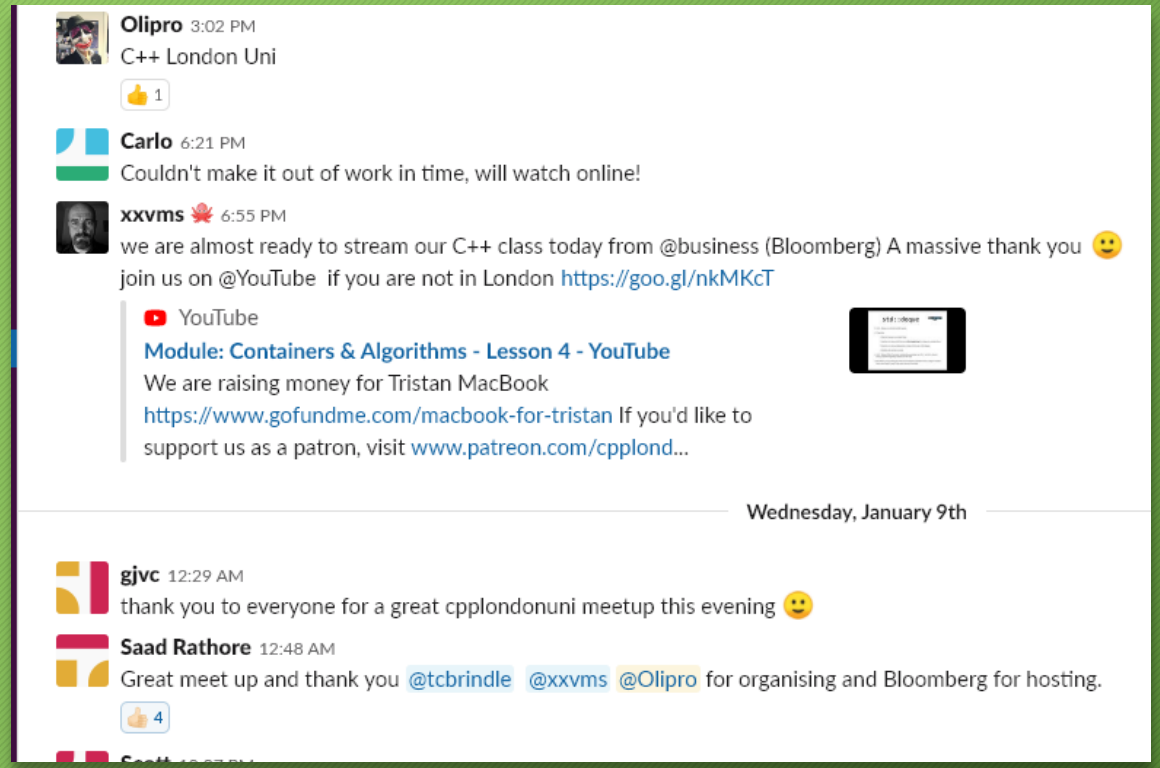
Initial C++ - Lesson 5

The Auto Keyword & References

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.



Previous Lesson Recap

- Last week, we covered:
  - Loops in C++
    - while()
      - Do/while()
    - for()
    - The range-based for() loop.
  - break;
  - continue;

Last Week's Homework

- Write a program which reads in a sequence of strings from the user with std::cin, and stores them in a std::vector
- Continue reading strings until the user enters the string "quit"
- Once the user has typed "quit", print out all the strings that they entered, EXCEPT those which begin with the letter 'b'.
  - For example, given the input:
    - Apricot banana cherry date quit
  - The program should print:
    - apricot cherry date

Possible Homework Solution

```
#include <iostream>
#include <string>
#include <vector>

int main() {
    std::string item;
    std::vector<std::string> items;
    while (std::cin >> item && item != "quit")
        items.push_back(item);
    for (std::string item : items)
        if (!item.empty() && item[0] != 'b' && item[0] != 'B')
            std::cout << item << ' ';
}
```

The auto keyword

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, you blow your whole leg off"

- Bjarne Stroustrup

Automatic Type Deduction

- So far, we have been explicitly stating the type of our variables.
  - For example:
    - for(std::string word : words)
- However, the compiler can deduce the type for us.
- We do this with the **auto** keyword.
  - For example, we can write the same loop above using auto:
    - for(auto word : words)
- The code is identical; “word” is still of type std::string.

Some Examples of auto


```
int main() {
    auto x = 53; //x is an int.

    auto y = x; //y is also an int.

    std::vector<std::string> words;

    for (auto word : words) // word is an std::string.
        std::cout << word;
}
```

The auto Keyword - Caveats

- Auto will only work when the compiler has a valid type to deduce from.
  - Consequently, writing “auto x;” will not compile.
    - This can be useful as it prevents you from forgetting to initialise the value.
- Additionally, `auto str = “Hello”;` will not be std::string.
  - This is because the actual type is `const char*`
  - Don’t worry about what that is for now, just be aware of it.
    - Continue using std::string when you need it.
- Remember, auto is for convenience, it’s not a requirement.

