An introduction to C++

day 7

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center

Slides: Hannes Hauswedell

Timeframe

day	What	State
Monday	Questions, Recap, Debugging, **Test**	✓
Tuesday	Object-oriented programming 1, Essential operations, Regular types	✓
Wednesday	Memory management, lifetime, pointers	✓
Thursday	Object-oriented programming 2, move-semantics, forwarding	←
Friday	Type conversions and casts + **Test**	

Move semantics

Forwarding

Inheritance and virtual functions

Variables and reference binding

```
size_t          i1{1};
size_t const    i2{1};
```

```
size_t          s1a = i1;
size_t          s1b = i2;
size_t          s1c = size_t{1};

size_t const    s2a = i1;
size_t const    s2b = i2;
size_t const    s2c = size_t{1};
```

```
size_t       & s3a = i1;
size_t       & s3b = i2;
size_t       & s3c = size_t{1};

size_t const & s4a = i1;
size_t const & s4b = i2;
size_t const & s4c = size_t{1};
```

Q: Which of these do you think is valid? Why?

`= size_t{1}` means assigning a *temporary object*, same as `= 1` in this context

Variables and reference binding

```
size_t          i1{1};
size_t const    i2{1};
```

```
size_t          s1a = i1;
size_t          s1b = i2;
size_t          s1c = size_t{1};

size_t const    s2a = i1;
size_t const    s2b = i2;
size_t const    s2c = size_t{1};
```

```
size_t       & s3a = i1;
size_t       & s3b = i2;
size_t       & s3c = size_t{1};

size_t const & s4a = i1;
size_t const & s4b = i2;
size_t const & s4c = size_t{1};
```

- If we copy, we don't care about `const`-ness.

Variables and reference binding

```
size_t          i1{1};
size_t const    i2{1};
```

```
size_t          s1a = i1;
size_t          s1b = i2;
size_t          s1c = size_t{1};

size_t const    s2a = i1;
size_t const    s2b = i2;
size_t const    s2c = size_t{1};
```

```
size_t       & s3a = i1;
//size_t       & s3b = i2;
//size_t       & s3c = size_t{1};

size_t const & s4a = i1;
size_t const & s4b = i2;
size_t const & s4c = size_t{1}; // ???
```

- If we copy, we don't care about `const`-ness.
- We can only bind non-`const`-ref to non-`const`-objects
- `&` cannot bind to a temporary value, but `const &` apparently can!

Variables and reference binding

```
void print(std::string const & s)
{
    std::cout << s:
}


std::string       s1{"foo"};
std::string const s2{"bar"};

print(s1);    // pass by ref
print(s2);    // pass by ref
print("bax"); // <- temporary
```

- The `const &` binds to the temporary and *extends its lifetime* to match its own lifetime.
- This feature of `const &` allows us to write interfaces that handle references *and* values.
- Q: Why can't non-`const &` do this?

Variables and reference binding

```
void print(std::string const & s)
{
    std::cout << s;
}


std::string       s1{"foo"};
std::string const s2{"bar"};

print(s1);    // pass by ref
print(s2);    // pass by ref
print("bax"); // <- temporary
```

- The `const &` binds to the temporary and *extends its lifetime* to match its own lifetime.
- This feature of `const &` allows us to write interfaces that handle references *and* values.
- Q: Why can't non-`const` `&` do this?

- A: `void print(std::string & s)` would suggest that the function's purpose is to permanently change an outside parameter. Since the temporary does not exist outside of the function, this would not make sense.

MyVector

```
public:
    MyVector() = default;
    MyVector(MyVector const & rhs) { operator=(rhs); }
    MyVector & operator=(MyVector const & rhs)
    {
        size = rhs.size;                    // copy size
        delete[] data; data = new T[size];  // deallocate and reallocate
        for (size_t i = 0; i < size; ++i)
            data[i] = rhs.data[i];          // copy every element
        return *this;
    }
```

The behaviour of `const &` is the reason the copy constructor / copy assignment operator take temporaries, too. But do they handle them optimally?

