Introduction to OOP in C++

Introduction to Modern C++

Ryan Baker

March 1, 2025

"*Software and cathedrals are much the same – first we build them, then we pray.*"

Lecture Objectives

- To understand arrays and their relationship to pointers.
- To know how to define and use a `struct` and a `class`.
- To understand various access-specifiers that C++ provides, and how they make a distinction between a `class` and a `struct`.
- To become familiar with constructors and destructors in C++.
- To know how to design with `static` within a `class`.

Contents

1 Arrays

Arrays are used to store multiple values in a single variable. Arrays are declared using the variable type and the number of elements stored in the array with square brackets `[]`:

```cpp
int arr[5]; // declares an array with 5 ints
```

This is often much more straightforward than maintaining separate variables for each value.

1.1 Subscript Operator `[]`

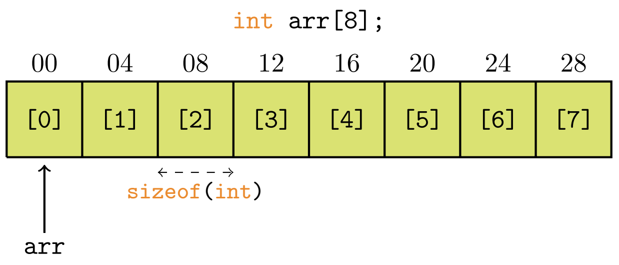
You can access an array element using the subscript operator `[]` with the index of the element you'd like to refer to:

```cpp
#include <iostream>

int main()
{
    using namespace std;
    int arr[4];
    for (int i = 0; i < 4; ++i)
        arr[i] = i; // write to the array

    // read from the array
    cout << arr[0] << " " <<  arr[3] << endl;
}
```

Output:

```
0 3
```

Note that array indices begin with 0. `arr[0]` is the first element, `arr[1]` is the second element, etc. This is because of an array's relationship to pointers: An array is a pointer to its first element.



This means that an array variable is equal `==` to the address of its first element:

```cpp
#include <iostream>

int main()
{
    int arr[100];
    std::cout << (arr==&arr[0]) << std::endl; // 1
}
```

For an array `arr`, and an index `i`, the expression `arr[i]` is equivalent to `*(arr + i)`. Hence why `arr == &arr[0] == &(*(arr + 0))`. This fact, along with the commutativity of addition, can be abused in the following manner:

```cpp
int arr[4];
3[arr] = 3000; // *(3 + arr) == *(arr + 3)
2[arr] = 2000; // *(2 + arr) == *(arr + 2)
```

1.2 Array Initialization

Arrays can be initialized when declared using an initializer list `{}`:

```cpp
#include <iostream>

int main()
{
    int arr[3] { 10, 20, 30 };
    std::cout << arr[0] << std::endl; // 10
    std::cout << arr[1] << std::endl; // 20
    std::cout << arr[2] << std::endl; // 30
}
```

An assignment operator `=` is optional for array initialization. While brace initialization `{}` typically prevents narrowing conversions, narrowing conversions are already disallowed within initializer lists. Hence, the choice between `int arr[10] = {/* ... */}` and `int arr[10] {/* ... */}` is entirely stylistic.

You may recall that the reason narrowing conversions are allowed for normal variable initialization is to maintain backwards compatibility with C. However, because narrowing conversions are disallowed within initializer lists in C++, the following code is valid in C but not in C++:

```cpp
int arr[1] { 0.5 };
```

Initializer lists can be used by the compiler to infer the size of the array:

```cpp
// arr will contain 7 elements
int arr[] { 1, 1, 2, 3, 5, 8, 13 };
```

Providing more initializers than elements will cause a compiler error:

```cpp
// error: excess elements in array initializer
int arr[5] { 0, 1, 2, 3, 4, 5 };
```

If fewer initializers than elements are provided, remaining elements will be 0:

```cpp
int arr[4] { 1 }; // holds { 1, 0, 0, 0 }
```

