An introduction to C++

day 2

Chris Bielow

Bioinformatics Solution Center, FU Berlin

slides: Hannes Hauswedell *AG Algorithmische Bioinformatik* et al.

Lambda functions

User-defined types

Enumerations

Class types

Separate compilation

Lambda functions

```
template <typename TElem>
void square_all_elements(std::vector<TElem> & vec)
{
    for (TElem & elem : vec)
        elem = elem * elem;
}
```

Lambda functions

```
template <typename TElem>
void square_all_elements(std::vector<TElem> & vec)
{
    for (TElem & elem : vec)
        elem = elem * elem;
}
```

```
template <typename TElem>
void squareroot_all_elements(std::vector<TElem> & vec)
{
    for (TElem & elem : vec)
        elem = std::sqrt(elem);
}
```

Lambda functions

```
template <typename TElem>
void square_all_elements(std::vector<TElem> & vec)
{
    for (TElem & elem : vec)
        elem = elem * elem;
}
```

```
template <typename TElem>
void squareroot_all_elements(std::vector<TElem> & vec)
{
    for (TElem & elem : vec)
        elem = std::sqrt(elem);
}
```

Seems tedious. What if we want to create function that just does "X_on_all_elements" and define X separately?

Lambda functions

```
template <typename TElem, typename TLambda>
void on_all_elements(std::vector<TElem> & vec, TLambda const & l)
{
    for (TElem & elem : vec)
        elem = l(elem);
}
```

Lambda functions

```cpp
template <typename TElem, typename TLambda>
void on_all_elements(std::vector<TElem> & vec, TLambda const & l)
{
    for (TElem & elem : vec)
        elem = l(elem);
}
```

```cpp
int main()
{
    auto square      = [] (auto const & elem) { return elem * elem;     };
    auto square_root = [] (auto const & elem) { return std::sqrt(elem); };

    std::vector<double> ds{0.2, 1.5, 2};

    on_all_elements(ds, square);         // ds == { 0.04, 2.25, 4 }
    on_all_elements(ds, square_root);    // ds == { 0.20, 1.50, 2 }
}
```

Lambda functions

```cpp
auto square = [] (auto const & elem) { return elem * elem; };
```

- Lambdas are *objects* and each lambda has a distinct type – that's why we can only save them in a variable with deduced type and why functions need to take them as template parameters.

Lambda functions

```
auto square = [] (auto const & elem) { return elem * elem; };
```

- Lambdas are *objects* and each lambda has a distinct type – that's why we can only save them in a variable with deduced type and why functions need to take them as template parameters.
- A minimal Lambda that does nothing is `[](){}`.

Lambda functions

```
auto square = [] (auto const & elem) { return elem * elem; };
```

- Lambdas are *objects* and each lambda has a distinct type – that's why we can only save them in a variable with deduced type and why functions need to take them as template parameters.
- A minimal Lambda that does nothing is `[](){}`.
- `[]` introduces a Lambda definition (other things can go into the `[]`, but we won't cover that now).
- `()` contains the parameters, just like with ordinary functions except that `auto` is valid (even before C++20).
- `{}` contains the body of the lambda function.
- The return type of the Lambda is deduced by default.

Lambda functions

User-defined types

Enumerations

Class types

Separate compilation

User-defined types (~ "A Tour of C++")

- The arithmetic types, possibly modified by `const` and/or `&`, as well as arrays thereof are **built-in types**.
- They are *low-level*; efficiently reflect the capabilities of conventional hardware.

- C++ provides various mechanisms of abstraction for the design of higher level applications.
- You can combine these abstraction mechanisms with the built-in types to create **user-defined types**.
- User-defined types are either *enumerations* or *class types*.

Lambda functions

User-defined types

Enumerations

Class types

Separate compilation

Enumerations

```cpp
enum class Color
{
    red, blue, green, yellow
}; // ; is important!

Color invertMe(Color const c)
{
    switch (c)
    {
        case Color::red:
            return Color::green;
        case Color::green:
            return Color::red;
        case Color::blue:
            return Color::yellow;
        case Color::yellow:
            return Color::blue;
    }
}
```

- Enumerations are simple user-defined types that represent a small set of integer values by giving them names.
- Helps to make code more expressive, making it easier to read and write and prevents errors.
- Enumerations can be introduced by `enum NAME` or `enum class NAME`, the latter is a *strongly-typed* enumeration.
- Prefer strongly-typed enums!

