

# Some awesome modern C++ features that every developer should know

 [freecodecamp.org/news/some-awesome-modern-c-features-that-every-developer-should-know-5e3bf6f79a3c](https://freecodecamp.org/news/some-awesome-modern-c-features-that-every-developer-should-know-5e3bf6f79a3c)

8 May 2019

[Forum](#) [Donate](#)

[Learn to code — free 3,000-hour curriculum](#)

freeCodeCamp ()

8 May 2019 / [#Programming](#)



**M Chowdhury**



As a language, C++ has evolved a lot.

Of course this did not happen overnight. There was a time when C++ lacked dynamism. It was difficult to be fond of the language.

But things changed when the C++ standard committee decided to spin up the wheel.

Since 2011, C++ has emerged as a dynamic and ever-evolving language that a lot of people have been hoping for.

Don't get the wrong idea that the language has become easier. It still is one of the hardest programming languages, if not the hardest one, that are used widely. But C++ has also become much more user friendly than its previous versions.

In my last post, I talked about the C++ algorithm library that has been enriched over the last couple of years.

Today, we shall look into some new features (starting from C++11, which is already 8 years old by the way) that every developer would like to know.

Also note that I have skipped some advanced features in this article, but I'm willing to write about them in future. ?

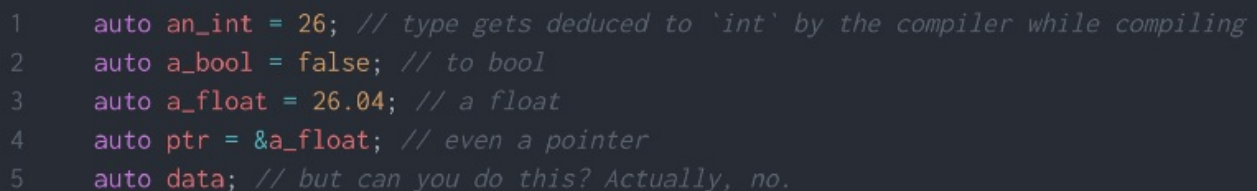
Go!

## The auto keyword

---

When C++11 first introduced **auto**, life became easier.

The idea of **auto** was to make the C++ compiler deduce the type of your data while compiling — instead of making you declare the type *every-freaking-time*. That was so convenient when you have data types like **map<string,vector<pair<int,int>>>** ?



```
1  auto an_int = 26; // type gets deduced to 'int' by the compiler while compiling
2  auto a_bool = false; // to bool
3  auto a_float = 26.04; // a float
4  auto ptr = &a_float; // even a pointer
5  auto data; // but can you do this? Actually, no.
```

Look at the line number 5. You cannot declare something without an **initializer**. That actually makes sense. Line 5 doesn't let the compiler know what can the data type be.