Quick Revision: Value Semantics

- In the absence of qualifiers, C++ uses value semantics.
- This means (roughly) that copies of variables are distinct: modifying a copy does not affect the original
  - That is, copies are “deep”
- This applies to built-in (fundamental) types like **int** and **float**, and also to library types like `std::string` and `std::vector`
- Some other languages have differing behavior between “*value types*” and “*reference types*” - C++ does not

Value Semantics Explained


```cpp
int main() {
    int a = 1;
    int b = a;
    // b has value 1

    a = 42;
    // a now has value 42
    // b still has value 1

    std::string s = "Hello ";
    std::string s2 = "World";

    s += s2;
    // s now has value "Hello World"
    // s2 still has value "World"
}
```

Quick Revision: Pass-By-Value

- Typically, when we call a function, it receives a copy of the arguments we pass.
  - This is known as pass-by-value.
    - The function gets a copy of each *value* that is *passed* to it.
- This has the caveat that copying can take time and slow us down.
  - An std::vector of 10 million elements might take a long time to copy.
- When we initialize a variable, such as “int x = y;”, this is also value semantics.
  - The variable “x” will copy whatever is in ”y”.

Reference Semantics

- Instead of creating a new copy, we can instead append “&” onto the end of the type to indicate we want a reference.
  - A reference type creates another name for the same thing (an alias)
- For example:
  - `void example(float val, int& ref);`
- Any changes the function makes to `ref` will be seen by the caller.
  - `val` is still a copy since it lacks the “&” qualification on its type.

Reference Semantics Example

```
void say_hello(std::string& name) { // here, name is passed BY REFERENCE
    name = "Hello " + name; // operates on the passed-in string *directly*, not a copy
}

int main() {
    std::string tom = ”Tom";
    say_hello(tom);

    std::cout << tom << '\n’; // prints "Hello Tom"
}
```

Reference Semantics

- The reference (&) is considered to be part of the type.
- Thus, float& is a *different type* to float.
- We typically pronounce it as "*reference to float*" or "*float ref*" for brevity.
- References are very common in function parameters but can of course be used elsewhere too.
- We can also combine it with auto to give us auto&.
  - This gives us type deduction and makes it a reference to that type!

Reference Semantics - Caveats

- A function can also return a reference type.
- However, this can be dangerous – for example:
  - Your function creates a variable inside its code body.
  - Your function then returns that variable by reference.
  - When we return from the function, all of its values are destroyed.
  - You have passed a reference to a destroyed value!
- If you manage to do this, anything can happen.
  - Your program might appear to work normally.
  - It might unceremoniously crash immediately.
- This is known as a "*dangling reference*" and is a form of UB.

Dangling Reference - Example

```
int& bad_function(int x) {
    int ret = x + 5;
    return ret;
}

int main() {
    int boom = bad_function(2); //UB!!!
}
```

Exercise

- In CLion, create a new C++ executable project
- In main.cpp, write a function void increment(int& i) which adds one to i
- In your main() function, call increment() on a local int variable and print the result to see that the function directly modifies the argument you pass to it
- Try removing the & from the definition of increment(). Does the code still compile? What happens? Why?
- Try calling increment(42). Does the code still compile? What happens? Why?

Solution

```
void increment(int& i) { ++i; }

void increment2(int i) { ++i; }

int main() {
    int val = 0;
    increment(val);

    std::cout << val << '\n'; // prints 1
    increment2(val);
    std::cout << val << '\n'; // prints 1(!)

    // increment(42) -- does not compile
    increment2(42); // compiles (but useless)
}
```

The const Keyword

The Const Keyword

- In C++, sometimes we don’t need or want data to be modified.
- We might also want to indicate in a function that we won’t modify it.
- There is a keyword that allows us to achieve this: const.
  - The keyword is part of the type, much like a ref qualifier is (&)
- When a value is const, it becomes read-only.
  - If we try to write to or otherwise change it, the code will not compile.
- For example “const int x = 5;” will be read-only.

Exercise 2

- Create a function which takes a reference to a std::vector<std::string> and a single std::string.
  - The function should insert the std::string into the vector unless the std::string is empty.
  - See what happens if you add const to one or both of the parameters.
- Create another function which takes a **const** reference to a std::vector<std::string> and a const std::string reference.
  - The function should only print elements that don't match the string argument that was passed in.
- Write some code in main() to try out your functions.

Solution

```
void insert_in_vector(std::vector<std::string>& vec, std::string value) {
    if (!value.empty())
        vec.push_back(value);
}

void print_non_matching(const std::vector<std::string>& vec, conststd::string& skip) {
    for (auto& str : vec)
        if (str != skip)
            std::cout << str << ' ‘;
}

int main() {
    std::vector<std::string> vec;
    insert_in_vector(vec, "Tom");
    insert_in_vector(vec, "Oliver");
    insert_in_vector(vec, "Bob");
    insert_in_vector(vec, "Tristan");
    std::string skipThisGuy = "Bob”
    print_non_matching(vec, skipThisGuy);
}
```

Thank You!

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