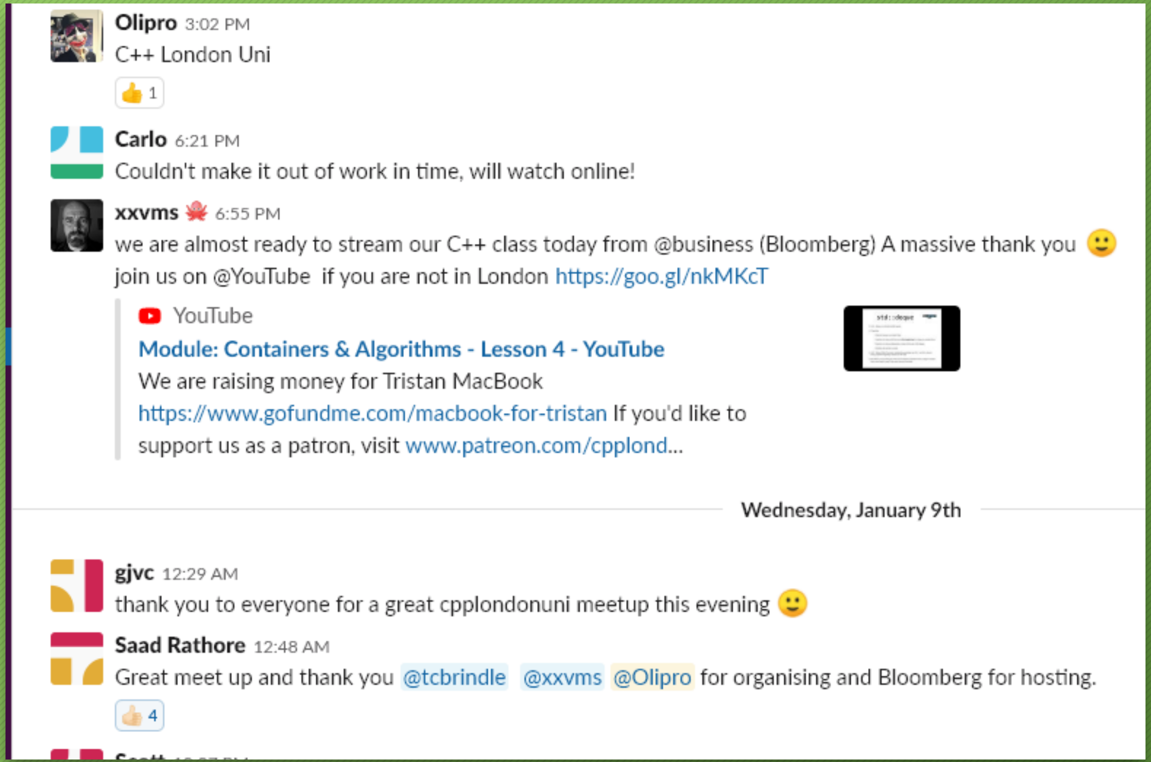
Initial C++ — Session 6

Tristan Brindle

Feedback

- We’d love to hear from you!
- The easiest way is via the *CPPLang* Slack organisation. Our chatroom is #cpplondonuni
- If you already use Slack, don’t worry, it supports multiple workgroups!
- Go to https://slack.cpp.al to register.



Olipro 3:02 PM
C++ London Uni
👍 1

Carlo 6:21 PM
Couldn't make it out of work in time, will watch online!

xxvms 6:55 PM
we are almost ready to stream our C++ class today from @business (Bloomberg) A massive thank you 🙂 join us on @YouTube if you are not in London https://goo.gl/nkMKcT

> YouTube
> **Module: Containers & Algorithms - Lesson 4 - YouTube**
> We are raising money for Tristan MacBook https://www.gofundme.com/macbook-for-tristan If you'd like to support us as a patron, visit www.patreon.com/cpplond...

Wednesday, January 9th

gjvc 12:29 AM
thank you to everyone for a great cpplondonuni meetup this evening 🙂

Saad Rathore 12:48 AM
Great meet up and thank you @tcbrindle @xxvms @Olipro for organising and Bloomberg for hosting.
👍 4

Last week

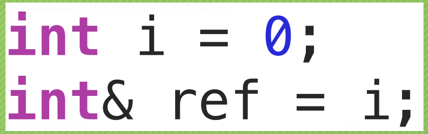
- The auto keyword
- Pass-by-reference
- The const keyword

This week

- Revision: const and references
- Pass-by-reference-to-const
- Const, references and auto
- Namespaces

Revision: references

- C++ uses *value semantics* by default
    - modifying a copy does not affect the original variable
- We can also create a *reference* by using the & symbol after the type name



```
int i = 0;
int& ref = i;
```