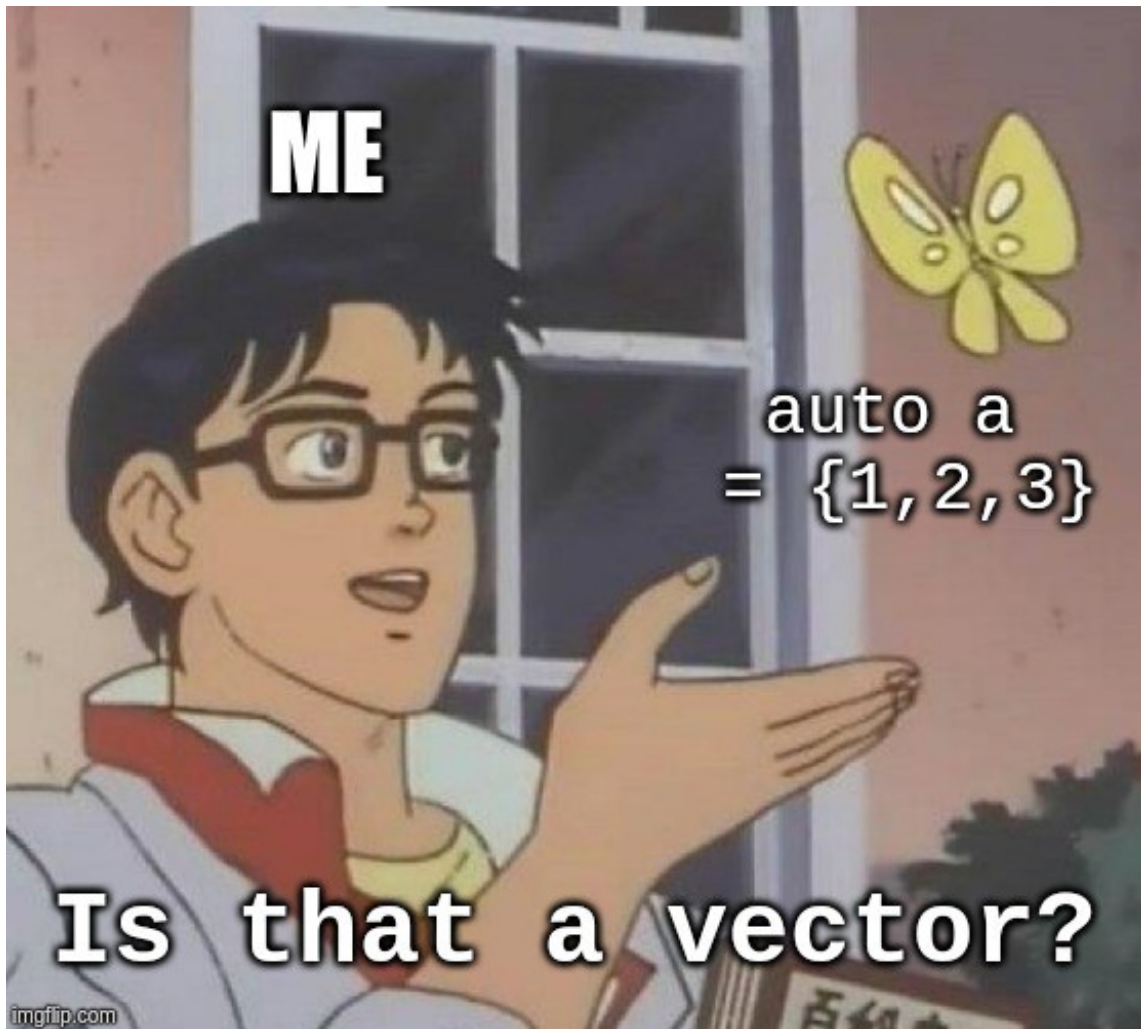
Initially, **auto** was somewhat limited. Then in the later versions of the language, more power was added to it!

```
1 auto merge(auto a, auto b) // Parameters and return types of the function can be auto!
2 {
3     std::vector<int> c = do_something(a, b);
4     return c;
5 }
6
7 std::vector<int> a = { ... }; // some data
8 std::vector<int> b = { ... }; // some more data
9 auto c = merge(a,b); // type deduced by the received data!
```

In lines 7 and 8, I used bracketed initialization. This was also a new feature added in C++11.

Remember, in case of using **auto**, there must be some way for the compiler to deduce your type.

Now a very nice question, *what happens if we write* **auto a = {1, 2, 3}** ? Is that a compile error? Is that a vector?



smh ?

Actually, C++11 introduced `std::initializer_list<type>`. Braced initialized list will be considered as this lightweight container if declared `auto`.

Finally, as I previously mentioned, type-deducing by compiler can be really useful when you have complex data structures:

```

1 void populate(auto &data) { // see!
2     data.insert({"a",{1,4}});
3     data.insert({"b",{3,1}});
4     data.insert({"c",{2,3}});
5 }
6
7 auto merge(auto data, auto upcoming_data) { // don't write long declaration again
8     auto result = data;
9     for(auto it: upcoming_data) {
10         result.insert(it);
11     }
12     return result;
13 }
14
15 int main() {
16     std::map<std::string, std::pair<int,int>> data;
17     populate(data);
18
19     std::map<std::string, std::pair<int,int>> upcoming_data;
20     upcoming_data.insert({"d",{5,3}});
21
22     auto final_data = merge(data,upcoming_data);
23
24     for(auto itr: final_data) {
25         auto [v1,v2] = itr.second; // structured bindings discussed below
26         std::cout << itr.first << " " << v1 << " " << v2 << std::endl;
27     }
28     return 0;
29 }

```

Don't forget to check out line 25! The expression `auto [v1,v2] = itr.second` is literally a new feature in C++17. This is called **structured binding**. In previous versions of the language, you had to extract each variable separately. But structured binding has made it much more convenient.

Moreover, if you wanted to get the data using reference, you would just add a symbol — `auto &[v1,v2] = itr.second` .

Neat.

## The lambda expression

---

C++11 introduced lambda expressions, something like anonymous functions in JavaScript. They are function objects, without any names, and they capture variables on various *scopes* based on some concise syntax. They are also assignable to variables.