A *designator*, denoted by `[i]` within in initializer list, causes the following initializers to begin filling the array at the index specified by the designator. Initialization continues from the next element after the one described by the designator:

```cpp
#include <iostream>

#define SIZE 10

int main()
{
    // holds { 1, 2, 3, 0, 0, 0, 0, 3, 2, 1 }
    int arr[SIZE] { 1, 2, 3, [SIZE-3] = 3, 2, 1 };

    // holds { 0, 1, 2, 3, 4 }
    int arr2[5] = { [4] = 4, [0] = 0, 1, 2, 3 };
}
```

`char` type arrays can be initialzed with string literals:

```cpp
#include <iostream>

int main()
{
    // holds { 'h', 'e', 'l', 'l', 'o', '\0' }
    char arr[] { "hello" };

    std::cout << arr << std::endl; // hello
}
```

1.3 sizeof Arrays

Recall that the `sizeof` operator returns the number of bytes occupied by a datatype or variable. `sizeof`, when called on an array variable, does the same:

```cpp
#include <iostream>

int main()
{
    int arr[] { 0, 1, 4, 9 };
    std::cout << sizeof(int) << std::endl;
    std::cout << sizeof(arr) << std::endl;
}
```

Output:

```
4
16
```

We can use this fact to calculate the number of elements in an array:

```cpp
#include <iostream>

int main()
{
    int arr[] { 0, 1, 4, 9, [32] = 1024 };
    std::cout << "n elements = "
              << (sizeof(arr) / sizeof(*arr))
              << std::endl;
}
```

Output:

```
33
```

The expression `sizeof(arr) / sizeof(*arr)` cannot be used to determine the number of elements in a dynamically allocated array. This is because `sizeof`, when called on a dynamically allocated array, returns the size of the pointer (`sizeof(type*)`) rather than the size of the array.

```cpp
// dynamically allocated array
int* arr = new int[3];

std::cout << sizeof(arr) << std::endl; // 8
```

This is because operator new may, for a variety of reasons, allocate more bytes than you explicitly request. Hence, if sizeof were to return the number of bytes allocated for the array, the result would be useless.

To avoid this confusion and to improve your code's readability and safety, you can (and likely should) use `std::size()`, defined in the `<iterator>` library:

```cpp
#include <iostream>
#include <iterator>

int main()
{
    int arr[] { 0, 1, 8, 27, 64 };
    std::cout << std::size(arr) << std::endl;
}
```

Output

```
5
```

1.4 Multidimensional Arrays

A multidimensional array is an array of arrays. To declare a multidimensional array, we simply append more pairs of brackets `[N]`, each denoting the array's size in a different dimension:

```cpp
int board[8][8]; // an 8x8 2D array of integers
```

To access elements within a multidimensional array, specify an index in each of the array's dimensions:

```cpp
// expands to *(*(board + 3) + 4)
int e4 { board[3][4]; };
```

Nested initializers can be used to initialize multidimensional arrays:

```cpp
char tictactoe[3][3]
{
    { 'x', 'x', ' ', },
    { ' ', 'o', ' ', },
    { ' ', 'o', 'x', },
};
```

2 Structs

A structure, or a `struct`, is a *type* that stores several related variables in one place. Each variable in the `struct` is known as a *member* of the `struct`. Unlike an array, `struct` members can be of different types:

```cpp
struct MyStruct // declare 'MyStruct' as a type
{
    int a;    // member variable (integer)
    char b;   // member variable (character)
    float c;  // member variable (floating point)
};
```

We can instantiate a `struct` as follows:

```cpp
// MyStruct definition ...

int main()
{
    MyStruct m; // variable 'm' of type 'MyStruct'
}
```

2.1 Accessing `struct` Members `.`, `->`

The member access operator, or the *dot operator*, is used to access members of a `struct` variable:

```cpp
#include <iostream>

struct MyStruct
{
    int member1;
    float member2;
};

int main()
{
    MyStruct object;

    object.member1 = 42;
    object.member2 = 3.14159;

    std::cout << object.member1 << std::endl; // 42
}
```

The indirect member access operator, also known as the *arrow operator*, is used to access members through a pointer to a struct variable.

```cpp
#include <iostream>

int main()
{
    struct S { int member; };

    S s1;
    S* ptr { &s1 };

    s1.member = 10;
    // access 'member' thru pointer to struct type
    std::cout << ptr->member << std::endl;
}
```

