



C++

...Easy, Elegant, Powerful!

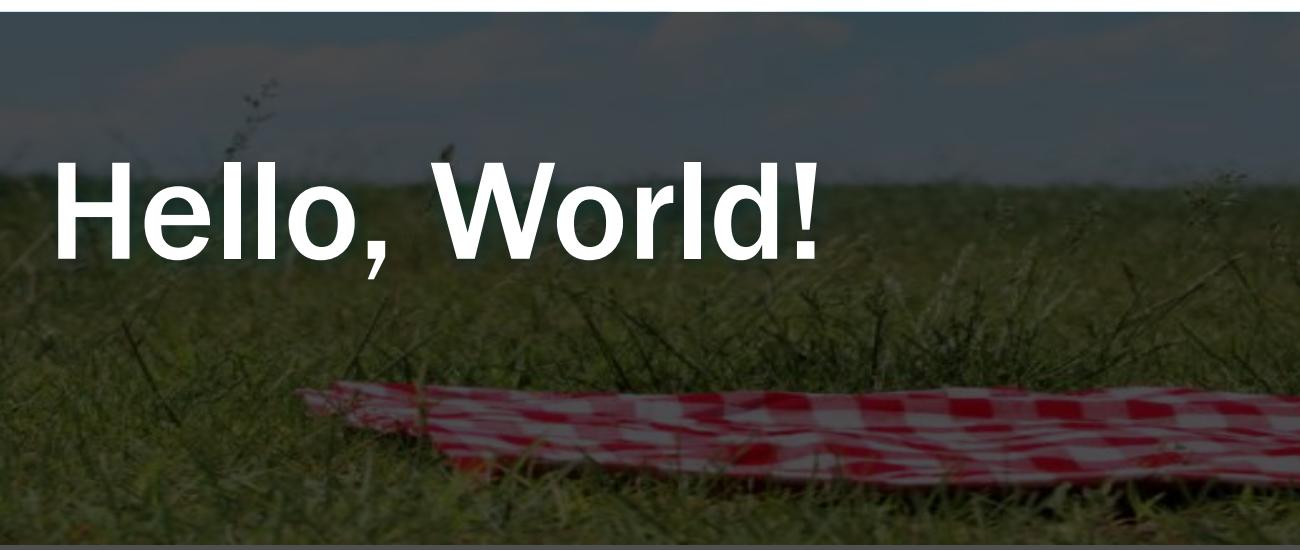


charley bay
charleyb123 at gmail dot com

Today's Agenda

1. Data Types, Data Objects
2. Pointers
3. Strong Typing
4. Value Semantics
5. Lexical Scoping
6. Conclusion





My First Program



Hello, World!

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

Output:

```
Hello, World!
```

*Exit value from process
(often ignored, but may be
captured by shell scripts)*

Hello, World!

- **Well-defined:**

- Bootstrap through **main()**
- Execution sequence stops when **main()** ends

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

Output:

```
Hello, World!
```

*Exit value from process
(often ignored, but may be
captured by shell scripts)*

Hello, World!

- **Well-defined:**
 - Bootstrap through **main()**
 - Execution sequence stops when **main()** ends
- **Some “setup”** can occur before **main()**:
 - OS loads your program into memory, and initializes process resources
 - Initialization of “global” and “static” data objects

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

Output:

Hello, World!

*Exit value from process
(often ignored, but may be
captured by shell scripts)*

Hello, World!

- **Well-defined:**
 - Bootstrap through `main()`
 - Execution sequence stops when `main()` ends
- **Some “setup”** can occur before `main()`:
 - OS loads your program into memory, and initializes process resources
 - Initialization of “global” and “static” data objects
- **Some “shutdown”** can occur after `main()`:
 - Global objects are destructed
 - OS reclaims system resources

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

Output:

Hello, World!

*Exit value from process
(often ignored, but may be
captured by shell scripts)*

Hello, World!

- **Well-defined:**
 - Bootstrap through `main()`
 - Execution sequence stops when `main()` ends
- **Some “setup”** can occur before `main()`:
 - OS loads your program into memory, and initializes process resources
 - Initialization of “global” and “static” data objects
- **Some “shutdown”** can occur after `main()`:
 - Global objects are destructed
 - OS reclaims system resources

“When `main()` begins,
the program begins.”

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

Output:

Hello, World!

*Exit value from process
(often ignored, but may be
captured by shell scripts)*

When `main()` is over,
the program is over.”

Undefined Behavior?

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```



WHERE'S THE
UNDEFINED BEHAVIOR?

Q: Where is the
undefined behavior?

Undefined Behavior?

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

- The program executes by doing *exactly* what you specify
 - If you tell the process to jump off a cliff, ***then it will***



WHERE'S THE
UNDEFINED BEHAVIOR?

Q: Where is the
undefined behavior?

Undefined Behavior?

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

- The program executes by doing *exactly* what you specify
 - If you tell the process to jump off a cliff, *then it will*



Q: Where is the **undefined** behavior?

A: No **undefined** behavior is present

Undefined Behavior?

```
// FILE: MyHello.cpp
#include <cstdio>

int main() {
    puts("Hello, World!");
    return 0;
}
```

- The program executes by doing exactly what you specify
 - If you tell the process to jump off a cliff, ***then it will***



WHERE'S THE
UNDEFINED BEHAVIOR?

Q: Where is the **undefined** behavior?

A: No **undefined** behavior is present

You must do something **wrong** to **add** undefined behavior to your program

Undefined Behavior

- **GOOD:** Your code should **RELY** upon those things the **C++ Language Guarantees**

Your System:

- Is based on **Well-Defined Behavior**
- Does **not** invoke **Undefined Behavior**



Undefined Behavior

- **GOOD**: Your code should **RELY** upon those things the **C++ Language Guarantees**
- **BAD**: Your code can **ASSUME** something that the C++ Language **does not guarantee**

Your System:

- Is based on **Well-Defined Behavior**
- Does **not** invoke **Undefined Behavior**



Undefined Behavior

- **GOOD**: Your code should RELY upon those things the C++ Language Guarantees
- **BAD**: Your code can ASSUME something that the C++ Language does not guarantee
- If you do something Undefined, You Are Wrong
 - The C++ Language provides all necessary guarantees to build any kind of system
 - But not a system:
 - that is inconsistent
 - that does not reflect reality (*of the underlying hardware: “The Abstract Machine”*)

Your System:

- Is based on Well-Defined Behavior
- Does not invoke Undefined Behavior



Undefined Behavior

- **GOOD**: Your code should RELY upon those things the C++ Language Guarantees
- **BAD**: Your code can ASSUME something that the C++ Language does not guarantee
- If you do something Undefined, You Are Wrong
 - The C++ Language provides all necessary guarantees to build any kind of system
 - But not a system:
 - that is inconsistent
 - that does not reflect reality (*of the underlying hardware: “The Abstract Machine”*)

Your System:

- Is based on Well-Defined Behavior
- Does not invoke Undefined Behavior



Undefined Behavior:

An assumption you might make,
but which is not guaranteed
by the C++ Language
(*by definition, you cannot reason about it*)

Undefined Behavior

- **GOOD**: Your code should RELY upon those things the C++ Language Guarantees
- **BAD**: Your code can ASSUME something that the C++ Language does not guarantee
- If you do something Undefined, You Are Wrong
 - The C++ Language provides all necessary guarantees to build any kind of system
 - But not a system:
 - that is inconsistent
 - that does not reflect reality (*of the underlying hardware: “The Abstract Machine”*)

Well-Defined Behavior:

That which the C++ Language specifies
(so you can reason about what is going on)

Your System:

- Is based on Well-Defined Behavior
- Does not invoke Undefined Behavior



Undefined Behavior:

An assumption you might make,
but which is not guaranteed
by the C++ Language
(*by definition, you cannot reason about it*)

Confusion Is Understandable?



- You might try to use components that are:
 - Stupid (*bad abstractions*)



Confusion Is Understandable?

- You might try to use components that are:
 - Stupid (*bad abstractions*)
 - Not designed well (*poor API and behavior*)



Confusion Is Understandable?

- You might try to use components that are:
 - Stupid (*bad abstractions*)
 - Not designed well (*poor API and behavior*)
 - Are complicated (*perhaps warranted, perhaps not*)



Confusion Is Understandable?

- You might try to use components that are:
 - Stupid (*bad abstractions*)
 - Not designed well (*poor API and behavior*)
 - Are complicated (*perhaps warranted, perhaps not*)
- In such cases, it may be unclear: “What should I do?”



Confusion Is Understandable?

- You might try to use components that are:
 - Stupid (*bad abstractions*)
 - Not designed well (*poor API and behavior*)
 - Are complicated (*perhaps warranted, perhaps not*)
- In such cases, it may be unclear: “What should I do?”



When (Not “If”) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost

When (Not “If”) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost
- **Usually:**
 1. You cannot interfere with other processes
 - *But, is possible:* Hackers work hard to interfere with other processes

When (*Not “If”*) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost
- **Usually:**
 1. You cannot interfere with other processes
 - *But, is possible:* Hackers work hard to interfere with other processes
 2. You do not hurt your hardware
 - *But, is possible:* Can damage hardware if you are writing device drivers (*may require privileged permissions*)

When (*Not “If”*) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost
- **Usually:**
 1. You cannot interfere with other processes
 - *But, is possible:* Hackers work hard to interfere with other processes
 2. You do not hurt your hardware
 - *But, is possible:* Can damage hardware if you are writing device drivers (*may require privileged permissions*)
 3. You do not hurt your files-on-disk
 - *But, is possible:* If your program writes files to disk, you could overwrite with empty or useless data

When (Not “If”) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost
- **Usually:**
 1. You cannot interfere with other processes
 - *But, is possible:* Hackers work hard to interfere with other processes
 2. You do not hurt your hardware
 - *But, is possible:* Can damage hardware if you are writing device drivers (*may require privileged permissions*)
 3. You do not hurt your files-on-disk
 - *But, is possible:* If your program writes files to disk, you could overwrite with empty or useless data

Try! Experiment!
Usually, you have to
work “extra hard”
to really hurt your
hardware or file system

When (Not “If”) We Are Stupid

- **If your process (program) does something stupid:**
 1. It is usually killed by the operating system
 2. The data in your program is probably lost
- **Usually:**
 1. You cannot interfere with other processes
 - *But, is possible:* Hackers work hard to interfere with other processes
 2. You do not hurt your hardware
 - *But, is possible:* Can damage hardware if you are writing device drivers (*may require privileged permissions*)
 3. You do not hurt your files-on-disk
 - *But, is possible:* If your program writes files to disk, you could overwrite with empty or useless data

These issues exist to some extent with all programming languages

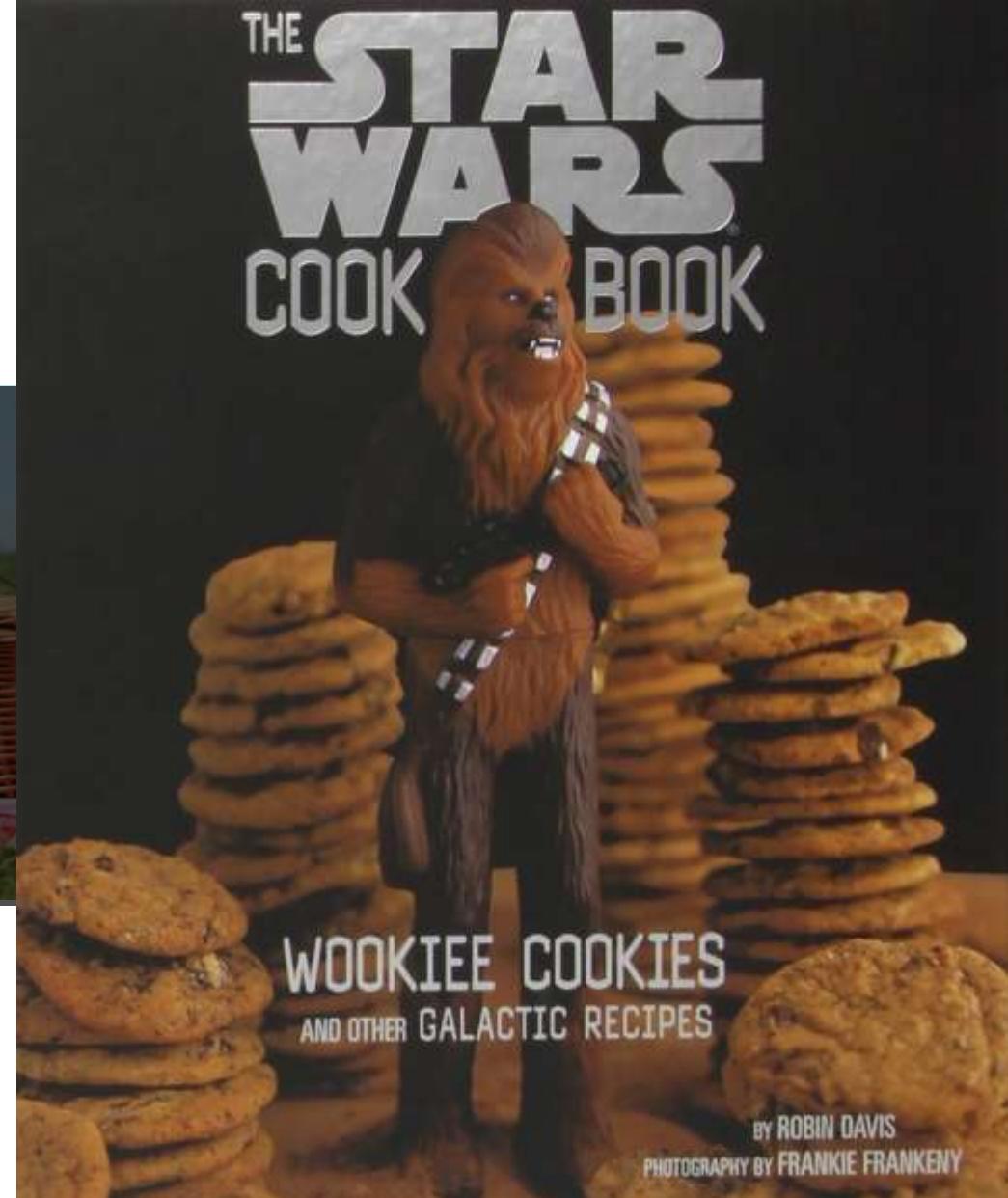
(but C++ gives you more power for Good or Evil)

Try! Experiment!
Usually, you have to work “extra hard” to really hurt your hardware or file system



Creating Data Objects

All You Can Eat!



Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    ...
}
```



Instantiation: All You Can Eat!

Instantiation: Bringing into existence

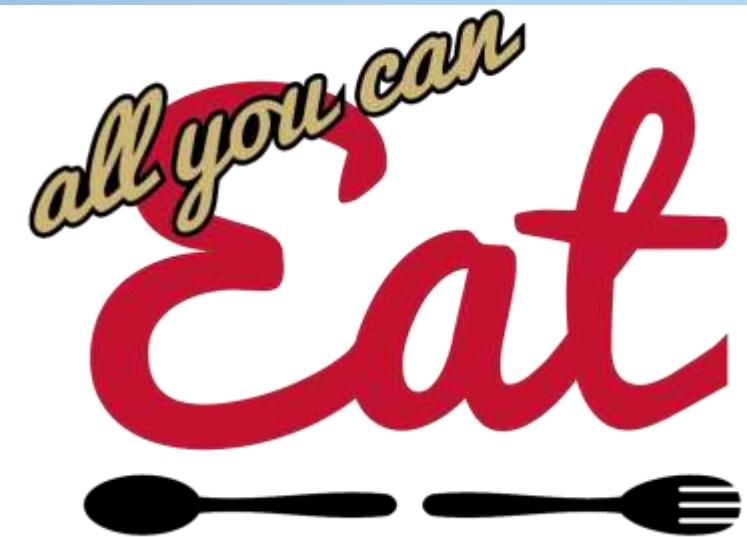
```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    ...do some more stuff...
    ...
    return my_int;
}
```



Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    ...do some more stuff...
    return my_int;
}
```



You **instantiate** one of:

- A function
- A data object

"Define" a function
"Define" a data object

Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    int another_int;
    ...do some more stuff...

    return my_int;
}
```



You **instantiate** one of:

- A function
- A data object

"Define" a function
"Define" a data object

Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    int another_int;
    int my_array_of_ints[100];
    ...do some more stuff...
    return my_int;
}
```



You **instantiate** one of:

- A function
- A data object

"Define" a function
"Define" a data object

Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    int another_int;
    int my_array_of_ints[100];
    ...do some more stuff...
    return my_int;
}
```



You **instantiate** one of:

- A function
- A data object

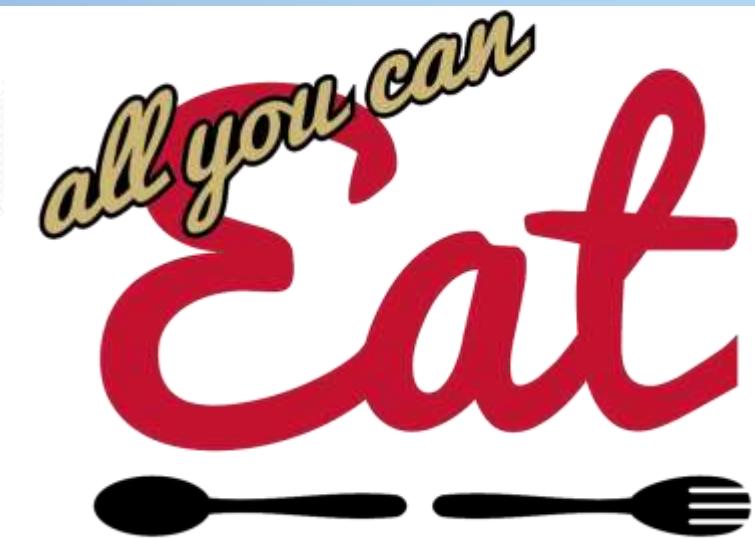
"Define" a function
"Define" a data object

- You can **instantiate as many things as you want!**
(Up to system resource limits)

Instantiation: All You Can Eat!

Instantiation: Bringing into existence

```
// Let's create a function "foo"!
int foo() {
    // Let's create some data objects!
    int my_int = 42;
    int another_int;
    int my_array_of_ints[100];
    ...do some more stuff...
    return my_int;
}
```



You **instantiate** one of:

- A function
- A data object

"Define" a function
"Define" a data object

- You can **instantiate as many things as you want!**
(Up to system resource limits)
- **Virtual Memory** – modern Operating Systems even let you instantiate more things than physically fit in memory!

Saying vs. Doing

Declare: Saying it exists (*it might not*)

Define: Instantiation (*i.e., allocating memory*)

Saying vs. Doing

Declaring something
DOES NOT
allocate memory!

Declare: Saying it exists (*it might not*)

Define: Instantiation (*i.e., allocating memory*)



Saying vs. Doing

Declaring something
DOES NOT
allocate memory!

Declare: Saying it exists (*it might not*)

Define: Instantiation (*i.e., allocating memory*)



```
// Say 'my_int' exists
extern int my_int;
```

```
// Say 'foo()' exists
int foo();
```

Saying vs. Doing

Declaring something
DOES NOT
allocate memory!

Declare: Saying it exists (*it might not*)

Define: Instantiation (*i.e., allocating memory*)

```
// Really define 'my_int'  
int my_int;
```

```
// Really define 'foo'  
int foo () {  
    ...do stuff...  
    return 42;  
}
```

```
// Say 'my_int' exists  
extern int my_int;
```

```
// Say 'foo ()' exists  
int foo ();
```



Saying vs. Doing

Declaring something
DOES NOT
allocate memory!

Declare: Saying it exists (*it might not*)

Define: Instantiation (*i.e., allocating memory*)



```
// Really define 'my_int'  
int my_int;
```

```
// Really define 'foo'  
int foo () {  
    ...do stuff...  
    return 42;  
}
```

```
// Say 'my_int' exists  
extern int my_int;
```

```
// Say 'foo ()' exists  
int foo ();
```

Because “declaring” something
does not allocate memory:
This becomes a **TOOL** that
we use to **“hook-up” data objects
and functions** across libraries!



Organizing Your Instantiations?

- If you can instantiate as much as you want, how do you organize your system?



Organizing Your Instantiations?

- If you can instantiate as much as you want, how do you organize your system?
- This is: Software Engineering



Organizing Your Instantiations?

- If you can instantiate as much as you want, how do you organize your system?
- This is: Software Engineering
- Your job is to define a strategy (*this is called Design*)



Dis plan

not work like I hoped...

Organizing Your Instantiations?

- If you can instantiate as much as you want, how do you organize your system?
- This is: Software Engineering
- Your job is to define a strategy (*this is called Design*)
- This is easier in C++ than other languages, because you are explicit about how you spend resources



Organizing Your Instantiations?

- If you can instantiate as much as you want, how do you organize your system?
- This is: Software Engineering
- Your job is to define a strategy (*this is called Design*)
- This is easier in C++ than other languages, because you are explicit about how you spend resources



Concern: You spent your resources
in a way that did not address your problem

Concern for ANY system,
implemented with ANY language!

Taking Control

- Taking Control in your C++ system is usually through:

Your Object Model



Your Control Flow



Taking Control

- Taking Control in your C++ system is usually through:

Your Object Model

1. What objects exist?



2. Who owns a given object?

Your Control Flow



Taking Control

- Taking Control in your C++ system is usually through:

Your Object Model

1. What objects exist?

- You define the types that exist
(and rules for those types)
- You instantiate objects

2. Who owns a given object?



KEEP IT SiMPLE



KEEP IT SiMPLE

Taking Control

- Taking Control in your C++ system is usually through:

Your Object Model

1. What objects exist?

- You define the types that exist
(and rules for those types)
- You instantiate objects

2. Who owns a given object?



KEEP IT SiMPLE



KEEP IT SiMPLE

Taking Control

- Taking Control in your C++ system is usually through:

Your Object Model

1. What objects exist?

- You define the types that exist (*and rules for those types*)
- You instantiate objects

2. Who owns a given object?



KEEP IT SiMPLE

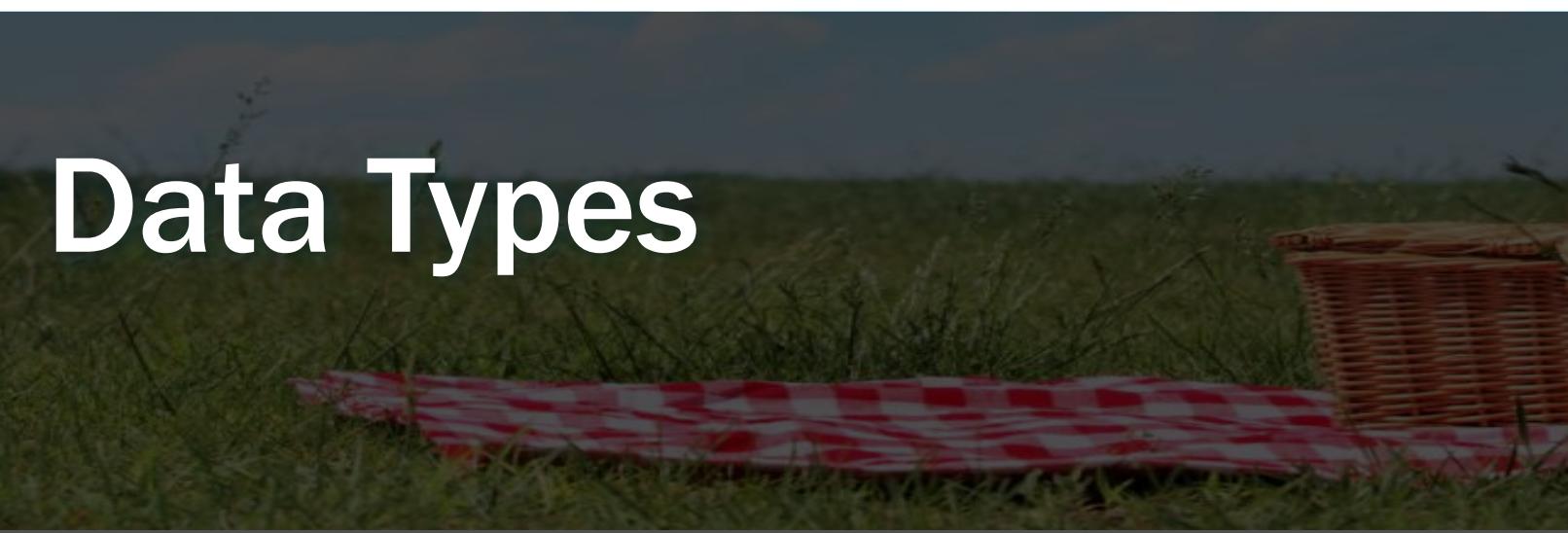


KEEP IT SiMPLE

Your Control Flow

1. Your Object Lifecycle

- What causes objects to be created, mutated, and destroyed?



Data Types

Primitive Types, And Your Types



Data Types

*Also called
“Fundamental Type”*

Primitive Type

- Defined by the C++ Language
- Supported by the underlying hardware

Data Types

Also called
Fundamental Type

Primitive Type

- Defined by the C++ Language
- Supported by the underlying hardware

User-Defined Type

- Primitive Data Object “*with your rules applied on-top*”
- May be a collection of Primitive and User-Defined types

Data Types

Primitive Type

- Defined by the C++ Language
- Supported by the underlying hardware

Also called
Fundamental Type

```
// Primitive type
int my_int = 42;
```

Defined by C++ Language Standard

User-Defined Type

- Primitive Data Object “*with your rules applied on-top*”
- May be a collection of Primitive and User-Defined types

Data Types

Primitive Type

- Defined by the C++ Language
- Supported by the underlying hardware

Also called
Fundamental Type

```
// Primitive type
int my_int = 42;
```

Defined by C++ Language Standard

User-Defined Type

- Primitive Data Object “*with your rules applied on-top*”
- May be a collection of Primitive and User-Defined types

```
// User-defined type
Dog my_dog = "Fluffy";
```

Defined by You

Primitive Types (“*Fundamental Types*”)

1

Integral (*whole number*)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

Primitive Types (“Fundamental Types”)

1

Integral (whole number)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

```
int my_int = 42;
```

Primitive Types (“Fundamental Types”)

1

Integral (whole number)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

```
int my_int = 42;
```

2

Floating Point (real number)

- Stores a real-number (i.e., *fractional value*)
- May be Positive or Negative

Primitive Types (“Fundamental Types”)

1

Integral (whole number)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

```
int my_int = 42;
```

2

Floating Point (real number)

```
float my_float = 3.14;
```

- Stores a real-number (i.e., *fractional value*)
- May be Positive or Negative

Primitive Types (“Fundamental Types”)

1

Integral (whole number)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

```
int my_int = 42;
```

2

Floating Point (real number)

```
float my_float = 3.14;
```

- Stores a real-number (i.e., *fractional value*)
- May be Positive or Negative

3

Pointer (address)

- Stores an address
- Identifies location of data-object on the local machine

Primitive Types (“Fundamental Types”)

1

Integral (whole number)

- Stores a whole number
- May be Positive or Negative (*depending on type*)

```
int my_int = 42;
```

2

Floating Point (real number)

```
float my_float = 3.14;
```

- Stores a real-number (i.e., *fractional value*)
- May be Positive or Negative

3

Pointer (address)