Enumerations

Strongly-typed enums:

```
enum class Color
{
    red, blue, green, yellow
};

//    scoped ↓
Color c = Color::blue;

// not implicitly convertible:
//int i = c;

// no arithmetic operators:
//Color cc = red + blue;
```

C-style enum:

```
enum Color
{
    red, blue, green, yellow
};

// unscoped ↓
Color c = blue;

// implicitly convertible:
int i = c;               // == 1

// arithmetic operators:
Color cc = red + blue;   // == green
```

Lambda functions

User-defined types

Enumerations

Class types

Class templates

Separate compilation

Class types - struct

```cpp
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

```cpp
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{};

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

    }
};

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 c{1, 4};
Complex c2{2, 5}; Complex c3{1, 1};

c += c2 += c3;     // c == {4, 10}
                   // c2 ?
```

- Customarily those arithmetic operators that change an object, return a reference the object itself after it was changed.
- This enables them to be "chained" and used in expressions.
- *At this point you don't need to understand what* `*this` *does except "reference to self"*.

Class types - member functions

```cpp
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}; Complex c3{1, 1};
// equivalent to without ()
c += (c2 += c3);  // c == {4, 10}
                  // c2 == {3, 6}
```

- Customarily those arithmetic operators that change an object, return a reference the object itself after it was changed.
- This enables them to be "chained" and used in expressions.
- *At this point you don't need to understand what* `*this` *does except "reference to self".*

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}; Complex c3{1, 1};
// not equivalent to without ()
(c += c2) += c3;  // c == {4, 10}
                  // c2 unchanged
```

- Customarily those arithmetic operators that change an object, return a reference the object itself after it was changed.
- This enables them to be "chained" and used in expressions.
- *At this point you don't need to understand what* `*this` *does except "reference to self".*

Class types - member functions

```cpp
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

```cpp
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);
    }
};
int i = a + b + c; // same as (a + b) + c;
```

- 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};
// and direct access:
// std::cout << c.re; // can't do: private now
```

- 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};
// and direct access:
// std::cout << c.re; // can't do: private now
```

- 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 in the 2nd week of the course!

Class types - Quiz

```cpp
struct Dog
{
    void bark()
    {
        std::cout << "WUFF\n";
    }
};

void ignore(Dog const & d)
{
    d.bark();
}

int main()
{
    Dog d{};
    ignore(d);
}
```

- What's going wrong here?

Class types - Quiz

```
struct Dog
{
    void bark()
    {
        std::cout << "WUFF\n";
    }
};

void ignore(Dog const & d)
{
    d.bark();
}

int main()
{
    Dog d{};
    ignore(d);
}
```

- What's going wrong here?
- The member function `bark()` was not `const`-qualified so it can't be called inside of `ignore()`, because inside `ignore()` the argument is accessed via `const &`!

Lambda functions

User-defined types

Enumerations

Class types

Class templates

Separate compilation

Class templates

```
template <typename T>
struct Complex
{
    T re{};
    T im{};
};

Complex<double> c{3.3, 4.4};
Complex<int32_t> c2{3, 4};
```

- Similar to function templates: a class template is not a "complete type", it's a template for a type.
- By specifying all template arguments, e.g. `Complex<double>` you *instantiate* the template and declare a type.
- `Complex<double>` and `Complex<int32_t>` are different types!
- the template argument for class templates can sometimes be deduced (as for function templates), but rules are compilicated

Class templates

```
template <typename T>
struct Complex
{
    T re{};
    T im{};
};

Complex<double> c{3.3, 4.4};
Complex<int32_t> c2{3, 4};
```

```
std::vector<size_t> vec;
```

- Now you know what `std::vector` is!

- Similar to function templates: a class template is not a "complete type", it's a template for a type.
- By specifying all template arguments, e.g. `Complex<double>` you *instantiate* the template and declare a type.
- `Complex<double>` and `Complex<int32_t>` are different types!
- the template argument for class templates can sometimes be deduced (as for function templates), but rules are compilicated

Lambda functions

User-defined types

Enumerations

Class types

Class templates

Separate compilation

Separate compilation

- The more code you write, the less organised your `.cpp` will get.
- To increase readability and maintainability of your code, you should split it into multiple files.
- C++ knows `.cpp` files and header files (`.hpp`, sometimes `.h`).
- Header files are like `.cpp` files, but they don't contain a `main()` function.

Separate compilation

- The more code you write, the less organised your `.cpp` will get.
- To increase readability and maintainability of your code, you should split it into multiple files.
- C++ knows `.cpp` files and header files (`.hpp`, sometimes `.h`).
- Header files are like `.cpp` files, but they don't contain a `main()` function.

- Header files are included by `#include`; headers can include other headers.
- Including a header literally results in the entire contents of the header being pasted into the current source file!
- This also means you need to avoid including a header twice (why?).
- To do that, write `#pragma once` into the header file.

Separate compilation

There are two different "styles" for organising source code:

1. A single `.cpp` file and many header files (or in the case of libraries only headers)
2. A `.cpp` file with `main()` and pairs of cpp+hpp files where declaration and definition are split:
    - declaration in the header
    - definition in the cpp

The second style is more common, but it depends also on the paradigm of programming.

Separate compilation -- header-only

`main.cpp`:

