rvalue reference to a variable even if that variable was defined as an rvalue reference type.

```
int &&rr1 = 42; // ok: literals are rvalues
```

```
int &&rr2 = rr1; // error: the expression rr1  
                // is an lvalue!
```

- These facts together mean that code that uses an rvalue reference is free to take over resources from the object to which the reference refers.
-



# rvalues are ephemeral

A variable is an lvalue; we cannot directly bind an rvalue reference to a variable even if that variable was defined as an rvalue reference type.

```
int &&rr1 = 42; // ok: literals are rvalues
```

```
int &&rr2 = rr1; // error: the expression rr1  
                // is an lvalue!
```

• These facts together mean that code that uses C++11 lets you bind rvalue references to rvalues, effectively prolonging their lifetime as if they were lvalues  
resources from the object to which the reference refers.

---





# lvalue/rvalue overload

- When a function has both rvalue reference and lvalue reference overloads, the rvalue reference overload binds to rvalues, while the lvalue reference overload binds to lvalues:

```
#include <iostream>
#include <utility>
void f(int& x) {
    std::cout << "lvalue reference overload f(" << x << ")\n";
}
void f(const int& x) {
    std::cout << "lvalue reference to const overload f(" << x << ")\n";
}
void f(int&& x) {
    std::cout << "rvalue reference overload f(" << x << ")\n";
}
```



# lvalue/rvalue overload

```
int main() {
    int i = 1;
    const int ci = 2;
    f(i); // calls f(int&)
    f(ci); // calls f(const int&)
    f(3); // calls f(int&&)
           // would call f(const int&) if
           // f(int&&) overload wasn't provided
    f(std::move(i)); // calls f(int&&)
}
```



# rvalue reference and move

- We can obtain an rvalue reference bound to an lvalue by calling a new library function named `std::move`, which is defined in the `utility` header.
- The `move` function returns an rvalue reference to its given object.

```
int&& rr1 = 42; // ok: literals are rvalues
int&& rr3 = std::move(rr1); // ok, even if
                               // rr1 is an lvalue
```



# std::move - effects

- Calling `std::move` tells the compiler that we have an lvalue that we want to treat as if it were an rvalue. A call to `move` promises that we do not intend to use the lvalue again except to assign to it or to destroy it.  
After a call to `move`, we cannot make any assumptions about the value of the moved-from object.
  - We can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved-from object.
-



# std::move - effects

`std::move(x)` is just a cast that means “you can treat `x` as an rvalue”.

- Calling `std::move` tells the compiler that we have an lvalue that we want to treat as if it were an rvalue. A call to `move` promises that we do not intend to use the lvalue again except to assign to it or to destroy it.  
After a call to `move`, we cannot make any assumptions about the value of the moved-from object.
  - We can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved-from object.
-



# move vs. copy - why ?

- In many real-world scenarios, you don't copy objects but move them.
  - When paying (cash or electronic), we move money from our account into the seller's account. Similarly, removing the SIM card from your mobile phone and installing it in another mobile is a move operation, and so are cutting-and-pasting icons on your desktop, or borrowing a book from a library.
-



# move vs. copy - why ?

- In many real-world scenarios, you don't copy objects but move them.

`unique_ptr` can be moved with `std::move`, they can not be copied (or they would not be the unique owners of a resource...

account. Similarly, removing the SIM card from your mobile phone and installing it in another mobile is a move operation, and so are cutting-and-pasting icons on your desktop, or borrowing a book from a library.

---



# rvalue reference - why ?

- Copying has been the only means for transferring a state from one object to another (an object's state is the 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***.
-





# Useless copy - example

```
template <class T>
swap(T& a, T& b) {
    T tmp(a); // now we have
                // two copies of a
    a = b;     // now we have
                // two copies of b
    b = tmp;   // now we have
                // two copies of tmp (aka a)
}
```



# rvalue reference - why ?

- 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.



# rvalue reference - why ?

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

When `func()` returns, C++ constructs a temporary copy of `S` on the caller's stack memory. Next, `S` is destroyed and the temporary is used for copy-constructing `mystr`. After that, the temporary itself is destroyed. Moving achieves the same effect without so many copies and destructor calls along the way.



# move constructors and assignment

- In C++11, we can define “move constructors” and “move assignments” to move rather than copy their argument.
  - The idea behind a move assignment is that instead of making a copy, it simply takes the representation from its source and replaces it with a cheap default.
  - The compiler provides default implementations in addition to the standard default implementations of copy and assignment.
-



# move constructors and assignment

- In C++11, we can define “move constructors” and “move assignments” to move rather than copy their argument.
- The idea behind a move assignment is that instead of making a copy, it simply takes the representation from its source and replaces it with a cheap default.
- What happens to a moved-from object?  
The state of a moved-from object is unspecified. Therefore, always assume that a moved-from object no longer owns any resources, and that its state is similar to that of an empty (as if default-constructed) object.



