Class 04

More on Functions & Intro to STL Containers

Kinematics

Outline

- Announcements/Discussion
  - Assignment-01/Grades
- More on Functions
  - Return Types
  - Arguments
  - Usage
- Intro to STL
  - std::vector
  - Inserting, Indexing, Looping

Assignment-01

- Method for distributing feedback is being updated, by tomorrow night everyone will have all point deductions detailed.

- **Some of you still had uninitialized numeric variables. Womp.**

- For the first time since I have started giving this assignment (~5 years)
  - <u>No one used the pow function.</u>
  - <u>No one used printf.</u>

- **Every** infraction is worth -2.5 points.

Final Projects

- Keep an eye out for an announcement this week detailing the final project. You will be creating a simulated Predator-Prey Model
  - Simulate a predator(s) entity tracking/eating prey entities
  - Prey entities should *intelligently navigate* away from nearby predators
  - Add a twist and make the predators "vampires" that turn their prey into more vampires
- In addition to your project working correctly it must satisfy performance requirements.
  - Runtime, memory, cache efficiency

Functions

- Functions are self contained blocks of code that perform specific tasks.
- We use functions to isolate code into callable blocks which we can call as needed.
  - We can call a function as many times as we want!
- Functions perform *some* task
  - Functions can optionally accept inputs to be worked on
  - Functions can optionally return new data that is a product of the work it performed.

Functions

- Functions take the following form:

```
auto function_name(type1 input1, type2 input2, ...) -> return_type
{
    action(s);     // 0 or more statements can be performed
    return output; // only return something if we need to
}
```

- When we "call" a function, we are invoking it. This requires us to use the functions name, followed by parentheses.
  - If the function has inputs, then we can place values or variables inside of the parentheses.
  - If the function has no inputs, then we just use empty parentheses.
  - If the function has a return, we need to do something with it! We can store the result in variables, or use it in other expressions.

Functions

- Functions take the following form:

```
auto function_name(type1 input1, type2 input2, ...) -> return_type
{
    action(s);     // 0 or more statements can be performed
    return output; // only return something if we need to
}
```

- function_name: name of the function, can by anything
- return_type: the type of data the function returns, e.g. int, double, *void*
- inputs: variables (with their type) input to the function, if any
- action(s): any number of actions to take
- output: the data to return from the function, if any.

- Collectively the return type, function name, and inputs create a function signature.

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto say_hello() -> void
{
    cout << "hello!" << endl;
}
```

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto say_hello() -> void
{
    cout << "hello!" << endl;
}
```

- This function just prints the string "hello!"

- Its signature is: `auto say_hello() -> void`
  - Function name – say_hello
  - Return type – void
  - Inputs – N/A

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto is_even(int x) -> bool
{
    return x % 2 == 0;
}
```

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto is_even(int x) -> bool
{
    return x % 2 == 0;
}
```

- This function checks if the input int is even. It returns true when it is, and false otherwise

- Its signature is: `auto is_even(int x) -> bool`
  - `Function name – is_even`
  - `Return type – bool`
  - `Inputs – int x`

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto factorial(int y) -> int
{
    for (auto i = y - 1; i > 1; --i) {
        y *= i;
    }
    return y;
}
```

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto factorial(int y) -> int
{
    for (auto i = y - 1; i > 1; --i) {
        y *= i;
    }
    return y;
}
```

- This function computes the factorial of the input integer
- Its signature is: `auto factorial(int y) -> int`
  - Function name – factorial
  - Return type – int
  - Inputs – int y

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto multiply(double x, double y) -> double
{
    return x * y;
}
```

Functions

- Example. What is it doing? What is the signature (return type, name, inputs)?

```
auto multiply(double x, double y) -> double
{
    return x * y;
}
```

- This function computes the product of two doubles. This sort of function is not necessary to write however, as we can just do the multiplication using * to begin with.

- Its signature is: `auto multiply(double x, double y) -> double`
  - `Function name – multiply`
  - `Return type – double`
  - `Inputs – double x, double y`

Functions

- To use a function, it must be ***declared***!
  - This is just like using variables – we cannot use a variable until it has been ***defined***!
- Functions are always ***declared*** *outside and above* of the main function.
- Functions can be ***defined*** separately from their declaration.
  - This means we can declare a function above main but define it somewhere else!
  - When we do this, it is called a ***forward declaration***, as the declaration is "in front" of the definition.
- When we declare a function, we only specify its ***signature*** (the name , return type, and inputs).

Functions

- Example.

