An introduction to C++

day 1

Sandro Andreotti, Chris Bielow

Bioinformatics Solution Center, FU Berlin

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

Day	Content	State
Monday	Variables, Constness, I/O, STL (vector, string, …)	✓
Tuesday	Control flow, functions (overloading), function templates	←
Wednesday	Lambdas, Enums, struct/class, class template, compilation	
Thursday	STL: tuples, more data containers, algorithms	
Friday	Holiday	
Monday	Questions, Recap, Debugging, **Test**	

Control flow

References

Functions

Function overloading

Function templates

Control flow - if

- the specific blocks are only executed if the conditions are met
- `else` blocks are optional
- `{ }` are optional if only one statement follows, but it's good style to always "balance braces", i.e. if one block has braces, all of them should have braces

Logical operators:

- `!` negation
- `&&` logical AND
- `||` logical OR

```
if (i > 3)
{
    // ...
}
else if ((i > 5) && (i < 100))
{
    // ...
}
else if (!(i > 100))
{
    // ...
}
else
{
    // ...
}
```

Control flow - if

- if you need to evaluate some function and do the check based on the return, and also need that value inside the block, you can use an `if` statement with an "init statement"
- if the expression used as the condition is a constant expression, you can have the decision take place at compile-time with `if constexpr`

```
// with init statement
if (double j = sqrt(i); j > 3)
{
    // ...
    // use j in here again
}
```

```
size_t constexpr k = 42;

// decision at compile-time
if constexpr (k - 7 > 3)
{
    // ...
}
```

Control flow - switch

- instead of many successive `if` checks, do a single `switch` statement
- only works for expressions of integral or enum type
- each case begins with a `case` label and ends with a `break;`
- if you omit the `break;` statement, subsequent block will also be executed!
- if you introduce local variables, you must enclose the block in `{ }`

```
switch (i)
{
    case 0:
        std::cout << "ZERO\n";
        break;
    case 1:
        std::cout << "ONE\n";
        break;
    case 42:
    {
        char c = 'C';
        // ...
        break;
    }
    default:
        std::cout << "OTHER\n";
}
```

Control flow - for

```cpp
std::vector<uint16_t> vec{1, 22, 333, 4444};

//      "for e in vec"
for (uint16_t e : vec)
    std::cout << e << ' ';

//   initial state. ;   run condition;    iteration expr.
for (size_t i = 0;      i < vec.size();     ++i)
{
    if (i % 2 == 0) continue;
    std::cout << "elem no." << i << " is " << vec[i] << '\n';
}
```

- To iterate over a range (vector, array...) use the short syntax
- to exit the innermost loop early, call `break;`
- to jump to the next iteration early, call `continue;`

Control flow - while, do-while

- similiar to `for`-loops but used less frequently
- also support `break;` and `continue;`
- often as `while (true)` and later `if (complex_condition) break;`

```
while (condition)
{
    // ...
}

while (init; condition)
{
    // ...
}

do
{
    // ...
} while (condition);
```

Control flow

References

Functions

Function overloading

Function templates

References

What happens when you initialise (or assign) one variable with another variable of the same type?

```
size_t i = 145;
size_t j = i;                // i == 145

std::string s{"A text much longer than this example"};
std::string t{s};            // t == "A text...."
```

- A **copy** is being made! That's because C++ has "value semantics".
- Different to other programming languages.
- Copying data is an expensive operation.

References

Sometimes we don't want to copy, we just want to introduce a new name, that's what **references** are for:

```
std::string s{"A text much longer than this example"};
std::string & t{s};
//          ^  reference symbol

std::cout << t << '\n';     // prints "A text..."
t = "a shorter text";

std::cout << s << '\n';     // prints "a shorter text"
//           ^ original variable has changed
```

- A variable of reference type behaves exactly like the original.
- A variable of reference type has to be initialised when declared.

References

You can create a reference to `const` type, even if the original is not `const`:

```
std::string s{"A text much longer than this example"};
std::string const & t{s};

// t = "a shorter text";        // doesn't work, t is const
s = "a shorter text";           // works, also "updates" t
```

The reverse is not true:

```
std::string const s{"A text much longer than this example"};
// std::string & t{s};         // doesn't work, s is const
std::string const & u{s};
```

References - when are they useful?

Avoid copies during iteration:

```
std::vector<std::string> chromosomes{/*...*/};

// iterate over the chromosomes WITHOUT copying on each iteration:
for (std::string const & chromosome : chromosomes)
    std::cout << "Length: " << chromosome.size() << '\n';
```

References - when are they useful?