```
#include "example.hpp"


int main()
{
    example_func();
}
```

`example.hpp`:

```
#pragma once
#include <iostream>

void example_func()
{
    std::cout << "!!!";
}
```

- simpler build process; entire library shipped as headers
- necessary for function templates and members of class templates
- slower build-times, because all headers are parsed every time and the entire project is rebuilt on every small change

Separate compilation -- cpp+hpp

`main.cpp`:

```cpp
#include "example.hpp"


int main()
{
    example_func();
}
```

`example.hpp` (decl.):

```cpp
#pragma once

void example_func();
```

`example.cpp` (def.):

```cpp
#include <iostream>
#include "example.hpp"

void example_func()
{
    std::cout << "!!!";
}
```

- every pair of cpp+hpp is compiled separately (called *translation unit*)
- faster builds (only those parts are rebuilt that changed)
- libraries used by many programs are shared (less memory used)
- doesn't work for templates

Separate compilation -- cpp+hpp

Build-process unix

First we build an *object file* for every cpp other than main:

```
% g++ -std=c++17 -Wall -Wextra -Werror -pedantic example.cpp -c
```

(note the `-c`!) This created `example.o`.

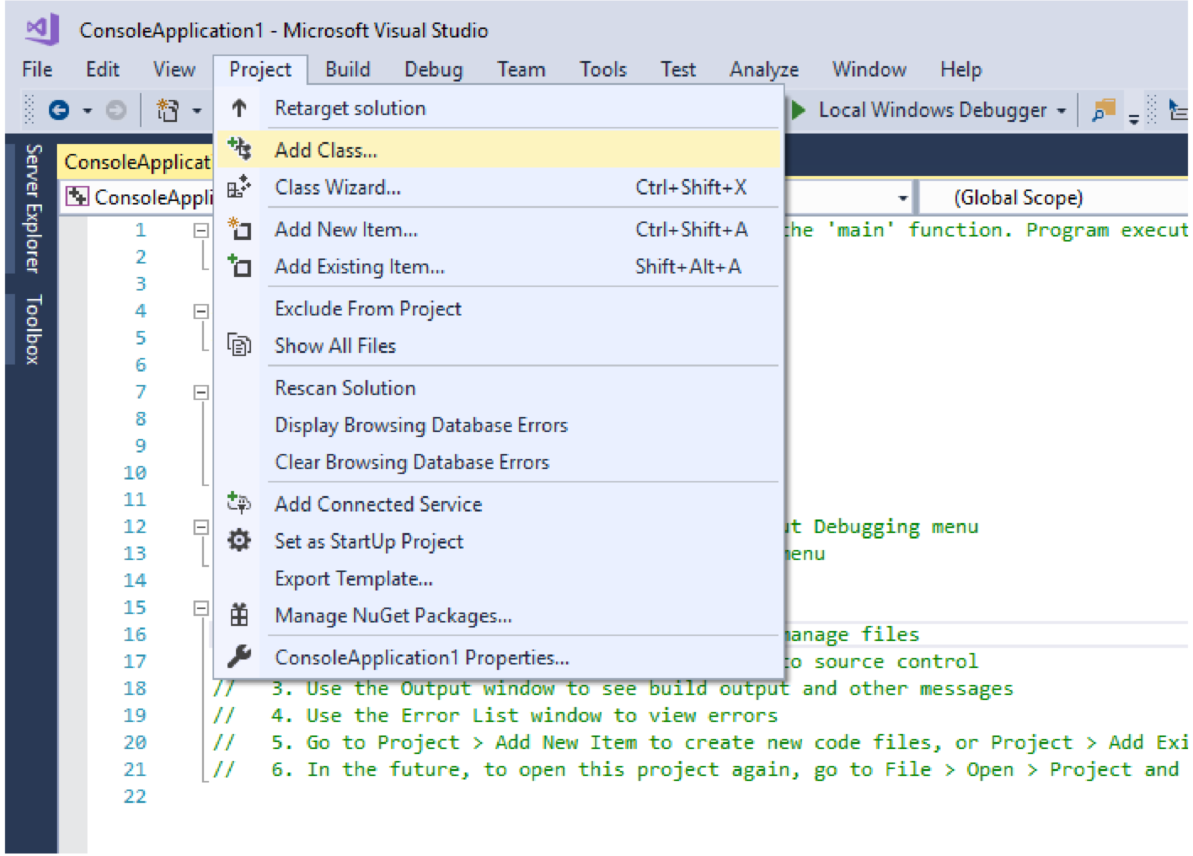
Finally we build `main.cpp` and *link* it with the existing object files:

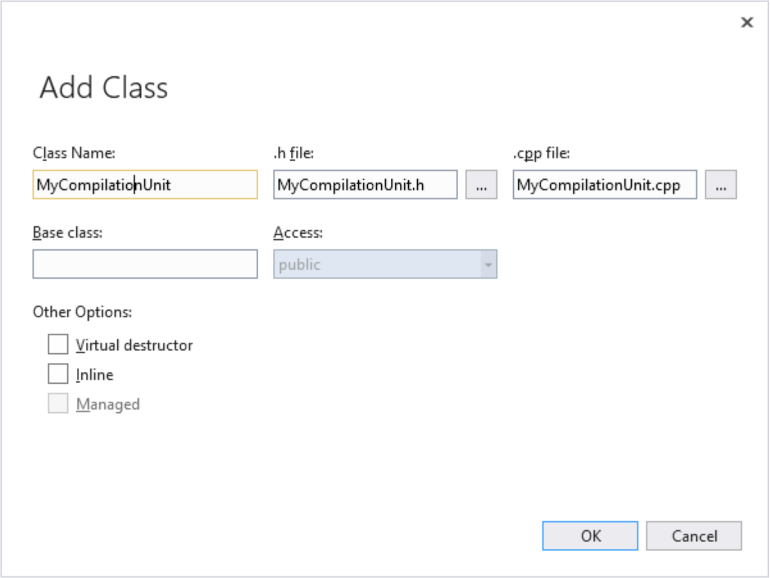
```
% g++ -std=c++17 -Wall -Wextra -Werror -pedantic main.cpp example.o
```

In larger projects *build-systems* like CMake, Meson, Gnu-Make, Ninja or a combination thereof handle this for us.

Separate compilation -- cpp+hpp

Build-process Windows





- Delete all contents from the new pair of files
- MSVC will take care of the linking

Tasks for the computer lab

Tasks for the computer lab I

- Implement a function with the following signature:

