An introduction to C++

day 5

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center

Slides: Hannes Hauswedell

day	What
Monday	Recap week 1 & 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

Recap from week 0

Essential operations and Regular types

Simple inheritance

Class types - struct

```
struct Complex
{
    double re;
    double im;
}; // ; is important!

Complex c{1, 4};

std::cout << "real:"
          << c.re
          << " imaginary:"
          << c.im
          << '\n';

Complex c2;   // undefined
Complex c3{}; // like {0, 0}
```

- The definition on the left introduces the `Complex` class type for complex numbers.
- It consists of two *member variables*, a real part (`.re`) and an imaginary part (`.im`).
- The member variables can be accessed via the dot-operator.
- Objects of type `Complex` can be brace-initialised, both with values and default

Class types - struct

```
struct Complex
{
    double re{};
    double im{};
};

Complex c{1, 4};

std::cout << "real:"
          << c.re
          << " imaginary:"
          << c.im
          << '\n';

Complex c2;   // like {0, 0}
Complex c3{}; // like {0, 0}
```

- The definition on the left introduces the `Complex` class type for complex numbers.
- It consists of two *member variables*, a real part (`.re`) and an imaginary part (`.im`).
- The member variables can be accessed via the dot-operator.
- Objects of type `Complex` can be brace-initialised, both with values and default
- Member variables of built-in type could (and should!) be *member-initialised*.

Class types - member functions

```
struct Complex
{
    double re{};
    double im{};

    void add(Complex const & c)
    {
        re += c.re;
        im += c.im;

    }
};

Complex c{1, 4};
Complex c2{2, 5};

c.add(c2);          // == {3, 9}
```

- What if we would like to be able to add two complex numbers?
- To do that, we can add a *member function*!
- Member functions are like other functions, but declared inside the body of the class.
- They can access member variables.
- They are called via `.` on an object of the type.

Class types - member functions

```
struct Complex
{
    double re{};
    double im{};

    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }
};

Complex c{1, 4};
Complex c2{2, 5};

c.operator+=(c2);    // == {3, 9}
c += c2;             // == {5, 14}
```

- But an `.add()` function is ugly...
- ... instead we can define an *operator*!
- Operators can be invoked via their name like other member functions.
- **But** they can also be invoked directly via their operator so user defined types *appear* similar to built-in types.

Class types - member functions

```
struct Complex
{
    double re{};
    double im{};

    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }

    Complex operator+(Complex const & c)
    {
        Complex tmp{re, im};
        tmp += c;
        return tmp;
    }
};
```

- Often some operators can be used to simplify the definition of others.
- Be aware of the different return values:
  - arithmetic+assignment: reference to self
  - regular arithmetic: new object
  - comparison: bool

Class types - member functions

```
struct Complex
{
    double re{};
    double im{};

    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }

    Complex operator+(Complex c) const
    {
        return (c += *this);
    }
};
```

- Often some operators can be used to simplify the definition of others.
- Be aware of the different return values:
  - arithmetic+assignment: reference to self
  - regular arithmetic: new object
  - comparison: bool
- Member functions that don't change an object should be marked `const`; otherwise can't be called on objects of `const` type.

Class types - protection of members

```
struct Complex
{
private:
    double re{};
    double im{};

public:
    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }
};

// private members disable easy initial.
// Complex c{1, 3.4};
```

- Sometimes you may want to protect your member variables so that they are only accessible to member functions.
- You can use the `private` and `public` keywords to denote this difference.

Class types - protection of members

```cpp
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }
};

// private members disable easy initial.
// Complex c{1, 3.4};
```

- Sometimes you may want to protect your member variables so that they are only accessible to member functions.
- You can use the `private` and `public` keywords to denote this difference.
- A `class` is a `struct` whose members are `private` by default.
- More on classes soon…

Recap from week 0

Essential operations and Regular types

Simple inheritance

Essential operations, conventional operations

Essential:

1. construct: create an object, possibly initialise
2. assignment: change the state of existing object
3. destruct: free the resources of an object

Conventional:

1. compare: (in-)equality, less-than…
2. swap: exchange contents of two objects (covered later)

Constructors

```
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex & operator+=(Complex const & c)
    {
        re += c.re;
        im += c.im;
        return *this;
    }
    // other member functions...
};

// private members disable easy initial.
// Complex c{1, 3.4};
```

- Sometimes you may want to protect your member variables so that they are only accessible to member functions.
- You can use the `private` and `public` keywords to denote this difference.
- A `class` is a `struct` whose members are `private` by default.
- More on classes soon …

Constructors

```cpp
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex(double const _re, double const _im)
    {
        re = _re;
        im = _im;
    }

    // other member functions...
};

// now this works again:
Complex c{1, 3.4};
```

- A constructor is a special member function with the name of the type and no return type.
- Called when attempting to create a new object of a given type.
- Here the constructor enables us to initialise the object's members on creation.

