Class 07

Classes

Outline

- Classes
  - Defining Classes
  - Constructors
  - Class Memory Layout & Optimization (time permitting)

Setting Up a Particle Simulation

- Consider the following initial attempt at making a 4-particle simulation. We represent particles as x & y positions and velocities.

```
auto main() -> int
{
    auto x1 = double{1.0};
    auto y1 = double{0.0};
    auto vx1 = double{0.0};
    auto vy1 = double{0.0};

    auto x2 = double{0.0};
    auto y2 = double{1.0};
    auto vx2 = double{0.0};
    auto vy2 = double{0.0};

    auto x3 = double{-1.0};
    auto y3 = double{0.0};
    auto vx3 = double{0.0};
    auto vy3 = double{0.0};

    auto x4 = double{0.0};
    auto y4 = double{-1.0};
    auto vx4 = double{0.0};
    auto vy4 = double{0.0};

    // ...
```

- Is something problematic? Now? Later?

Setting Up a Particle Simulation

- How can we improve this? Every time we need to add a new particle to the system, we need to create 4 more variables for its position and velocity.

```
auto main() -> int
{
    auto x1 = double{1.0};
    auto y1 = double{0.0};
    auto vx1 = double{0.0};
    auto vy1 = double{0.0};

    auto x2 = double{0.0};
    auto y2 = double{1.0};
    auto vx2 = double{0.0};
    auto vy2 = double{0.0};

    auto x3 = double{-1.0};
    auto y3 = double{0.0};
    auto vx3 = double{0.0};
    auto vy3 = double{0.0};

    auto x4 = double{0.0};
    auto y4 = double{-1.0};
    auto vx4 = double{0.0};
    auto vy4 = double{0.0};

    // ...
```

- This gets even worse when we need to compute the actual physics in the problem.
- There are 4 particles, each interacting with the other 3, giving us 12 interactions that we must compute.

Setting Up a Particle Simulation

- We can try using vectors, and that helps us a little...

```
#include <vector>

auto main() -> int {
    auto x = std::vector<double>{1.0, 0.0, -1.0, 0.0};
    auto y = std::vector<double>{0.0, 1.0, 0.0, -1.0};
    auto vx = std::vector<double>{0.0, 0.0, 0.0, 0.0};
    auto vy = std::vector<double>{0.0, 0.0, 0.0, 0.0};
```

Setting Up a Particle Simulation

- We can try using vectors, and that helps us a little... Here the vectors group the common components, making it easier to add more.
- However, we are grouping physically unrelated data though. This ultimately fragments our data and reduces performance (only with how we intend to use this; this is not necessarily bad)!
- We also can end up with vectors missing data – that would be bad!

```cpp
#include <vector>

auto main() -> int {
    auto x = std::vector<double>{1.0, 0.0, -1.0, 0.0};
    auto y = std::vector<double>{0.0, 1.0, 0.0, -1.0};
    auto vx = std::vector<double>{0.0, 0.0, 0.0, 0.0};
    auto vy = std::vector<double>{0.0, 0.0, 0.0, 0.0};
```

Classes

- C++ is an *object-oriented* programming language, which means that data is stored and referenced in memory.
  - That data can be used in different ways depending on the *type* of that data.
- Ints, doubles, vectors, functions, etc. are all object types. C++ allows us to design and create our own objects types with a great degree of flexibility.
- In C++, these custom object types are called *classes*.
- A *class* is a *definition* of an object, and may include variables, functions, operators, etc. to describe how that object behaves.

Classes

- When we define a new class, we are defining a blueprint for the new object. The code inside of the class ***is not*** executed like a function ***until it is used***!

- As the class is just a blueprint of a type, we need to create variables of that new type in order to use it!

- The process of creating an object from a class is called *instantiating an object* – or - *creating an instance of the class*.

Classes

- We define new classes using the key-words *class* or *struct*, followed by the *name* of the class.
- We then add curly braces to denote the body of the class – this is not creating a block of scope because the class is not executed like other code!
- We can then fill the class with variables and/or functions.
- Lastly, we put a semicolon after the closing curly-brace of the class.

