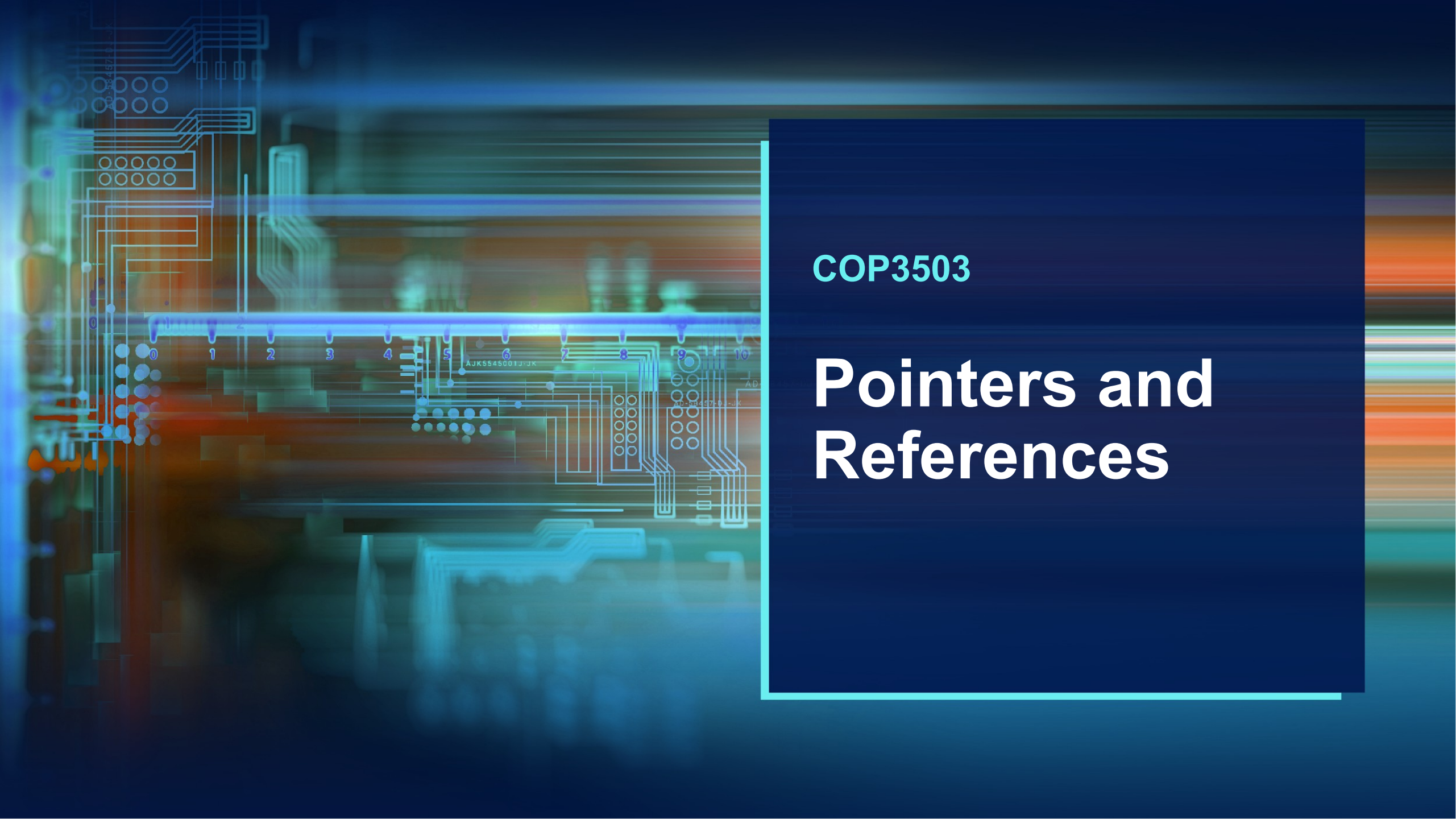


COP3503

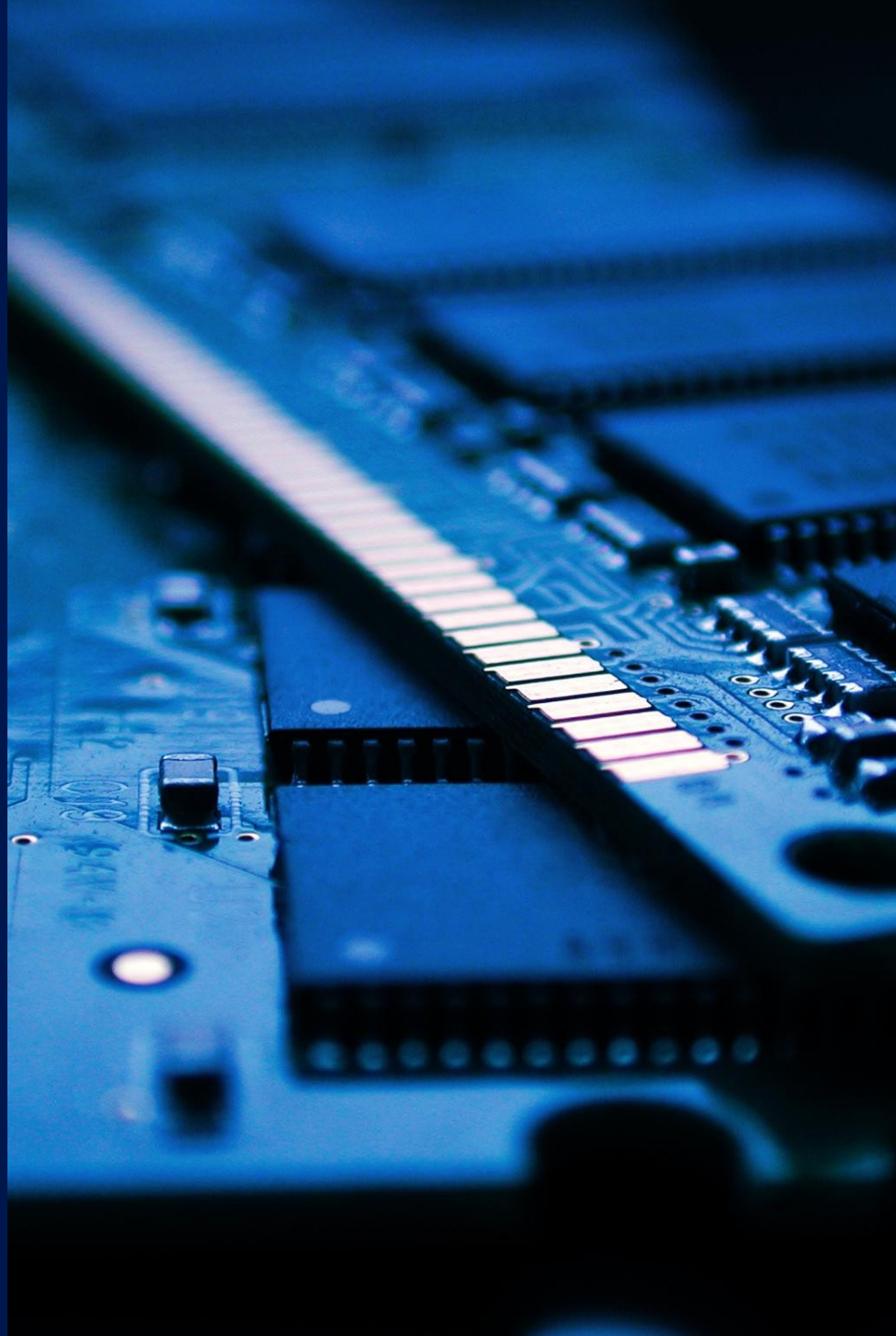
Pointers and References

Pointers

First, Let's Talk About Memory

- All of the variables we use (and more!) reside somewhere in **memory**.
- Exactly where is called its memory **address**.