Avoid copies during iteration:

```cpp
std::vector<std::string> chromosomes{/*...*/};

// iterate over the chromosomes WITHOUT copying on each iteration:
for (std::string const & chromosome : chromosomes)
    std::cout << "Length: " << chromosome.size() << '\n';
```

To be able to write to the underlying data:

```cpp
std::vector<size_t> vec{1, 2, 3};

for (size_t i : vec)
    i = i * i;
// vec is unchanged!
```

```cpp
std::vector<size_t> vec{1, 2, 3};

for (size_t & i : vec)
    i = i * i;
// vec is { 1, 4, 9}
```

References

Summary:

- by default things are copied in C++
- avoid copying anything except arithmetic types
- instead of copying you can create references to existing variables
- references behave just like the original (same syntax)
- the referred object has to be set at time of declaration and **cannot** be changed later on
- `const`-ness can be "added" via a reference, but not "removed"
- `const`, `&` and `const &` are part of a variable's type, e.g. `int const & i;` means `i` is of type "reference to integer constant"

Control flow

References

Functions

Function overloading

Function templates

Functions

```
#include <iostream>

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

int main()
{
    std::cout << "Enter number: ";

    double e{};
    std::cin >> e;

    std::cout << "The square is: " << square(e) << '\n';
}
```

Functions

```
return_type function_name(param1_type param1_name,
                          param2_type param2_name)  // (1)
{
    // ...
}
```

- (1) is called the *function signature*
- everything between the `{ }` is called *function body*
- the whole thing is called *function definition*
- a function signature without body is called a *function declaration* (a definition is implicitly also a declaration)
- **a function must be declared, before it is first used**
- a function may be declared multiple times, but defined only once (**one-definition-rule**, ODR)

Functions

```cpp
#include <iostream>

double square(double const d); // declaration

int main()
{
    std::cout << "Enter number: ";

    double e{};
    std::cin >> e;
    //                                  ↓ must be declared before use
    std::cout << "The square is: " << square(e) << '\n';
}

double square(double const d) // definition (can happen after use)
{
    return d * d;
}
```

Function return type

```
// ↓ this thing!
double square(double const d)
{
    return d * d;
}
```

```
void print(std::string const & str)
{
    std::cout << str << '\n';
}
```

- Adding `const` to a return type has no effect, because a new object is generated on returning anyway.
- Can be `void` if you don't want to return anything – the return statement is optional in that case, although a blank `return;` is sometimes useful.
- If it is `auto`, the compiler will deduce the type – usually only helpful with templates (more on this later).

(Play around with this during the exercise!)

Function return type

```
double & square()
{
    double d{5.3};
    return d;           // 💣
}

double & d = square(); // 💥
```

- You can return references, but you will need this rarely outside of *member functions* (more on this tomorrow); and it can be dangerous!

Function return type

```
double & square()
{
    double d{5.3};
    return d;           // 💣
}

double & d = square(); // 💥
```

- You can return references, but you will need this rarely outside of *member functions* (more on this tomorrow); and it can be dangerous!

```
std::vector<std::string> foo()
{
    return { "LONG_STRING",
             "EVEN_LONGER"};
}

std::vector<std::string> v = foo();
```

- Contrary to C and old C++ it is accepted and recommended to also return large objects from functions; with some limitations - read on return value optimization (RVO)

Function parameters

```
//             ↓"parameter"↓
double square(double const d)
{
    return d * d;
}
//       "argument" ↓
double de = square(3.4);
```

```
void print(std::string const & str)
{
    std::cout << str << '\n';
}

std::string s{"test"};
print(s);
```

- Parameters declare new variables in the scope of the function that are initialised with the respective *arguments* of the function call.
- All rules we discussed to declaring variables apply, especially:
- **If you forget the `&`, values are copied to the function!**
- make parameters `const` whenever possible

Function parameters

Recommendations

1. By default make all function parameters `const &`
2. Ask yourself: do I want to change it?
    1. no
        1. arithmetic type? → just `const`
        2. else → keep `const &`
    2. yes
        1. so that change is visible outside? → just `&`
        2. change only inside function? → neither `const` nor `&` (copy)

Function parameters – more examples

In general:

```
void print(std::string const & str)
{
    std::cout << str << '\n';
}
std::string s{"test"};
print(s);     // prints "test"
```

```
void append_foo(std::string & str)
{
    str += "foo";
}

std::string s{"test"};
append_foo(s); // foo == "testfoo"
```

Arithmetic types:

```
double square1(double const d)
{
    return d * d;
}

double de = square1(3.4);
```

```
void square2(double & d)
{
    d = d * d;
}

double df{3};
square2(df); // df == 9
```

Function parameters – more examples

In general:

```
void print(std::string const & str)
{
    std::cout << str << '\n';
}
std::string s{"test"};
print(s);     // prints "test"
```

```
void append_foo(std::string & str)
{
    str += "foo";
}

std::string s{"test"};
append_foo(s); // foo == "testfoo"
```