Classes

- Going back to our initial problem… let us create a particle class.

```
struct Particle {
    double x = 0.0;
    double y = 0.0;
    double vx = 0.0;
    double vy = 0.0;
};
```

- Here we are defining a new *data type*, *not a new variable*.

- This data type is called *Particle*, and it contains 4 doubles named *x*, *y*, *vx*, *and vy*, and they all have a **default value** of 0.0.

Classes

- In our programs we can now create variables of the type *Particle* just like we can create *ints*, or *std::vector*

```
struct Particle {
    double x = 0.0;
    double y = 0.0;
    double vx = 0.0;
    double vy = 0.0;
};

auto main() -> int {
    auto p1 = Particle{};
    auto p2 = Particle{};
    fmt::print("p1 @ <{}, {}>\n", p1.x, p1.y);
    fmt::print("p2 @ <{}, {}>\n", p2.x, p2.y);
}
```

Classes

- Every time we create an instance of the class, that object's variables, functions, etc. are all specific to that instance.
  - e.g. if we create two vectors and push back a variable into one vector, the other vector is unaffected.
  - e.g. if we create two particles and update the position of one particle, the other particle is unaffected.

- In other words, each instance of a class has its own copy of the variables and functions defined by the class.

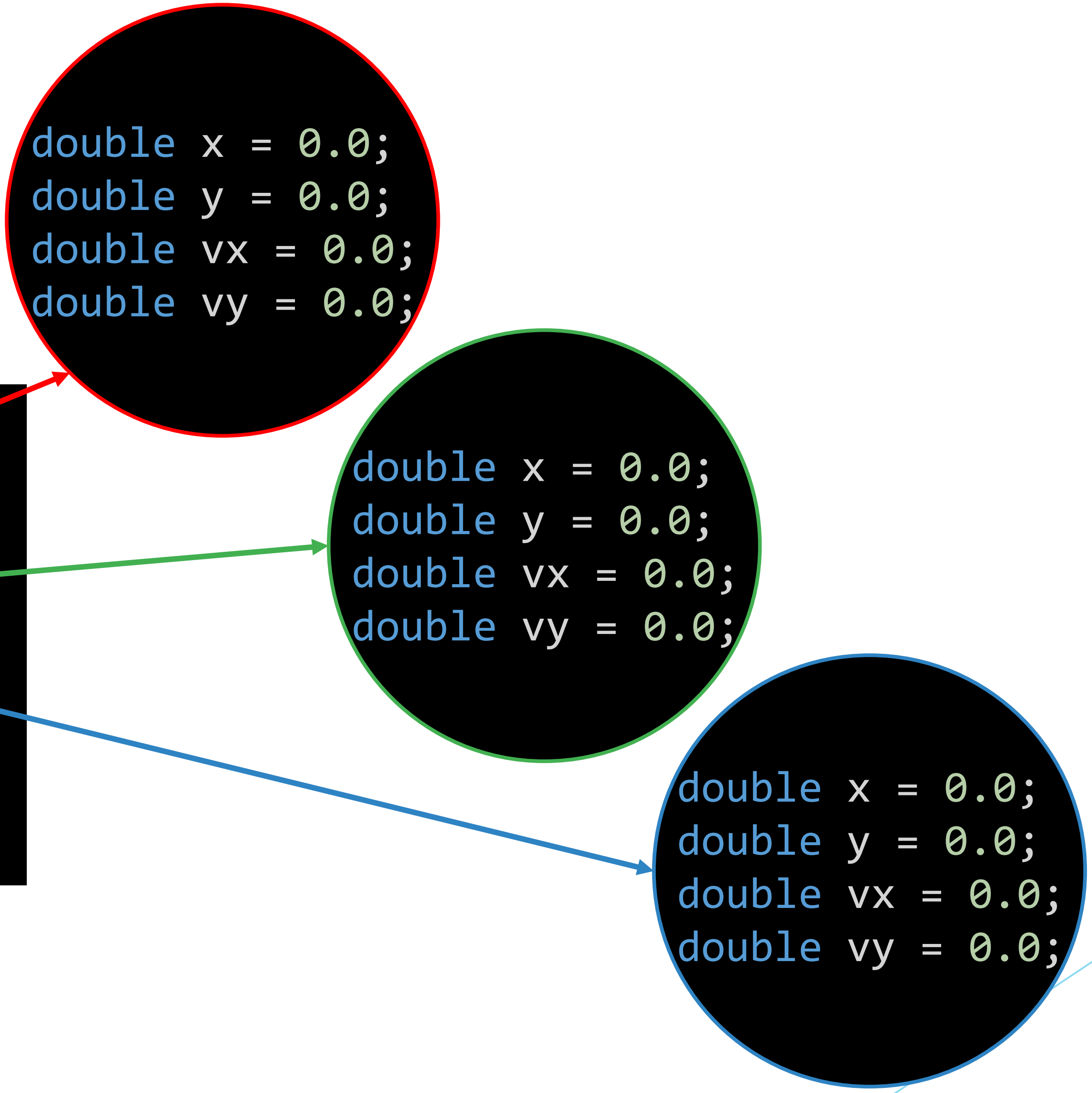
Classes

```cpp
auto main() -> int {
    Particle p1;

    Particle p2;

    Particle p3;

    // ...
```