- Addresses are numeric values, typically represented in hexadecimal format:
  - `0x00000000`
  - `0x1C33f198`
  - `0x00002fa1`

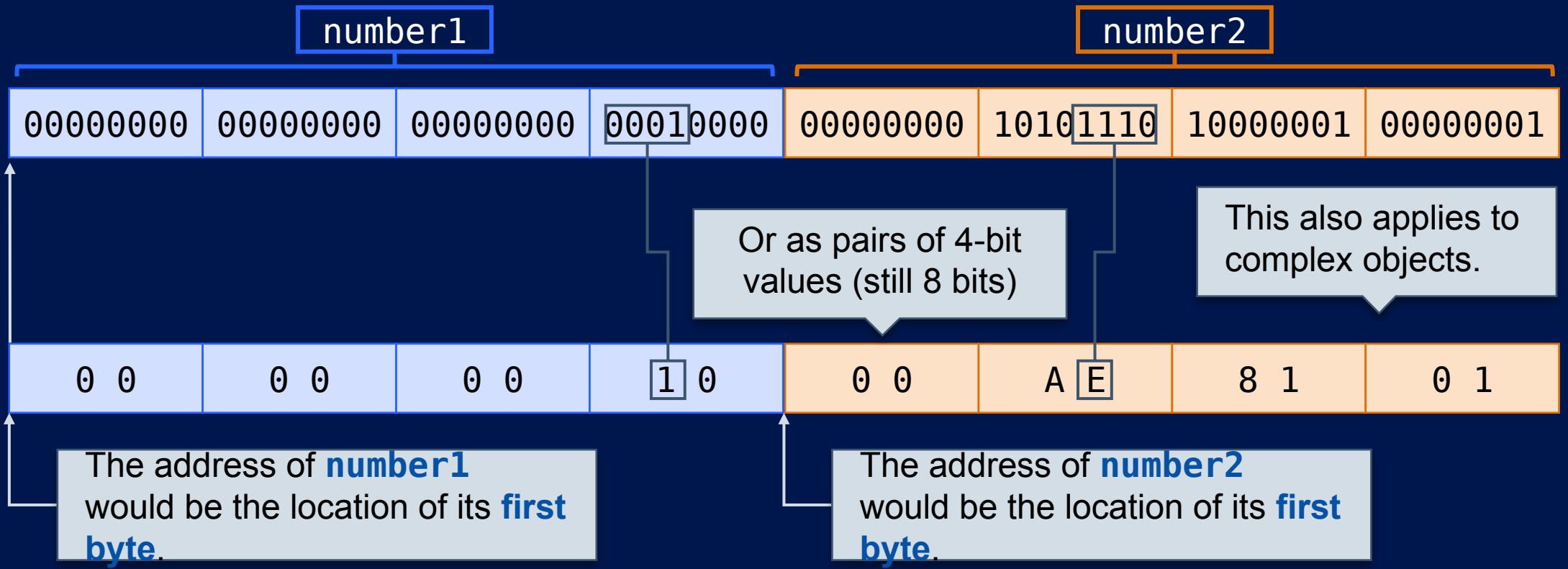
Hexadecimal is used to represent addresses because it converts from binary very easily.