Lambdas are very useful if you need some small quick thing to be done inside your code but you are not willing to write a whole separate function for that. Another pretty common use is to use them as compare functions.

```
1 std::vector<std::pair<int,int>> data = {{1, 3}, {7, 6}, {12, 4}}; // note the bracketed initialization
2 std::sort(begin(data), end(data), [](auto a, auto b) { // auto!
3     return a.second < b.second;
4 });
```

The above example has a lot to say.

Firstly, notice how curly braced initialization is lifting the weight for you. Then comes generic **begin()**, **end()** that is also an addition in C++11. Then comes the lambda function as a comparator for you data. The parameters of the lambda function are declared **auto** which was added in C++14. Before that, we could not use **auto** for function parameters.

Note how we start the lambda expression with a square bracket **[]**. They define the scope of the lambda — how much authority it has over the local variables and objects.

As defined in this [awesome repository](#) on modern C++:

- **[]** — captures nothing. So you cannot use any local variable of the outer scope inside your lambda expression. You can only use the parameters.
- **[=]** — captures local objects (local variables, parameters) in scope by value. You can use them, but cannot modify them.
- **[&]** — capture local objects (local variables, parameters) in scope by reference. You can modify them. Like the following example.
- **[this]** — capture **this** pointer by value.
- **[a, &b]** — capture objects **a** by value, **b** by reference.

So if, inside your lambda function, you want to transform your data into some other format, you can use lambda by taking the advantage of the scoping. For example:



```

1 std::vector<int> data = {2, 4, 4, 1, 1, 3, 9};
2 int factor = 7;
3 for_each(begin(data), end(data), [&factor](int &val) { // capturing factor by reference
4     val = val * factor;
5     factor--; // this works because lambda has scope to change this variable
6 });
7
8 for(int val: data) {
9     std::cout << val << ' '; // 14 24 20 4 3 6 9
10 }

```

In the above example, if you had captured local variables by value ( **[factor]** ) in your lambda expression, you could not change **factor** in line 5. Because simply, you have no right to do that. Don't misuse your rights! ?

Finally, notice that we take **val** as reference. This ensures that any change inside the lambda function actually changes the **vector** .



They feel joyous after learning about modern C++! (Photo by [Ian Schneider](#) on [Unsplash](#))

## Init statements inside if & switch

---

I really liked this feature of C++17 immediately after I got to know of it.

```
1 std::set<int> input = {1, 5, 3, 6};
2
3 if(auto it = input.find(7); it==input.end()) { // first part is init, second is condition
4     std::cout << 7 << " not found!" << std::endl;
5 }
6 else {
7     // this else block has scope of 'it'!
8     std::cout << 7 << " is there!" << std::endl;
9 }
```

So apparently, now you can do initialization of variables and check condition on that — simultaneously inside the **if/switch** block. This is really helpful to keep your code concise and clean. The general form is:

```
if( init-statement(x); condition(x)) {
    // do some stuff here
} else {
    // else has the scope of x
    // do some other stuff
}
```

### Do it in compile time by constexpr

---

**constexpr** is cool!

Say you have some expression to evaluate and its value won't change once initialized. You can pre-calculate the value and then use it as a macro. Or as C++11 offered, you can use **constexpr** .

Programmers tend to reduce runtime of their programs as much as possible. So if there are some operations you can make the compiler do and take the load off runtime, then the runtime can be improved.



```

1 constexpr double fib(int n) { // function declared as constexpr
2     if(n == 1) return 1;
3     return fib(n-1) * n;
4 }
5
6 int main() {
7     const long long bigval = fib(20);
8     std::cout<<bigval<<std::endl;
9 }

```

The above code is a very common example of **constexpr**.

Since we declared the fibonacci calculation function as **constexpr**, the compiler can pre-calculate **fib(20)** in compile time. So after compilation, it can replace the line