Classes

- We use classes to represent entities and objects in a collective and intuitive way.
- We can also define functions within classes that allow us to not just define what an object is made of, but what it is capable of.
- Classes allow us to bundle "packages" of code that we can reuse; this in the end lets us write less code and gives us access to paradigms and patterns we did not have access to before.
- The order of the variables and functions ***does not matter***. Within functions normal rules of scope apply, but not to variables and functions defined in the class itself.

Classes

- Every class has direct access to its memory address (its pointer) by using the special `this` *pointer.*

- `this` is a pointer and we can reach into it to access functions and variables in the class.
  - However, use of `this` is completely optional!

- Let's add a function to our class to handle updating the position of the particle.

Classes

```
struct Particle {
    double x = 0.0;
    double y = 0.0;
    double vx = 0.0;
    double vy = 0.0;

    // this function is inside the class!
    auto update(double time_delta) -> void {
        this->x += this->vx * time_delta;
        this->y += this->vy * time_delta;
    }
};

auto main() -> int {
    auto p = Particle{};
    p.vx = 10.0;
    p.vy = 10.0;

    p.update(0.25);  // call the function from within the instance!
}
```

Classes

```
struct Particle {
    double x = 0.0;
    double y = 0.0;
    double vx = 0.0;
    double vy = 0.0;

    // this function is inside the class!
    auto update(double time_delta) -> void {
        x += vx * time_delta;
        y += vy * time_delta;
    }
};

auto main() -> int {
    auto p = Particle{};
    p.vx = 10.0;
    p.vy = 10.0;

    p.update(0.25);  // call the function from within the instance!
}
```

Classes - Constructors

- Now what we need is a way to create instances of our class.
- *Constructors* are special functions that make it easy to create instances with values.
- There are many rules surrounding constructors, and so for now we will just default to using what *C++ provides for us*.

Classes - Constructors

```
struct Particle {
    double x = 0.0;
    double y = 0.0;
    double vx = 0.0;
    double vy = 0.0;
    auto update(double time_delta) -> void {
        x += vx * time_delta;
        y += vy * time_delta;
    }
};

auto main() -> int {
    auto p1 = Particle{};              // default initialization:    x, y, vx, vy given defaults
    auto p2 = Particle{1, 2, 3, 4};    // initialization:            x=1, y=2, vx=3, vy=4
    auto p3 = Particle{1, 2};          // partial initialization:    x=1, y=2; vx, vy given defaults
    auto p4 = Particle{.vx=3, .vy=4}; // designated initialization: x, y given defaults; vx=3, vy=4
}
```

- We can initialize a Particle in a few different ways:
  - Default initialization uses the default values
  - Initialization let's us specify values for members
  - Partial initialization let's us specify some of the values for members, but we cannot skip a member, only omit them after a specific point.
  - Designated initialization let's us specify some of the values for members by name.

Improper Use of Classes

- What's wrong here?

```
struct AtomicElement {
    number = 0;
    name = "";
    mass = 0.0;
};
```

Improper Use of Classes

```
struct AtomicElement {
    int number = 0;
    std::string name = "";
    double mass = 0.0;
};
```

- We are missing the types for each of our members!