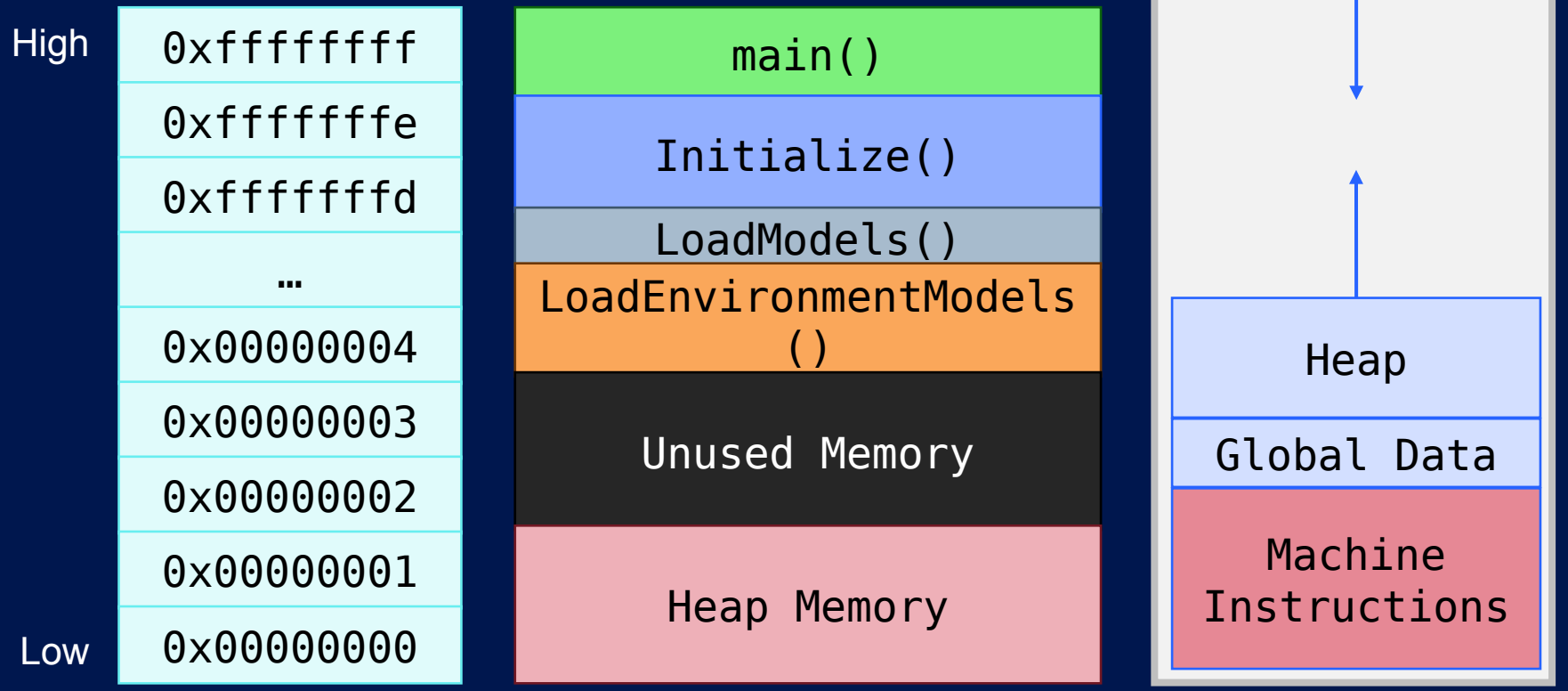
Binary / Hex Representation of Values

```
int number1 = 16;
int number2 =
98734;
```

All data types are made of **some number of bytes**.
Integers are commonly **4 bytes**.
A byte is made of **8 bits** (8 values of either 1 or 0).



Memory Diagrams



All of these can be valid, there isn't one "correct" or universal representation.

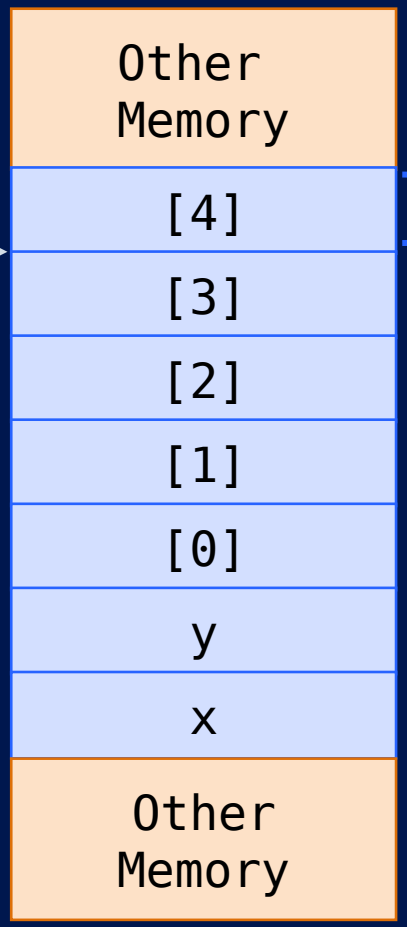
Diagrams are just an approximation; operating systems or hardware determine the exact details.

We'll focus more on how to **work with the system** (and less on how the system of memory itself works).

Basic Representation

```
int main()
{
	float x = 2.13f;
	double y = 4891.479;
	int someArray[5];

	return 0;
}
```

This would be the “address of someArray[4]”.

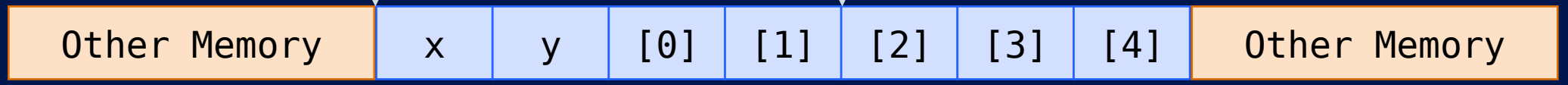


someArray[4] occupies these bytes.

This is true for **all** variables: They all live **somewhere** and occupy **some** amount of memory.

This would be the “address of x”.

This would be the “address of someArray[2]”.



x occupies these bytes.