```
#include <iostream>
using namespace std;

auto celsius(double f) -> double
{
    cout << "Converting to Celsius..." << endl;
    return (f - 32.0) * 5.0 / 9.0;
}

auto main() -> int
{
    cout << celsius(100.0) << endl;
}
```

Functions

- Example with forward declaration.

```
#include <iostream>
using namespace std;

auto celsius(double f) -> double;  // this is the forward declaration!

auto main() -> int
{
    cout << celsius(100.0) << endl;
}

auto celsius(double f) -> double
{
    cout << "Converting to Celsius..." << endl;
    return (f - 32.0) * 5.0 / 9.0;
}
```

Improper Use of Functions

- What's wrong here (assume we have the proper includes and a main function)?

```
f(double x)
{
    return 2.0 * x + 3.0;
}
```

Improper Use of Functions

- What's wrong here (assume we have the proper includes and a main function)?

```
auto f(double x) -> double
{
    return 2.0 * x + 3.0;
}
```

- Our function is not valid – it is missing a return type!

Improper Use of Functions

- What's wrong here (assume we have the proper includes and a main function)?

```
auto f(double x) -> double
{
    double y = 2.0 * x + 3.0;
}
```

Improper Use of Functions

- What's wrong here (assume we have the proper includes and a main function)?

```
auto f(double x) -> double
{
    double y = 2.0 * x + 3.0;
    return y;
}
```

- Our function is not valid – while it has a return type, we are not returning anything!

Improper Use of Functions

- What's wrong here?

```
#include <iostream>
using namespace std;

auto main() -> int
{
    cout << volume(1.0, 2.0, 3.0) << endl;
}

auto volume(double h, double w, double l) -> double
{
    return h * w * l;
}
```

Improper Use of Functions

- What's wrong here?

```
#include <iostream>
using namespace std;

auto main() -> int
{
    cout << volume(1.0, 2.0, 3.0) << endl;// what is volume?
}

auto volume(double h, double w, double l) -> double
{
    return h * w * l;
}
```

- We are attempting to use the volume function before it has been declared!

Improper Use of Functions

- What's wrong here?

```
#include <iostream>
using namespace std;

auto volume(double h, double w, double l) -> double;

int main() {
    cout << volume(1.0, 2.0, 3.0) << endl;
}
```

Improper Use of Functions

- What's wrong here?

```
#include <iostream>
using namespace std;

auto volume(double h, double w, double l) -> double;

int main() {
    cout << volume(1.0, 2.0, 3.0) << endl; // I have volume...
                                           // but what does it do?
}
```

- While we have declared the volume function, we have not defined it... but this code is *perfectly valid* though! Let's see what happens...

Functions

- What's wrong here?

```
#include <iostream>
using namespace std;

auto volume(double h, double w, double l) -> double;

int main() {
    cout << volume(1.0, 2.0, 3.0) << endl; // I have volume...
                                           // but what does it do?
}
```

- Remember from Class-01 when talking about building C++ programs that there are two steps: Compiling and Linking. This code will *compile*, but it will not *link*.

STL

- STL stands for Standard Template Library. It is a collection of containers and algorithms that provide developers with common structures and algorithms.

- Collections are types (like double, int) that represent more than one instance of some type. We can have collections of doubles, ints, and even other collections (a collection of collections!).
  - There are quite a few different types of collections that all have different implications when it comes to speed, memory consumption, and usability.

- The algorithms that the STL provides are all meant to work with STL collections, and provide functionality like searching, sorting, and slicing (among other things).

std::vector

- The vector container is a STL container that provides the following:
  - Every element is adjacent in memory
  - It is resizable
  - The amortized complexity to insert something into the vector is O(1)
    - This is not complex at all!
  - It can manage only a single type at a time.
    - e.g. a vector cannot contain doubles, chars, and floats simultaneously
  - Size and capacity are differentiated:
    - size – how many elements are in the vector currently
    - capacity – how many elements can fit in the vector before it is full

std::vector

- A vector object contains functions *inside of it* that we can access using the *access operator*, which is just a single *period*.
- Of these functions are methods for adding elements to the vector and getting the size of the vector (how many objects are in it).
  - push_back
  - size
- Example

```
auto my_numbers = vector<int>{};
cout << my_numbers.size() << endl;  // prints 0

my_numbers.push_back(1337);
cout << my_numbers.size() << endl;  // prints 1
```

std::vector::size

- A note about the size method
  - This function returns an *unsigned int*, not an *int*.
  - This means when we get the size of a vector, we must ensure that we are using it as an unsigned int and not an int.
    - There is a special type to use here called `size_t` that we will use instead.
  - It is not valid to compare an integer and an unsigned integer (size_t)!

- E.g.