```
void* my_ptr = nullptr;
```

- Stores an address
- Identifies location of data-object on the local machine

Lots Of Different Hardware!

The C++ Language Standard provides primitive types so you can:

1. Make design choices for what resources you wish to use
2. Reason about the real-world behavior (*because primitive types directly map your data objects to underlying hardware*)

Lots Of Different Hardware!

The C++ Language Standard provides primitive types so you can:

1. Make design choices for what resources you wish to use
2. Reason about the real-world behavior (*because primitive types directly map your data objects to underlying hardware*)

- Lots of different hardware exist ← **C++ runs on them all!**

- CPUs of different architectures (x86, AMD64, MIPS, ARM, PowerPC, ...)
- GPUs, APUs, FPGAs, DSPs, Intel MICs, microcontrollers, ...

NOTE: “Containers” and Virtual Machines (VMs) still present a virtual hardware platform with specific resources and capabilities onto which your software is mapped

Lots Of Different Hardware!

The C++ Language Standard provides primitive types so you can:

1. Make design choices for what resources you wish to use
2. Reason about the real-world behavior (*because primitive types directly map your data objects to underlying hardware*)

- Lots of different hardware exist ← **C++ runs on them all!**
 - CPUs of different architectures (x86, AMD64, MIPS, ARM, PowerPC, ...)
 - GPUs, APUs, FPGAs, DSPs, Intel MICs, microcontrollers, ...
- Each hardware has different limitations
 - Primitive types supported, valid ranges, alignment requirements, etc.

NOTE: “Containers” and Virtual Machines (VMs) still present a virtual hardware platform with specific resources and capabilities onto which your software is mapped

Lots Of Different Hardware!

The C++ Language Standard provides primitive types so you can:

1. Make design choices for what resources you wish to use
2. Reason about the real-world behavior (*because primitive types directly map your data objects to underlying hardware*)

- Lots of different hardware exist ← **C++ runs on them all!**
 - CPUs of different architectures (x86, AMD64, MIPS, ARM, PowerPC, ...)
 - GPUs, APUs, FPGAs, DSPs, Intel MICs, microcontrollers, ...
- Each hardware has different limitations
 - Primitive types supported, valid ranges, alignment requirements, etc.

NOTE: “Containers” and Virtual Machines (VMs) still present a virtual hardware platform with specific resources and capabilities onto which your software is mapped

The underlying hardware is reality:

Failure to account for mapping to system resources
is a common source of issues in software systems

C++ Fundamental Types

void	// Type "with an empty set of values"
std::nullptr_t	// Type of null pointer literal
bool	// 1-bit (0 or 1, false or true)
char	// usually 8-bit (-128..+127)
unsigned char	// usually 8-bit (0..+256)
wchar_t	// 16+bits, wide-string code point
char8_t	// 8-bits, for UTF-8 (since C++20)
char16_t	// 16-bits, for UTF-16
char32_t	// 32-bits, for UTF-32
short	// 16+bits, usually 16-bit (-32768..+32767)
unsigned short	// 16+bits, usually 16-bit (0..+65535)
int	// 16+bits, usually 32-bit (-32768..+32767)
unsigned int	// 16+bits, usually 32-bit (0..+65535)
long	// 32+bits, usually 64-bit (-2147483648..+2147483647)
unsigned long	// 32+bits, usually 64-bit (0..+4294967295)
long long	// 64+bits, usually 64-bit (-9223372036854775808..+9223372036854775807)
unsigned long long	// 64+bits, usually 64-bit (0..+18446744073709551615)
std::size_t	// 64+bits, usually 64-bit (0..+18446744073709551615)
float	// usually 32-bits
double	// usually 64-bits
long double	// usually 80-bits

C++ Fundamental Types

<code>void</code> // Type "with an empty set of values"		"Void-ish"
<code>std::nullptr_t</code> // Type of null pointer literal		"Pointer-ish"
<code>bool</code> // 1-bit (0 or 1, false or true)		
<code>char</code> // usually 8-bit (-128..+127)		
<code>unsigned char</code> // usually 8-bit (0..+256)		
<code>wchar_t</code> // 16+bits, wide-string code point		
<code>char8_t</code> // 8-bits, for UTF-8 (since C++20)		
<code>char16_t</code> // 16-bits, for UTF-16		
<code>char32_t</code> // 32-bits, for UTF-32		
<code>short</code> // 16+bits, usually 16-bit (-32768..+32767)
<code>unsigned short</code> // 16+bits, usually 16-bit (0..+65535)
<code>int</code> // 16+bits, usually 32-bit (-32768..+32767)
<code>unsigned int</code> // 16+bits, usually 32-bit (0..+65535)
<code>long</code> // 32+bits, usually 64-bit (-2147483648..+2147483647)
<code>unsigned long</code> // 32+bits, usually 64-bit (0..+4294967295)
<code>long long</code> // 64+bits, usually 64-bit ((-9223372036854775808..+9223372036854775807))
<code>unsigned long long</code> // 64+bits, usually 64-bit (0..+18446744073709551615))
<code>std::size_t</code> // 64+bits, usually 64-bit (0..+18446744073709551615))
<code>float</code> // usually 32-bits		
<code>double</code> // usually 64-bits		
<code>long double</code> // usually 80-bits		

C++ Fixed Width Types (since C++11)

Integer type
with bit-count exact

```
int8_t // 8-bits  
int16_t // 16-bits  
int32_t // 32-bits  
int64_t // 64-bits
```

```
uint8_t // 8-bits  
uint16_t // 16-bits  
uint32_t // 32-bits  
uint64_t // 64-bits
```

fastest integer type
with bit-count or more

```
int_fast8_t // 8+bits  
int_fast16_t // 16+bits  
int_fast32_t // 32+bits  
int_fast64_t // 64+bits
```

```
uint_fast8_t // 8+bits  
uint_fast16_t // 16+bits  
uint_fast32_t // 32+bits  
uint_fast64_t // 64+bits
```

smallest integer type
with bit-count or more

```
int_least8_t // 8+bits  
int_least16_t // 16+bits  
int_least32_t // 32+bits  
int_least64_t // 64+bits
```

```
uint_least8_t // 8+bits  
uint_least16_t // 16+bits  
uint_least32_t // 32+bits  
uint_least64_t // 64+bits
```

```
intmax_t // maximum width integer type  
uintmax_t // maximum width unsigned integer type  
intptr_t // unsigned integer type capable of holding a pointer  
uintptr_t // unsigned integer type capable of holding a pointer
```

C++ Fixed Width Types (since C++11)

Integer type
with bit-count exact

fastest integer type
with bit-count or more

smallest integer type
with bit-count or more

```
int8_t // 8-bits
int16_t // 16-bits
int32_t // 32-bits
int64_t // 64-bits

int_fast8_t // 8+bits
int_fast16_t // 16+bits
int_fast32_t // 32+bits
int_fast64_t // 64+bits

int_least8_t // 8+bits
int_least16_t // 16+bits
int_least32_t // 32+bits
int_least64_t // 64+bits

intmax_t // maximum width integer type
uintmax_t // maximum width unsigned integer type
intptr_t // unsigned integer type capable of holding a pointer
uintptr_t // unsigned integer type capable of holding a pointer
```

“Integer-ish”

“Integer-ish” and “Pointer-ish”

C++ Fixed Width Types (since C++11)

Integer type
with bit-count exact

*Can reason!
(for serialization,
wire protocols)*

fastest integer type
with bit-count or more

Optimize for “speed”

smallest integer type
with bit-count or more

Optimize for “size”

```
int8_t // 8-bits  
int16_t // 16-bits  
int32_t // 32-bits  
int64_t // 64-bits
```

```
int_fast8_t // 8+bits  
int_fast16_t // 16+bits  
int_fast32_t // 32+bits  
int_fast64_t // 64+bits
```

```
int_least8_t // 8+bits  
int_least16_t // 16+bits  
int_least32_t // 32+bits  
int_least64_t // 64+bits
```

```
intmax_t // maximum width integer type  
uintmax_t // maximum width unsigned integer type  
intptr_t // unsigned integer type capable of holding a pointer  
uintptr_t // unsigned integer type capable of holding a pointer
```

“Integer-ish”

```
uint8_t // 8-bits  
uint16_t // 16-bits  
uint32_t // 32-bits  
uint64_t // 64-bits
```

```
uint_fast8_t // 8+bits  
uint_fast16_t // 16+bits  
uint_fast32_t // 32+bits  
uint_fast64_t // 64+bits
```

```
uint_least8_t // 8+bits  
uint_least16_t // 16+bits  
uint_least32_t // 32+bits  
uint_least64_t // 64+bits
```

“Integer-ish” and “Pointer-ish”

Which one?

- How to select my primitive type?



C++ Fundamental Types

```
void // Type "with an empty set of values"           ← "Void-ish"
std::nullptr_t // Type for null pointer literal      ← "Pointer-ish"
bool // 1-bit (0 or 1, false or true)
char // usually 8-bit (-128..+127)
unsigned char // usually 8-bit ( 0..+256)             ← "Integer-ish"
wchar_t // 16+bits, wide string code point
char8_t // 8-bits, for UTF-8 (int C++11)
char16_t // 16-bits, for UTF-16 (int C++11)
char32_t // 32-bits, for UTF-32 (int C++11)
short // 16+bits, usually 16-bit ( -32768..+32767 )
unsigned short // 16+bits, usually 16-bit ( 0..+65535 )
int // 16+bits, usually 32-bit ( -32768..+32767 )    ← "Int"
unsigned int // 16+bits, usually 32-bit ( 0..+65535 )
long // 32+bits, usually 64-bit ( -2147483648..+2147483647 )
unsigned long // 32+bits, usually 64-bit ( 0..+4294967295 )
long long // 64+bits, usually 64-bit ( -9223372036854775808..+9223372036854775807 )
unsigned long long // 64+bits, usually 64-bit ( 0..+18446744073709551615 )
std::size_t // 64+bits, usually 64-bit ( 0..+18446744073709551615 )
float // usually 32-bits
double // usually 64-bits
long double // usually 80-bits                         ← "Float-ish"
```

When in doubt,
pick this one

Primitive Types You Will Use

- Use *lots*:
 - **void** : “*nothing here*”



Primitive Types You Will Use

- Use *lots*:
 - **void** : “*nothing here*”
 - **int** : “*whole number*”



Primitive Types You Will Use

- Use lots:
 - **void** : “nothing here”
 - **int** : “whole number”
 - **float** : “floating-point (real) number”



Primitive Types You Will Use

- Use lots:

- **void** : “*nothing here*”
- **int** : “*whole number*”
- **float** : “*floating-point (real) number*”

```
void doSomething(int i, float f);
```



Primitive Types You Will Use



- Use **lots**:

- **void** : “*nothing here*”
- **int** : “*whole number*”
- **float** : “*floating-point (real) number*”

```
void doSomething(int i, float f);
```

- Use **occasionally**:

- **bool** : “*yes-or-no*”

Primitive Types You Will Use



- Use **lots**:

- **void** : “nothing here”
- **int** : “whole number”
- **float** : “floating-point (real) number”

```
void doSomething(int i, float f);
```

- Use **occasionally**:

- **bool** : “yes-or-no”
- **unsigned int** : “never-negative-int”

Primitive Types You Will Use



- Use **lots**:

- **void** : “*nothing here*”
- **int** : “*whole number*”
- **float** : “*floating-point (real) number*”

```
void doSomething(int i, float f);
```

- Use **occasionally**:

- **bool** : “*yes-or-no*”
- **unsigned int** : “*never-negative-int*”
- **double** : “*bigger-float*”

Primitive Types You Will Use



- Use *lots*:

- **void** : “*nothing here*”
- **int** : “*whole number*”
- **float** : “*floating-point (real) number*”

```
void doSomething(int i, float f);
```

- Use *occasionally*:

- **bool** : “*yes-or-no*”
- **unsigned int** : “*never-negative-int*”
- **double** : “*bigger-float*”

```
bool is_open(MyLogFile log_file);
```

Pointers

Pointers Hold Addresses



What Is Stored?

Circle One:

1. An int holds: integral number floating-point number address nothing

What Is Stored?

Circle One:

1. An int holds:

integral number

floating-point number

address

nothing

What Is Stored?

Circle One:

- | | | | | |
|--------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

What Is Stored?

Circle One:

1. An int holds: integral number floating-point number address nothing
2. A float holds: integral number floating-point number address nothing

What Is Stored?

Circle One:

- | | | | | |
|--------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 3. A <u>void</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

What Is Stored?

Circle One:

- | | | | | |
|--------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 3. A <u>void</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

What Is Stored?

Circle One:

- | | | | | |
|----------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 3. A <u>void</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 4. A <u>pointer</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

What Is Stored?

Circle One:

- | | | | | |
|----------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 3. A <u>void</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 4. A <u>pointer</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

What Is Stored?

Circle One:

- | | | | | |
|----------------------------|------------------------|------------------------------|----------------|----------------|
| 1. An <u>int</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 2. A <u>float</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 3. A <u>void</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |
| 4. A <u>pointer</u> holds: | <u>integral number</u> | <u>floating-point number</u> | <u>address</u> | <u>nothing</u> |

Take-Away for Today:

A pointer holds an address

Address Of What?

Given:

```
void* my_void_ptr; // what is held?
```

- Q: What is stored in `my_void_ptr`?
- A:



Address Of What?

Given:

```
void* my_void_ptr; // what is held?
```

- Q: What is stored in `my_void_ptr`?
- A: An address ✓ *Full Credit*



Address Of What?

Given:

```
void* my_void_ptr; // what is held?
```

- Q: What is stored in `my_void_ptr`?
- A: An address ✓ *Full Credit*

BONUS Question:

- Q: Address of what?
- A:

RECALL: A pointer holds an address



Address Of What?

Given:

```
void* my_void_ptr; // what is held?
```

- Q: What is stored in `my_void_ptr`?
- A: An address ✓ *Full Credit*

BONUS Question:

- Q: Address of what?
- A1: Address of “nothing” ✓ *Full Credit*
- A2:

RECALL: A pointer holds an address



Address Of What?

Given:

```
void* my_void_ptr; // what is held?
```

- Q: What is stored in `my_void_ptr`?
- A: An address ✓ *Full Credit*

BONUS Question:

- Q: Address of what?
- A1: Address of “nothing” ✓ *Full Credit*
- A2: Address of something, but we do not know what ✓ *Full Credit*



Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds what value?
```

Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds what value?
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (*i.e.*, “garbage value”)

Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (*i.e.*, “garbage value”)

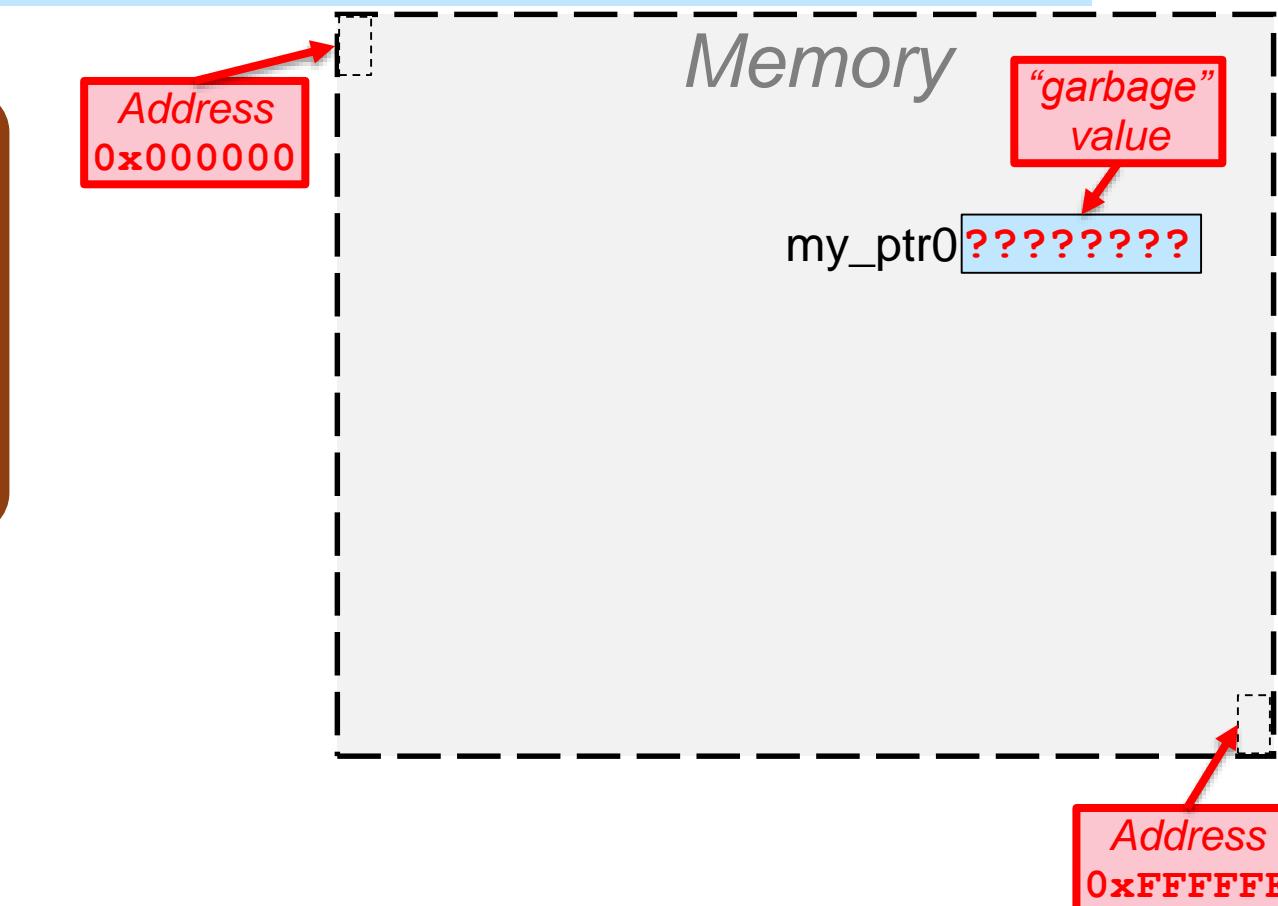
Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., “garbage value”)