MyVector

For temporaries, we would like to just transfer ownership instead of reallocating:

```
MyVector & operator=(MyVector ??? rhs)
{
    size = rhs.size;                      // copy size
    delete[] data; data = rhs.data;       // point to rhs' data
    rhs.data = nullptr;                   // prevent rhs from deleting
    return *this;
}
```

This looks correct, but we have two problems:

- we still need the old assignment operator for copying from references
- we can't take `rhs` by something that is `const`, because we are changing it

MyVector – move assignment

For temporaries, we would like to just transfer ownership instead of reallocating:

```
MyVector & operator=(MyVector && rhs)
{
    size = rhs.size;                        // copy size
    delete[] data; data = rhs.data;         // point to rhs' data
    rhs.data = nullptr;                     // prevent rhs from deleting
    return *this;
}
```

This looks correct, but we have two problems:

- we still need the old assignment operator for copying from references
- we can't take `rhs` by something that is `const`, because we are changing it

Variables and reference binding

```
size_t         i1{1};
size_t const   i2{1};
```

```
size_t         s1a = i1;
size_t         s1b = i2;
size_t         s1c = size_t{1};

size_t const   s2a = i1;
size_t const   s2b = i2;
size_t const   s2c = size_t{1};
```

```
size_t       & s3a = i1;
//size_t       & s3b = i2;
//size_t       & s3c = size_t{1};

size_t const & s4a = i1;
size_t const & s4b = i2;
size_t const & s4c = size_t{1};

//size_t      && s5a = i1;
//size_t      && s5b = i2;
size_t      && s5c = size_t{1};
```

- `size_t &&` is an *rvalue reference*
- It only binds to *rvalues* i.e. "temporary objects".

Function overloading [from day1]

```
double square(double const d)
{
    return d * d;
}

uint32_t square(uint32_t const i)
{
    return i * i;
}

uint64_t square(uint64_t const i)
{
    return i * i;
}

uint64_t i = 7;
i = square(i); // picks third one
```

- You can have multiple functions with the same name, provided
  - they have a different number of parameters;
  - **and/or the parameters have different types**
  - (a different return type is not sufficient!)
- This is called *function overloading*.
- This is very useful when the functions *do different things*, but…

Function overloading

```
void foo(size_t          i) {} // (1)
void foo(size_t const    i) {} // (2)
void foo(size_t        & i) {} // (3)
void foo(size_t const  & i) {} // (4)
void foo(size_t       && i) {} // (5)
```

- `const`-ness and "reference"-ness are part of the type, i.e. `size_t` and `size_t const` are different types.
- Which of the above overloads do you think can be declared together?
- Which combinations would make sense?
- And when would which overload be chosen?

Overload resolution

```
void foo(size_t          i) {} // (1)

void foo(size_t const    i) {} // (2)

void foo(size_t        & i) {} // (3)
void foo(size_t const  & i) {} // (4)
void foo(size_t       && i) {} // (5)
```

- (1) and (2) can never be declared together.
- (1) is ambiguous with (3), (4), (5)
- (2) is ambiguous with (3), (4), (5)

	(1)	*or*	**(2)**	*or*	**(3)**	**(4)**	**(5)**
`size_t s1; foo(s1);`	?		?		?	?	?
`size_t const s2; foo(s2);`	?		?		?	?	?
`foo(size_t{});`	?		?		?	?	?