# Implicit move constructor/assignment

- If no user-defined move constructors are provided for a class type (struct, class, or union), and all of the following is true:
    - there are no user-declared copy constructors;
    - there are no user-declared copy assignment operators;
    - there are no user-declared move assignment operators;
    - there are no user-declared destructors;
  - then the compiler will declare a move constructor as inline public member of its class with the signature `T::T(T&&)`.
  - then the compiler will declare a move assignment operator as an inline public member of its class with the signature `T& T::operator=(T&&)`.
-



## Rule of five:

because the presence of a user-defined destructor, copy-constructor, or copy-assignment operator prevents implicit definition of move constructor+assignment operators, any class for which move semantics are desirable, has to declare all five special member functions

- there are no user-declared copy constructors;
  - there are no user-declared copy assignment operators;
  - there are no user-declared move assignment operators;
  - there are no user-declared destructors;
  - then the compiler will declare a move constructor as inline public member of its class with the signature `T::T(T&&)`.
  - then the compiler will declare a move assignment operator as an inline public member of its class with the signature `T& T::operator=(T&&)`.
-



**Note:** a class designed for polymorphic use (i.e. with virtual methods, typically requires a public and virtual destructor. This blocks implicit moves.

However, we can use a new C++11 method declaration that asks the compiler to create a default method, adding `= default` at the end of the declaration:

```
class Widget {  
public:  
    Widget(const Widget&) = default;  
    Widget(Widget&&) = default;  
  
    Widget& operator=(const Widget&) = default;  
    Widget& operator=(Widget&&) = default;  
  
    ...  
};
```

In general, we can have the default versions of the six special member functions of C++11: Default constructors, Destructors, Copy constructors, Copy assignment operators, Move constructors, Move assignment operators





# move constructors - example

```
template <class T>
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```



# move constructors - example

```
template <class T>
swap(T& a, T& b) {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

No more useless copies, thanks to move and move constructors



# C++ I I libraries

- STL and standard C++ I I library use move constructors and assignment to speedup operations.
- E.g. `std::string` has move constructor, thus in C++ I I the following code is optimized:

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

---



# C++11 libraries

In most modern compilers, the compiler will see that `S` is about to be destroyed and it will first move it into the return value.

Then this temporary return value will be moved into `mystr`.

If `std::string` did not have a move constructor (e.g. prior to C++11), it would have been copied for both transfers instead.

- E.g. `std::string` has move constructor, thus in C++11 the following code is optimized:

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

---



# C++ | | STL

- We can add rvalues to STL containers, e.g. vector has `push_back(T&&)` method
- Move constructors allow us to write:

```
vector<int> makeBigVector() {  
    vector<int> result;  
    for(int i=0; i<1024; i++) {  
        result[i] = rand();  
    }  
    return result;  
}  
auto result = make_big_vector();  
// guaranteed not to copy the vector
```



# C++11 STL

In the C++11 standard library, all containers are provided with move constructors and move assignments, and operations that insert new elements, such as `insert()` and `push_back()`, have versions that take rvalue references.

The net result is that the standard containers and algorithms quietly – without user intervention – improve in performance because they copy less.

```
vector<int> makeBigVector() {  
    vector<int> result;  
    for(int i=0; i<1024; i++) {  
        result[i] = rand();  
    }  
    return result;  
}  
auto result = make_big_vector();  
// guaranteed not to copy the vector
```



# C++11 STL

In the C++11 standard library, all containers are provided with move constructors and move assignments, and operations that insert new elements, such as `insert()` and `push_back()`, have versions that take rvalue references.

The net result is that the standard containers and algorithms quietly – without user intervention – improve in performance because they copy less.

```
vector<int> makeBigVector() {  
    vector<int> result;  
    for(int i=0; i<1024; i++) {  
        result[i] = rand();  
    }  
    return result;  
}
```

The C++11 STL move constructor avoids to make a full copy

```
auto result = make_big_vector();  
// guaranteed not to copy the vector
```



# Move parameters

- Move semantics is useful in methods that receive temporaries (i.e. rvalues):

```
class MyBuffer {  
public:  
    MyBuffer(const MyBuffer& orig);  
    MyBuffer operator+(const MyBuffer& right);  
}
```

```
MyBuffer x, y;  
MyBuffer a(x);  
MyBuffer b(x+y);  
MyBuffer c(function_returning_MyBuffer());
```





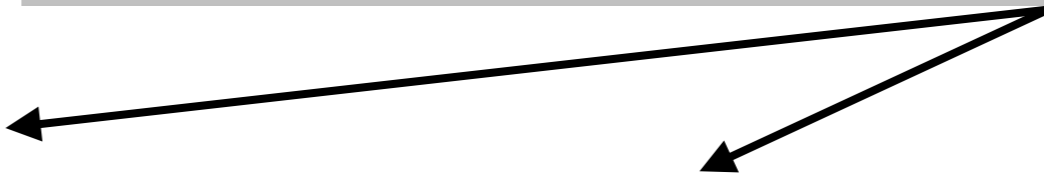
# Move parameters

- Move semantics is useful in methods that receive temporaries (i.e. rvalues):

```
class MyBuffer {  
public:  
    MyBuffer(const MyBuffer& orig);  
    MyBuffer operator+(const MyBuffer& right);  
}
```

```
MyBuffer x, y;  
MyBuffer a(x);  
MyBuffer b(x+y);  
MyBuffer c(function_returning_MyBuffer());
```

MyBuffer(MyBuffer&& temp);  
would be useful here...