Using An Address

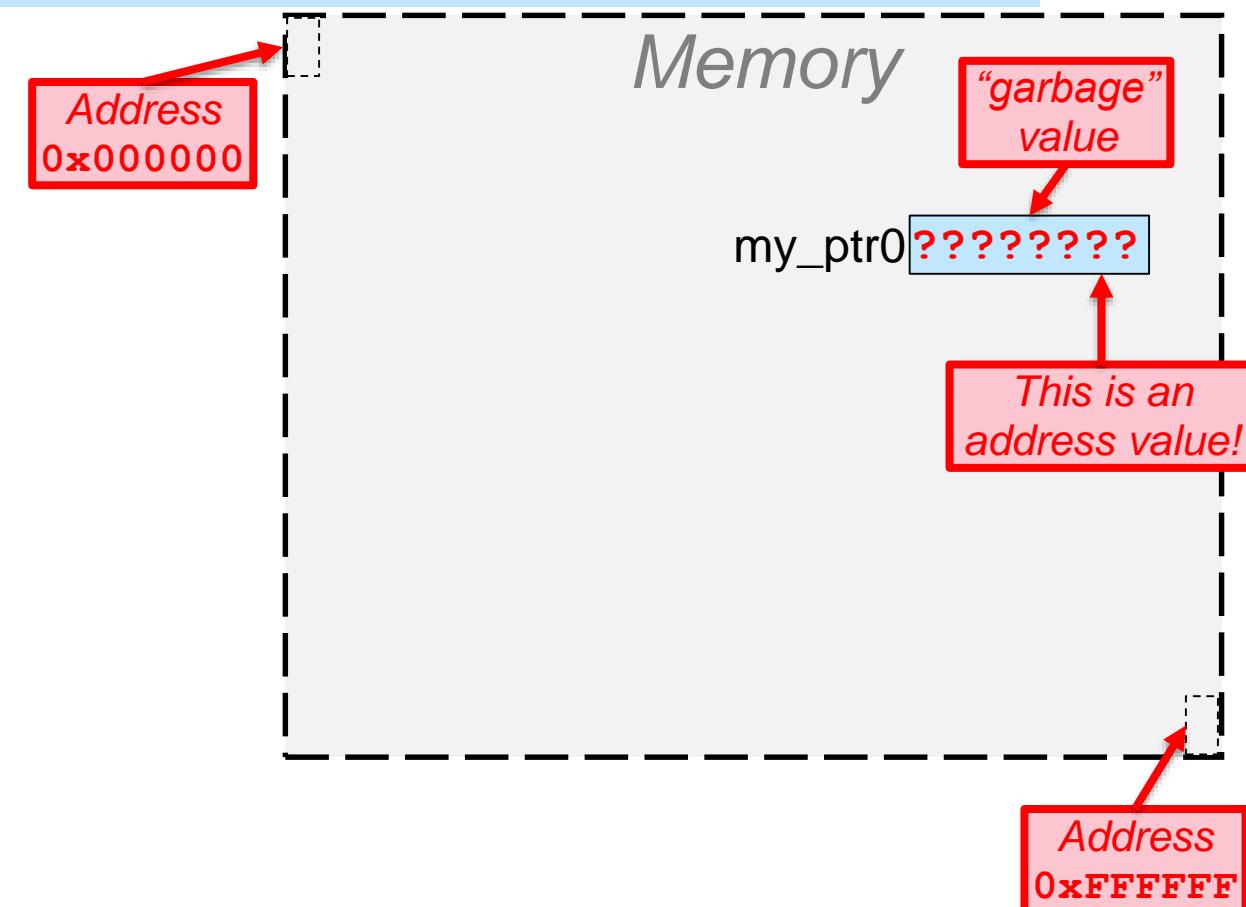
RECALL: A pointer holds an address

```
void* my_ptr0;
```

// holds "garbage" value

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., "garbage value")



Using An Address

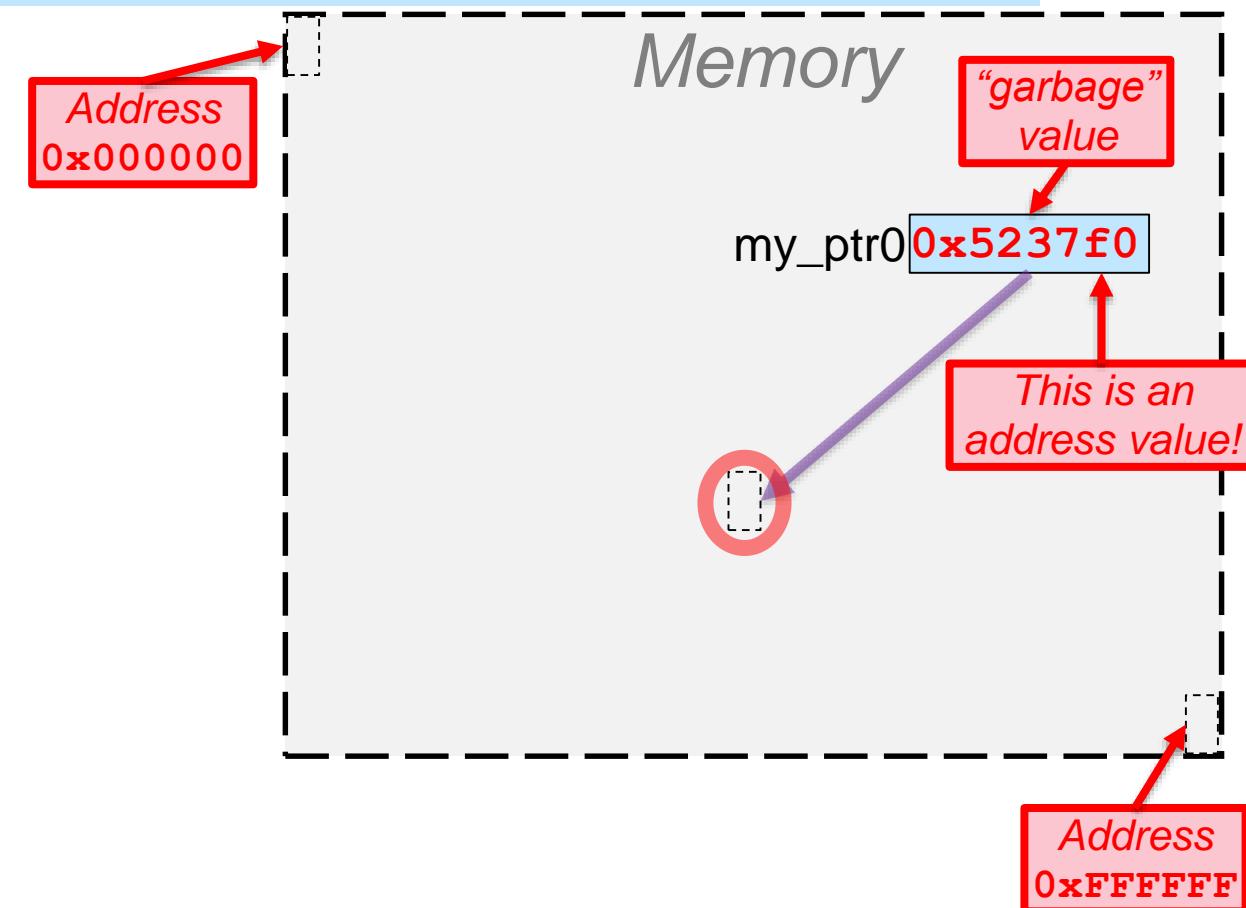
RECALL: A pointer holds an address

```
void* my_ptr0;
```

// holds "garbage" value

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., "garbage value")



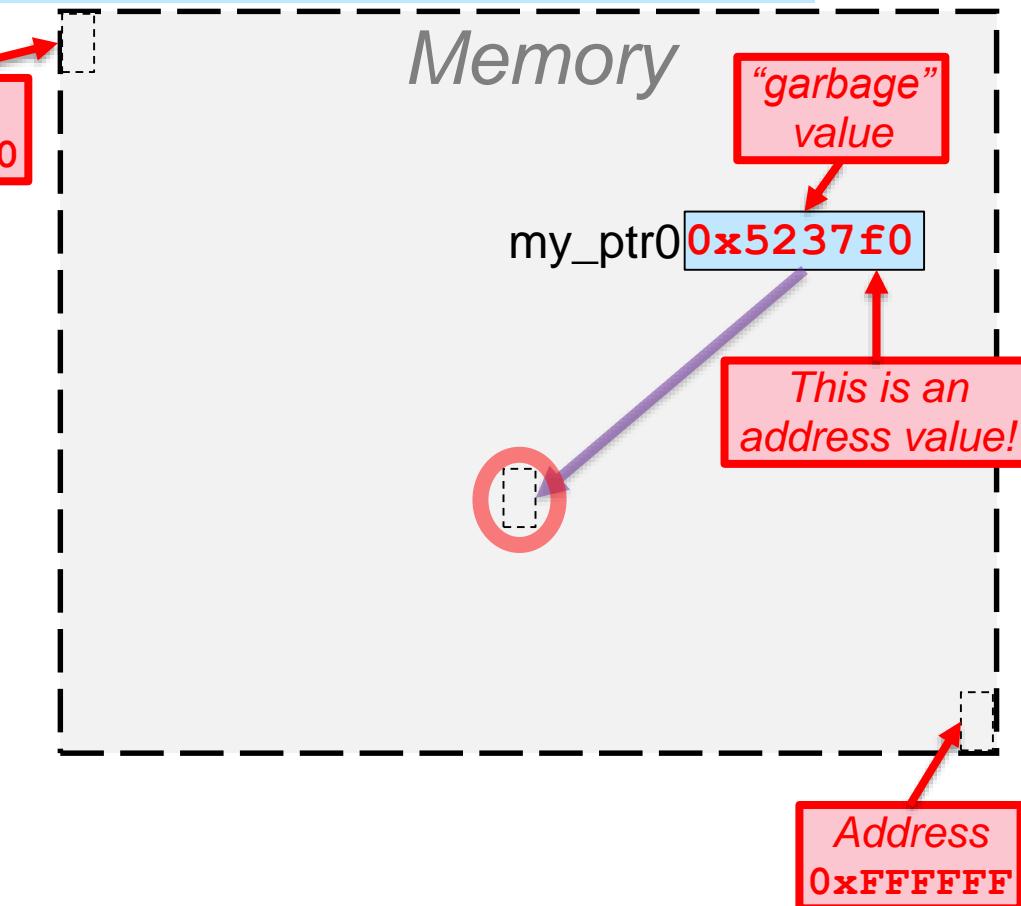
Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value  
void* my_ptr1 = nullptr; // holds "null" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., “garbage value”)



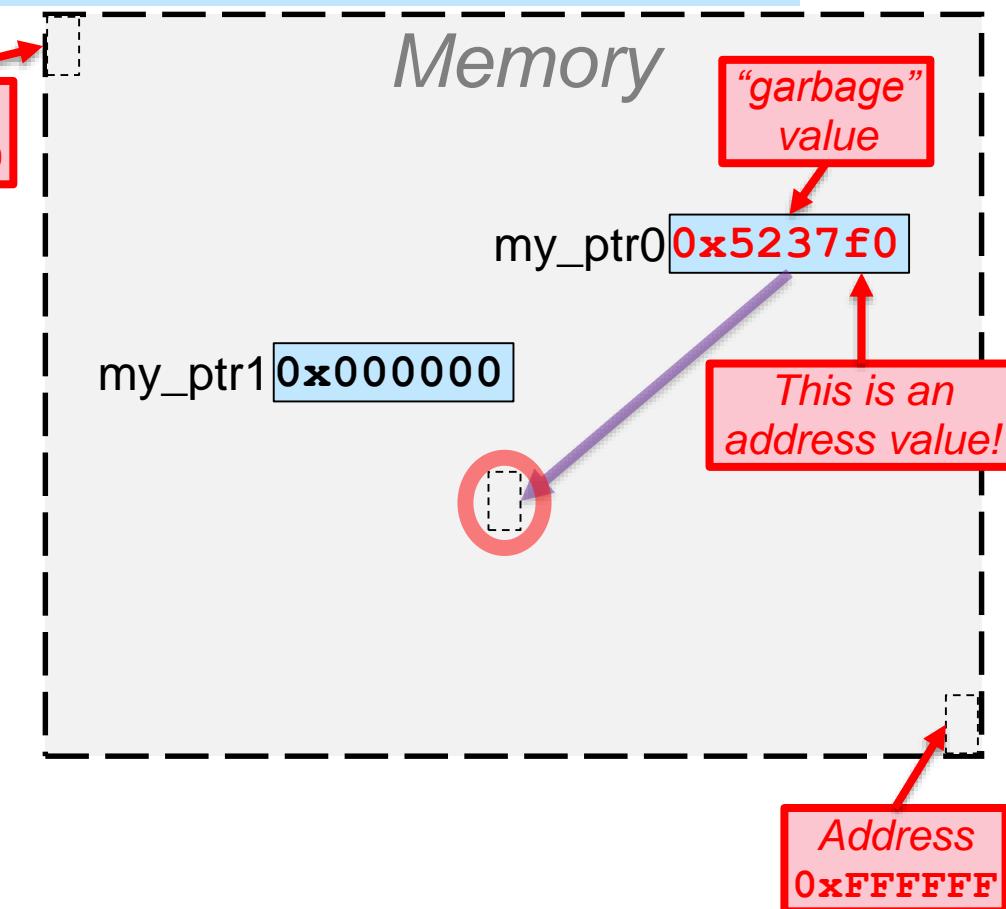
Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value  
void* my_ptr1 = nullptr; // holds "null" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., “garbage value”)