```
// assume we have some vector named data
auto number_of_items = int{data.size()};  // bad!
auto number_of_items = data.size();       // correct!
```

Accessing Elements of std::vector

- Once our vector has data inside of it, we can access its content using the *subscript operator*, which is represented by square brackets [and].
- Each element of the vector has a position, also known as an *index*. This index represent the order of the elements and starts counting at 0 (not 1!).
  - e.g. index 0 is the 1st position, index 1 is the 2nd position, index 2 is the 3rd position, etc.
  - For a vector with N elements in it, the last index is N-1
- Example

```
auto my_numbers = vector<double>{1.0, 4.0, 9.0, 16.0, 25.0};
auto first = my_numbers[0];
auto last = my_numbers[4];
```

std::vector

- Example

```cpp
#include <vector>
using namespace std;

auto main() -> int
{
    auto numbers = vector<int>{};
    for (auto i = int{0}; i < 10; ++i)
    {
        numbers.push_back(2 * i);
    }
}
```

std::vector

- Example

```
#include <vector>
using namespace std;

auto main() -> int
{
    auto xs = vector<int>{1, 2, 3, 4};
    auto ys = vector<int>{};
    for (auto i = size_t{0}; i < xs.size(); ++i)
    {
        ys.push_back(xs[i]);
    }
}
```

Improper Use of std::vector

- What's wrong here?

```
#include <iostream>
#include <vector>
using namespace std;

auto main() -> int
{
    auto data = vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
    cout << data << endl;
}
```

Improper Use of std::vector

- What's wrong here?

```
#include <iostream>
#include <vector>
using namespace std;

auto main() -> int
{
    auto data = vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
    cout << data << endl;
}
```

- C++ does not know what it means to print a vector! There are so many ways to do this, so C++ assumes ***you*** will tell it how.

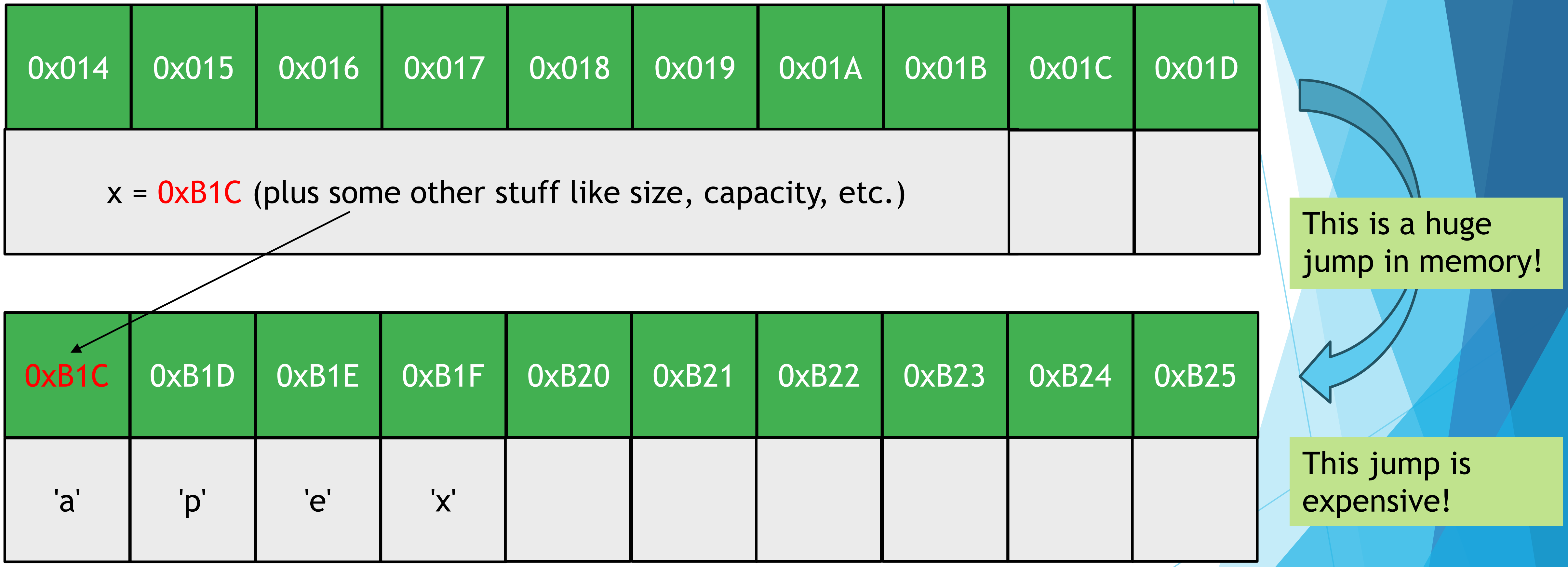
Improper Use of std::vector

- This is *one* way to do it

