# Wrapping a static array

# About

Early on when starting with C++, we cover static arrays. Later on, we create dynamic arrays by using pointers.

To introduce some of the basic ideas of making a data structure, we are going to start with wrapping a static array, then a dynamic array.

# Topics

1. Review: Static arrays
2. Pitfalls of a static array
3. Making a “smart” static array

# 1. Review: Static arrays

# 1. Review: Static arrays

Just in case it's been a while since you've done some programming with static arrays, let's look at how they're used.

Notes

# 1. Review: Static arrays

**Static arrays…**

- Must have a defined size at compile-time
- Cannot be resized while the program is running
- Have the risk of crashing the program if you go out of bounds of the array

Notes

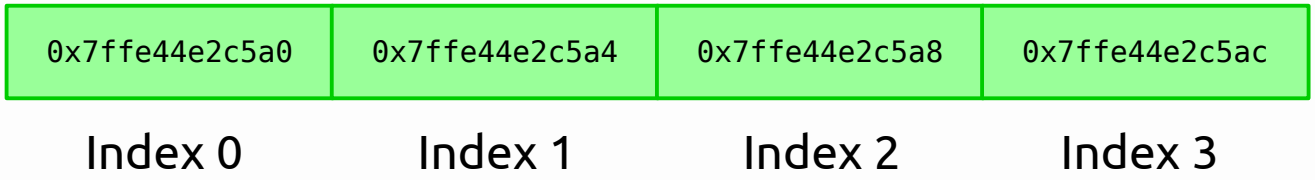
**Static arrays…**

1) Size must be known at compile time
2) Cannot be resized
3) Problems happen if you go outside of bounds

# 1. Review: Static arrays

## Static arrays…

- Elements of the array are stored contiguously in memory




- C++ arrays don't keep track of their amount of elements; we have to track that ourselves, or use the sizeof() function – which gives size in bytes.
  *If you know the size of the data type the array is storing, you can figure out the amount of elements from that.*

Notes

**Static arrays…**
1) Size must be known at compile time
2) Cannot be resized
3) Problems happen if you go outside of bounds
4) Elements are stored contiguously in memory
5) Arrays don't keep track of their # of elements

# 1. Review: Static arrays

- Each **element** of an array is essentially a single variable, but as part of an array we can access each of the "element variables" by their position.

- The **index** is the numeric position in the array. The first index of arrays in C++ is always 0.

- If an array is of size ***n***, then the valid indexes are ***0 through n-1***.

Notes

**Static arrays…**

1) Size must be known at compile time
2) Cannot be resized
3) Problems happen if you go outside of bounds
4) Elements are stored contiguously in memory
5) Arrays don't keep track of their # of elements
6) For an array of size *n*, valid indexes are *0* through *n-1*.

# 2. Pitfalls of a static array

# 2. Pitfalls of a static array

Let’s look at an example of using a static array in a program, and some things we have to look out for.

Notes

# 2. Pitfalls of a static array

```
switch( choice )
{
    case 1: // Add game
    cout << "Game name: ";
    getline( cin, games[gameCount] );
    gameCount++;
    break;

    case 2: // Edit game
    cout << "Game index: ";
    int index;
    cin >> index;
    cin.ignore();

    cout << "Name update: ";
    getline( cin, games[index] );
    break;

    case 3: // Quit
    done = true;
    break;
}
```

Using a static array

1. If the array is full (gameCount > max size), this will crash.

2. If the user enters an invalid index (index < 0 or index >= size), the program will crash.

Notes

# 2. Pitfalls of a static array

```
const int GAME_MAX = 5;
int gameCount = 0;

string games[GAME_MAX];
```

Declaring the array

3. If we want to protect against going outside of bounds of the array, we need to keep track of both the **maximum size** (the array's size), and the **element count**.

Notes

# 2. Pitfalls of a static array

```
case 1: // Add game
if ( gameCount == GAME_MAX )
{
    cout << "ERROR: Array is full";
}
else
{
    cout << "Game name: ";
    getline( cin, games[gameCount] );
    gameCount++;
}
break;

case 2: // Edit game
cout << "Game index: ";
int index;
cin >> index;
if ( index < 0 || index >= GAME_MAX )
{
    cout << "ERROR: Invalid index";
}
else
{
    cin.ignore();
    cout << "Name update: ";
    getline( cin, games[index] );
}
break;
```

Using just a vanilla static array, we would have to check that the array is not full, and validate any user-entered index, every time each of these happens.

