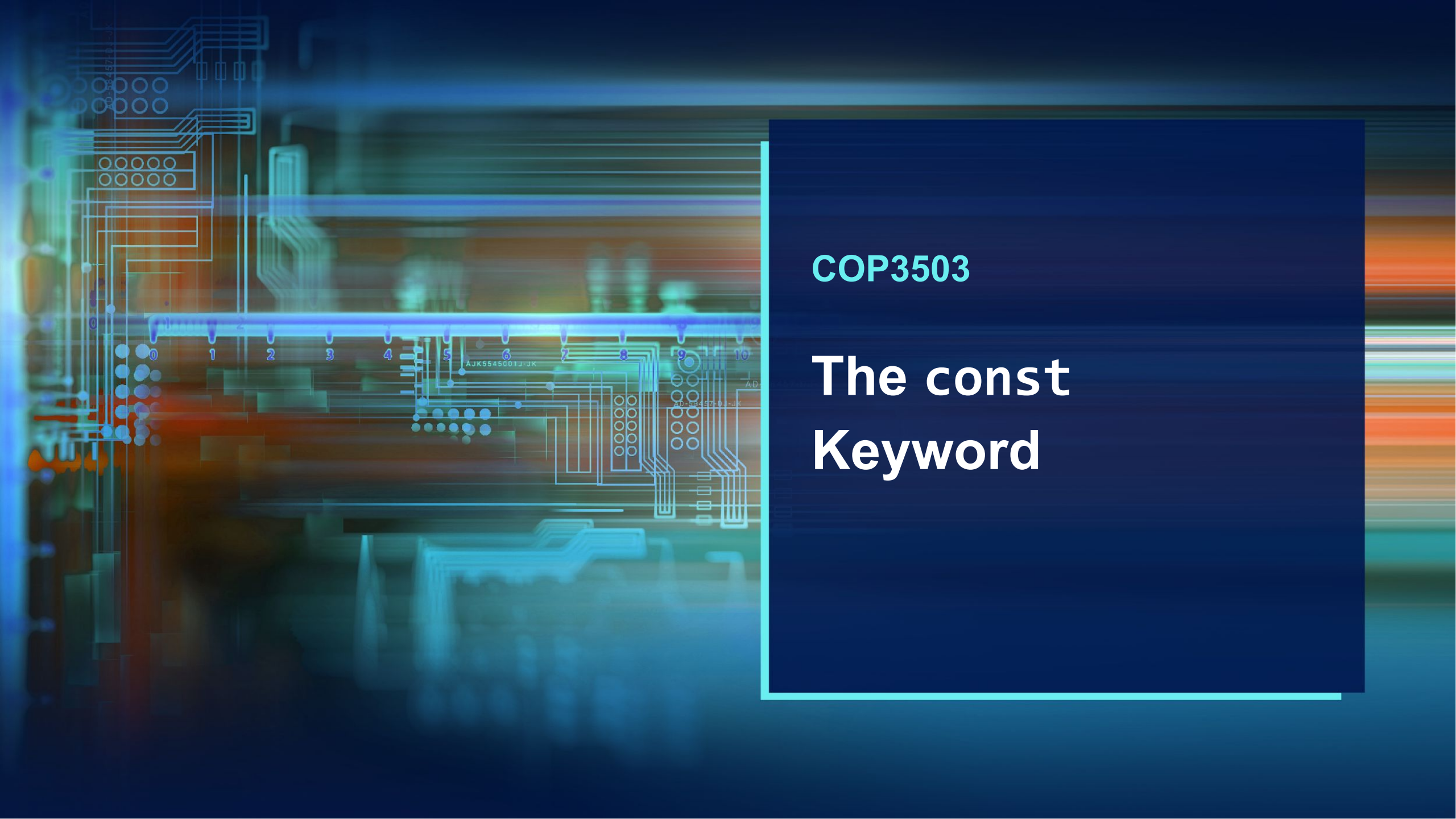


COP3503

The const Keyword

What is const?