- **Modifying a reference also affects the original variable**
- You can think of a reference as a *new name* for an existing variable

References example

```
int main()
{
    int i = 0;
    int& ref = i;

    ref = 42;
    std::cout << i << '\n';
    // prints 42

    std::vector<int> vec = { 1, 2, 3 };
    for (int& i : vec) {
        i = i * 2;
    }
    // vec now contains { 2, 4, 6 }
}
```

Revision: pass-by-reference

- Usually, when we pass data to a function, it operates on a copy of each value
  - We call this *pass-by-value*
- We can also declare function parameters to have reference type
- This is called *pass-by-reference*
- When we use pass-by-reference, the function operates directly on the passed-in variable, with no copies taking place

Pass-by-reference example

```cpp
void my_function(int val, float& ref)
{
    val = 100;
    ref = 3.142f;
}

int main()
{
    int i = 0;
    float f = 0.0f;

    my_function(i, f);

    std::cout << i << '\n';
    // prints 0
    std::cout << f << '\n';
    // prints 3.142
}
```

Revision: immutable values

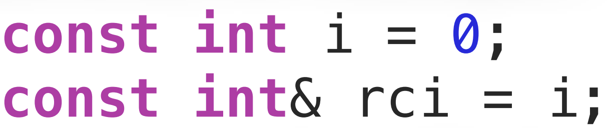
- Frequently, we do not need to modify the value of a variable after we have first created it
- In C++, we can use the `const` keyword to mark a variable as immutable (“read only”)
- If we try to modify a `const` variable, we will get a compile error
- The const keyword helps ensure the *correctness* of our programs, by preventing us from accidentally modifying something we shouldn’t
- Get into the habit of making variables “*const by default*”

Const example

```cpp
void print_float(float f) { std::cout << f << ‘\n'; }

int main() {
    const float pi = 3.142f;

    pi = 3.0f; // Okay?
    // NO: compile error (attempt to modify const value)

    const float e;
    e = 2.718f;    // Okay?
    // NO: two compile errors!

    float pi2 = pi; // Okay?
    // YES: creates a copy

    print_float(pi); // Okay?
    // YES: pass by value
}
```

Read-only references

- It is possible to combine the `const` and reference (`&`) type modifiers, for example:



```
const int i = 0;
const int& rci = i;
```

- This is properly called a “*reference to const*”, but is usually just known as a “*const reference*”
  - I prefer the term “*read-only reference*”
- Read-only references are particularly useful as function parameters

Read-only references

- We cannot form a mutable reference to an immutable variable:

```
const int ci = 0;
int& ref = ci; // ERROR!
```

- If permitted, this would allow us to modify `ci` via the name `ref`
- However, we *can* form a read-only reference to a mutable variable:

```
int i = 0;
const int& cref = i; // Okay
```

- Note that this does *not* mean that `i` itself is now immutable: it just means that the name `cref` cannot be used to change it

Pass-by-const-reference

- By far the most common use of read-only references is for function parameters
- This is the best of both worlds: we avoid copying potentially large objects, but have the assurance that the value will not be changed
- A good rule of thumb: pass built-in types *by value* (e.g. `int`, `float`), everything else *by const reference* (e.g. `std::vector`, `std::string`)
- Use mutable (non-const) references rarely, only when you actually need to modify the passed-in value

Pass-by-const-reference example

```
void print_vector(const std::vector<std::string>& vec)
{
    for (const std::string& str : vec) {
        std::cout << str << '\n';
    }
}

int main()
{
    std::vector<std::string> vec = { "a", "b" };
    vec.push_back("c");

    print_vector(vec);
}
```

Const, references and auto

- As we saw last week, the `auto` keyword may be used to *deduce the type* of a local variable from its initialiser
- However, plain `auto` **never deduces a const or reference type**!
  - Remember, C++ uses value semantics by default, and auto is consistent with that
- This means that if we want a reference of deduced type, we need to say `auto&`
- Likewise, if we want an immutable variable, we need to say `const auto`

Const, references and auto example (1)

```cpp
int main()
{
    int i = 0;
    const int ci = 1;
    int& ref = i;
    const int& cref = ci;

    auto a1 = i;
    auto a2 = ci;
    auto a3 = ref;
    auto a4 = cref;
    // a1-a4 are all int
}
```