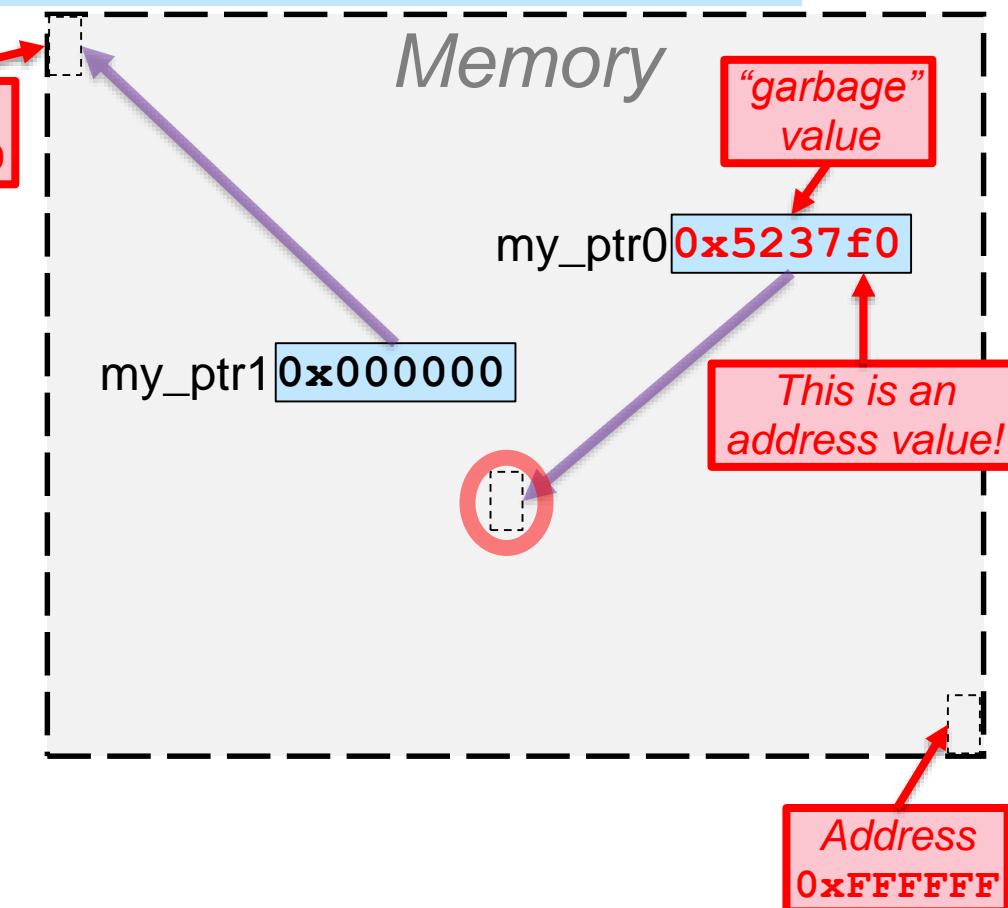
Using An Address

RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value  
void* my_ptr1 = nullptr; // holds "null" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., “garbage value”)



Using An Address

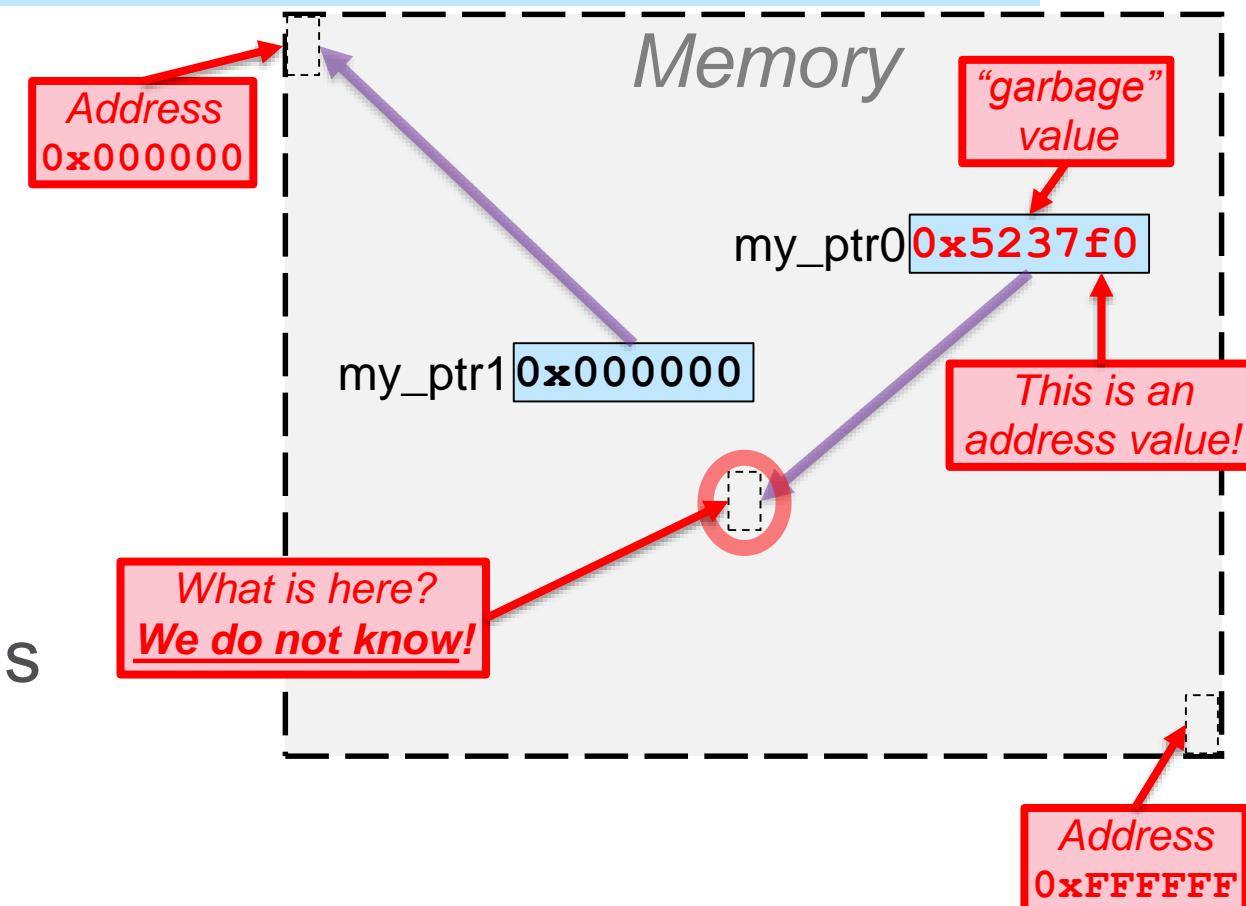
RECALL: A pointer holds an address

```
void* my_ptr0; // holds "garbage" value  
void* my_ptr1 = nullptr; // holds "null" value
```

A data object always has a value, which is one of:

1. The value you assigned
2. (*If you did not assign a value*), the bit-pattern at that location before the data object was created (i.e., “garbage value”)

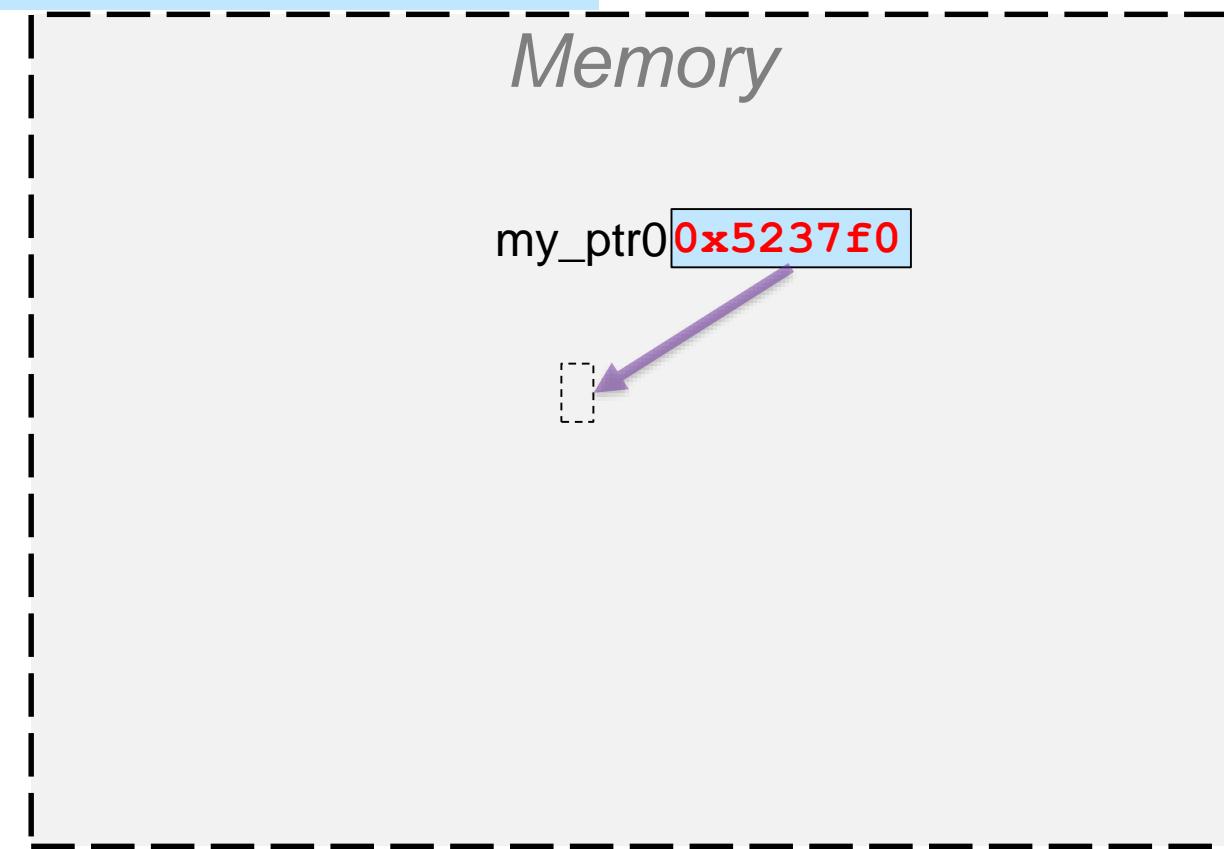
- It would be nice to know “what” is at the address stored in `my_ptr0`