Output:

```
10
```

2.2 struct Initialization

C++ provides several ways to initialize a struct. Initializer lists can assign values to members in the order they were declared within the struct definition:

```cpp
struct Point
{
    int x;
    int y;
};

Point p { 3, 4 }; // p.x = 3, p.y = 4
```

Designators can be used to initialize struct members by name:

```cpp
struct Point
{
    int x;
    int y;
};

Point p { .y = 4, .x = 3 }; // p.x = 3, p.y = 4
```

Default member values can be provided directly in the struct definition. If no initializer is used when creating an instance, these default values will be used.

```cpp
struct Point
{
    int x = 1;
    int y = 1;
};

Point p;             // p.x = 1, p.y = 1
Point q { 2, 2 };    // q.x = 2, q.y = 2
Point r { 0 };       // r.x = 0, r.y = 1
```

If you have an existing instance, you can initialize a new instance by *copying* the values of the existing one:

```cpp
struct Point
{
    int x = 1;
    int y = 1;
};

Point p { 15, 16 }; // p.x = 15, p.y = 16
Point q { p };      // q.x = 15, q.y = 16
Point r = q;        // r.x = 15, r.y = 16
```

2.3 Member Functions

We may want to define a function to act on a struct variable:

```cpp
#include <iostream>

struct Point
{
    int x, y;
};

void print_point(Point p)
{
    std::cout << "{" << p.x << ", " << p.y << "}";
    std::cout << std::endl;
}
```

This function can be called with an instance of the struct:

```
int main()
{
    Point p { 14, 6 };
    print_point(p);
}
```

Output:

```
{14, 6}
```

Alternatively, in C++, a `struct` can contain functions as members. These functions can operate directly on the data members of the `struct`.

```
#include <iostream>

struct Point
{
    int x, y;
    void print();
};

// define Point.print() with resolution operator ::
Point::print()
{
    std::cout << "{" << p.x << ", " << p.y << "}";
    std::cout << std::endl;
}

int main()
{
    Point p { 14, 6 };
    p.print(); // {14, 6}
}
```

The member function can also be defined within the `struct` definition:

```
struct Point
{
    int x, y;
    void print()
    {
        // ...
    }
};
```

3 Classes

A class is a user-defined datatype that typically has a collection of data and functionality associated with it. An instance of a class is known as an object:

```
class MyClass // define a class
{
    // ...
};

int main()
{
    MyClass MyObject; // instantiate an object
                      // of type ‘MyClass’
}
```

Classes can have data and functionality associated with them.

3.1 Access Specifiers public, private, protected

Access specifiers in C++ control the visibility of class and struct members. There are three types: public, private, and protected.

- public: Accessible from anywhere, both inside and outside the class.
- private: Accessible only from within the same class.
- protected: Accessible from within the same class and subclasses.

```
class MyClass
{
public:
    int public_int;
    void foo() { private_int = 10; } // ok

private:
    int private_int;
}

int main()
{
    MyClass m;
    m.public_int = 10;  // ok
    m.private_int = 10; // error: cannot access
}
```

3.1.1 class vs. struct

Functionally, the only difference between a class and a struct is the default member visibility. struct members are public by default, while class members are private by default. This means that the following definitions:

```
struct Point
{
    int x, y;
    Point();
}
```

and

```
class Point
{
public:
    int x, y;
    Point();
}
```

are entirely equivalent, at least as far as C++ is concerned. However, most developers make a strong mental distinction. A struct is (or ought to be) plain-old-data; a group of bits with very little in the way of encapsulation or complex functionality. A class, on the other hand, is a functional body with encapsulation, intelligent functionality, and maybe a Costco membership. Hence, you should probably use struct for types with few methods and only public data, and use class otherwise.