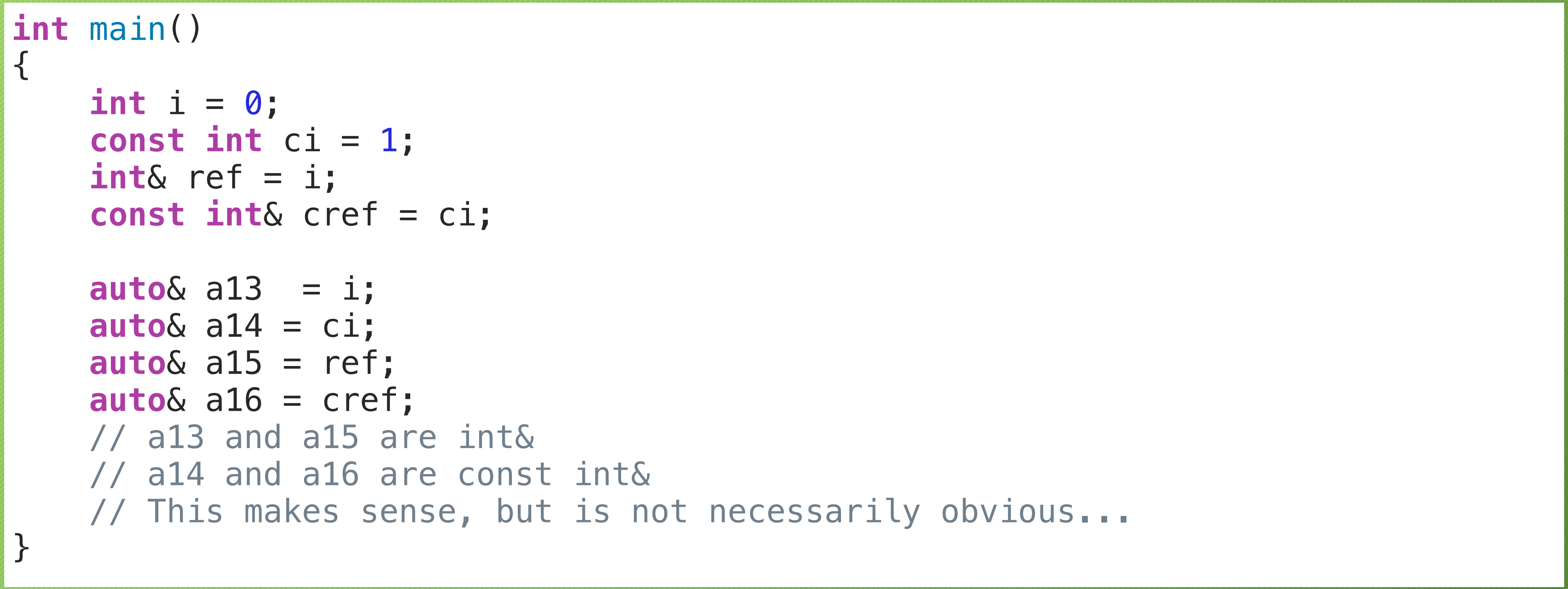
Const, references and auto example (2)

```cpp
int main()
{
    int i = 0;
    const int ci = 1;
    int& ref = i;
    const int& cref = ci;

    const auto a5  = i;
    const auto a6 = ci;
    const auto a7 = ref;
    const auto a8 = cref;
    // a5-a8 are all const int
}
```

Const, references and auto example (3)

```cpp
int main()
{
    int i = 0;
    const int ci = 1;
    int& ref = i;
    const int& cref = ci;

    const auto& a9  = i;
    const auto& a10 = ci;
    const auto& a11 = ref;
    const auto& a12 = cref;
    // a9-a12 are all const int&
}
```

Const, references and auto example (4)



```
int main()
{
    int i = 0;
    const int ci = 1;
    int& ref = i;
    const int& cref = ci;

    auto& a13  = i;
    auto& a14 = ci;
    auto& a15 = ref;
    auto& a16 = cref;
    // a13 and a15 are int&
    // a14 and a16 are const int&
    // This makes sense, but is not necessarily obvious...
}
```

Exercise

- Write a function `print_string()` that takes a read-only (const) reference to a `std::string` as an argument, and prints the string using `std::cout`
- Call your function with a local string variable. Verify that it works correctly.
- Call your function with a string literal. What happens? Why?
- Modify your function so that it takes the string argument by value instead. Does the code still compile? What happens? Why?
- Modify your function so that it takes the string argument by non-const reference instead. Does the code still compile? What happens? Why?

Namespaces

- When we create our programs, we usually don’t want to have to write everything from scratch ourselves
- We’d prefer to build on top of other *libraries* that other people have written
- When writing a large program, we might use many different libraries from many different sources
- Problem: what happens if two different libraries use the *same names* for different types or functions?