```
#include <iostream>
#include <vector>
using namespace std;

auto main() -> int
{
    auto data = vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
    for (auto i = size_t{0}; i < data.size(); ++i)
    {
        cout << data[i] << endl;
    }
}
```

std::vector



- Consider the following:

```
auto x = vector<char>{'a', 'p', 'e', 'x'}; // note the separation in memory!
```

std::vector::push_back Efficiency

- Imagine that our vector's data is sitting right next to some other data in memory.

0x014 … 0x01B
x = 0xB1C, capacity=4

0xE11 … 0xE19
y = 0xB20

0xB1C	0xB1D	0xB1E	0xB1F	0xB20	0xB21	0xB22	0xB23	0xB24	0xB25
'a'	'p'	'e'	'x'	1	22	333	444		

- What happens if we try to push another char into x?

  e.g.
  ```
  x.push_back('!')
  ```

std::vector::push_back Efficiency

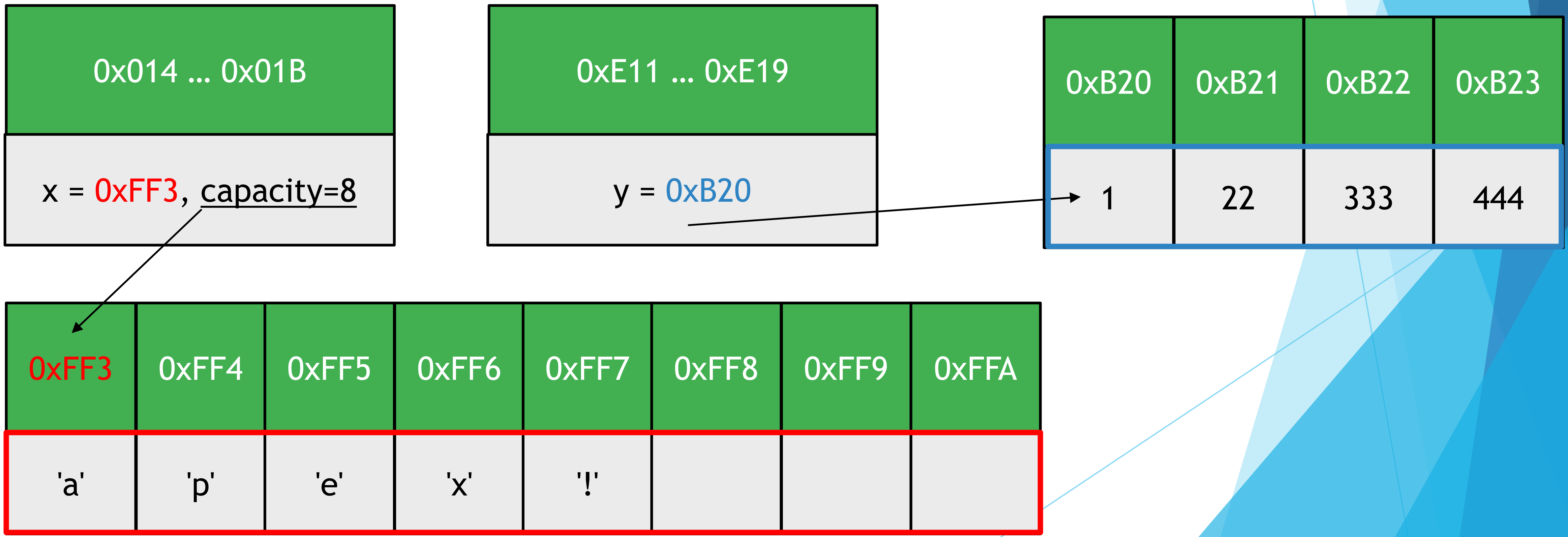
- The problem with our last example is that vectors require all their elements to be <u>adjacent in memory</u>, but when we go to add another char, '!', <u>there is no room at the end.</u>