✓ = overload can accept this ✓✓ = preferrred when competing

Overload resolution

```
void foo(size_t          i) {} // (1)

void foo(size_t const    i) {} // (2)

void foo(size_t        & i) {} // (3)
void foo(size_t const  & i) {} // (4)
void foo(size_t       && i) {} // (5)
```

Typically either

- (1) *or* (2) *or* (3) *or* (4) *or* (4)+(5)
- (3)+(4)+(5) possible, but unusual

	(1)	***or***	**(2)**	***or***	**(3)**	**(4)**	**(5)**
`size_t s1; foo(s1);`	✓		✓		✓✓	✓	X
`size_t const s2; foo(s2);`	✓		✓		X	✓✓	X
`foo(size_t{});`	✓		✓		X	✓	✓✓

✓ = overload can accept this ✓✓ = preferrred when competing

MyVector – Summary move construction/assignment

```
public:
    MyVector() = default;
    MyVector(MyVector const & rhs)              { /*…*/ }
    MyVector(MyVector && rhs)                   { /*…*/ }
    MyVector & operator=(MyVector const & rhs)  { /*…*/ }
    MyVector & operator=(MyVector && rhs)       { /*…*/ }
    ~MyVector() = default; // if using smart pointer
```

- The Move-constructor and the move-assignment operator allow us to avoid unnecessary copies, because they are chosen automatically by overload resolution over the copy-constructor / copy-assignment operator when the source is a *temporary*.
- By default they move-assign/move-construct all members.
- If you provide you own copy-constructor / copy-assignment operator, move-* will never be declared implicitly, you should also define or explicitly `=default` them.

Explicitly moving

```cpp
std::vector v0{1, 2, 3, 5};
std::vector v1{v0};              // v1 is copy of v0
std::vector v2{std::move(v0)};  // contents of v0 moved into v2
```

- It is possible to "mark something as being a temporary" using `std::move`.
- Note that `std::move` does not actually move something by itself, it merely casts its argument to `&&` so that other functions (e.g. the move constructor) are chosen and can then "steal" the contents.
- Attention: this leaves the moved-from object in "valid, but unspecified state" → you cannot use them afterwards, potential for errors!
- So, when is this useful?

Explicitly moving

```
void foo(MyVector<size_t> && s) { /*…*/ }

void bar(MyVector<size_t> && s)
{
    s[0] = 42;
    foo(std::move(s));
}

bar(MyVector<size_t>{7});
```

The only frequent use-case:

- When you want to pass a temporary to the next function, you need to explicitly call `std::move`, because it looses it's "temporary-ness" in every scope again.

Use-cases for `&&` outside of construction/assignment?

```
MyVector<size_t> foo(MyVector<size_t> const & s)
{
    MyVector<size_t> ret_value{s};
    ret_value[0] = 42;
    return ret_value;
}

MyVector<size_t> foo(MyVector<size_t> && s)
{
    MyVector<size_t> ret_value{std::move(s)};    // avoid copy
    ret_value[0] = 42;
    return ret_value;
}
```

- Q: Do you need to do that for all functions that copy input?

Use-cases for `&&` outside of construction/assignment?

```
MyVector<size_t> foo(MyVector<size_t> ret_value) // copy or move by constructor
{
    ret_value[0] = 42;
    return ret_value;
}
```

- Q: Do you need to do that for all functions that copy input?
- A: If you need to copy the input, take it by value and trust the constructors!

Use-cases for `&&` outside of construction/assignment?

```
MyVector<size_t> foo(MyVector<size_t> ret_value)
{
    ret_value[0] = 42;
    return ret_value;                               // move on return?
}
```

- Q: What about the return value, should we move it out?

Use-cases for `&&` outside of construction/assignment?

```
MyVector<size_t> foo(MyVector<size_t> ret_value)
{
    ret_value[0] = 42;
    return ret_value;                               // move on return?
}
```

- Q: What about the return value, should we move it out?
- A: No, copies of the return value are avoided automatically[1].

[1] in most cases

`std::swap`

```
MyVector<size_t> v1{7};
// …
MyVector<size_t> v2{9};

std::swap(v1, v2); // contents now swapped
```

- Two objects of a type that is move-constructible can be efficiently swapped via `std::swap`.
- Other types would need custom `swap()` overload, but I don't see why a non-movable object should be swappable.

