C++ Tips

Edwin Kofler

El Camino College Computer Science Club

Unknown, 2023

Table Of Contents

Introduction

Have you ever wondered if there is a better way of doing things while writing C++? Or are you tired of debugging the same errors over and over again?

I'll introduce features that:

1. show you more errors at compile-time (so you don't have to debug them at runtime)
2. improve code maintainability
3. save time
4. you'll come across when reading others' code

Assume C++11

Compiling with more errors enabled

```
$ gcc ./main.cpp && ./a.out
$ gcc ./main.cpp -Wall -Wextra -pedantic && ./a.out
```

- VSCode
- Visual Studio
- Xcode

Using nullptr

- Use `nullptr` instead of `NULL`
- More errors at compile time

```
1  #include <iostream>
2
3  void print_number(int i) {
4      std::cout << "number: " << i << '\n';
5  }
6
7  int main() {
8      print_number(40); // ?
9      print_number(NULL); // ?
10     print_number(nullptr); // ?
11 }
```

Initialization with auto

Before

```
1  int main() {
2      int i = 10; // int
3      long j = 2L; // long
4      float k = 8.0F; // float
5      float l = 15.1; // float
6  }
```

After

```
1  int main() {
2      auto i = 10; // ?
3      auto j = 2L; // ?
4      auto k = 8.0F; // ?
5      auto l = 15.1; // ?
6  }
```

- term: placeholder type specifier

Initialization with auto: Use case 1

1. Initializing a variable with a long type name

```
1  class SomeVeryLongClassNameWow {
2      ...
3  };
4
5  int main() {
6      // A
7      SomeVeryLongClassNameWow myInstance1 = SomeVeryLongClass
8
9      // B
10     auto myInstance2 = SomeVeryLongClassNameWow();
11 }
```

Initialization with auto: Use case 2

2. Avoid repetition

```
1  #include <vector>
2
3  int main() {
4      // A
5      std::vector<int> myVector1 = std::vector<int>{1, 2, 3};
6
7      // B
8      auto myVector2 = std::vector<int>{1, 2, 3};
9  }
```

Trailing Return Types

- What is the return type of *sayHi*?

Before

```
1  void sayHi() {
2      std::cout << "Hi!\n";
3  }
4
5  int main() {
6      sayHi();
7  }
```

Trailing Return Types

- What is the return type of *sayHi*?

Before

```
1  void sayHi() {
2      std::cout << "Hi!\n";
3  }
4
5  int main() {
6      sayHi();
7  }
```

After

```
1  auto sayHi() {
2      std::cout << "Hi!\n";
3  }
4
5  int main() {
6      sayHi();
7  }
```

- Note: Deduction isn't always what you want (`const`, `volatile`)

```
1  auto returnOne() {
2      int const i = 1;
3      return i;
4  }
```

Trailing Return Types (cont.)

Usually, you would specify the type
Can also specify manually:

```
1  auto sayHi() -> void {
2      std::cout << "Hi!\n";
3  }
4
5  auto returnOne() -> const int {
6      int const i = 1;
7      return i;
8  }
```

- term: trailing return types
- term: return type deduction

Using `std::span` (Before)

How to pass an array of ints to a function?

Using `std::span` (Before)

How to pass an array of ints to a function?

```
1   void print_array(int* arr, int arr_len) {
2     for (int i = 0; i < arr_len; ++i) {
3       std::cout << arr[i] << '\n';
4     }
5   }
6
7   int main() {
8     int my_arr[] = {94, 33, 12};
9     int my_len = sizeof(my_arr) / sizeof(my_arr[0]);
10    print_array(my_arr, my_len);
11  }
```

- Must pass in pointer *and* its length separately
- Must use *sizeof*
- Array is "converted" into a pointer (pointer decay)

Using std::span (After)

With std::span

```
1  #include <iostream>
2  #include <span>
3
4  void print_arr(std::span<int> arr) {
5    for (int i = 0; i < arr.size(); ++i) {
6      std::cout << arr[i] << '\n';
7    }
8  }
9
10 int main() {
11   int* my_arr{94, 33, 12};
12   print_arr(my_arr);
13 }
```

- **Requires C++20**

End