- So that the element can be added, the vector's content must be moved in memory to a location large enough to contain the old content along with the new content.
  - In our example, we need to have enough room for 5 characters.

- <u>**C++ will find a block of memory *twice as large, not just 1 larger*, to guarantee fewer reallocations over time. This is *always* done when a vector is automatically resized.**</u>

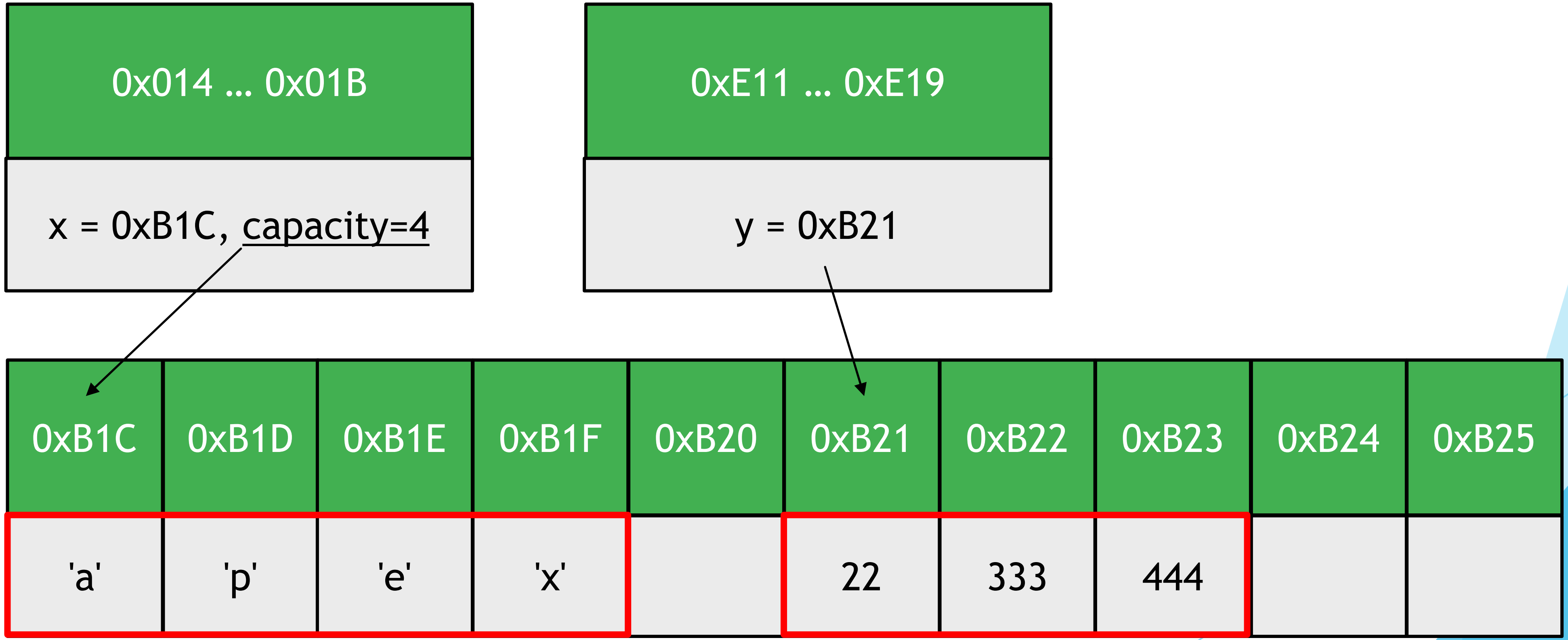
std::vector::push_back Efficiency

- Here is our memory after a successful call to `x.push_back('!')`



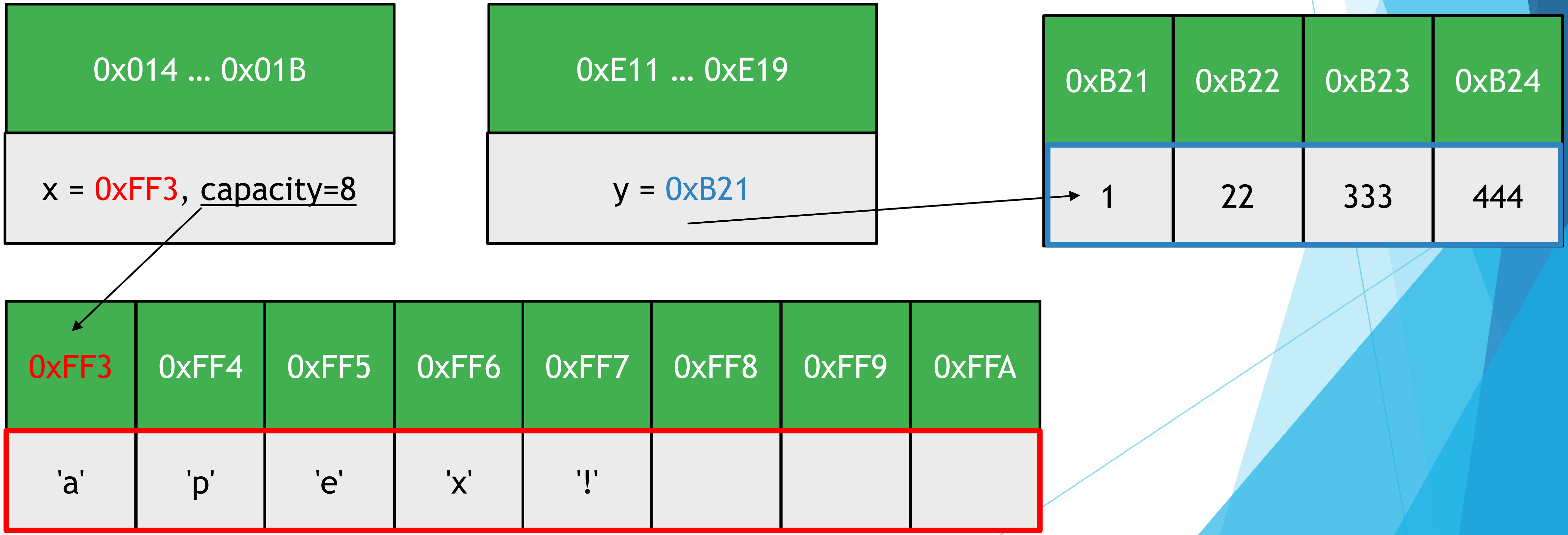
std::vector::push_back Efficiency

- What would happen in this scenario if we try to call `x.push_back('!')` here?



std::vector::push_back Efficiency

- A reallocation still needs to happen! Even though there was enough room, C++ will necessitate that a reallocation happen to facilitate later push_backs.



Looping Over Elements of a Vector

- Example using traditional loop