Knowing What Is At An Address

```
void* my_ptr0; // pointer to ??
```

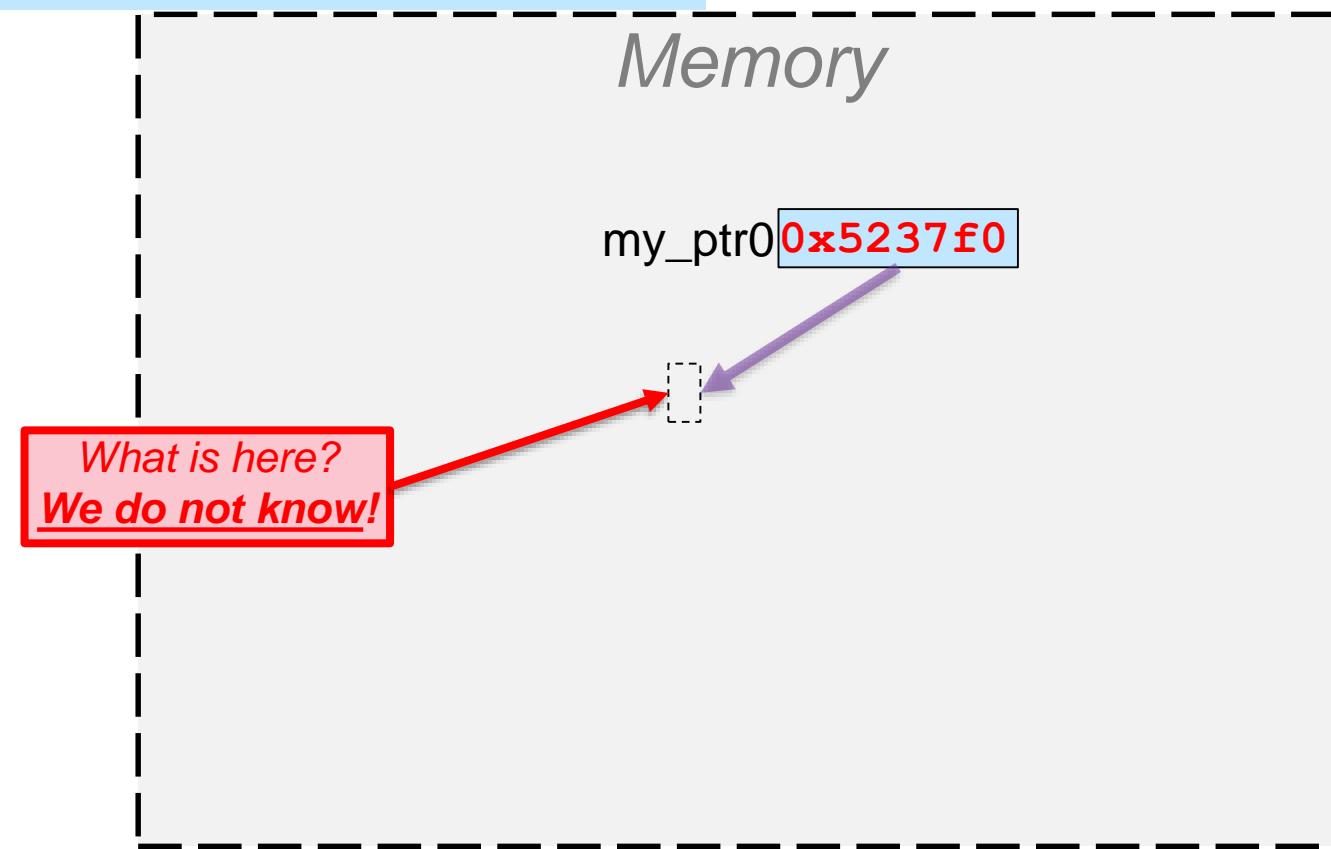
RECALL: A pointer holds an address



Knowing What Is At An Address

```
void* my_ptr0; // pointer to ??
```

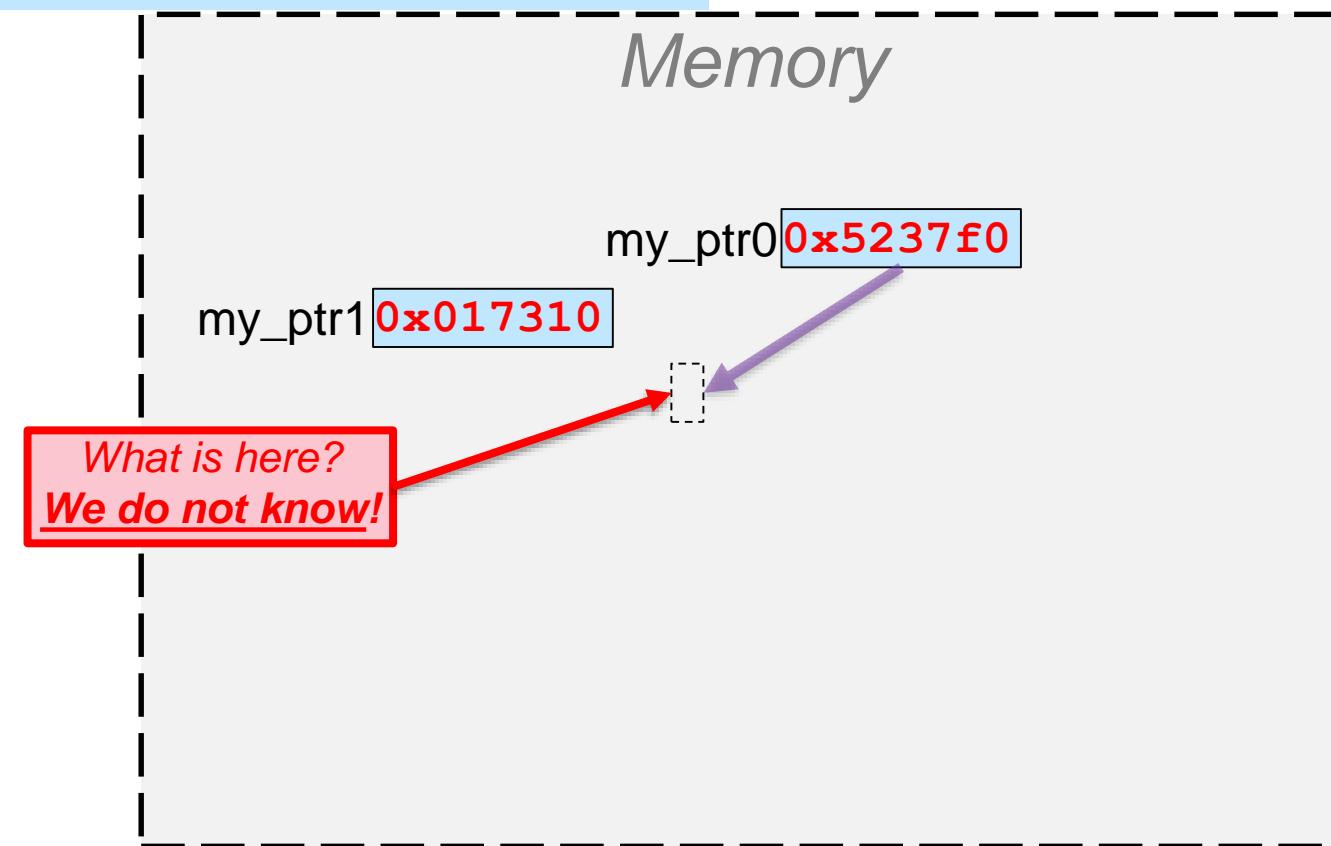
RECALL: A pointer holds an address



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int*  my_ptr1;           // pointer to int
```

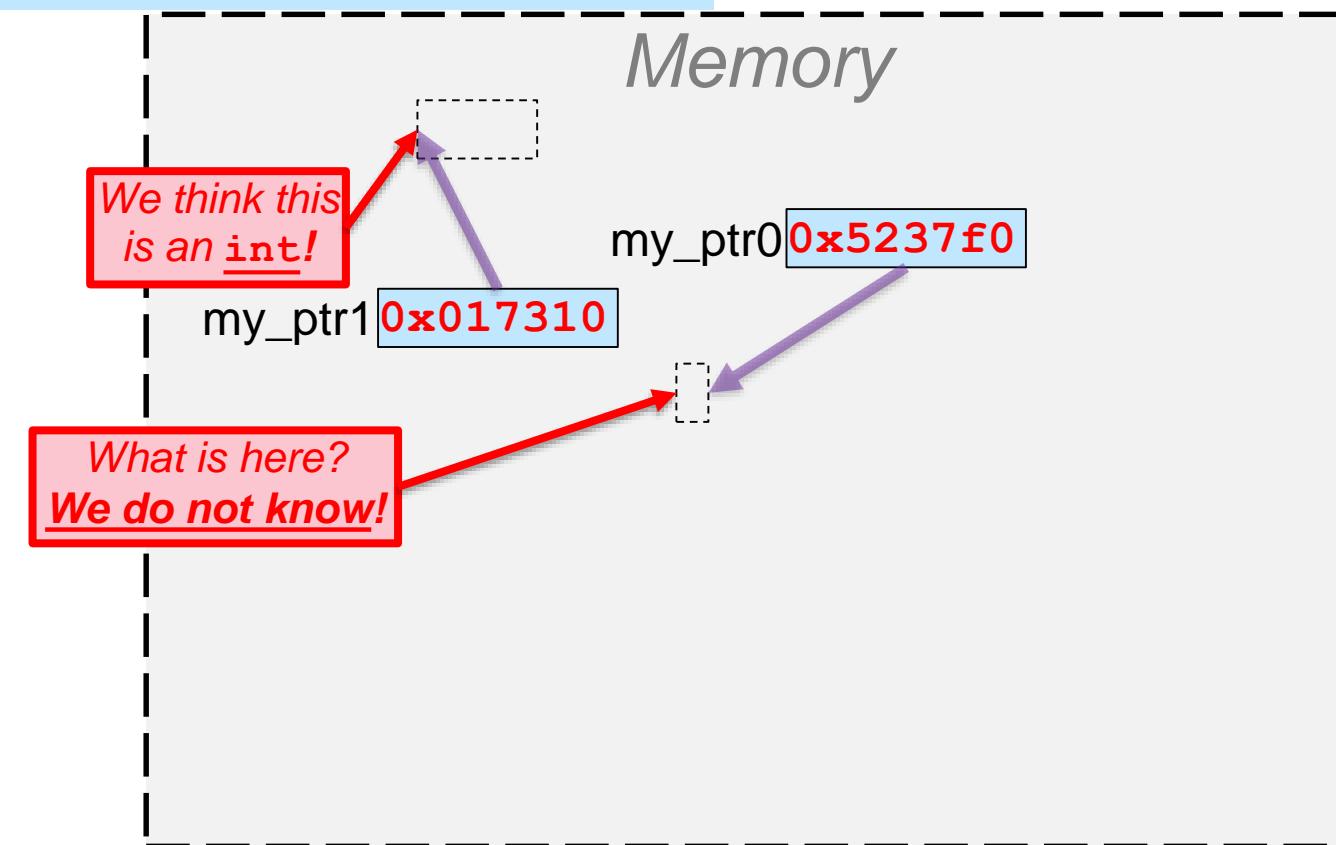
RECALL: A pointer holds an address



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int*  my_ptr1;           // pointer to int
```