Name clashing example

```
// In graphics.hpp
void draw(const Cowboy& cb); // draws Cowboy on screen

// In gameplay.hpp
void draw(const Cowboy& cb); // unholsters pistol

// In processing.cpp
#include <graphics.hpp>
#include <gameplay.hpp>

void process_character(const Cowboy& cowboy)
{
    draw(cowboy); // bang?
}
```

Name clashing: prefixes

- One solution might be to *prefix* the names of all functions and types within a library with the (possibly abbreviated) name of that library
- For example, for a graphics library we might say gfx_draw(const Cowboy&)
- This is the solution used by C and some other languages
- Some older C++ libraries (for example Qt) also use this approach

Namespaces

- C++ has a built-in solution to the problem of clashing names, called *namespaces*
- We can declare our functions, types, aliases, templates etc as belonging to a particular *namespace*
- The same names can be safely re-used in different namespaces without the risk of clashes
- To call a function from a different namespace, we use the namespace name, followed by a double-colon (::), followed by the function name
- We've already seen this! All functions and types in the standard library are in the `std` namespace

Namespace example

```cpp
// In graphics.hpp
namespace gfx {

void draw(const Cowboy& cb); // draws Cowboy on screen

}

// In gameplay.hpp
namespace game {

void draw(const Cowboy& cb); // unholsters pistol

}

// In processing.cpp
#include <graphics.hpp>
#include <gameplay.hpp>

void process_character(const Cowboy& cowboy)
{
    gfx::draw(cowboy); // draws
    game::draw(cowboy); // shoots
}
```

Namespaces

- To declare things as belonging to a namespace, we use the keyword `namespace`, followed by its name, and then a list of declarations in curly braces { }
- Within a namespace, we can refer to other things in the same namespace *without needing to use a prefix*
- This can lead to code which is much less verbose compared with C-style prefixes
- Code which does not belong to any namespace is said to be in the *global namespace*

Namespace example

```cpp
namespace maths { // open maths namespace

float sum(const std::vector<float>& vec) {
    float total = 0.0f;
    for (auto f : vec) {
        total += f;
    }
    return total;
}

float average(const std::vector<float>& vec) {
    const auto total = sum(vec); // same namespace, no need for maths:: prefix
    return total/vec.size();
}

} // close maths namespace

int main() {
    const std::vector<float> vec = { 1.1f, 2.2f, 3.3f };
    const auto mean = maths::average(vec);
}
```

Namespace aliases

- In **very** large projects, namespaces may be nested: for example, the standard library has a `std::filesystem` *sub-namespace*
- This can lead to `very::long::function::names::indeed!`
- One way to avoid this is to use a *namespace alias* — that is, a new local shorthand for an existing (sub-)namespace name
- For example, we might say

```
namespace fs = std::filesystem;
```

- Now we can use names from std::filesystem by saying, for example

```
fs::path my_file = "C:\Documents\important.doc";
// my_file is a std::filesystem::path object
```

`using namespace`

- To avoid having to type prefixes everywhere, C++ allows us to say "`using namespace` *example*`;`"
- This makes all the declarations inside the namespace *example* available without needed to type `example::` first
- This appears attractive, but puts us at risk of name clashes and unexpected functions being called – the reason we used namespaces in the first place
- This is particularly the case for large namespaces like `std`
- Recommendation: avoid `using namespace` unless you're sure you know what you're doing
- NEVER put a `using namespace` declaration in a header

~Module();

- Congratulations on making it to the end of *Initial C++*!
- Next week we’ll begin a new module, “Custom Types”
- Also next week: end of module fun time questionnaire!

Homework/revision

1. Write a function lower_case() which takes a reference to a std::string, and modifies the string so that every character is changed to lower case.
   - For example, the string “MiXeD” should be changed to “mixed”.
2. Write a function headline_case() which takes a reference to a string. Change the first character of every word of the string into a capital letter.
   - For example, the string “this is a headline” should be changed to “This Is A Headline”
3. Write a function sentence_case() which changes the first character of every sentence into a capital letter. Sentences are separated by a full stop followed by a single space character.
   - For example, the string “this is sentence 1. this is sentence 2.” should be changed to “This is sentence 1. This is sentence 2.”
4. Write a function sentence_case2() which does the same as sentence_case(), except that sentences may be separated by an arbitrary number of whitespace characters.
   - For example, the string “sentence 1. \n\t\r sentence 2.” should be changed to “Sentence 1. \n\t\r Sentence 2.”
   - (HINT: the function std::isspace() can be used to check for whitespace characters.)
5. Write a function sentence_case3() which does the same as sentence_case2(), except that all whitespace characters between sentences are replaced with a single space character.
   - For example, the string “sentence 1. \n\t\r sentence 2.” should be changed to “Sentence 1. Sentence 2.”

Thank You!

As usual, we will be going to the pub! Support us @ https://patreon.com/CPPLondonUni