# Create a move constructor

- A move constructor looks like this:

```
C::C(C&& other);
```

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



# Create a move assignment

- A move assignment operator has the following signature:

```
C& C::operator=(C&& other);
```

- 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:
    - Release any resources that `*this` currently owns.
    - Pilfer other's resource.
    - Set other to a default state.
    - Return `*this`.
-



# Full example

- Let us consider a class representing a buffer:

```
class MemoryPage {  
private:  
    size_t size;  
    char * buf;  
public:  
    explicit MemoryPage(int sz=512): size(sz),  
                                     buf(new char [size]) {}  
    ~MemoryPage( delete[] buf; }  
    //typical C++03 copy ctor and assignment operator  
    MemoryPage(const MemoryPage&);  
    MemoryPage& operator=(const MemoryPage&);  
};
```



# A move constructor

- A typical move constructor definition would look like this:

```
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;  
}
```



# A move constructor

The move constructor is much faster than a copy constructor because it doesn't allocate memory nor does it copy memory buffers.

```
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;  
}
```



# A move constructor

Basically a move constructor is a shallow copy followed by a nullification of the source... this latter step is needed, since the shallow copy has stolen the pointer/resources and we do not want that the destructor of the source destroys them

```
        buf(nullptr) {  
    // pilfer other's resource  
    size=other.size;  
    buf=other.buf;  
    // reset other  
    other.size=0;  
    other.buf=nullptr;  
}
```



# A move assignment

```
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;  
}
```





# Rvalues and returns

- Rvalue (and move semantics) greatly simplifies returning objects in C++11:

```
std::vector<int> return_vector(void) {  
    std::vector<int> tmp {1,2,3,4,5};  
    return tmp;  
}
```

```
std::vector<int> &&rval_ref = return_vector();  
// or more simply:  
std::vector<int> rval_ref = return_vector();
```

- The temporary object is caught by the rvalue reference, that extends its life beyond the `rval_ref` definition.
  - It works because, as said before, C++11 STL implements move semantics (constructor and assignment)
-



# Rvalues and returns

- In C++98/C++03, programmers would often resort to returning a large object by pointer just to avoid the penalty of copying its state.
  - In C++11, normally we should just return by value, because we will incur only a cheap move operation, not a deep copy, to hand the result to the caller, **if** the object is movable.
  - Guideline: most of the time, return movable objects by value.
-



# Dangling references



# Pitfall: dangling reference

- Although references, once initialized, always refer to valid objects or functions, it is possible to create a program where the lifetime of the referred-to object ends, but the reference remains accessible (dangling). Accessing such a reference is undefined behavior:

```
std::string& wrong_lvalue_ref() {  
    std::string s = "Example";  
    return s; // exits the scope of s:  
              // its destructor is called and its storage deallocated  
}  
  
std::string& r = wrong_lvalue_ref(); // dangling reference  
std::cout << r; // undefined behavior: reads from a dangling reference  
std::string s = wrong_lvalue_ref(); // undefined behavior:  
                                     // copy-initializes from a dangling reference
```



# Pitfall: dangling reference

- Although references, once initialized, always refer to valid objects or functions, it is possible to create a program where the lifetime of the referred-to object ends, but the reference remains accessible (dangling). Accessing such a reference is undefined behavior:

```
std::string& wrong_lvalue_ref() {  
    std::string s = "Example";  
    return s; // exits the scope of s:  
              // its destructor is called and its storage deallocated  
}  
  
std::string& r = wrong_lvalue_ref(); // dangling reference  
std::cout << r; // undefined behavior: reads from a dangling reference  
std::string s = wrong_lvalue_ref(); // undefined behavior:  
                                     // copy-initializes from a dangling reference
```

**Simply avoid to return references to function-local objects**



# Pitfall: dangling reference

- The same issue may happen with rvalue references:

```
std::string&& wrong_rvalue_ref() {  
    std::string r = "foo";  
    r += "bar";  
    return std::move(r);  
}
```



# Pitfall: dangling reference

- The same issue may happen with rvalue references:

```
std::string&& wrong_rvalue_ref() {  
    std::string r = "foo";  
    r += "bar";  
    return std::move(r);  
}
```

Simply avoid to return references to function-local objects



# Reading material

- M. Bertini, “Programmazione Object-Oriented in C++”, parte I, cap. 8
- <https://isocpp.org/wiki/faq/cpp11-language>





# Credits

- These slides are (heavily) based on the material of:
    - C++ FAQ
    - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo, “C++ primer”, Addison Wesley
-