Improper Use of Classes

- What's wrong here?

```
struct AtomicElement {
    int number = 0;
    std::string name = "";
    double mass = 0.0;
};

auto main() -> int {
    auto hydrogen = AtomicElement{1, 1.00784};
}
```

Improper Use of Classes

```
struct AtomicElement {
    int number = 0;
    std::string name = "";
    double mass = 0.0;
};

auto main() -> int {
    auto hydrogen = AtomicElement{.number = 1, .mass = 1.00784};
}
```

- We did not call the constructor properly!
- While partial initialization is allowed, we skipped the name member, and C++ tried to assign 1.00784 to it. We can use designated initialization or...

Improper Use of Classes

```
struct AtomicElement {
    int number = 0;
    std::string name = "";
    double mass = 0.0;
};

auto main() -> int {
    auto hydrogen = AtomicElement{1, "hydrogen", 1.00784};
}
```

- … we can just use complete initialization

Improper Use of Classes

- What's wrong here?

```
struct CustomData {
    int x = 0;
};

auto main() -> int {
    auto cd = CustomData{};
    x = 1337;
}
```

Improper Use of Classes

```
struct CustomData {
    int x = 0;
};

auto main() -> int {
    auto cd = CustomData{};
    cd.x = 1337;
}
```

- We are using x, but x is not defined! We need to reach into *cd* to access x!

Organizing Code – Headers & Source Files

- Using header and source files for classes are a little more complicated.

- When we write the class, we simply declare its functions in the class definition, but do not define the functions (just a forward declaration!).

- When we define the functions of the class in the source file, we need to indicate to C++ that the functions we are defining are a part of the class and not some global function.

- We do this by *qualifying* the function name in the source file with the name of the class and a double colon.

Organizing Code - Headers & Source Files

- Example

```cpp
// foo.h
#pragma once

struct Foo {
    auto hello() -> void;
};
```

```cpp
// foo.cpp
#include <fmt/format.h>
#include "foo.h"

auto Foo::hello() -> void {
    fmt::print("Hello from Foo!\n");
}
```

```cpp
// main.cpp
#include "foo.h"

auto main() -> int {
    auto f = Foo{};
    f.hello();
}
```

Classes – Memory

- Whenever we create *an instance* of our classes its member variables need to be allocated in memory.

- The member variables are allocated in the order they are defined in the class.

```
struct Engine {
    int mpg = 0;
    int hp = 0;
    bool is_on = false;
};

// ...

auto e = Engine{};
```

0x014	0x015	0x016	0x017	0x018	0x019	0x01A	0x01B	0x01C	0x01D	0x01E	0x01F
e.mpg= 0				e.hp = 0				e.is_on = false			

Classes – Memory

- Whenever we create *an instance* of our classes its member variables need to be allocated in memory.

- The member variables are allocated in the order they are defined in the class.

```
struct Engine {
    int mpg = 0;
    int hp = 0;
    bool is_on = false;
};

// ...

auto e = Engine{};
```

0x014	0x015	0x016	0x017	0x018	0x019	0x01A	0x01B	0x01C	0x01D	0x01E	0x01F
e.mpg= 0				e.hp = 0				e.is_on = false			

- Everything surrounded by red is the memory consumed by our variable e.

- Is there anything weird here?

Classes – Memory

- Our Engine class is consuming 12 bytes, not 9 (4 + 4 + 1)!

- This is due to *alignment*. This alignment helps ensure that objects do not cross *cache-line boundaries*.

```
struct Engine {
    int mpg = 0;
    int hp = 0;
    bool is_on = false;
};

// ...

auto e = Engine{};
```

0x014	0x015	0x016	0x017	0x018	0x019	0x01A	0x01B	0x01C	0x01D	0x01E	0x01F
e.mpg= 0				e.hp = 0				e.is_on = false			

- Classes are aligned in memory by the largest member of the class.

- Here the largest member is an int, 4 bytes, and so the memory of the instance needs to be split into chunks of that size.