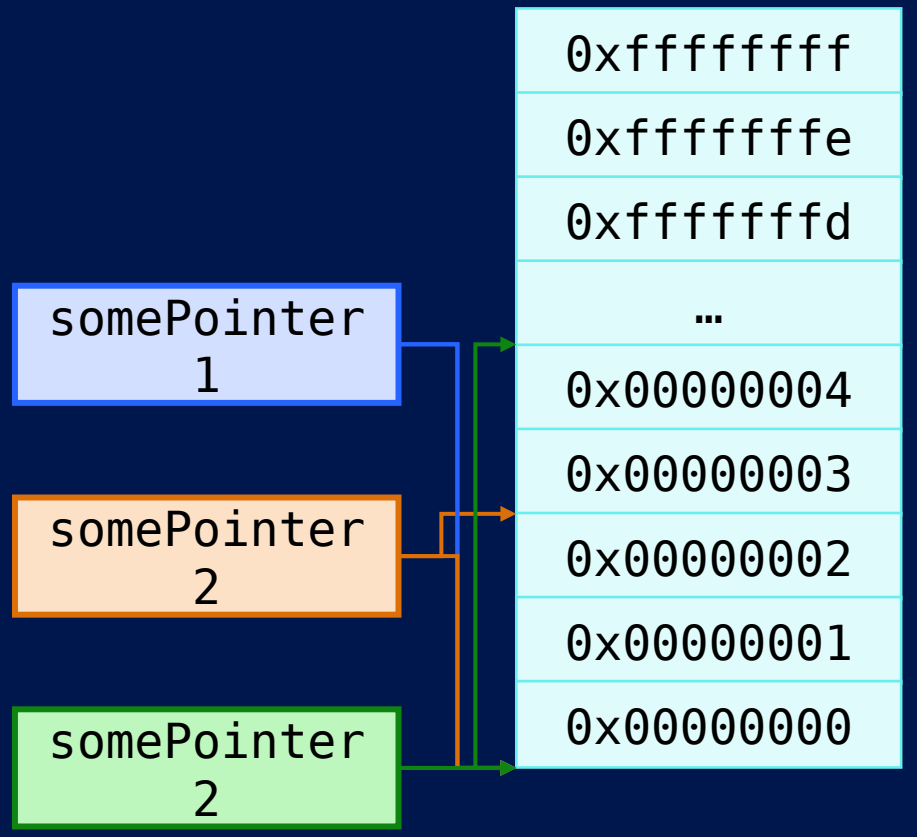
someArray[2] occupies these bytes.

Using Memory Address

- The exact address of something is often irrelevant (to us!)
  - **For example**:
    Does it matter if we know a variable is stored at `0x0CF3F219` or `0x0CF3F210`? (Most of the time: no)
- The operating system determines what goes where in memory.
  - Trust it to “hand out” the correct addresses
  - As long as you properly **store** and **use** the address of something, you’ll be fine.
- Memory addresses can be used to **share access** to existing variables.
- This requires the use of **pointers**.

What Is a Pointer?

- Fundamentally:
  A variable that contains a memory address
- A pointer “points to” a memory address, and something (i.e., some value) exists at that location.
- That something may be referred to as the **pointee**.
- A pointer can only point to **one** memory address at a time.
- Multiple pointers can point to the same address (i.e., one pointee, multiple pointers pointing to it).

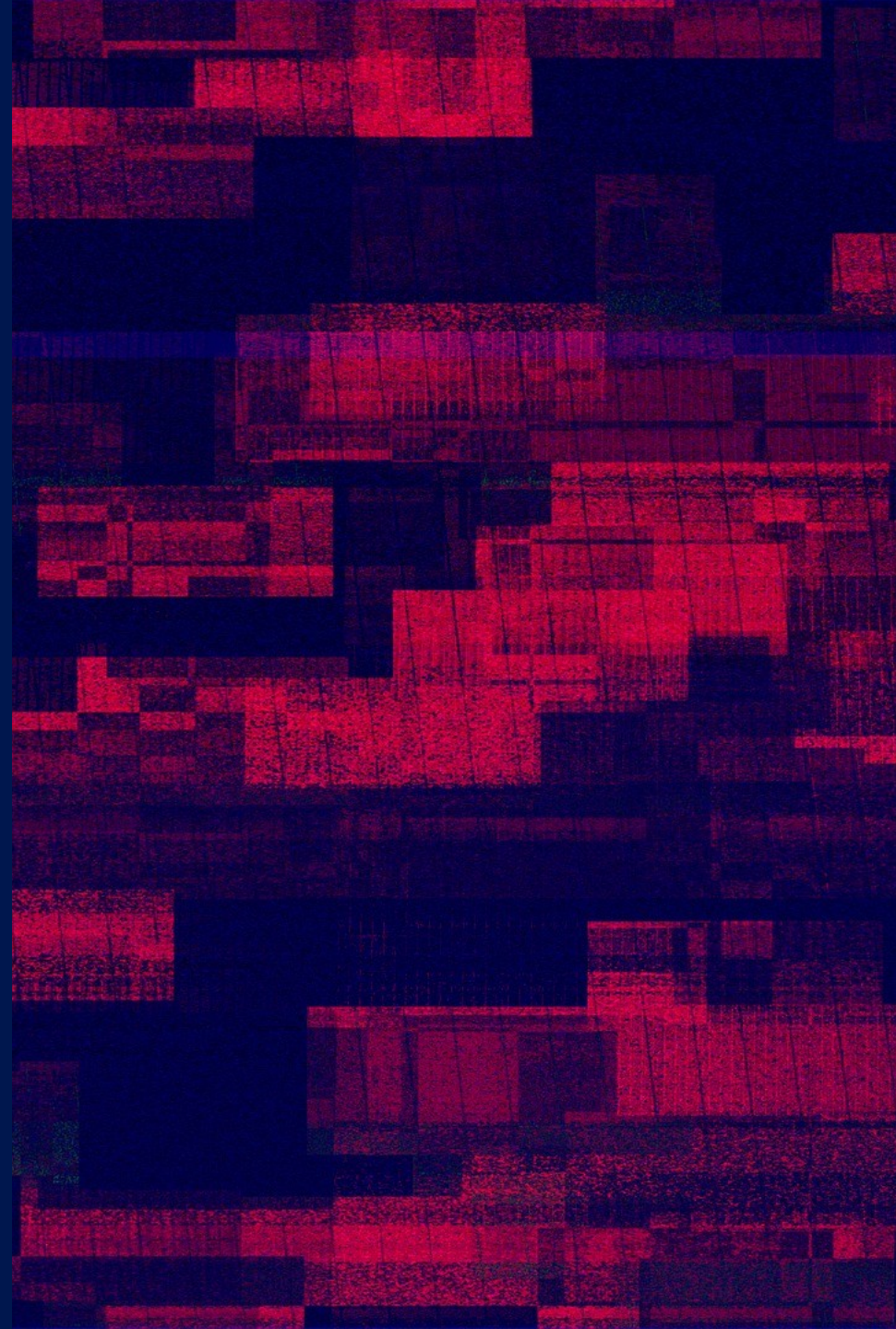


The Pointer Mantra

- Pointers point to **memory addresses**.
  - Not to objects
  - Not to variables
  - But to **memory addresses**
- Repeat this until it sticks!