**const long long bigval = fib(20);** with

**const long long bigval = 2432902008176640000;**

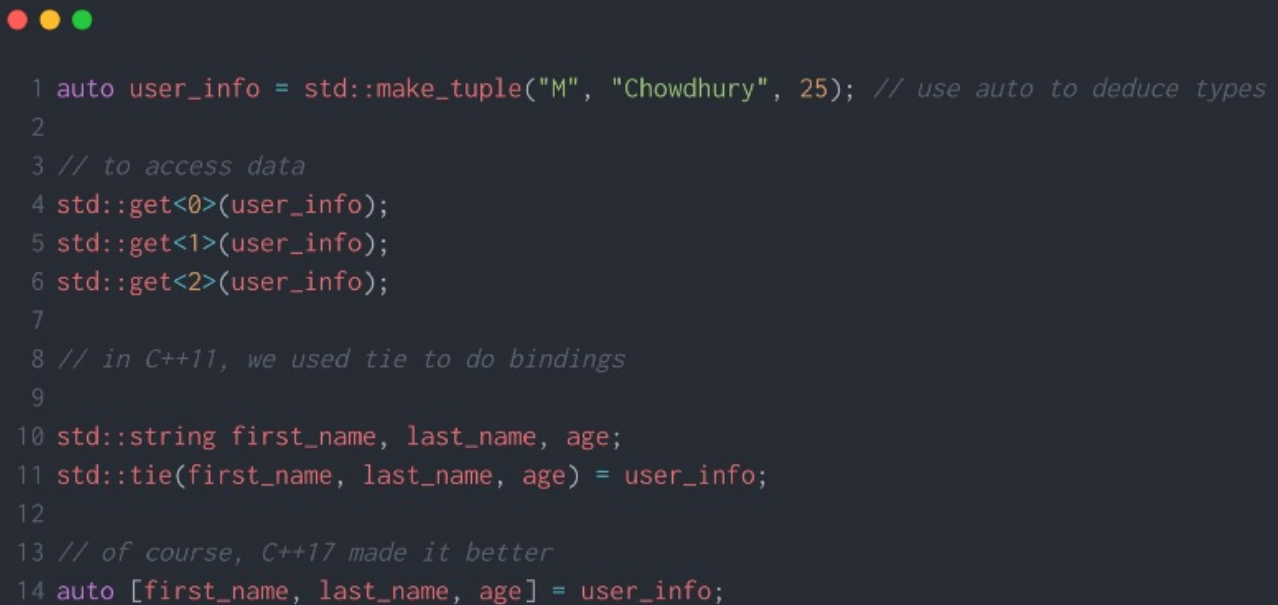
Note that the passed argument is a **const** value. This is one important point of functions declared **constexpr** — the arguments passed should also be **constexpr** or **const**. Otherwise, the function will behave as a normal function which means no pre-calculation during compile time.

Variables can also be **constexpr**, too. In that case, as you can guess, those variables have to be evaluable in compile time. Otherwise, you get a compilation error.

Interestingly, later in C++17, **constexpr-if** and **constexpr-lambda** were introduced.

## Tuples

Much like **pair**, **tuple** is a collection of fixed size values of various data types.



```

1 auto user_info = std::make_tuple("M", "Chowdhury", 25); // use auto to deduce types
2
3 // to access data
4 std::get<0>(user_info);
5 std::get<1>(user_info);
6 std::get<2>(user_info);
7
8 // in C++11, we used tie to do bindings
9
10 std::string first_name, last_name, age;
11 std::tie(first_name, last_name, age) = user_info;
12
13 // of course, C++17 made it better
14 auto [first_name, last_name, age] = user_info;

```

Sometimes it is more convenient to use `std::array` instead of `tuple`. `array` is similar to plain C type array along with couple of functionalities of the C++ standard library. This data structure was added in C++11.

## Class template argument deduction

---

A very verbose name for a feature. The idea is, from C++17, argument deduction for templates will also happen for standard class templates. Previously, it was supported for only function templates.

As a result,

```

std::pair<std::string, int> user = {"M", 25}; // previous
std::pair user = {"M", 25}; // C++17

```

The type of deduction is done implicitly. This becomes even more convenient for `tuple`.

```

// previous
std::tuple<std::string, std::string, int> user ("M", "Chy", 25);
// deduction in action!
std::tuple user2("M", "Chy", 25);

```

This feature above won't make any sense if you are not quite familiar with C++ templates.

## Smart pointers

---

Pointers can be hellish.

Due to the freedom that languages like C++ provide to programmers, it sometimes becomes very easy to shoot yourself in the foot. And in many cases, pointers are responsible for the harm.

Luckily, C++11 introduced smart pointers, pointers that are far more convenient than raw pointers. They help programmers to prevent memory-leaks by freeing it when possible. They also provide exception safety.

I thought of writing about the smart pointers in C++ in this post. But apparently, there are lots of important details about them. They deserve their own post and I am certainly willing to write one about them in near future.

That's all for today. Remember that C++ actually added a lot more newer features in the latest versions of the language. You should check them out if you feel interested. Here is an awesome repository on modern C++ which is literally named [Awesome Modern C++!](#)

Adios!

---



---

**M Chowdhury**

Read [more posts](#) by this author.

---

If this article was helpful, [tweet it](#).

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. [Get started](#)