Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

C++11 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 - 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 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];
```

`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

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.

Move semantics / &&

lvalue

- An **lvalue** is an expression that yields an object or function.
- The name is an old C mnemonic that means that **lvalues** could stand on the left-hand side of an assignment
  - In C++ not all **lvalues** 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*x, 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*x, 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

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 introduce 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.

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.

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***.

Unuseful 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.

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 unuseful copies, thanks to move and move constructors

C++11 libraries

- STL and standard C++11 library use move constructors and assignment to speedup operations.
  - 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++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++11 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;
}
auto result = make_big_vector();
// guaranteed not to copy the vector
```

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

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 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;
}
```

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

- 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