Classes – Memory

- Member variables are always aligned to the following member and chunked into blocks equal to the largest member.

```
struct Shipment
{
    short id;               // 2 bytes, aligned to 4 bytes
    int units;              // 4 bytes,
    double cost_per_unit;   // 8 bytes
    double weight;          // 8 bytes
    bool is_shipped;        // 1 bytes, aligned to 8 bytes
};

// sizeof(Shipment) == 32
```

Classes – Memory

- Each red section indicates an 8-byte block

```
struct Shipment
{
    short id;                  // 2 bytes, aligned to 4 bytes
    int units;                 // 4 bytes,
    double cost_per_unit;      // 8 bytes
    double weight;             // 8 bytes
    bool is_shipped;           // 1 bytes, aligned to 8 bytes
};

// sizeof(Shipment) == 32
```

Classes – Memory

- We can reorder the class to allow for better natural alignment

```
struct Shipment
{
    bool is_shipped;        // 1 bytes, aligned to 2 bytes
    short id;               // 2 bytes
    int units;              // 4 bytes
    double cost_per_unit;   // 8 bytes
    double weight;          // 8 bytes
};

// sizeof(Shipment) == 24
```

Classes – Memory

- In fact, we can add another bool and still not change the size of the class!

```
struct Shipment
{
    bool is_paid;           // 1 bytes
    bool is_shipped;        // 1 bytes
    short id;               // 2 bytes
    int units;              // 4 bytes
    double cost_per_unit;   // 8 bytes
    double weight;          // 8 bytes
};

// sizeof(Shipment) == 24
```

Classes – Memory

- The general rule is to order your members in a strictly increasing or decreasing order according to size – this allows for the least number of padded bits!

- All of this is necessary due to how memory is allocated and laid out – the CPU wants to avoid scenarios where an object is fragmented in its cache.
  - Data is loaded into a CPU via a ***cache-line***; this is a fixed size (usually 64 bytes) where data being processed is loaded.

- Simply put (grossly over simplified), a CPU really wants to load a variable all at once into a single cache-line. If data is misaligned then we run the risk of loading only part of a variable into the cache-line, which then means the CPU requires another cache-line for the remainder of that variable. This is called a ***cache-split***.

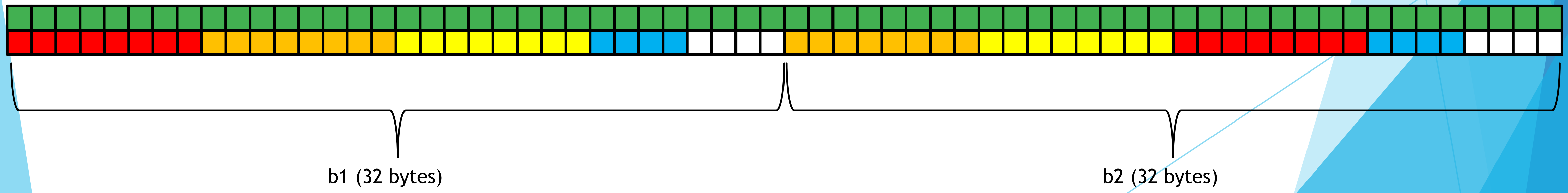
- The padding ensures that misalignment does not (or more rarely) occurs. This consumes more memory, but results in fast execution!

Classes – Memory

- Imagine the **green blocks** are a 64-byte CPU cache-line and imagine there is padding used when our objects are constructed (aligned).

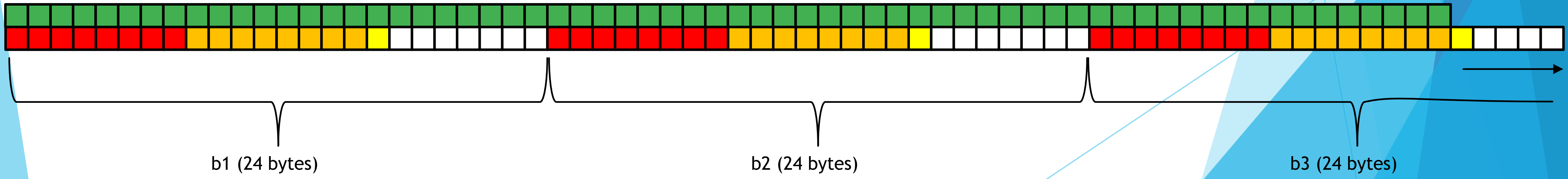
- Let our custom class be 32 bytes large (double, double, double, int), and let's say we have 2 instances of our custom class: b1 and b2 (total 64 bytes) – we could imagine these being in a vector.