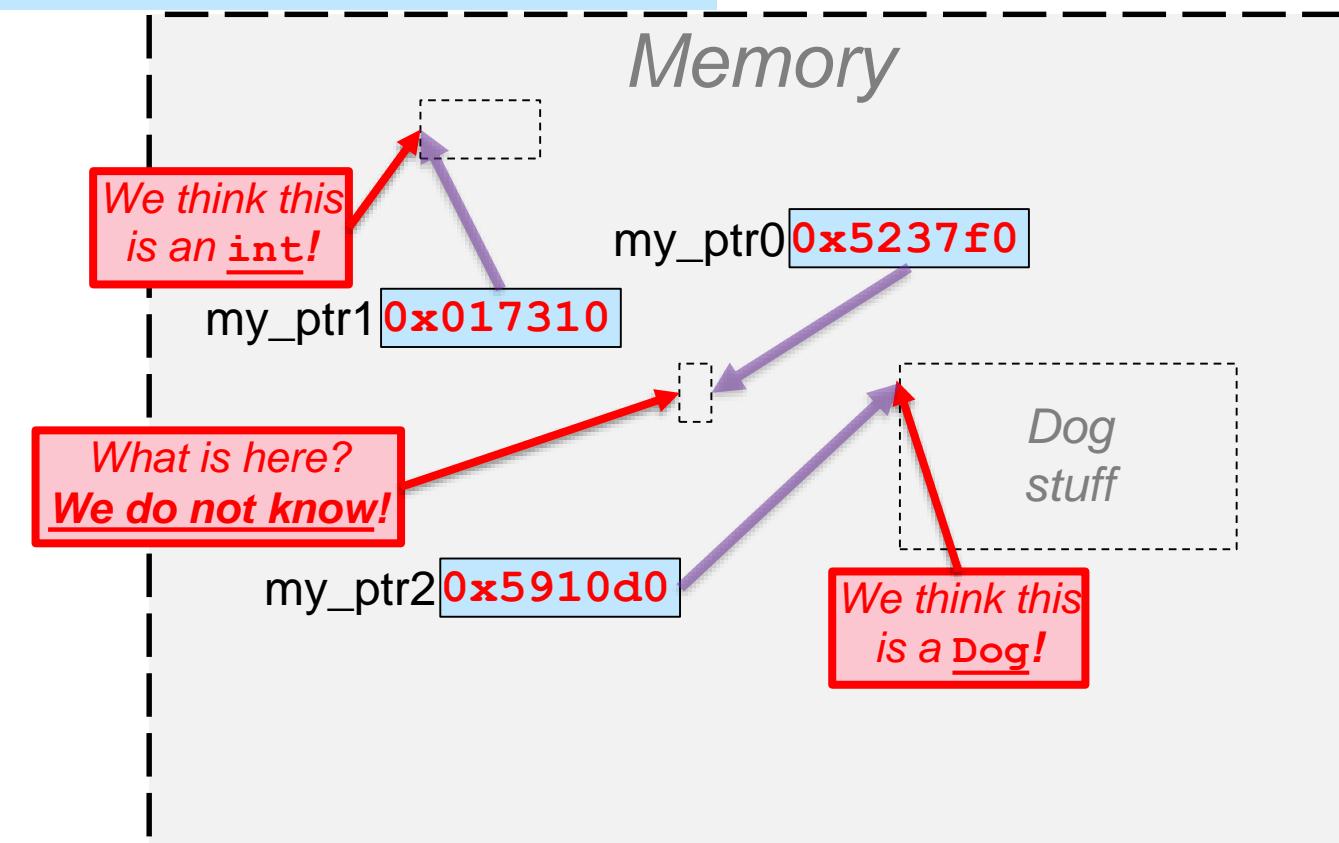
RECALL: A pointer holds an address



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int*  my_ptr1;           // pointer to int  
Dog*  my_ptr2;           // pointer to Dog
```

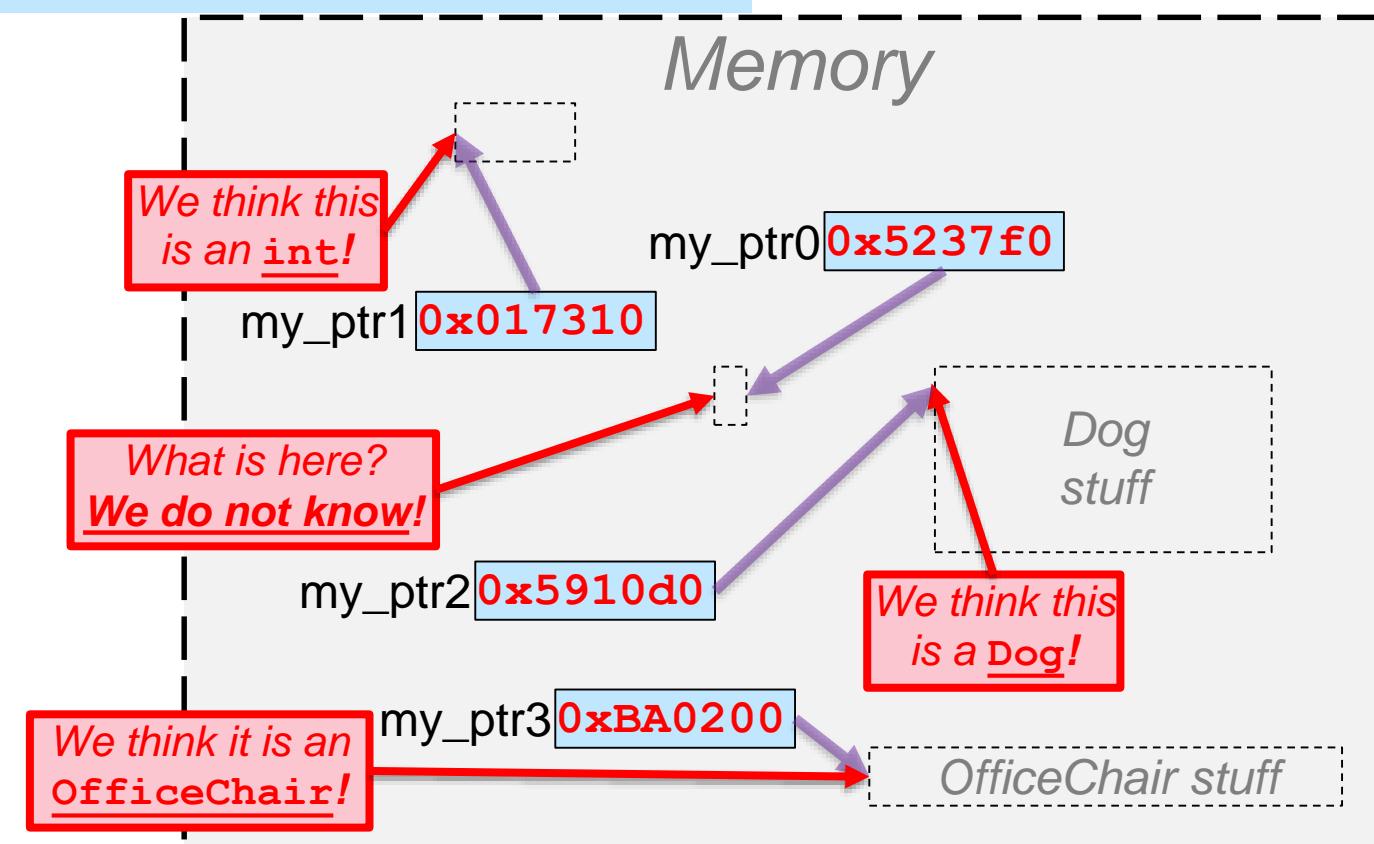
RECALL: A pointer holds an address



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

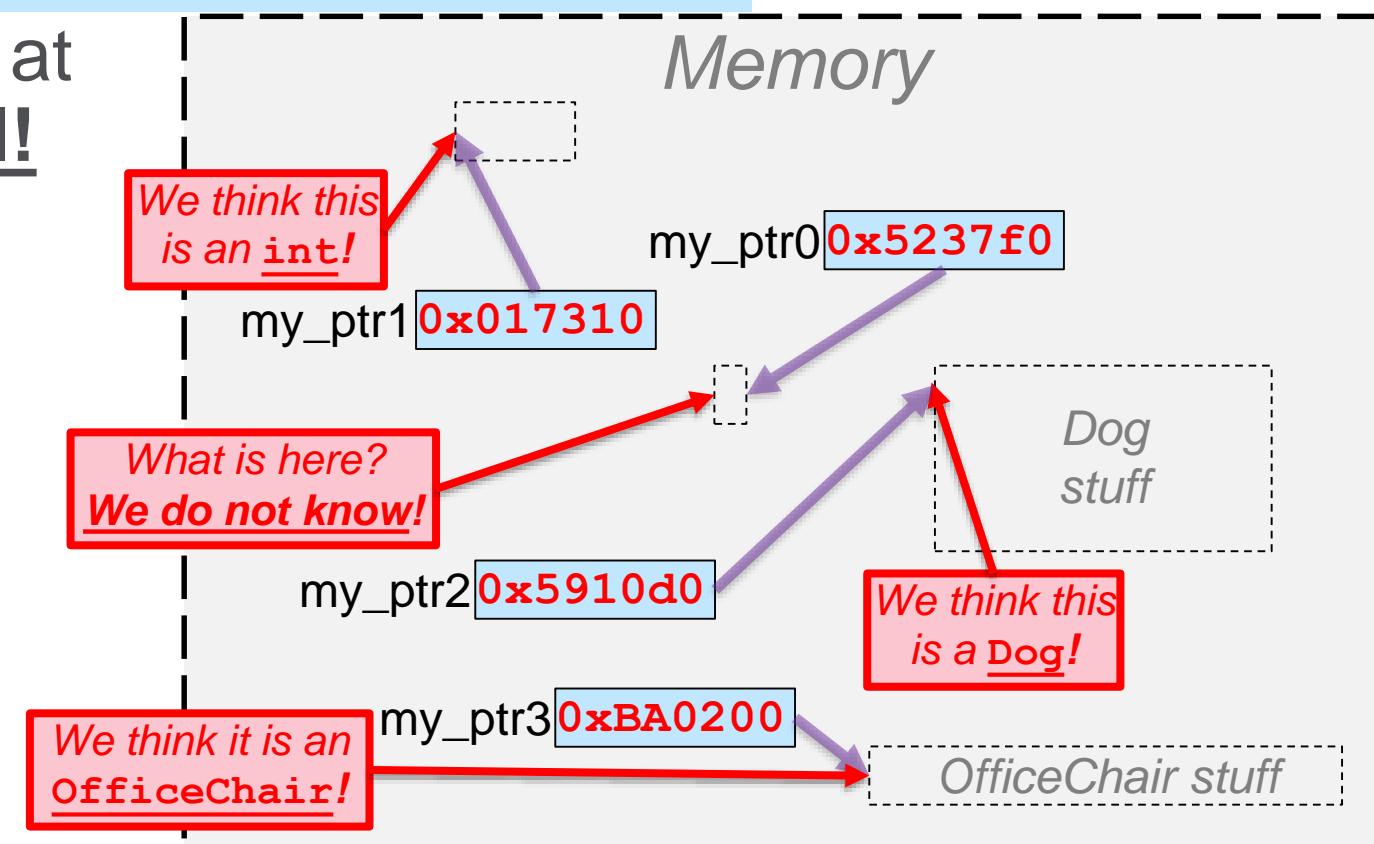


Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

- Knowing the “type” of object at the target-address is powerful!

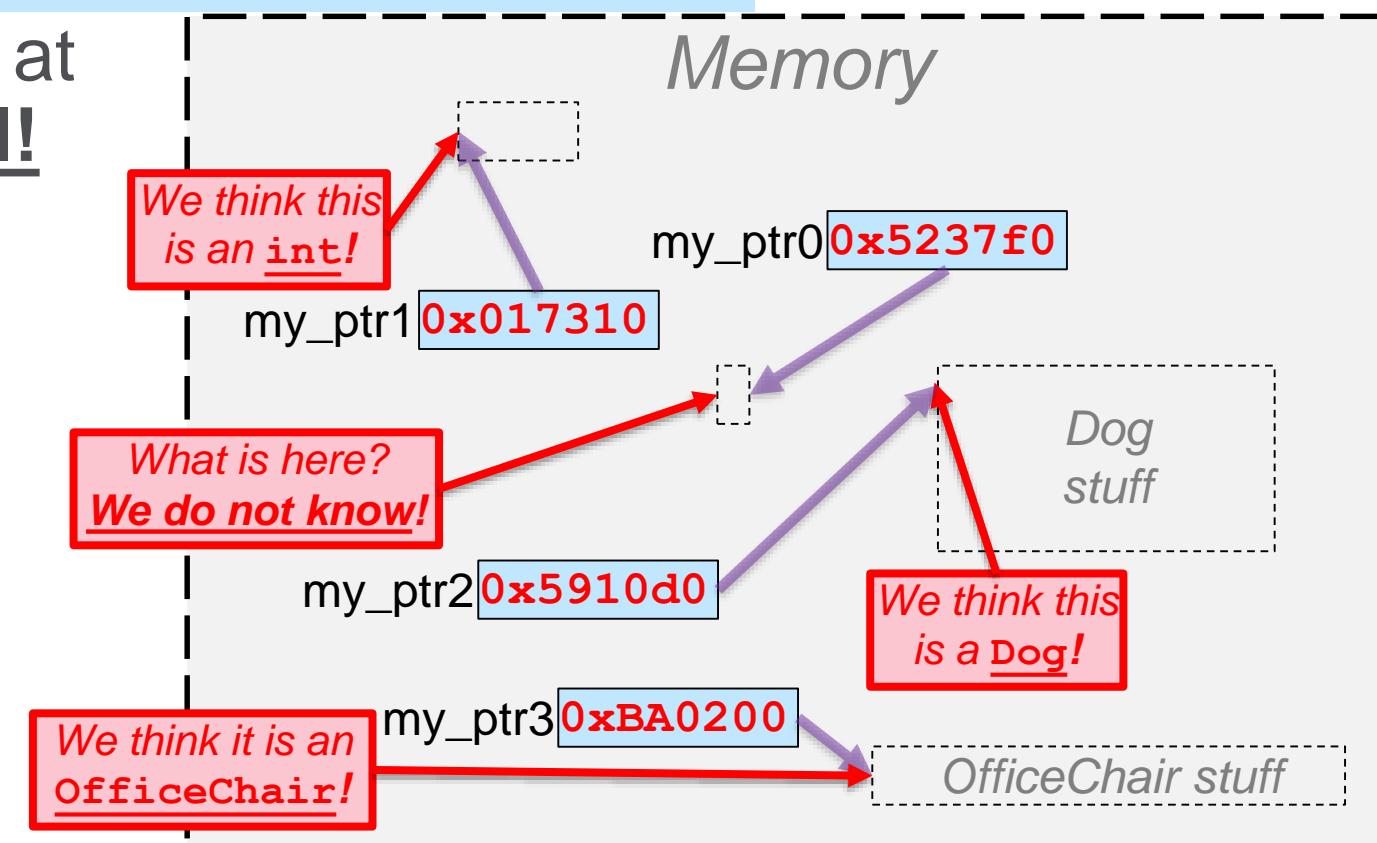


Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

- Knowing the “type” of object at the target-address is powerful!
- Type information provides:

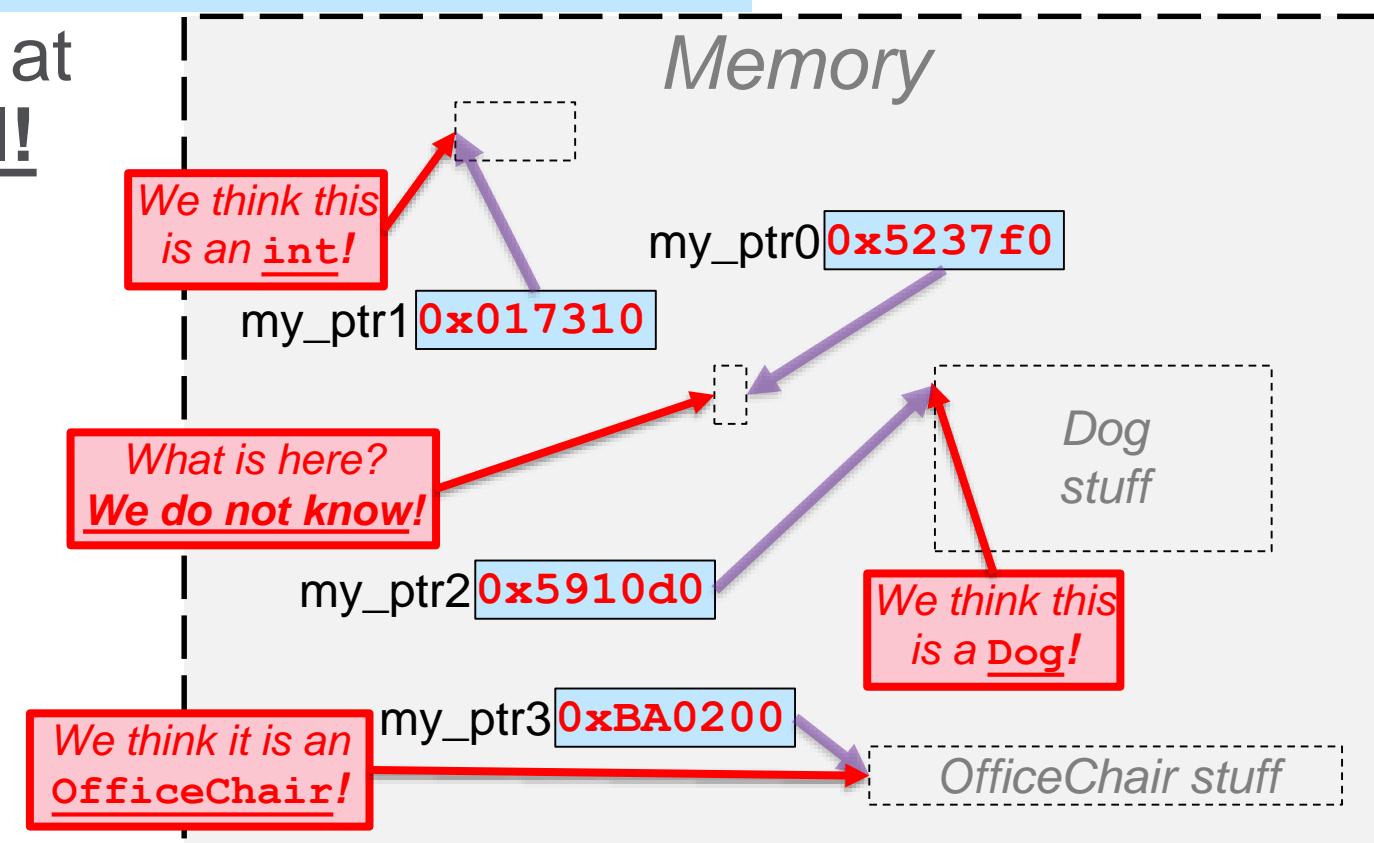


Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

- Knowing the “type” of object at the target-address is powerful!
- Type information provides:
 1. How big it is
(target object size)



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

- Knowing the “type” of object at the target-address is powerful!

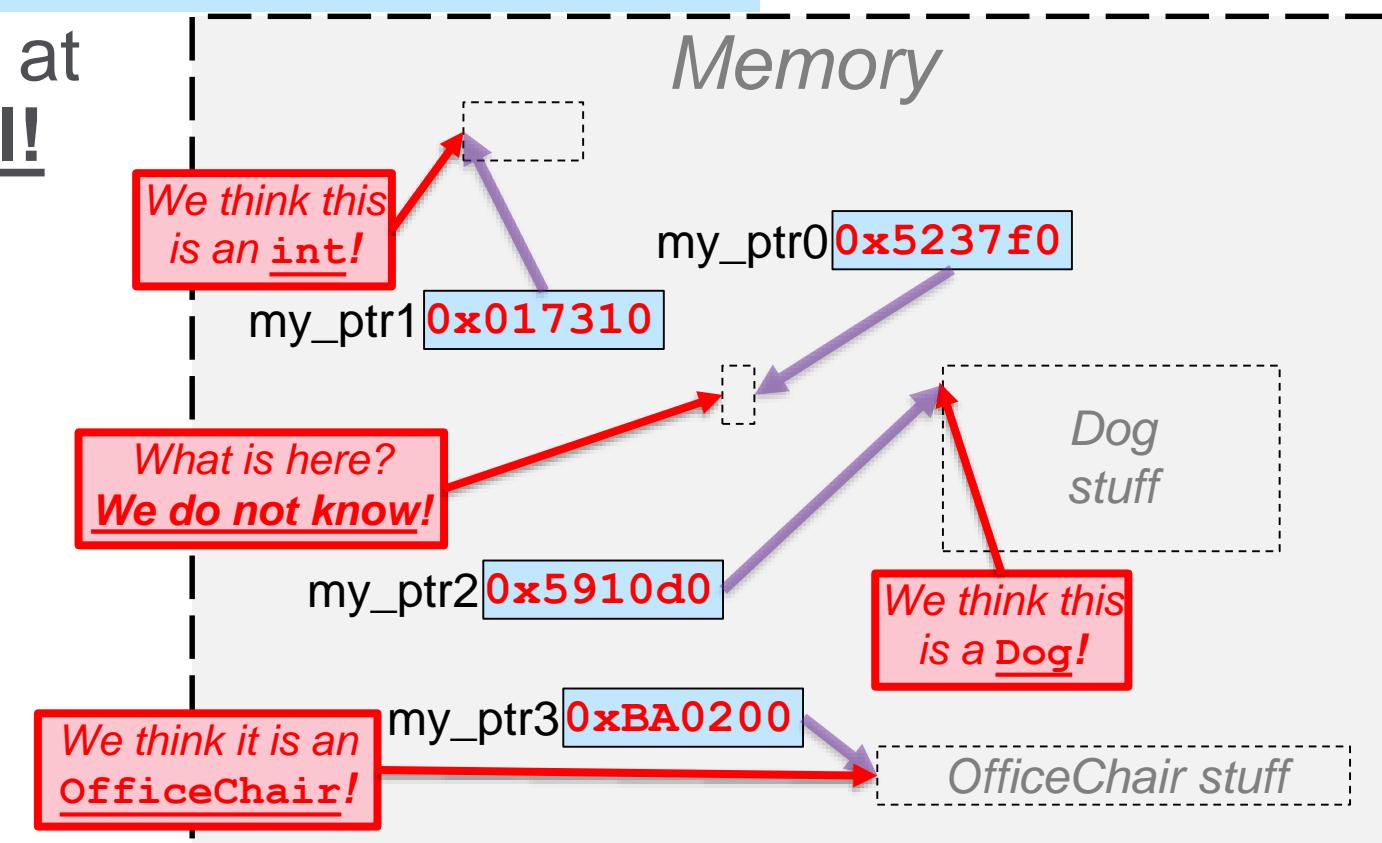
- Type information provides:

1. How big it is

(target object size)

2. What's in there

(what nested fields are present)



Knowing What Is At An Address

