Modern C++ (The Good Parts)

CS1570 — Introduction to Programming

Illya Starikov

June 30, 2025

Since its creation, C++ has gone through many different "versions" — more formally, known as standards. With `fg++`, you've been using C++03[1]. Since then, there have been two major standards released: C++11 and C++14.

C++11 introduced many modern programming paradigms from other programming languages, and C++14 built on these principles. Namely, some of these features are type inference, lambda expressions, constant expressions, `default` and `deleted` keywords, and much more!

To compile C++11 or C++14 code, the `-std=c++11` or `-std=c++14` flags must be added (respectively). So, to compile with C++14, the command would be

```
g++ -std=c++14 *.cpp
```

1 Type Inference (`auto`)

Type inference is a way for a language to systematically "choose" the type you are trying to use. This can intuitively be thought of as you, the programmer, guessing the type of a math equation:

$$\text{answer} = 42 \tag{1}$$

Mathematically, we know answer to be an integer — subsequently, the compiler (GCC) should guess its type to be an `int`. Wouldn't it be great if C++ could just magically do that? Well, now it can.

$$\texttt{auto}\ \text{answer} = 42\texttt{;} \tag{2}$$

The actual syntax is

[1] Yes, that is "hella" old

```
auto variable = inference;
```

Now, we no longer have to type out those pesky return types (very useful if it's constantly changing). Let's do an example.

```cpp
template <typename T>
T oneUp(const T value) {
    return value + 1;
}

int main(int argc, char *argv[]) {
    auto integer = oneUp(1);
    auto character = oneUp('P');
    auto string = oneUp("vs NP");
    auto boolean = oneUp(false);
}
```

Compare `main` to it's pre-C++11/14 version.

```cpp
template <typename T>
T oneUp(const T value) {
    return value + 1;
}

int main(int argc, char *argv[]) {
    int integer = oneUp(1);
    char character = oneUp('P');
    string std_string = oneUp("vs NP");
    bool boolean = oneUp(false);
}
```

So there it is efficient for the programmer. How efficient is it for the program? Let's take another example.

```cpp
    char charArray[42] = "Hello";
    auto length = strlen(charArray);
```

Can you guess the type of `length`? Spoiler alert, it's not `int`. It's actually `size_t`, a different type leftover from the C era. It's not vital knowing what `size_t` is or how it works, it's just vital knowing *it is not an* `int`. So, every time `strlen` it is used as an `int`, i.e.,

```cpp
    for (int i = 0; i < strlen(charArray); i++) {
        /* do something here */
    }
```

this has a performance cost. Why? `size_t` has to be downcast to an integer.

A note, in C++14, `auto` can be used as the return type of a function.

1.1 Range Based For Loops (`for (auto)`)

With `auto`, range based for loops are possible! What are these strange loops? They are just a way to iterate through all the element of a collection type (i.e., arrays).

```
auto  coolTeachers = { "Illya", "Andrea", "Not Price" };

for (auto teacher : coolTeachers) {
    cout << teacher << " ";
}
```

As you can guess, the output is "`Illya Andrea Not Price`". Raaad. Some reasons it might be useful.

- Iterating through more complex data structures becomes much easier.
- It's more readable.
- You don't have to worry about proper indexing.

2 Lambda Expressions

Such a scary name for a scary topic. The simplest definition that will make thinking about them easier:

> Functions as variables and types.

That's it. Now, the syntax:

```
[ capture ]( parameters ) { functionBody }
```

For now, ignore the capture part. We will not be using it for this course. But now here is where shit gets crazy. Let's do an example.

```
std::function<bool(int, int)> lessThan = [](int x, int y) { return x < y; };
```

First, notice the ugly return type `std::function<bool(int, int)>` — how can we get rid of it? `auto`! Second, it's not so bad! It sort of like a function but we are assigning it to a variable called `lessThan`.

```
auto lessThan = [](int x, int y) { return x < y; };

cout << lessThan(128, 256); /* Prints 1 for true */
```