Constructors

```
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex(double const _re, double const _im)
        : re{_re}, im{_im} //<-initializer list
    {
    }


    // other member functions...
};

// now this works again:
Complex c{1, 3.4};
```

- A constructor is a special member function with the name of the type and no return type.
- Called when attempting to create a new object of a given type.
- Here the constructor enables us to initialise the object's members on creation.
- There is a convenience syntax (initializer list) for initialising members.

Constructors

```
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex(double const _re, double const _im)
        : re{_re}, im{_im}
    {}
};
```

Why all this code…

Constructors

```
class Complex
{
private:
    double re{};
    double im{};

public:
    Complex(double const _re, double const _im)
        : re{_re}, im{_im}
    {}
};
```

Why all this code…

🤔

```
struct Complex
{
    double re{};
    double im{};
};
```

… just to get this?

Constructors - `struct` vs `class`

- Both `class` and `struct` are "class types", you *can* do everything with both, by definition the only difference is that members are `private` by default in classes and `public` by default in structs.
- **BUT** there is a very strong convention to
    1. use `class` instead of `struct` when you have *class invariants* (this usually implies custom constructors)
    2. use `class` if you want to hide implementation details, e.g. an internal data structure, which the user shall not have access to

A *class invariant* is a restriction on the state of objects of your class, e.g.

- "The `float` member `.f` may only hold positive values"
- "If the value of member `.a` changes, member `.b` needs to be updated"

This usually means: Explicitly declare all constructors/destructors in classes and none in structs

Constructors - Invariants

```
#include <cassert>                  // provides the assert() macro/function
class Birthday
{
private:
    uint16_t month{0}; // member initializer: here to `0` (explicit)
    uint16_t day{};    // member initializer: also 0, but implicit
public:
    Birthday(uint16_t const m, uint16_t const d)
    {
        set_month(m); set_day(d);
    }

    uint16_t get_day() const         { return day; }
    uint16_t get_month() const       { return month; }

    void set_day(uint16_t const d)   { assert(d <= 31); day = d; }   // lazy 😉
    void set_month(uint16_t const m) { assert(m <= 12); month = m; }
};
```

Note: Using `uint8_t` as members would suffice but causes trouble with std::cin and std::cout (`uint8_t == unsigned char`)

Q: How to enforce these invariants?

Constructors - Invariants

There are different ways to enforce invariants:

- `assert()` from `<cassert>` causes program to abort; **error only in DEBUG mode!**
- throwing exceptions (more on this next week); **always cause error**, but can be caught by program.
- silently restore correct state on invalid input (e.g. convert values), **never errors**.

Constructors - which are there?

```
// Default constructor
Birthday() : month{}, day{} { /* ... */ }

// Copy constructor
Birthday(Birthday const & rhs)
{ set_day(rhs.get_day()); set_month(rhs.get_month()); }

// Move constructor
Birthday(Birthday && rhs) { /* ... */ }

// Custom constructor
Birthday(uint16_t const m, uint16_t const d)
{ set_month(m); set_day(d); }

// more custom constructors

// Destructor
~Birthday() {}
```

```
Birthday d0;
Birthday d1{};


Birthday d2{d0};


// more on this
// in a few days


Birthday d3{1,-3};




// automatically called when
// object goes out of scope
```

Constructors - "explicitly defaulted"

```
// Default constructor
Birthday() = default;

// Copy constructor
Birthday(Birthday const &) = default;


// Move constructor
Birthday(Birthday &&) = default;

// Custom constructor
Birthday(uint16_t const m, uint16_t const d)
{ set_month(m); set_day(d); }

// more custom constructors

// Destructor
~Birthday() = default;
```

```
Birthday d0;
Birthday d1{};


Birthday d2{d0};


// more on this
// in a few days


Birthday d3{1,-3};




// automatically called when
// object goes out of scope
```