Verbal shortcut:
We might **say** a pointer points to a particular variable

While technically not true on paper, it may be good enough in conversation.



Viewing the Address of Something

& am-per-sand

- In C++, we can use the **ampersand** (**&**) to get the address of a variable.
- Put an ampersand in front of a variable to indicate you want to use the **address**, not the value.

```
int x = 25;

// Print the value of a variable
cout << "The value of x: " << x << endl;

// Print the ADDRESS of a variable
cout << "The address of x: " << &x << endl;
```

```
C:\WINDOWS\system32\cmd.exe
The value of x: 25
The address of x: 00EFFA1C
```

00EFFA1C may not mean anything right now, but a memory address makes many other things possible.

Same for All Variables

```
// Create some things
int x       = 25;
float pi    = 3.14f;
string str = "Spider-Man";
short val   = 1024;

vector<int> numbers;
for (int i = 0; i < 5; i++)
        numbers.push_back(i);
```

```
// Print VALUES
cout << x << endl;
cout << pi << endl;
cout << str << endl;
cout << val << endl;
for (unsigned int i = 0; i < numbers.size();
i++)
        cout << numbers[i] << " ";
cout << endl << endl;
```

```
// Print ADDRESSES
cout << &x << endl;
cout << &pi << endl;
cout << &str << endl;
cout << &val << endl;
cout << &numbers << endl;
for (unsigned int i = 0;
i++)
        cout << &numbers[
```

```
C:\WINDOWS\system32\cmd.exe
25
3.14
Spider-Man
1024
0 1 2 3 4
```

```
C:\WINDOWS\system32\cmd.exe
004FF918
004FF90C
004FF8E8
004FF8DC
004FF8C4
00586AA8
00586AAC
00586AB0
00586AB4
00586AB8
```

How can we **store** these addresses?

Everything has an address.

Address of the vector

Addresses of the elements stored **in** the vector

Pointers Are Just Variables

- In C++, we use the **asterisk** to create a pointer variable.
  - We also use this to access the pointee (more on this later).
- Just add an asterisk to the data type when declaring the variable.

```
int* example; // Integer pointer (or: pointer TO an integer)
```

Value type	Pointer type

`int x;`
`float pi;`
`SomeClass object;`
	`int* pointerOne;`
	`float* pointerTwo;`
	`SomeClass* pointerThree;`

Pointer to an integer (`int` pointer)

Pointer to a float (`float` pointer)

Pointer to an instance of SomeClass (`SomeClass` pointer)

Pointers Are Just Variables

Which One Is Correct?

```
int* ptr1;
```

Some say this is more explicit: the type is clearly **`pointer-to-int`**.

```
int *ptr2;
```

Some argue this is better—you're dealing with integers, but ptr1 is a pointer to that type.

- In the end, syntactically, it doesn't matter. Pick what you like and stick with it. **Consistency** is more important.

What about this?
Best of both worlds?

```
int * intPointer;
float * floatPointer;
SomeClass *
someClassPointer;
```

No. **Never**. Pick what you like, as long as it isn't this.

Both "sides" team up to oppose this monstrosity.

Basic Pointer Usage

- If pointers point to addresses, and the ampersand (&) can get the address of a variable…

```
int number = 0;
int* ptr=
        &number;
```

Formal Description:
The variable **`ptr`** points to the **address of number**.

`ptr` is the **pointer**,
`number` is the **pointee**.

Informal Description:
“**`ptr`** points to **`number`**.”
This is fine for “everyday” use.
However… technically, this is not exactly correct.

Passing Pointers

- A pointer is just a new type of data.

```
int vs int*;
SomeObject vs SomeObject*;
```

```
// Assume we have these functions
void Foo(SomeObject obj)
void Bar(SomeObject* ptr);
```

```
SomeObject example;
Foo(example);     // Pass by value (i.e. by copy)
Bar(&example);    // Pass by pointer (by address of the object)
```

Pointer Are Just Variables

They can be declared, initialized, and reassigned.

`int x;` `SomeClass object, object2;`	Declare uninitialized variables

`int* pointer;`
`SomeClass* ptr1;`
`SomeClass* ptr2;`
	Declare uninitialized pointer variables

`int x = 5;`	**Declare and initialize a variable**
`int* pointer = &x;` `SomeClass* ptr1 = &object;`	Declare **and** initialize a pointer to the address of another variable
`x = 10;`	**Assign a new value to a variable**
`ptr1 = &object2;`	Assign a new value to a pointer variable
`ptr2 = ptr1;`	Assign one pointer's value to another

The rules of programming haven't suddenly changed.

Don’t forget the old stuff when you see something new!

It takes time and practice to keep track of everything.

Default Value of a Pointer?

- No default value, just “uninitialized”
  - This could be any random value, which means any random memory address… not a good idea!
- Pointers should be initialized to the address of something.
- If not something specific, then **`nullptr`** (make it a **null pointer**).

```
int* bad; // Points to… who knows what?
int* good = nullptr; // Points to NOTHING, explicitly

// Or, in the ancient tongue (i.e, older C++)
int* ptr = NULL; // NULL is just 0
int* foo = 0;
```

You should use **nullptr** over **NULL** or 0.

Why? **nullptr** is a real keyword, it’s more “official” than **NULL** nowadays.