- A keyword to protect data – marks a variable as **constant**.

```
const int maxPlayers = 4;
const float pi = 3.14159f;
const int minimumScore = -10;

pi = 12.4f; // Compiler error

// maxPlayers MUST be const here
// compiler needs to know how big to make this array
Player players[maxPlayers];

int playerCount = 4;
Player players[playerCount]; // Compiler error
```

Also Used When Passing By Reference (Or By Pointer)

```
void Print(const Object& param);
void Print(const SomeClass* parameter);

void Print(SomeObject& param) // Non-const parameter
{
    cout << param.GetSomeData() << endl;
    param.SetSomeData(); // Is this what we want?
}

void Print(const SomeObject& param)
{
    cout << param.GetSomeData() << endl;
    param.SetSomeData(); // Compiler error
}
```

Also Used When Returning References and Pointers

```
class Container
{
    int* dynamicArray;
public:
    const int* GetData()
    {
        return dynamicArray;
    }
};

// data is a "read only" variable
Container example;
const int* data = example.GetData();

// Error, myPtr2 must be const
int* error = example.GetData();
```

You **must** catch this in a `const` variable – the compiler will enforce this.

The `const` keyword is a way to reduce unwanted changes in our programs.

const and Pointers

- The const keyword can be used in three ways with pointers:
  - Protect the thing the pointer points to (the pointee)
  - Protect the pointer itself
  - Using both the first and second simultaneously

Protecting the **pointer** is like laminating a piece of paper with an address on it.

You can't change what's on the piece of paper, but you **can** change the house.

Protecting the **pointee** is like building a wall around the building.

```
int someValue = 52;
const int* pointer = &someValue; // Pointer to const value

// Compiler error, can't change the pointee
*pointer += 18;

int valA = 52, valB = 24;
int* const pointer = &valA; // const pointer

*pointer = -6; // We CAN change the pointee, in this example
pointer = &valB; // Compiler error, can't change the pointer
```

const and Pointers

No changes of any kind

```
int someValue = 52;
int otherValue = 33;

// const pointer to a const int – no change to either pointer,
// or the pointee
const int* const pointer = &someValue;

*pointer = 2; // Compiler error, can't change the pointee
pointer = new int[5]; // Error, can't reassign the pointer
pointer = &otherValue; // Error, can't reassign the pointer
pointer = nullptr; // Error, can't reassign the pointer
```

When you want to be really sure there are no unexpected changes

In my own code I tend not to make pointers "double const" as it feels like a bit of overkill. Your experience may vary!

`const` and Class Functions

- `const` protects the safety of `this` inside a class function.

```
class Example
{
public:
    void Foo() const;
};

class Example
{
public:
    void Foo(Example* this); // The compiler passes "this" for us

    // const member functions use a const pointer for "this"
    void Foo(const Example* this) const;
};
```

The **`const`** keyword after a function's parameter list marks it as a **`const` member function**.

`const` Member Functions Protects `this` from Changes

- No code in that function can change the `this` pointer
- This essentially makes `this` “read-only” for this function.

No change to the object coming in.
No change to the invoking object (*this)

```
class Point
{
    float x, y;
public:
    float Distance(const Point& p) const;
};

float Point::Distance(const Point& p) const
{
    return sqrt((x - p.x)*(x - p.x) + (y - p.y)*(y - p.y));
}
```

The const function promises it can do its work, with no **side effects** (i.e., unexpected changes).

What if we removed **const** from everything?

`const` Has a Way of Spreading

```
void Example::Foo() const
{
    // this->someValue -= 5; No changes to this!
    Bar();
}

void Example::Bar()
{
    this->someValue -= 5;
}
```

Compiler error:
`const Example* this` calling non-**`const`** function `Bar()`

`Foo()` says "I promise to protect `*this`".

`Bar()` says "I make no such promise!" and invalidates protection claims of `Foo()`.

- What's the solution here?
  - Don't call `Bar()` from within `Foo()`.
  - Change `Bar()` to a const member function, and remove any code that changes `*this;`

All of these issues and considerations fall under the term **`const` correctness**.

const and Non-const Versions of Functions

```
class Example
{
    vector<int> someData;
public:
    vector<int>&          GetData();
    const vector<int>&    GetData() const;
};
```

It's not uncommon to see (or write) both **const** and non-**const** versions of a function.