Summary

- There are *rvalue references* denoted by `&&` that only bind to temporary objects.
- We can use these to define move-constructors and move-assignment operators that are then preferred overloads for temporaries.
- These can be used to "steal" dynamically allocated memory instead of copying which is more efficient (**memory on the stack is always copied!**)
- *rvalue references* are rarely used outside of move-construction/assignment.
- If you copy the argument-to-a-function inside that function, just take the input by value.
- If you "pass on" a temporary, you need to explicitly `std::move` it.

Further reading

"What are move semantics?"

- https://stackoverflow.com/a/11540204

"A Tour of C++", Second Edition, Bjarne Stroustrup

- "§5.2 Essential operations; Copy and Move"
- "§5.3 Resource Management"
- "§13.2 – §13.2.2 Resource Management"

Move semantics

Forwarding

Inheritance and virtual functions

Forwarding references

rvalue references:

```
MyVector<int> && v = /*…*/

void foobar(MyVector<int> && v) { /*…*/ }
```

forwarding references:

```
auto && v = /*…*/

template <typename T>
void foobar(T && v) { /*…*/ }
```

`&&` means something slightly different when there is type-deduction happening.

Variables and reference binding

```
size_t         i1{1};
size_t const   i2{1};
```

```
size_t         s1a = i1;
size_t         s1b = i2;
size_t         s1c = size_t{1};

size_t const   s2a = i1;
size_t const   s2b = i2;
size_t const   s2c = size_t{1};
```

```
size_t       & s3a = i1;
//size_t       & s3b = i2;
//size_t       & s3c = size_t{1};

size_t const & s4a = i1;
size_t const & s4b = i2;
size_t const & s4c = size_t{1};

//size_t      && s5a = i1;
//size_t      && s5b = i2;
size_t      && s5c = size_t{1};
```

- `size_t &&` is an *rvalue reference*
- It only binds to *rvalues* i.e. "temporary objects".

Variables and reference binding (auto)

```
size_t          i1{1};
size_t const    i2{1};
```

```
auto            s1a = i1;
auto            s1b = i2;
auto            s1c = size_t{1};

auto    const   s2a = i1;
auto    const   s2b = i2;
auto    const   s2c = size_t{1};
```

```
auto          & s3a = i1;
auto          & s3b = i2;  // == auto const &
//auto          & s3c = size_t{1};

auto    const & s4a = i1;
auto    const & s4b = i2;
auto    const & s4c = size_t{1};

auto         && s5a = i1;  // == auto &
auto         && s5b = i2;  // == auto const &
auto         && s5c = size_t{1};
```

- `auto &&` is an *forwarding reference*
- It binds to anything 🥳

Forwarding references - why?

```
template <typename TRange>
void foo(TRange && rng)      // can be `&`, `&&`, `const &`
{
    for (auto && elem : rng) // can be `&`, `&&`, `const &`
    {
        std::cout << elem;
    }
}
```

- Forwarding references are useful in generic code, because you can express logic independent of whether you are working with references or values (and independent of `const`-ness); use when you would otherwise use `const &`, but where you want to change the values.

Forwarding references - why?

```
template <typename TRange>
void foo(TRange && rng)      // can be `&`, `&&`, `const &`
{
    for (auto && elem : rng) // can be `&`, `&&`, `const &`
    {
        std::cout << elem;
    }
}
```

- Forwarding references are useful in generic code, because you can express logic independent of whether you are working with references or values (and independent of `const`-ness); use when you would otherwise use `const &`, but where you want to change the values.
- In the above example, `rng` could be a container (type of elem has `&`) or a "generator" (type of elem has `&&`).

Perfect forwarding

```
template <typename TRange>
void foo(TRange && rng) { /*…*/ }

template <typename TRange>
void bar(TRange && rng)
{
    /*…*/
    foo(std::forward<TRange>(rng));
}
```

- As with *rvalue references*, objects bound to *forwarding references* loose their "temporary-ness".
- To preserve it you cannot `std::move` them, because the function template handles both: temporaries and references.
- Instead you can `std::forward` them which preserves the original type independent of context.