```
#include <iostream>
#include <vector>
using namespace std;

auto main() -> int
{
    auto data = vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
    for (auto i = size_t{0}; i < data.size(); ++i)
    {
        cout << data[i] << endl;
    }
}
```

Range-Based Loop

- Whenever we want to loop over some container (like a vector), we typically would iterate across the *positions* of the elements, and then get the element from its position (e.g. something like *data[i]*, where *i* is the position).

- Range-Based Loops provide a cleaner way of iterating over containers where instead of getting the positions and getting the elements from those positions, we get the element directly. They have a new syntax that we can use:

```
for (auto element_name: my_collection)
{
    // do stuff with element_name, instead of my_collection[i]
}
```

Range-Based Loop

- Example using a range-based loop

```
#include <iostream>
#include <vector>
using namespace std;

auto main() -> int
{
    auto data = vector<int>{2, 3, 5, 7, 11, 13, 17, 19};
    for (auto element : data)
    {
        cout << element << endl;
    }
}
```

Basic Kinematics

- We can (and frequently will) use physics to define the *movement models* for entities in our simulations.
- To begin we will use physics to define basic 1^{st} order and 2^{nd} order movement; i.e. movements defined by velocity and sometimes acceleration.
- Applying this movement to 2D space requires that the movement be applied to **each dimension individually**.
  - Movement will be applied to the horizontal and vertical components **separately**.

Basic Kinematics – Euler Integration

- We have two equations, one for updating the position of an entity, and one for updating the velocity of an entity:
  - Position

$$x_{t+\Delta t} = x_t + v_t \Delta t$$

  - Velocity

$$v_{t+\Delta t} = v_t + a_t \Delta t$$

- The subscript t indicates the *time at which that value is observed*, and thus $t + \Delta t$ indicates the *following time at which the value is observed*.
  - E.g. given **x**, **v**, and **a** at t = 10, we can compute the vales of x and v at t = 11.
- Using these equations in this way performs *Euler Integration*.

Basic Kinematics - Euler Integration

- There are multiple variations of Euler Integration, with differences based on the order in which we update position and velocity.
- There are other *integrators* out there that we will eventually also explore.
  - i.e. Runge-Kutta
- Each of these different integrators have different degrees of accuracy, stability, and performance.
- More on this next week!