Constructors - "explicitly deleted"

```
// Default constructor
Birthday() = delete;

// Copy constructor
Birthday(Birthday const &) = delete;


// Move constructor
Birthday(Birthday &&) = delete;

// Custom constructor
Birthday(uint16_t const m, uint16_t const d)
{ set_month(m); set_day(d); }

// more custom constructors

// Destructor
~Birthday() = delete;
```

```
// prevents:
Birthday d0;
Birthday d1{};

// prevents copy
Birthday d0(d1);


// prevents move

// enables:
Birthday d3{1,-3};




// also prevents
// construction
```

Constructors - default constructor

```
Birthday() : month{}, day{}
{ /* ... */ }
```

```
Birthday d0;
Birthday d1{};
```

- Member initialisers are executed by all constructors unless the c'tor initializes the member explicitly
- A typical reason for a class not being default-constructible is one of it's data members not being default-constructible, e.g. data members of reference type are not!
- Mark it as `= default` and rely on member initialisers if the default constructor enforces no invariant. The defaulted default constructor(!) calls the default constructors of base-classes and members.
- Mark it as `= delete` only if your class absolutely cannot be created without some parameters.

Constructors - default constructor

```
Birthday() : month{}, day{}
{ /* ... */ }
```

```
Birthday d0;
Birthday d1{};
```

Attention:

- For class types **with user-provided constructors** `Birthday d0;` and `Birthday d1{};` behave the same, both call the default constructor.
- But for built-in types like `float` and class types **without user-provided constructors**, `{}` leads to initialisation while omitting it leads to no initialisation.

→ It's good practice to always write `{}`.

Constructors - copy constructor

```
Birthday(Birthday const & rhs)
{
    set_day(rhs.get_day());
    set_month(rhs.get_month());
}
```

```
Birthday d0;
Birthday d1{};


Birthday d2{d0};
```

- Copy constructors can often be explicitly defaulted (invariants are enforced by other constructors).
- Defaulted copy constructors copy every member.
- `Birthday d3 = d0` is also copy construction, not assignment!
- Mark it as `= delete` if your class manages a resource that doesn't allow multiple-access, e.g. a file.

Copy assignment and move assignment

- Assignment is closely related to Construction.
- We expect that if a new object can be constructed as the copy of another one, we should also be able to make an existing object the copy of another one.
- With very few exceptions, copy assign and copy construct should leave the object in the same state!
- In fact, one is usually implemented as using the other.
- Can be marked `= default` or `= delete`, too.
- There is also move assignment (more on this later).

```
Birthday & operator=(Birthday const & rhs)
{
    month = rhs.month;
    day = rhs.day;
    return *this;
}
```

```
Birthday b0{1, 10};
Birthday b1{2, 12};

// copy construct:
Birthday b2{b0};

// copy assign:
b2 = b1;

// attention:
Birthday b3 = b0;
```

Essential operations - Summary

Invariants and/or custom Constructors?

No:

- call it `struct`
- no user-provided constructors, destructor or assignment operators (assumed to be defaulted)
- "Rule-of-none" / "Rule-of-0"
- everything `public`
- no `virtual` functions
- e.g. plain data storage
- also called "aggregate types"

Yes:

- call it `class`
- always explicitly provide
  1. default constructor
  2. copy constructor
  3. move constructor
  4. copy assignment operator
  5. move assignment operator
  6. destructor
- "Rule-of-6"
- can be `= default` or `= delete` or user-defined

Conventional operations - comparison (C++<=17)

```
class Birthday { /**/ };

bool operator==(Birthday const & lhs,
                Birthday const & rhs)
{
    return std::tuple{lhs.get_month(),
                      lhs.get_day()}
        == std::tuple{rhs.get_month(),
                      rhs.get_day()};
}
```

- Comparison should be implemented as a free function (or a `friend`), not a member. You can't make 1 == X{} work with a member function - it has to be a free function!

- Comparison is strongly related to assignment and by convention you should make sure that `a = b` results in `a == b`. (This might sound trivial, but it isn't always, e.g. if `a` and `b` have different types).
- Always try to define `==` and `!=`, if possible also `<`, `<=`, `>` and `>=`.
- Cannot be `= default`, must be user-implemented, but can be made easier with `std::tuple` and `std::tie`; re-use one operator to implement another.

Conventional operations - comparison

```
class Birthday {

  auto operator<=>(const Birthday& rhs) const = default;

}
```

(C++20 or later):

- comparison operators can be defaulted
- <=> (spaceship operator) can be defaulted and allows 3-way comparison (returning -1,0,1 for less,equal,greater)
- this will implicitly define `==`, `<`, etc.
- rules on what type is returned by `<=>` are a bit complicated…
- 1 == X{} works with a member function `bool X::operator==(int);` since comparisons are symmetric in C++20 or later

Regular types

Regular type:

- copyable
- movable
- default-constructible
- (in-)equality-comparable

Semiregular type:

- copyable
- movable
- default-constructible

Try to make all of your types regular, it makes reasoning about them much easier.