Summary

- `&&` as part of template parameters or in combination with `auto` is not an *rvalue reference*, but a *forwarding reference*.
- Forwarding references can act as *rvalue references* and as regular references (single `&`) depending on context.
- Inconsistent, but makes writing generic code easier.
- Whenever you would `std::move` something bound to an *rvalue reference*, you should `std::forward` it if bound to a *forwarding reference*.
- Appears quite frequently in generic code.

Move semantics

Forwarding

Inheritance and virtual functions

Simple inheritance (from "A Tour of C++") [day5]

```
               Base Class

              /          \
             /            \

  Derived Class 1     Derived Class 2
```

We distinguish two roles of inheritance:

- *Interface inheritance:* An object of a derived class can be used wherever an object of the base class is required.
- *Implementation inheritance:* A base class provides functions or data that simplifies the definition of a derived class.

Simple inheritance

```cpp
struct Shape
{
    double area()      const { return NAN; } // returns not-a-number by default
    double perimeter() const { return NAN; } // returns not-a-number by default
};
```

```cpp
struct Square : Shape
{
    double width{};


    double area() const
    { return width*width; }
    double perimeter() const
    { return 4*width;     }
};
```

```cpp
struct Circle : Shape
{
    static constexpr double pi = 3.14;
    double radius{};

    double area() const
    { return pi*radius*radius; }
    double perimeter() const
    { return 2*pi*radius;      }
};
```

Interface inheritance

```
void printArea(Shape const & s)
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

Does this work? What will be printed?

Interface inheritance

```
void printArea(Shape const & s)
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

Does this work? What will be printed?

```
nan
nan
```

You told the compiler you are calling `.area()` on a `Shape` 🤔

Simple inheritance

```
struct Shape
{
    virtual double area()      const { return NAN; } // returns not-a-number
    virtual double perimeter() const { return NAN; } // returns not-a-number
};
```

```
struct Square : Shape
{
    double width{};


    double area() const override
    { return width*width; }
    double perimeter() const override
    { return 4*width;     }
};
```

```
struct Circle : Shape
{
    static constexpr double pi = 3.14;
    double radius{};

    double area() const override
    { return pi*radius*radius; }
    double perimeter() const override
    { return 2*pi*radius;      }
};
```

If you have `virtual` functions use `class` (not `struct`) and follow rule-of-six!

Interface inheritance

```
void printArea(Shape const & s)
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

And now?

Interface inheritance

```
void printArea(Shape const & s)
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

And now?

```
16
50.24
```

Virtual functions

- The `virtual` keyword tells the compiler that `Shape` might be a `Shape` *or* a *derivate* (derived class).
- When calling a **member** function that is marked `virtual` in the base class
  1. the compiler performs a look-up **at run-time** to see if the object you are referring to, is in fact a *derivate*
  2. if yes, checks if that class `override`s that function
  3. if yes, calls the derivate's function instead of its own
- The `override` keyword in derivates is optional (it will "override" without it), but highly recommended!
- The `final` keyword may be used instead of or additionally to `override` to state that (further-)derived classes may not override this function anymore.

Interface inheritance

```
void printArea(Shape const s)            // take by value instead of reference
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

And now?

Interface inheritance

```
void printArea(Shape const s)            // take by value instead of reference
{
    std::cout << s.area() << '\n';
}

Square sq{}; sq.width = 4; printArea(sq);

Circle ci{}; ci.radius = 4; printArea(ci);
```

And now?