- Imagine two separate functions you might write elsewhere in your program…

```
// Get some user input and store it in the object's list of data
// This WILL change the object
void GetUserInputForValues(Example& myObject);

// Count (and return) the number of even values in the object's
// list of data. This SHOULD NOT change the object
int CountEvenNumbers(const Example& myObject);
```

When You Need to Change the Data…

```
void GetUserInputForValues(Example& myObject);
{
    vector<int>& values = myObject.GetData();
    for (int i = 0; i < 5; i++)
    {
        int someNumber;
        cin >> someNumber;
        values.push_back(someNumber);
    }
}

// Same concept, slightly different approach
void GetUserInputForValues(Example& myObject);
{
    for (int i = 0; i < 5; i++)
    {
        int someNumber;
        cin >> someNumber;
        myObject.GetData().push_back(someNumber);
    }
}
```

To modify the data, we get a reference using the non-**const** version of **GetData()**.

Getting a reference variable makes this line of code a little cleaner.

Alternatively, we can just use the returned reference directly.

…and When You Want to Prevent Changes

Because the object is **const**…

```
int CountEvens(const Example& myObject);
{
    const vector<int>& values = myObject.GetData();
    int count = 0;
    for (unsigned int i = 0; i < values.size(); i++)
    {
        if (values[i] % 2 == 0)
            count++;
    }
    return count;
}
```

…if this function is non-**const**, a compiler error will be generated.

This variable must also be **const**.

Even if you aren't actually making any changes…

The compiler is concerned the data **might** be changed.

Writing Both Is Often the Correct Way

```
class Example
{
    vector<int> someData;
public:
    vector<int>& GetData();
    const vector<int>& GetData() const;
};

vector<int>& Example::GetData()
{
    return someData;
}

const vector<int>& Example::GetData() const
{
    return someData;
}
```

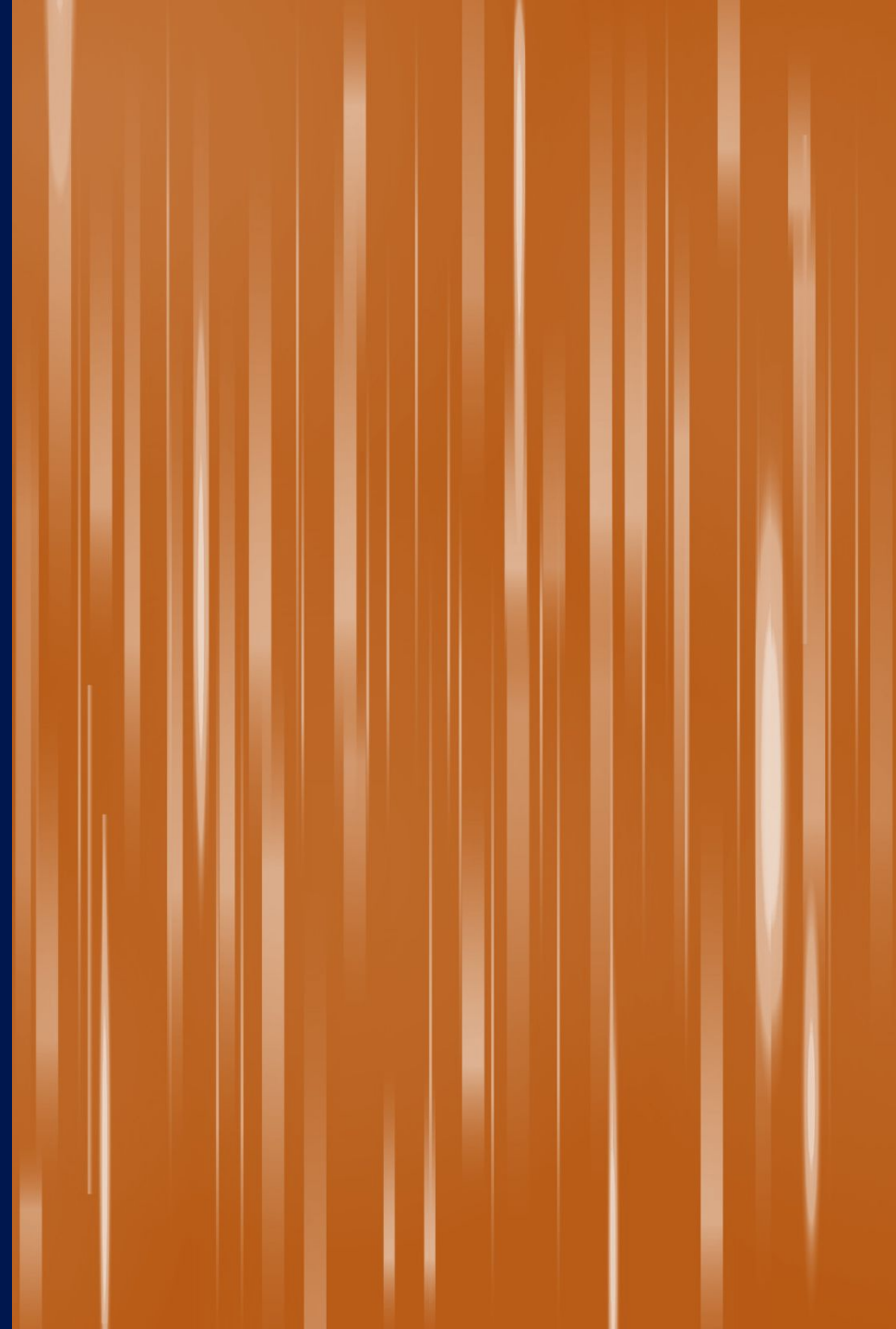
The compiler calls the right version based on the “const-ness” of the invoking object.