Regular types

Regular type:

- copyable
- movable
- default-constructible
- (in-)equality-comparable

Semiregular type:

- copyable
- movable
- default-constructible

Try to make all of your types regular, it makes reasoning about them much easier.

Exercise: How can you check?

Further reading

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

- "§5 Essential Operations"

"Fundamentals of Generic Programming"

- by James C. Dehnert and Alexander Stepanov:
- http://stepanovpapers.com/DeSt98.pdf

Recap from week 0

Essential operations and Regular types

Simple inheritance

Simple inheritance (from "A Tour of C++")

```
            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
#include <cassert>                  // provides the assert() macro/function
class Birthday
{
private:
    uint16_t month{};
    uint16_t day{};
public:
    Birthday() = default;
    Birthday(uint16_t const m, uint16_t const d)
    {
        set_month(m); set_day(d);
    }

    uint16_t get_day() const         { return day; }
    uint16_t get_month() const       { return month; }

    void set_day(uint16_t const d)   { assert(d <= 31); day = d; }   // lazy 😉
    void set_month(uint16_t const m) { assert(m <= 12); month = m; }
};
```

Simple inheritance

```
class ExactBirthday : public Birthday
{
private:
    uint16_t hour{};
public:
    ExactBirthday() = default;
    ExactBirthday(ExactBirthday const &) = default;
    ExactBirthday(ExactBirthday &&) = default;
    ExactBirthday & operator=(ExactBirthday const &) = default;
    ExactBirthday & operator=(ExactBirthday &&) = default;
    ~ExactBirthday() = default;

    // either:
    ExactBirthday(uint16_t const m, uint16_t const d, uint16_t const h)
    { set_month(m); set_day(d); set_hour(h); }

    // or:
    ExactBirthday(uint16_t const m, uint16_t const d, uint16_t const h)
      : Birthday(m, d)
    { set_hour(h); }


    uint16_t get_hour() const       { return hour; }
    void set_hour(uint16_t const h) { assert(h <= 23); hour = h; }
};
```

Simple inheritance

```
class ExactBirthday : public Birthday
{
```

- `class ExactBirthday` is *derivate* of the *base class* `Birthday`
- Members of `Birthday` are accessible:
  - `public` members: by everything
  - `protected` members: by self and derivate classes
  - `private` members: only by self
- Members not directly accessible can be changed indirectly through members that are.

Simple inheritance

```
class ExactBirthday : public Birthday
{
```

- `class ExactBirthday` is *derivate* of the *base class* `Birthday`
- Members of `Birthday` are accessible:
    - `public` members: by everything
    - `protected` members: by self and derivate classes
    - `private` members: only by self
- Members not directly accessible can be changed indirectly through members that are.
- *Type of inheritance* can be `public`, `protected` or `private`, but only `public` is commonly used.

Simple inheritance

```
private:
    uint16_t hour{};
```

An additional data member.

```
public:
    ExactBirthday() = default;
    ExactBirthday(ExactBirthday const &) = default;
    ExactBirthday(ExactBirthday &&) = default;
    ExactBirthday & operator=(ExactBirthday const &) = default;
    ExactBirthday & operator=(ExactBirthday &&) = default;
    ~ExactBirthday() = default;
```

Rule-of-6!

Simple inheritance

Q: Would this user defined assignment work?

```
ExactBirthday & operator=(ExactBirthday const & rhs)
{
    month = rhs.month; day = rhs.day; hour = rhs.hour; return *this;
}
```

Simple inheritance

Q: Would this user defined assignment work?

```
ExactBirthday & operator=(ExactBirthday const & rhs)
{
    month = rhs.month; day = rhs.day; hour = rhs.hour; return *this;
}
```

No, because `.month` and `.day` are private to the Birthday class!

Simple inheritance

Q: Would this user defined assignment work?

```
ExactBirthday & operator=(ExactBirthday const & rhs)
{
    month = rhs.month; day = rhs.day; hour = rhs.hour; return *this;
}
```

No, because `.month` and `.day` are private to the Birthday class! But this would:

```
ExactBirthday & operator=(ExactBirthday const & rhs)
{
    Birthday::operator=(rhs); hour = rhs.hour; return *this;
}
```

Explicitly calls the assignment operator of the base class (this is what the automatically generated definition looks like).