Nullptr Can Be Useful

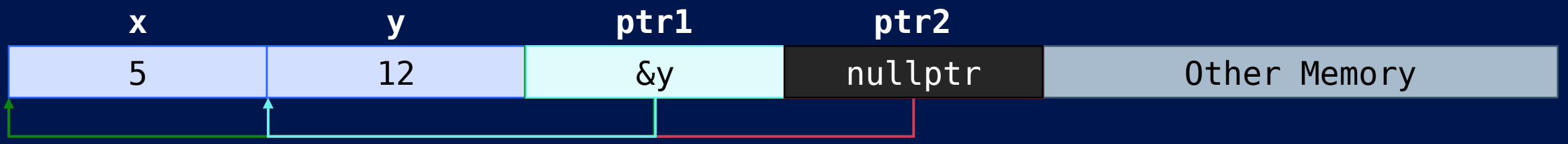
```
// If this pointer isn't pointing to anything…
if (ptr == nullptr)
{
        // Do your thing
}

if (ptr != nullptr)
{
        // Do your thing
}
```

Caution:
"Not null" does **not** mean it's valid!
We'll see examples of that later

Pointers in Memory

- Pointers are variables...
- ...and all variables reside in memory.
- Therefore, pointers reside in memory...? Right? Right!

```
int x = 5, y = 12;
int* ptr1;
int* ptr2;

ptr1 = &x; // Assign a value (i.e. an address) to the pointer
ptr2 = &y; // Ditto
ptr1 = ptr2; // Ditto (the value just comes from an existing pointer)
ptr2 = nullptr; // Set a pointer to null (doesn't point to anything)
```



Multiple Pointers to One Thing

You're Invited!
123 Main Street
Gainesville, FL
32611

You're Invited!
123 Main Street
Gainesville, FL
32611

You're Invited!
123 Main Street
Gainesville, FL
32611

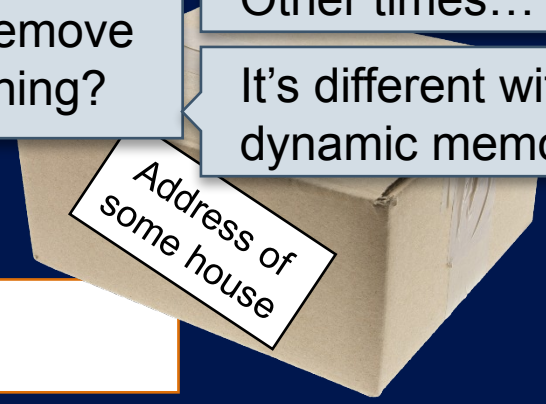
```
House theHouse;

House* partyAddress = &theHouse; // Store a copy of the address

vector<House*> lotsOfCopies;              // Store multiple copies, no
problem!
for (int i = 0; i < 20; i++)
       lotsOfCopies.push_back(&theHouse);
```

Changing a Pointer

What about the original object? Is it okay to remove a pointer from something?

Mostly, yes! Other times… no.

It's different with dynamic memory.




```
House*
deliveryTarget
```

```
House mainSt123;
House otherHouse;

House* deliveryTarget = &mainSt123;

// Change the pointer to another
house
deliveryTarget = &otherHouse;
```

Pointers and Arrays

- Some languages treat arrays as formal entities, typically as classes.
- C++ treats arrays as essentially pointers to the first element of the array.
  - This is an oversimplification, but good enough for this course.
- The **name of an array** is the same thing as **the address of the first element**.

```
int data[5] = { 2, 4, 6, 8, 10 };

int* ptr1 = data;           // array ==
&array[0]
int* ptr2 = &data[0]; // Same thing

cout << data << endl
cout << ptr1 << endl
cout << ptr2 << endl
```

```
00AFFAAC
00AFFAAC
00AFFAAC
Press any key to continue . . .
```

Pointers and Arrays

- A pointer only stores a single memory address.
- The number of elements beyond that isn't stored with the pointer.
- **Additional variables** (created by you) must be used to track that data.

```
void BadCodeExample(int* pointer)
{
        // Is this safe? Probably
not...
        cout << pointer[0] << endl;
        cout << pointer[1] << endl;
        cout << pointer[2] << endl;
}
```

```
void Example(int* pointer, unsigned int
count)
{
        // The provided parameter does the
        // work of counting
        for (unsigned int i = 0; i < count;
i++)
                cout << pointer[i] << endl;
}
```

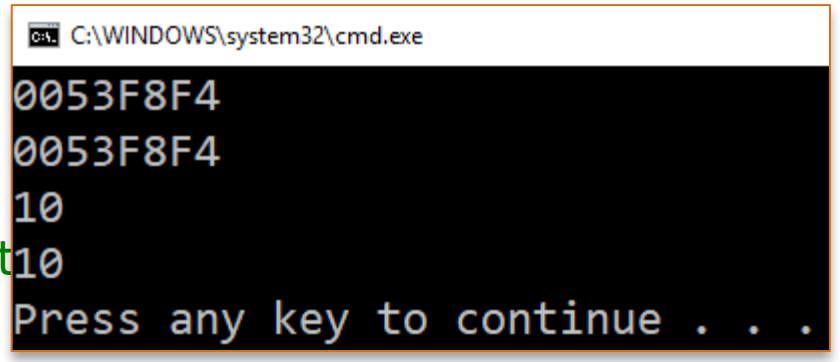
```
// How big is the array
pointer.size()?
pointer.length()?
pointer.count()?
```

None of these exist; you have to implement it yourself (or use containers like vectors)

Dereferencing Pointers

- To get the data from a memory address, we have to **dereference** a pointer.
- Dereferencing **retrieves the value of the pointee** from that location.
- To do so, we use the **asterisk** again—this time, on an already-existing pointer.

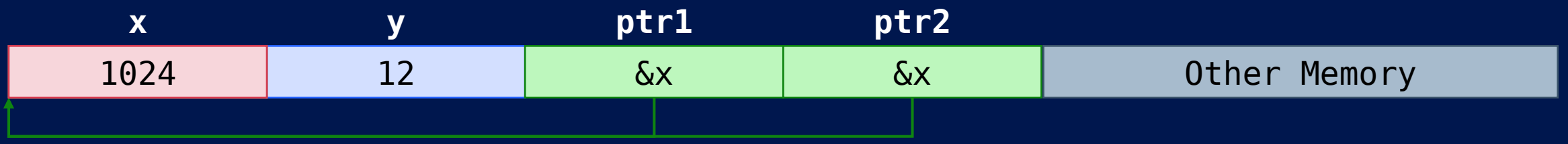
```
int x = 10;
int* ptr = &x;

cout << &x << endl; // Prints out the address of
cout << ptr << endl;       // Ditto (ptr is point

cout << x << endl;         // Print out 10
cout << *ptr << endl;      // Ditto--dereference a pointer to get a value
```



