

SEARCH

RESOURCES

CONCEPTS

1. Intro
2. Motion Planning
3. Maze
4. Maze 2
5. Coding the Shortest Path Algorithm
6. A\* Search
7. Lesson Code Structure
8. CODE: Starting A\* Search
9. CODE: Writing the A\* Heuristic
10. Pass by Reference in C++
11. CODE: Adding Nodes to the Ope...
12. CODE: Initialize the Open Vector
13. CODE: Create a Comparison Fun...
14. CODE: Write a While Loop for the...
15. CODE: Check for Valid Neighbors
16. Constants
17. CODE: Expand the A\* Search to ...
18. Arrays
19. CODE: Adding a Start and End to ...
20. Congratulations!!
21. How to Become More Proficient ...

## Constants

This example highlights how to use `const` to promise not to modify a variable, even though the variable can only be evaluated at run time.

The example also show how to use `constexpr` to guarantee that a variable can be evaluated at compile time.

```
In [ ]: #include <iostream>

int main()
{
    int i;
    std::cout << "Enter an integer value for i: ";
    std::cin >> i;
    const int j = i * 2; // i can only be evaluated at run time.
                        // But I promise not to change it after it is initialized.

    constexpr int k = 3; // k, in contrast, can be evaluated at compile time.

    std::cout << "j = " << j << "n";
    std::cout << "k = " << k << "n";
}
```

Compile & Run

Explain

Loading terminal (id\_nau33ys), please wait...

The compiler will catch a `const` variable that changes.

```
In [ ]: #include <iostream>

int main()
{
    const int i = 2; // i promise not to change this.
    i++;             // i just broke my promise.
}
```

Compile & Run

Explain

Loading terminal (id\_kxy40r), please wait...

Similarly, the compiler will catch a `constexpr` variable that changes.

```
In [ ]: #include <iostream>

int main()
{
    constexpr int i = 2; // i can be evaluated at compile time.
    i++;                 // But changing a constexpr variable triggers an error.
}
```

Compile & Run

Explain

Loading terminal (id\_ngdtbd2), please wait...

The major difference between `const` and `constexpr`, though, is that `constexpr` must be evaluated at compile time.

The compiler will catch a `constexpr` variable that cannot be evaluated at compile time.

```
In [ ]: #include <iostream>

int main()
{
    int i;
    std::cout << "Enter an integer value for i: ";
    std::cin >> i;
    constexpr int j = i * 2; // i can only be evaluated at run time.
                            // "constexpr must be evaluated at compile time."
                            // So this code will produce a compilation error.
}
```

Compile & Run

Explain

Loading terminal (id\_6207vcc), please wait...

A common usage of `const` is to guard against accidentally changing a variable, especially when it is passed-by-reference as a function argument.

```
In [ ]: #include <iostream>
#include <vector>

int sum(const std::vector<int> &v)
{
    int sum = 0;
    for(int i : v)
        sum += i;
    return sum;
}

int main()
{
    std::vector<int> v {0, 1, 2, 3, 4};
    std::cout << sum(v) << "n";
}
```

Compile & Run

Explain

Loading terminal (id\_x6tnbdj), please wait...

The distinction between `const` and `constexpr` is subtle.

In general, though, `const` is much more common than `constexpr`.

When in doubt, use `const`, especially to guard against accidentally modifying a variable.