Simple inheritance

```
    ExactBirthday(uint16_t const m, uint16_t const d, uint16_t const h)
    { set_month(m); set_day(d); set_hour(h); }

    uint16_t get_hour() const       { return hour; }
    void set_hour(uint16_t const h) { assert(h <= 23); hour = h; }
```

- Adds remaining part of the interface.
- Q: (Why) does the constructor work? Aren't month and day private?

Simple inheritance

```
    ExactBirthday(uint16_t const m, uint16_t const d, uint16_t const h)
    { set_month(m); set_day(d); set_hour(h); }

    uint16_t get_hour() const       { return hour; }
    void set_hour(uint16_t const h) { assert(h <= 23); hour = h; }
```

- Adds remaining part of the interface.
- Q: (Why) does the constructor work? Aren't month and day private?
- Q: Could you do this: `ExactBirthday b{11, 22}`, i.e. construct it without the hour like `Birthday`?

Simple inheritance

```
ExactBirthday(uint16_t const m, uint16_t const d, uint16_t const h)
{ set_month(m); set_day(d); set_hour(h); }

uint16_t get_hour() const       { return hour; }
void set_hour(uint16_t const h) { assert(h <= 23); hour = h; }
```

- Adds remaining part of the interface.
- Q: (Why) does the constructor work? Aren't month and day private?
- Q: Could you do this: `ExactBirthday b{11, 22}`, i.e. construct it without the hour like `Birthday`?
- No, the constructors of the base class are not inherited by default.

Simple inheritance

```
Birthday        b{11, 11};
ExactBirthday   e{11, 11, 11};

bool c = (b == e);
```

Q: What happens?

1. does not compile
2. `c` is `true`
3. `c` is `false`

Simple inheritance

```
Birthday        b{11, 11};
ExactBirthday   e{11, 11, 11};

bool c = (b == e);
```

Q: What happens?

1. does not compile
2. `c` is `true`
3. `c` is `false`

Derived types are *implicitly convertible* to their base types so

```
bool operator==(Birthday const & lhs, Birthday const & rhs)
```

is called which evaluates to `true`.

Exercise: what happens when you compare two `ExactBirthday` objects?

Simple inheritance - Summary

- Class types can inherit other class types to re-use code and to be used in similar situations (more on the latter in two days).

- The type of inheritance is almost always `public`.
- Whether or not members of the base type can be accessed by the derived type depends on their access protection.

- Constructors and assignment operators are not inherited by default.
- Derived types are implicitly convertible to base types, but not the other way around.

Tasks for the computer lab

Tasks for the computer lab I

- Create `birthday.hpp` that defines `class Birthday`.
- The class should be able to represent at least dates between 0000-00-00 (0 A.D.) and 3000-00-00.
- The class should enforce it's invariants (valid dates!) on user provided input (e.g. via `assert()`), you should handle simple leap years (every fourth year), but handling the more complicated rules is not required.
- Define default constructor, copy constructor and copy assignment operator yourself, don't use `= default`. You can omit move constructor and move-assignment for now.
- Implement `operator==` and `operator!=`. Bonus: also add `operator<`, `operator<=` …
- Test your implementation with a small program that reads user-given dates and creates objects from them.

Tasks for the computer lab II

- Explicitly default your constructors/operators, also add defaulted move-*
- Split the implementation of the class into `birthday.cpp` and `birthday.hpp` with declarations in the header and definitions in the cpp. See day2's slides if you forgot how to do this.

- How can you check from inside C++ if a type is default-constructible, copy-constructible, move-constructible?
- Is there a way to perform this check while the program is compiled?

- Add such a check to `birthday.hpp` and try commenting out your default constructor to see if it fails.

- Bonus: is there a way to check for comparability? What about testing directly if it's Regular?

Tasks for the computer lab III

- Create a new pair of files: `birthtime.cpp` and `birthtime.hpp`
- Inside create the class `Birthtime` that inherits `Birthday`, but add hour and minute as members with corresponding interfaces.
- Can you compare objects of type `Birthtime` with `Birthdate`? What are the semantics?
- What about copy-construction and copy-assignment (in both directions)?

- Write a free function `void print(Birthdate & d)` that streams out the date in "YYYY-MM-DD" notation.
- What happens when you print a Birthtime with this function? Does conversion take place?