- While we consume more space, our data fits very nicely into the CPU's cache-line, and should never (rarely) cache-split, *no matter how many elements we have in our vector*.



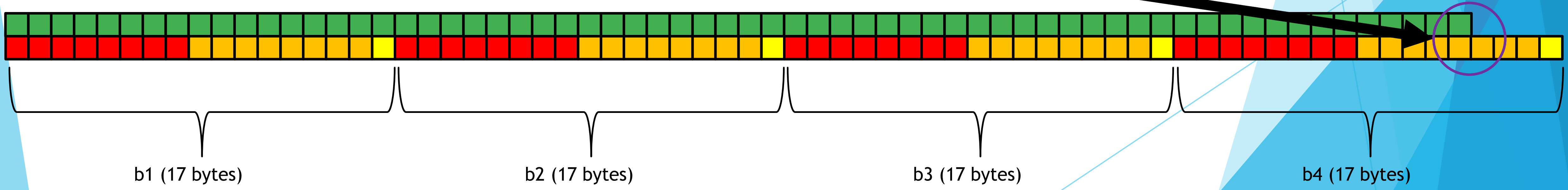
Classes – Memory

- Imagine the **green blocks** are a 64-byte CPU cache-line and imagine there is padding used when our objects are constructed (aligned).
- Let our custom class be 24 bytes large (double, double, bool), and let's say we have 3 instances of our custom class: b1, b2, and b3 (total 72 bytes) – we could imagine these being in a vector..
- While we consume more space, and b3 does not fit into a cache line along with b1 and b2, the split on b3 is on the *boundary of a member*. This is not great, but should be rare enough.



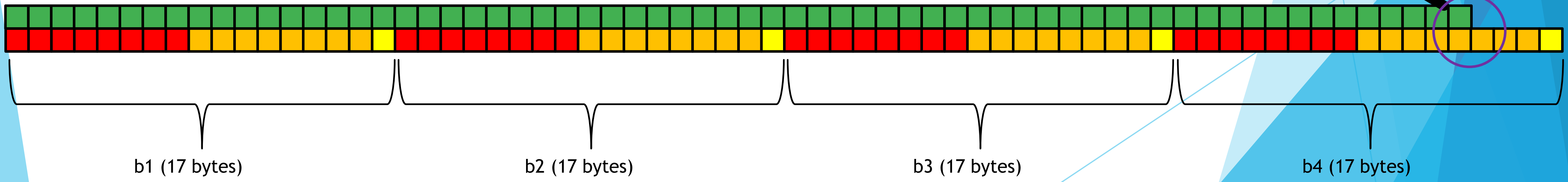
Classes – Memory

- Imagine the **green blocks** are a 64-byte CPU cache-line and imagine there was no padding used when our objects are constructed (misaligned).
- Let our custom class be 17 bytes large (double, double, bool), and let's say we have 4 instances of our custom class: b1, b2, b3, and b4 (total 68 bytes).
- What we want to avoid at all costs is cutting off a sequence of memory in the cache line. b4's orange blocks experiences a ***cache-split***!



Classes – Memory

- Cache-splits will happen frequently, but usually they occur at the beginning/end of variables.
- The **worst** cache-split is where a small region of memory (say, for a single double) is split across the boundary of the cache-line.
- This means that for the CPU to process that variable multiple load and stitching instructions are needed. **This is terrible for performance, and without alignment this will happen frequently.**



Classes – Memory

- C++ provides us with many tools for controlling bit alignment and cache-line allocations, but this is beyond the scope of this course.
- For now, we can rely on organizing our member variables in ways to help minimize our class sizes.
- This can go a long way to improving performance as we will be lowering the frequency of cache-splits!