```
nan
nan
```

Attention: if you take by value, a new object is created and there can be no derived function call!

Interface inheritance

- `virtual` function lookup works as expected on references and pointers, you can directly assign objects/`new` values of derived type to references/pointers of the base type.
- But when you implicitly convert to an object of the base type, *slicing* happens: you simply get an object of the base type and any state of the derivate is not represented in the new object.
- This means if you want to store a collection of possibly different related types in e.g. a vector, you need to store pointers (you cannot store references in a `std::vector`):

```
std::vector<std::unique_ptr<Shape>> vec{};
vec.push_back(std::make_unique<Circle>());
printArea(*vec[0]);  // prints `0` because object was default-initialised
```

Abstract base classes

```
struct ShapeInterface
{
    virtual double area()      const = 0;   // this function is "pure" virtual
    virtual double perimeter() const = 0;   // this function is "pure" virtual
};
```

```
struct Square : ShapeInterface
{
    double width{};


    double area() const override
    { return width*width; }
    double perimeter() const override
    { return 4*width;     }
};
```

```
struct Circle : ShapeInterface
{
    static constexpr double pi = 3.14;
    double radius{};

    double area() const override
    { return pi*radius*radius; }
    double perimeter() const override
    { return 2*pi*radius;      }
};
```

If you have `virtual` functions use `class` (not `struct`) and follow rule-of-six!

Abstract base classes

- `virtual` functions that are declared with `= 0` are *pure virtual* functions, they never have a definition.
- A class (type) with at least one *pure virtual* function is an *abstract class* (type).
- You cannot create objects of *abstract types*, but you can inherit from them and declare references or pointers to them.

Abstract base classes

- `virtual` functions that are declared with `= 0` are *pure virtual* functions, they never have a definition.
- A class (type) with at least one *pure virtual* function is an *abstract class* (type).
- You cannot create objects of *abstract types*, but you can inherit from them and declare references or pointers to them.
- You should separate *interface inheritance* from *implementation inheritance*, because this makes reasoning about inheritance much easier; one is the formal description of the properties of the type, the other is some helper to reduce code-duplication.
- Interfaces should be declared as *abstract classes* to highlight that they cannot be used on their own.
- This also prevents some *slicing* errors.

Summary

- Virtual functions facilitate *specialisation* similar to inheritance in other languages like Java.
- In C++ functions need to be explicitly marked as `virtual` for this to work.
- If you override a virtual function in a derived class, additionally mark it as `override`.
- With virtual functions, use `class` (not `struct`) and follow rule-of-six!
- Make sure to declare your destructor as `virtual` so that correct clean-up happens.
- Separate *interface inheritance* from *implementation inheritance*, make *interface classes* abstract by marking all member functions as *pure virtual* (`= 0`).

Further reading

"A Tour of C++", Second Edition, Bjarne Stroustrup

- "§4 Classes"

CPPReference:

- https://en.cppreference.com/w/cpp/language/virtual
- https://en.cppreference.com/w/cpp/language/override
- https://en.cppreference.com/w/cpp/language/abstract_class

Tasks for the computer lab I – MyVector

- Adapt your code from yesterday by adding move-constructor and move-assignment operator.
- *The lecture code refers to the implementation without smart pointer, but you should implement it with the smart pointer*.
- Add diagnostic code (e.g. `std::cout << "Being copied-assigned!\n";`) to all of your constructors, assignment operators and destructor and create tests that trigger each of the code paths.
- Can you explicitly default your move-*? Why or why not?

Tasks for the computer lab IIa – Shape

- Take the example from the lecture slides and define `ShapeInterface`, `Circle` and `Square`, but follow the lecture's advice and make them classes.
- Follow rule-of-six and provide an additional constructor for `Circle` and `Square` that can initialise the member variable.
- Make `ShapeInterface` an abstract base class.

- Implement a function `void print(ShapeInterface const & s)` that prints area and perimeter.
- What would you need to change to also print the name of the type?

Tasks for the computer lab IIb – Shape

- Implement a program that stores shapes in a collection and repeatedly asks for input.
- The text-based user interface should look like this:

```
What do you wish to do?

[c] Add Circle to collection.
[s] Add Square to collection.
[d] Remove last shape in collection.
[p] Print an item in the collection.
[q] Quit.
```

- [c] and [s] should result in a question for the width/radius
- [p] should result in a question for the index; it should print type, area and perimeter.