```
template <typename TLambda>
std::vector<size_t> filter(std::vector<size_t> const & input,
                           TLambda const & l)
```

- It should return a vector with those elements of the input for which the lambda function evaluates to `true`.
- Write a main()-function that tests this behaviour with three lambdas:
  1. a lambda that returns true for elements whose value is even.
  2. a lambda that returns true for elements whose value is odd.
  3. a lambda that returns true for elements that are not zero.

Tasks for the computer lab II

1. Implement a `struct Person` with the member variables `name` and `age` (what are appropriate types?).
2. Implement an `enum class` that represents gender with the three legal categories in Germany: `FEMALE`, `MALE` and `UNDEFINED` (in case you are not happy with those, you may add more 😉).
3. Add a member variable of that type to `struct Person`.
4. Add a member function called `print()` that prints the fields in some descriptive manner.

Tasks for the computer lab III

1. Use your type from the previous task!
2. Write a main function that repeatedly asks the user to input name, age and gender and then saves those as elements of a `std::vector<Person>`. Ask the user after every input if they want to quit adding persons.
3. If they quit, sort the vector of persons by age.
   - Use `std::sort` and read up the documentation on it!
   - Implement the actual comparison as `operator<` inside `struct Person`.
   - instead implement a lambda-function that does the comparison (and is passed to `std::sort`).
4. Which solution do you think is better? How difficult would it be to sort by name (alphabetically) instead?
5. Print the first and last persons from the vector.

Tasks for the computer lab IV

1. Implement the Complex type (template)
   - with the following operators:
     - *arithmetic operators*: `+`, `-`, `*`, `/`,
     - *arithmetic assignment operators*: `+=`, `-=`, `*=`, `/=`,
     - *comparison operators*: `==`, `!=`
   - Start with a non-templatised `struct`, later switch to the templatised version.
   - Write a main() function that properly tests the functionality!
2. Move the definition of the type into a separate header file and verify that everything works.
3. Why can templates not have their definitions in separate cpp files? Think about / find out!