Even if you don’t use that function yourself in a specific program, the class **might** be used in a way that requires a const version.

When Should You Use It?

- Think about what your code needs to do
- For member functions:
  - Is it to retrieve some information (accessor)? Use `const`!
  - Is a function meant to change the object (mutator)? Don't use `const`.
  - What about return types and parameters? Ask the same questions.

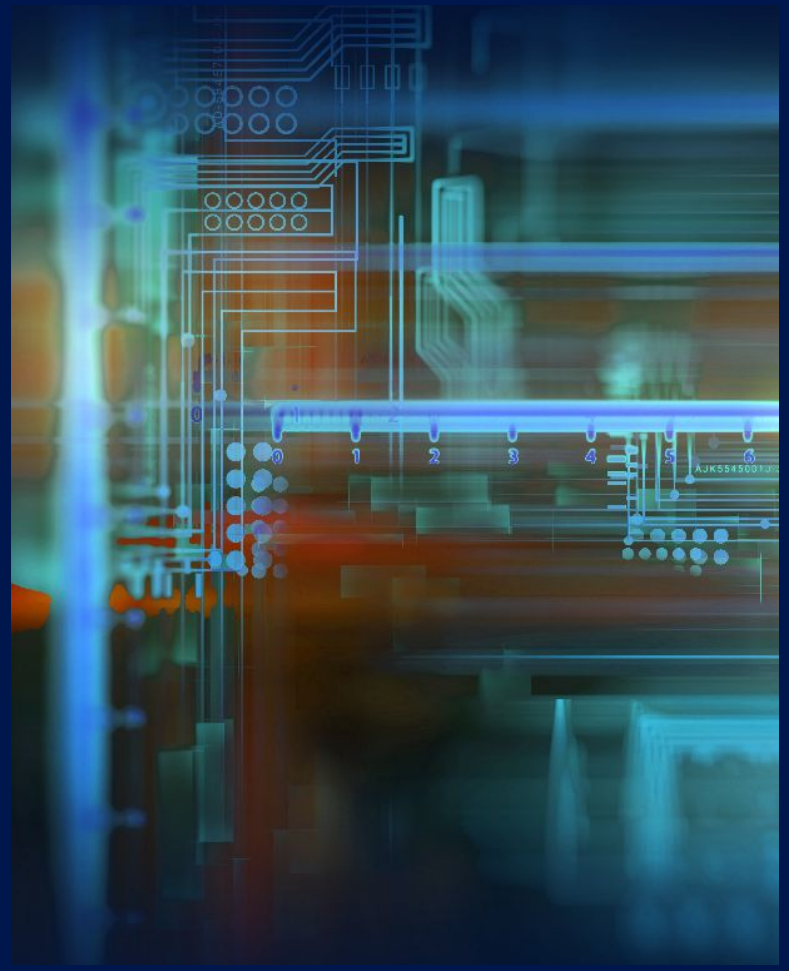
For parameters and return types:
Unless you can state reasons for doing it otherwise, default to passing all class objects by const reference or const pointer.



Recap

- `const` is a way marking variables or functions as **constant**, or “**read-only**”.
- Its purpose is to minimize **side effects** in your code and prevent unwanted changes.
- The **compiler enforces `const`**, and generates errors if we violate it.
- This is especially useful when **passing/returning pointers or references**.
- We can create **`const` variables**, or **`const` class member functions**.
- `const` class member functions to **protect the invoking object** by **preventing changes to `*this`**.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.