How else did we use variables? Functions!

```cpp
auto sort(std::function<bool(int, int)> compare, int array[], const int size) {
    // This is just a bubble sort
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size - i - 1; j++) {

            // Here's where we use compare
            if (compare(array[j], array[j + 1])) {
                int temporary = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temporary;
            }
        }
    }
}

int main(int argc, char *argv[]) {
    auto lessThan = [](int x, int y) { return x < y; };
    int array[5] = { 1, 2, 3, 4, 5 };
    sort(lessThan, array, 5);

    for (auto element : array) {
        cout << element << " ";
    }
}
```

We can also return lambda expressions! Don't mind the `[]`, again it's an advanced feature. For this, all you have to know is it signifies the variable is "held" in temporary memory.

```cpp
auto counter(const int start, const int incrementor) {
    return [=]() {
        static auto x = start;
        x += incrementor;

        return x;
    };
}

int main(int argc, char *argv[]) {
    auto count = counter(42, 2);

    cout << count() << "\n" << count() << "\n" << count();
    /* prints 44, 46, 48 */
}
```

We can use them exactly as we have used all variables — hold them in arrays

```cpp
    auto styleChecker = []() {
        ⋮
        std::cout << "Style Checking...\n";
    };
    auto plagarismChecker = []() {
```

```
        ⋮
        std::cout << "Checking For Plagarism (Caught 2)...\n";
    };
    auto inputOutputChecker = []() {
        ⋮
        std::cout << "Checking Input/Output...\n";
    };

    std::function<void()> graderStack[5] = { styleChecker,
   plagarismChecker, inputOutputChecker };

    for (auto gradeFunction: graderStack) {
        if (gradeFunction != NULL) {
            gradeFunction();
        }
    }
```

Notice the checking of line `gradeFunction != NULL`. We have three functions in the array, but the size is five. If try to call the function, without a function in there, *it will cause a runtime error*. So be careful.

And finally, we can add them as variables to classes. I won't give an example here, because it's pretty straightforward.

3 Constant Expressions `constexpr`

`constexpr` is a way to move calculations from runtime to compile time. So, if you have a function that's going to compute **known** values ahead of time. So, if you're going to be calculating $\ln 2$ ahead of time often,

```
constexpr double ln(const int x) {
    double sum = 0;

    for (int n = 1; n <= 100; n++) {
        sum += 1.0/n/(4*n - 2);
    }

    return sum;
}
```

Why might this be useful? Performance.

4 The `default` And `delete` Keywords

Remember when you implemented one constructor, and suddenly you lost the copy constructor and the assignment operator? Welp, it took roughly 30 years to figure out this was a pain in the ass for everyone. Now, if you overload the copy constructor, you can get the default constructor by prototyping it, then putting `= default`.

```
class Zombie {
    string nameOfGradStudent;

public:
    Zombie(const string name) {
        nameOfGradStudent = name;
    }


};

int main(int argc, char *argv[]) {
    Zombie Fred; // compiler error
}
```

This won't compile, Fred doesn't have a default constructor. But, if did something like

```
class Zombie {
    string nameOfGradStudent;

public:
    Zombie(const string name) {
        nameOfGradStudent = name;
    }

    Zombie() = default;
};

int main(int argc, char *argv[]) {
    Zombie Fred; // compiler error
}
```

`delete` works the same way, except the opposite. Instead of gaining the default, it simply deletes the function. So, if there is no way possible to have a default constructor for the `Zombie` class, something like this is possible:

```
class Zombie {
    string nameOfGradStudent;

public:
    Zombie(const string name) {
        nameOfGradStudent = name;
    }

    Zombie() = delete;
};

int main(int argc, char *argv[]) {
```

```
    Zombie Fred; // compiler error
}
```

This is useful when you're creating a class for someone else to use. It makes the deletion of certain functionality more explicit.

5 Further Topics

If time permitted, some topics of further discussion:

- Preventing Exception Propagation (`noexcept`)
- Smart Pointers
- Move Semantics
- Variadic Templates
- There's no way we'd get here.