Only copy if you want to modify the value inside the function **and** preserve it outside:

```
void app_print(std::string str)
{
    str += "foo";
    std::cout << str << '\n';
}

std::string s{"test"};
app_print(s);
// prints "testfoo"
// `s` remains "test"
```

Control flow

References

Functions

Function overloading

Function templates

Function overloading

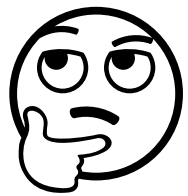
```
double square(double const d)
{
    return d * d;
}

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

```
float square(float const d)
{
    return d * d;
}
double square(double const d)
{
    return d * d;
}
uint8_t square(uint8_t const i)
{
    return i * i;
}
uint16_t square(uint16_t const i)
{
    return i * i;
}
```

```
uint32_t square(uint32_t const i)
{
    return i * i;
}
uint64_t square(uint64_t const i)
{
    return i * i;
}
// and now for int*_t ...
```



Control flow

References

Functions

Function overloading

Function templates

Function templates

```
template <typename T>
T square(T const n)
{
    return n * n;
}

// compiler generates no code
// for above template, until
// template is actually used:

int32_t i = 1;
i = square(i);

double d = 3.2;
d = square(d);
```

- definition is called a *function template* (**not ~~"template function"~~!**)
- when the template is used first, the compiler generates the overloads that we would have written otherwise (it actually produces the same machine code as when we would have written the overloads ourselves)
- this saves writing and maintaining lots of definitions
- also only those overloads are generated that are actually used!

Function templates

```
template <typename T1, typename T2>
???? add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- But what type does `int32_t` + `double` return?
- How do you generalise that?

Function templates

```
template <typename T1, typename T2>
???? add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- But what type does `int32_t` + `double` return?
- How do you generalise that?

```
template <typename T1, typename T2>
auto add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- This is the right place to use `auto`!
- Only works for the return type (in C++14/17)

Function templates

```
template <typename T1, typename T2>
???? add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- But what type does `int32_t` + `double` return?
- How do you generalise that?

```
template <typename T1, typename T2>
auto add(T1 const n1, T2 const n2)
{
    return n1 + n2;
}
```

- This is the right place to use `auto`!
- Only works for the return type (in C++14/17)

```
auto add(auto const n1, auto const n2)
{
    return n1 + n2;
}
```

- ... until C++20 (or when using `-fconcepts` with GCC)
- allows parameters as `auto`

Function templates

You can also templatise over the element type of a vector:

```
template <typename T>
T max_element(std::vector<T> const & vec_of_elements)
{
    T max{};
    for (T const & elem : vec_of_elements)
        if (elem > max)
            max = elem;

    return max;
}

std::vector<size_t> vec{1, 3, 55, 666, 44, 22, 1};
size_t m = max_element(vec); // == 666
```

Tasks for the computer lab

Tasks for the computer lab I

- Write a program that repeatedly asks the user to input a person's name and postal code.
- Immediately after an entry is submitted the program should respond with `{NAME} lives in Berlin` or `{NAME} lives outside of Berlin` depending on the postal code entered
- After that it should ask the user something like `Enter another person? [y/n]`;
- if `y` is entered it should repeat the last step;
- if `n` is entered it should print the percentage of entered people that lives in Berlin and the name of the person with the smallest postal code; and then quit.

Tasks for the computer lab II a

- Copy'n'paste some of the examples from the slides and check if they make sense to you!
- What happens when you return something of `const` type from a function? Can constants be initialised from functions that return a non-const type?
- What happens when you return a reference? What about the example on slide 20?
- What combinations of `const`, `const &`, `&` (and none) are there for variables and for function return values? Which combinations are valid/invalid for initialising the variable?

Tasks for the computer lab II b

```cpp
#include <iostream>

template <typename T>
void print(T const i)
{
    std::cout << "Integer!\n";
}

void print(float const i)
{
    std::cout << "Floating point!\n";
}

int main()
{
    print(3.3);
}
```

- What does the program print?
- Why?
- How can you "fix" it?

Tasks for the computer lab III

- Write a function that takes two initial numbers `n1` and `n2` and a counter `c` and returns the c-th "fibonacci" number based on the two initial numbers
- It should accept any numeric type for the initial numbers (assume both numbers are the same type)!
- Write a program that asks the user for two floating point variables and then returns the 42nd "fibonacci number" for these floats
- make your function 'constexpr', e.g. `constexpr auto fib(...) { ... }`. Is it sufficient to capture the return value as `const auto fib_value = fib(...);` or do you need `constexpr` here as well? Compute the 5th canonical fibonacci number at compile time. Can you also compute the 100th number at compile time?