This means a lot of duplicate code checking the same things if the array is used throughout the program.

Part of good program design is reducing duplicate code.

Notes

# 2. Pitfalls of a static array

```
case 1: // Add game
if ( gameCount == GAME_MAX )
{
    cout << "ERROR: Array is full";
}
else
{
    cout << "Game name: ";
    getline( cin, games[gameCount] );
    gameCount++;
}
break;

case 2: // Edit game
cout << "Game index: ";
int index;
cin >> index;
if ( index < 0 || index >= GAME_MAX )
{
    cout << "ERROR: Invalid index";
}
else
{
    cin.ignore();
    cout << "Name update: ";
    getline( cin, games[index] );
}
break;
```

So how do we cut down on duplicate code, and make the array easier to use?

Notes

# 3. Making a "smart" static array

# 3. Making a "smart" static array

We're going to write a class that "wraps" a static array. As part of a class, we can provide an **interface** to outside users to take care of basic functionality – such as insertions, accessing specific elements, and deleting.

Because the user doesn't have direct access to the array, and have to work with it via the interface of functions, we can use these functions to check for errors before they occur.

Notes

# 3. Making a "smart" static array

We're going to make a *really simple* smart static array for now, and it is just going to contain an array of strings – we will review **templates** later, which will allow us to create an array of *any data type*.

This class will just contain a **Set** and a **Get** function, but we can do error checking in these functions to make sure the user cannot misuse our class.

Notes

# 3. Making a "smart" static array

```
class SmartStaticArray
{
    public:
    void Set( int index, string value );
    string Get( int index );

    private:
    string m_array[10];
    const int MAX_SIZE = 10;
};
```

A very basic wrapper class

Here's the declaration for our smart static array. It stores an array of strings and keeps track of the MAX_SIZE, and the user is able to access elements via **Get** and **Set**, and must pass in some index.

Notes

# 3. Making a "smart" static array

```
void SmartStaticArray::Set( int index, string value )
{
    if ( index < 0 || index >= MAX_SIZE )
    {
        throw out_of_range( "Invalid index!" );
    }

    m_array[ index ] = value;
}
```

The **Set** function

The **Set** function is used to assign values to the array at specific positions. The user has to pass in an index and the value they want, and we can do an error check before updating the array value.

If the index is invalid, we throw an exception. Otherwise, we update the internal array.

Notes

# 3. Making a "smart" static array

```
bool SmartStaticArray::Set( int index, string value ) noexcept
{
    if ( index < 0 || index >= MAX_SIZE )
    {
        // Don't update the array, just return false
        return false;
    }
    else
    {
        m_array[ index ] = value;
        return true; // success
    }
}
```

If we didn't want to use exceptions, our **Set** function could just *ignore* the command to update the array. In this example, I'm at least returning **true** if the update was a success, or **false** if there was an error, but we're not throwing any exceptions.

Notes

# 3. Making a "smart" static array

```
string SmartStaticArray::Get( int index )
{
    if ( index < 0 || index >= MAX_SIZE )
    {
        throw out_of_range( "Invalid index!" );
    }

    return m_array[ index ];
}
```

The **Get** function

The **Get** function takes in an index and, if the index is in a valid range, it will return the value stored in the array at that position.

If the index is invalid, again it throws an exception.

Notes

# 3. Making a "smart" static array

Additional functions we can add to the smart static array:

- Append – Add something to the "end" of the array
- Insert – Insert something at some index, and push everything afterwards "right" in the array.
- Remove – Remove the value at some index, and push everything afterwards "left" in the array.
- Operator overloading – We can overload the =, ==, and [ ] operators to be used with our class.

We will do this in a programming assignment.

Notes

# Conclusion

In this lecture we talked about wrapping a static array within a class, using functions to create an interface and handle tasks for the structure and do error checking.

We will work more with this structure, as well as implement one with a dynamic array, before we get into Linked Lists, which is another type of linear structure we can use to store data.