3.2 Constructors

A constructor is a special member function whose purpose is to initialize the data within an object. Constructors share the same name as the class to which they belong. The basic syntax for a constructor is as follows:

```
class MyClass
{
public:
    MyClass() // constructor for MyClass
    {
        // ...
    }
};
```

Notice that the constructor has no return type, because its only purpose is to initialize the data within an object. A `default` constructor is one that either takes no arguments or has default values for all of its arguments. The `default` constructor can be defined by the programer, otherwise the compiler will create one. The programmer can also explicitly mark a constructor as `default`:

```
class Point
{
private:
    int x, y;

public:
    Point() = default;
};
```

Constructors can take input arguments, allowing for initialization:

```
class Point
{
private:
    int x, y;

public:
    Point() = default;
    Point(int x, int y)
    {
        this->x = x; // .x = x
        this->y = y; // .y = y
    }
};
```

Alternatively, constructors can initialize member variables with an initialization list. The initialization list appears before the constructor body and uses a colon : followed by a comma separated list of member variables:

```
class Point
{
private:
    int x, y;

public:
    Point() = default;
    // initializer list constructor
    Point(int x, int y): x(x), y(y) {}
};
```

Initializer lists are generally preferred to constructor bodies for initializing `class` data. They are strictly necessary for initializing `const` and reference members, because these members cannot be assigned within the constructor body. They may also be fore efficient because, rather than default initialization followed by assignment, initializer lists directly initialize member variables. Member variables are initialized in the order they are declared, regardless of the order they are in the list itself.

We may have a `class` where we want to ensure it is called with an initializer. C++ provides a very elegant way to do this, by deleting the default constructor:

```cpp
class Point
{
private:
    int x, y;

public:
    Point() = delete;
    // initializer list constructor
    Point(int x, int y): x(x), y(y) {}
};
```

3.3 Destructors

A destructor is a special member function within a `class` whose purpose is to perform cleanup actions when an object is destroyed. A destructor for a `class` called `MyClass` would have the name `~MyClass()`.

```cpp
class C
{
public:
    int* data;

    C(int size)
    {
        data = new int[size];
    }

    ~C()
    {
        delete[] data;
    }
};
```

3.4 static Members

Recall from *How C++ Works* that static memory is initialized only once before runtime. So what happens if we mark a class member as static?

```cpp
class Point
{
private:
    int x, y;

public:
    static int n; // static member variable

    Point() = delete;
    Point(int x, int y): x(x), y(y) {}
};
```

Because the variable n is given a place in memory at compile time, it exists even before a single instance of the Point class is created. This means we can access Point::n without needing an object:

```cpp
// class Point ...

int Point::n { 50 };

int main()
{
    std::cout << Point::n << std::endl;
}
```

Notice how, with respect to n, the Point class is behaving somewhat like a namespace. In essence, n belongs to the Point class itself, rather than any one Point object. A more realistic example of a static member is as follows:

```cpp
class Point
{
private:
    int x, y;

public:
    static int n_points;

    Point() = delete;
    Point(int x, int y): x(x), y(y) { ++n_points; }
    ~Point() { --n_points; }
};

int Point::n_points { 0 };
```

In the above code, we maintain a `static` variable representing the number of `Point` objects we have created. We increment this variable in the `Point()` constructor, and decrement it within the destructor.

```cpp
#include <iostream>

// class Point ...

int main()
{
    std::cout << Point::n_points << " ";
    {
        Point p(0, 0);
        std::cout << Point::n_points << " ";
        {
            Point q(1, 1);
            std::cout << Point::n_points << " ";
        }
        std::cout << Point::n_points << " ";
    }
    std::cout << Point::n_points << " ";
}
```

Output:

```
0 1 2 1 0
```

3.4.1 `static` Member Functions

The `static` keyword can also be used to qualify member functions. Similar to `static` member variables, `static` member functions belong to the `class` rather than to any one object.

```cpp
#include <iostream>

class Point
{
private:
    int x, y;

public:
    static int n_points;

    Point() = delete;
    Point(int x, int y): x(x), y(y) { ++n_points; }
    ~Point() { --n_points; }

    static void print_n() // static method
    {
        std::cout << Point::n_points << std::endl;
    }
};

int Point::n_points { 0 };
```

`static` member functions can only access `static` member variables. After all, how could they access the member variables belonging to an object when they themselves do not belong to an object?