Dereferencing and Changing a Pointee

- Pointers give access (indirectly) to some original value.
- Simply dereference the pointer, then operate on it like a normal variable.
- This is critical when we need to make changes in functions, and with classes.

```
int x = 5, y = 12;
int* ptr1 = &x;
int* ptr2 = &x;

*ptr1 = 20;    // Change x to 20 (indirectly, by way of a pointer)
cout << *ptr2; // Print the value of ptr2's pointee (it'll be 20)
*ptr2 = 1024;  // Change x to 1024 (indirectly, by way of a pointer)
cout << x;     // Print the value of x (it'll be 1024)
```



Dereferencing Null (or Bad) Pointers

- Dereferencing a pointer that is **`nullptr`** will probably throw an exception.
- The same goes for dereferencing a pointer that isn't `nullptr`, but also isn't valid (i.e. doesn't point to something usable…).
- Officially it is **undefined behavior**, a common C++ phrase.

```
float* pointerToNothing;
cout << *pointerToNothing; // What will happen here? Unknown!

int* nullPointer = nullptr;
cout << *nullPointer;              // Unknown… likely to thrown an exception
```

Undefined Behavior

- Literally what it sounds like: Behavior that is not defined.
- The C++ standard defines what the language **should** do in many situations, but it doesn't cover **everything**.
- In those "gray" areas, your program can behave unexpectedly.

```
float numbers[5];   // Just an array of 5 values
cout << numbers[50];       // What should happen here?
cout << numbers[-3];       // Or here? Crash? Bad output?
```

Many languages have safeguards for these situations, C++ does not.

Most classes **do** have them implemented, but simple arrays and pointers don't, so be careful!

Pointers to Class Objects

```
// Assume we have this class…
class Hero
{
	int hitpoints;		// How long before they die?
	int money;	// How rich are they?
	string name;	// It's… a name…
public:
	Hero();
	int GetMoney();
	int GetHitpoints();
	void TakeDamage(int damage);
};
```

```
// Using an object
Hero link;
link.TakeDamage(5);
int howMuchBling = link.GetMoney();
```

```
// Using a pointer to an object
Hero
ptr->TakeDamage(5);
int remaingLife = ptr-
>GetHitpoints();
```

One tiny difference

Dereferencing Pointers to Class Objects

- Membership operator

-> Indirect membership operator

- For objects, we use the **membership operator**.
- For pointers, instead we have to use the **indirect membership operator**.
- Why? Let's assume we have a pointer to an instance of a class object.

```
void PrintStats(Hero* hero)
{
        int hp = pointer.GetHitpoints();
        cout << "Hitpoints: " << hp <<
endl;
        // Repeat for other attributes...
}
```

Pointers don't have functions —class objects do.

We need to first **dereference** the pointer.

How to Dereference:

```
Hero* hero
```

```
int hp = *hero.GetHitpoints(); // Just deference, like normal?
```

This is dereferencing the **return** of `GetHitpoints()`
… which is just a number… essentially this:

```
int hp = *(hero.GetHitpoints()); // Error, "hero" is still a pointer
```

We need to dereference **first**, then use the result of **that**:

```
int hp = (*hero).GetHitpoints(); // Works, but…
```

Wrapping a class pointer with: (*) **every** time you want to use it?
Not ideal. Instead, use the **indirect membership operator**.

```
int hp = hero->GetHitpoints();
cout << "Our hero's cash: " << hero->GetMoney();
```

. **Membership operator** (For objects)

-> **Indirect membership operator** (For pointers to objects)

Pointers Provide Indirect Access

- A pointer is **not** the thing.
- A pointer is the location where the thing can be found in memory.
- Say you write down a house address on a piece of paper:

Super sweet party tonight at:

123 Main Street
Gainesville, FL
32611

This is not a house.

Here is an **address** where a house can be found. If you go to that address, the house will be **there**.

Here is where we store the **address of the house**, not the house itself.

Another Analogy

- Say you have a map to some buried treasure. Awesome!
- The map can help you get the treasure, but the treasure itself is somewhere else.
- Wait a minute… the map is not the treasure itself.
- The treasure map is just a “pointer” to the real prize.

Treasure map == Pointer

Treasure chest == Pointee

But… Why Pointers?

- So far, this is the **what** and **how** of pointers, but… why?
- The primary reason why: **efficiency**
- Pointers let us avoid making copies of everything.
- Pointers are **very small** in memory.
  - Which is larger? An address on a piece of paper or the house itself?
- Copying small data is trivial, but large objects are costly.
  - Copying a small address is easy, but copying a house…
- Many concepts and algorithms are basically impossible if we don’t use pointers.

References

The laid-back, easy-going cousin of pointers

References

Similar to pointers (kind of)

- Pointers tell us the **location** of something else.
- References **act as a stand-in (or alias)** for another variable.
- A reference **is the variable that it references**.
  - Any changes to a reference change the original.
  - They don't use memory addresses, **no dereferencing**.
- Created by using the **ampersand** (**&**) along with the regular data type.

```
Person somePerson;

// Create a reference TO another obje
Person& ref = somePerson;
```

After this line of code, **ref** and **somePerson** are now the **same entity**.

References Are the Thing They Reference

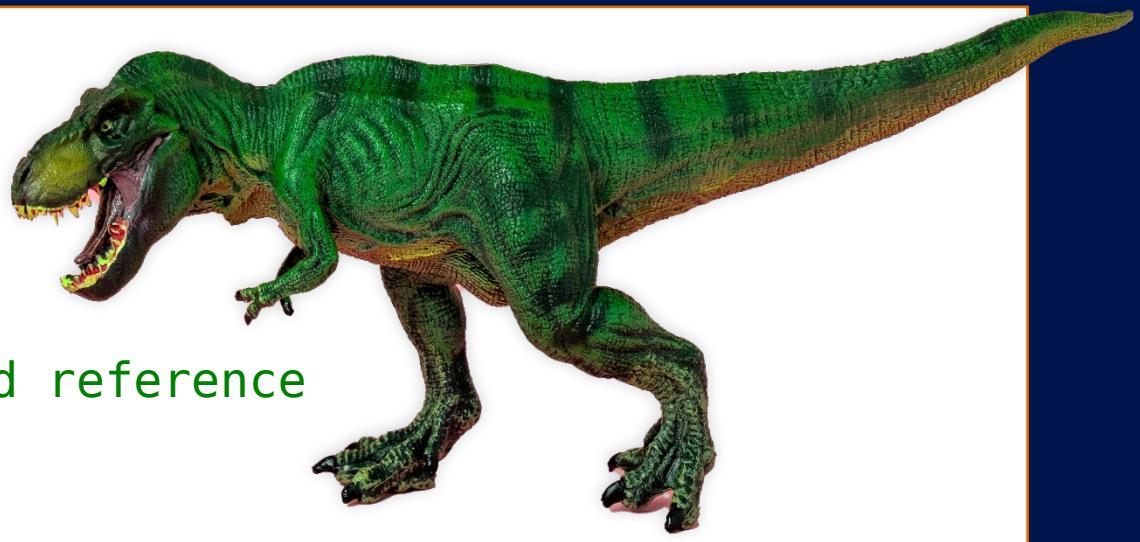
```
vector<int> values;

// Create/bind a reference to the original
vector<int>& alias = values;

// Add some data to the reference/original
for (int i = 1; i <= 10; i++)
       alias.push_back(i);

cout << "Number of elements: " << alias.size();
cout << "Number of elements: " << values.size();
```

Like we saw with pointers, you wouldn't create a reference right next to the original.

Reference Rules

- They **must** be initialized. No such thing as a “null reference”.
- Once bound, they cannot be reassigned or re-bound (the data they reference, however, **can** be changed).

```
Dinosaur tRex;

// You MUST initialize a reference
Dinosaur& dinoRef = tRex;

// Compiler error, can't have uninitialized reference
// “null reference” isn't a concept in C++
Dinosaur& otherRef;
otherRef = tRex;
```



Do References Have Their Own Memory?

```
void Foo(vector<int>& ref)
{
	cout << "Address of ref: " << &ref << endl;
}

int main()
{
	vector<int> original;
	cout << "Address of original: " << &original << endl;
	Foo(original);
	return 0;
}
```

Short answer:
No! References are not “other” memory

```
Address of original: 00AFFCAC
Address of ref: 00AFFCAC
Press any key to continue . . .
```

Where Does the Ampersand Go?

```
int& reference =
    someInt;
```

```
int &reference =
    someInt;
```

- Same as with pointers: Doesn’t matter, pick one and stick with it.

```
// Just don’t pick this
int & thisIsBad =
someInt;
```

Passing and Returning References

References as parameters:

```
void Foo(vector<int>& numbers);
void Bar(SomeClass& object);
```

Pass-by-reference is much faster than creating copies.

References as return types (for class member variables):

```
class Example
{
        vector<int> numbers;
        LargeObject lotsOfBytes;
public:
        vector<int>& GetNumbers();
        LargeObject& GetBytes_Original();
        LargeObject  GetBytes_Copy();
};
```

Return by reference

Return by copy

Passing Pointers vs. References vs. Copies

```
class Person
{
        string name;
public:
        string GetName();
};
```

```
// Calling those functions…
Person somePerson;

Foo_Ptr(&somePerson);
Foo_Cpy(somePerson);
Foo_Ref(somePerson);
```

No syntactical difference between copy and reference.

```
// Pass by pointer
void Foo_Ptr(Person* p)
{
        cout << p->GetName();
}
```

We have to dereference pointers.

```
// Pass by copy
void Foo_Cpy(Person p)
{
        cout << p.GetName();
}
```

```
// Pass by reference
void Foo_Ref(Person& p
{
        cout << p.GetName();
}
```

We don't dereference objects or references.

Why Should You Use References?

- They don’t copy objects, which is a performance boost!
- They don’t require addresses or dereferencing.
- **Most** of the benefits of pointers, with some of their own restrictions
- The **recommended way** of passing class objects around
- Use pointers when you **need** pointers, references in all other cases.

Is There Anything References Can't Do?!

- You can't create arrays of references.
  - No way to initialize all of them at the time of declaration
- You can't reassign references.
  - Need to change targets? Select a different item in a list, a piece on a game board, player in a game?
  - Use a pointer for this!
    (It's basically why they exist.)

```
// How to initialize all 10 of
these?
// Answer: you can't (sorry
dinos…)
Dinosaur& dinos[10];
```

```
Dinosaur tRex, triceratops;
Dinosaur& target = tRex;

// Copies triceratops to tRex
// Does NOT change the reference
target = triceratops;
```

```
Dinosaur tRex, triceratops;
Dinosaur* target = &tRex;

// Change targets to the
triceratops
// Point at a new memory address
target = &triceratops;
```

Ultimate Showdown! * vs. &

* Pointer

- **Indirect access** to something
  - (Must be dereferenced… with *)
- Store memory addresses
- Can point to nothing (`nullptr`)
- Flexible, can be reassigned
- Used to pass data quickly, avoids creating copies
- Must be used when dynamically creating new data (more on this later!)

& Reference

- **Direct access** to something
  - (No deferencing required)
- Act as an alias for another object
- Cannot be null
- Can’t be reassigned
- Used to pass data quickly, avoids creating copies

Use references everywhere you can, pointers when you can’t!

Recap

- Every exists in memory and has a **memory address**.
- **Pointers** are a way of storing and sharing those addresses, to share data efficiently.
- Pointers provide **indirect access** to something and must be **dereferenced** to access the “real” data they point to.
- Pointers are **powerful** but dangerous, and **easily misused**.

Recap

- References are an “**easier alternative**” to pointers and allow for **direct access**.
- They act as a **stand-in (or alias)** for other variables, and don’t have to be dereferenced.
- They’re similar but are a **different tool for different situations**.
- We’ll look at a lot of examples and uses of both throughout the rest of the semester.

Conclusion

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

Thank you for watching.