```
void* my_ptr0;           // pointer to ??  
int* my_ptr1;           // pointer to int  
Dog* my_ptr2;           // pointer to Dog  
OfficeChair* my_ptr3;   // pointer to OfficeChair
```

RECALL: A pointer holds an address

- Knowing the “type” of object at the target-address is powerful!

- Type information provides:

1. How big it is

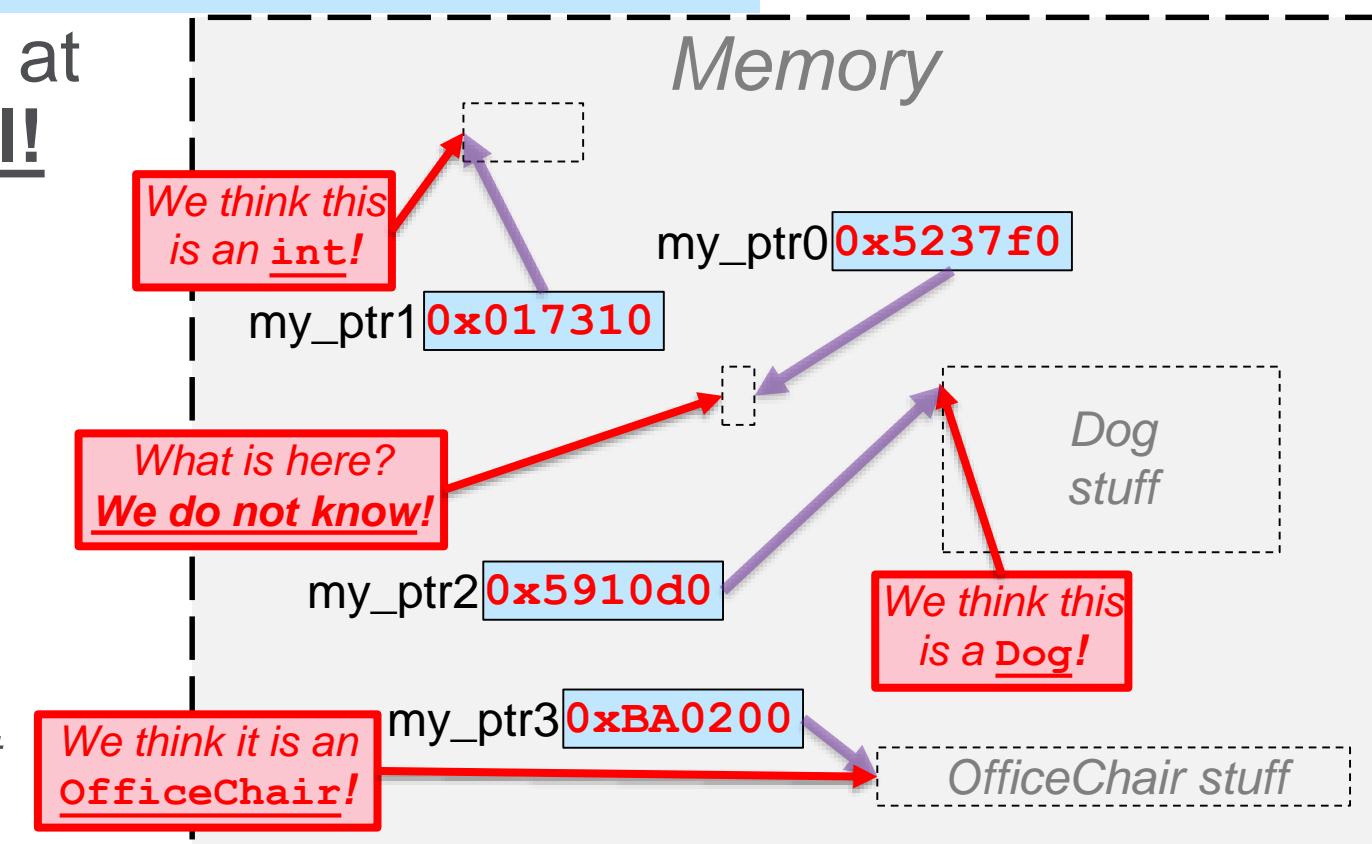
(target object size)

2. What's in there

(what nested fields are present)

3. What we can do with it

(math, relational expressions, extract properties, etc.)



Initializing Your Pointer

- “The Address Of” operator: &

RECALL: A pointer holds an address

Initializing Your Pointer

- “The Address Of” operator: &

```
int my_int = 42;
```

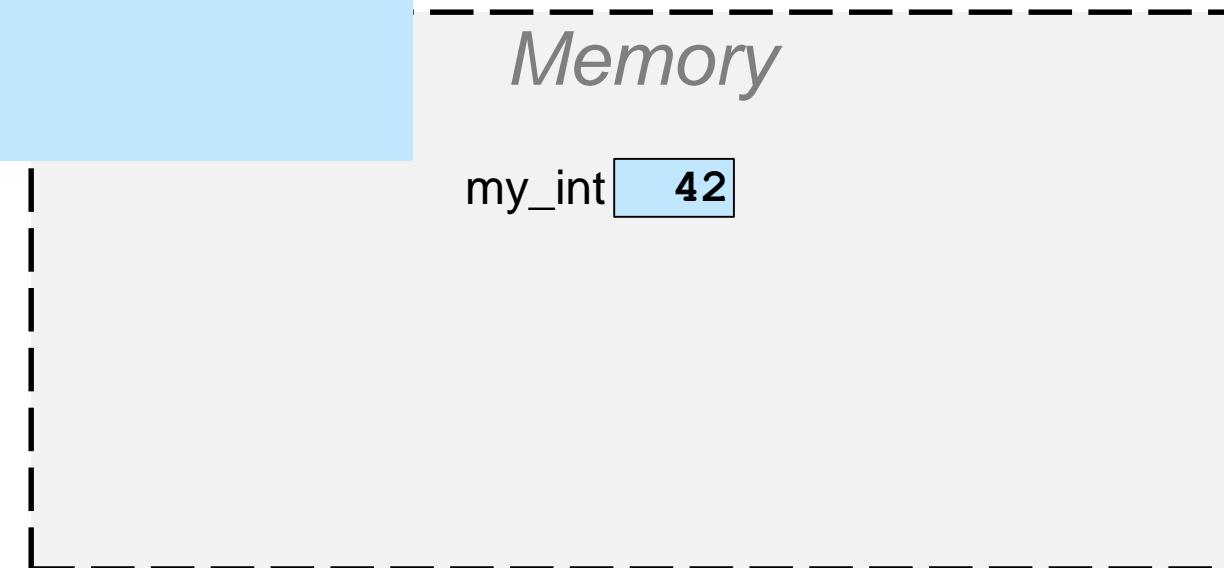
RECALL: A pointer holds an address

Initializing Your Pointer

- “The Address Of” operator: &

```
int my_int = 42;
```

RECALL: A pointer holds an address

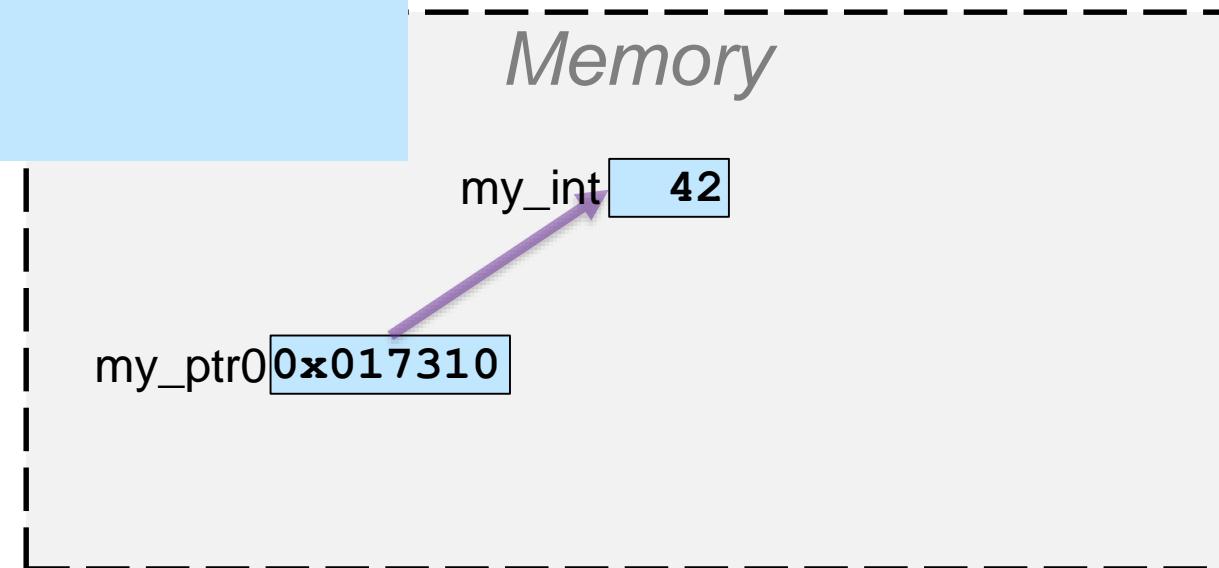


Initializing Your Pointer

- “The Address Of” operator: &

```
int my_int = 42;  
int* my_ptr0 = &my_int; // pointer to int
```

RECALL: A pointer holds an address

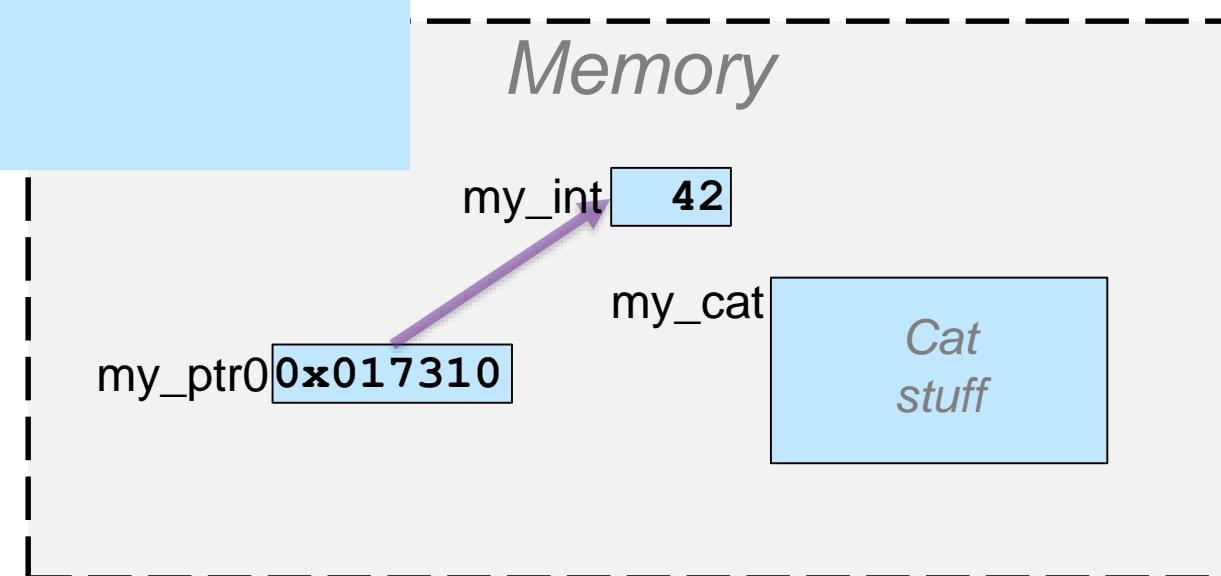


Initializing Your Pointer

- “The Address Of” operator: &

```
int my_int = 42;  
int* my_ptr0 = &my_int; // pointer to int  
Cat my_cat;
```

RECALL: A pointer holds an address

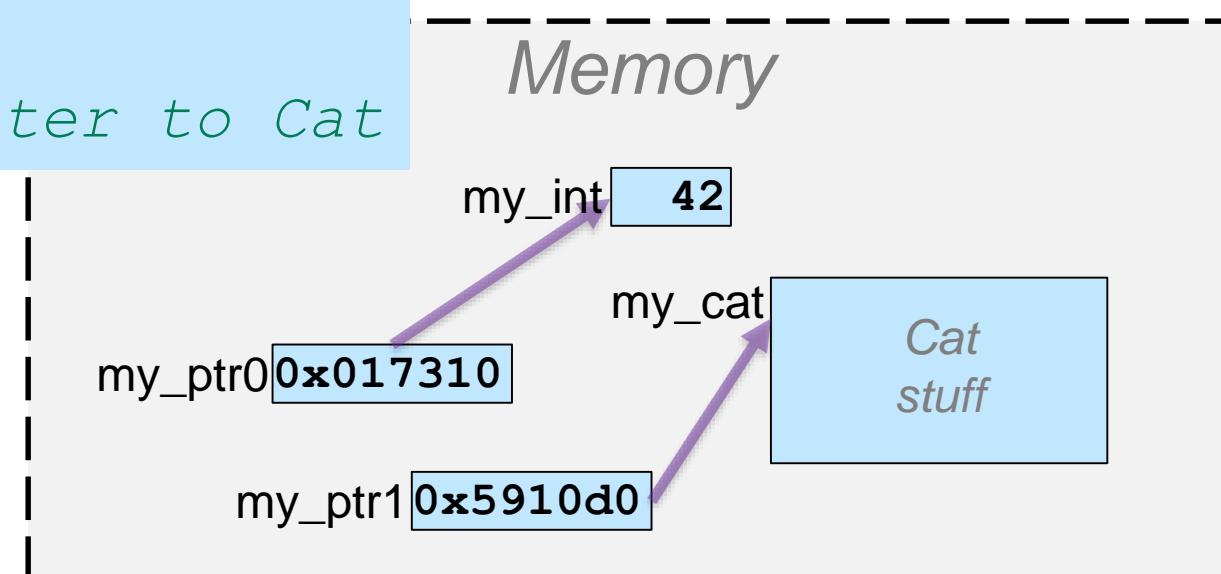


Initializing Your Pointer

- “The Address Of” operator: &

```
int my_int = 42;  
int* my_ptr0 = &my_int; // pointer to int  
Cat my_cat;  
Cat* my_ptr2 = &my_cat; // pointer to Cat
```

RECALL: A pointer holds an address



Initializing Your Pointer

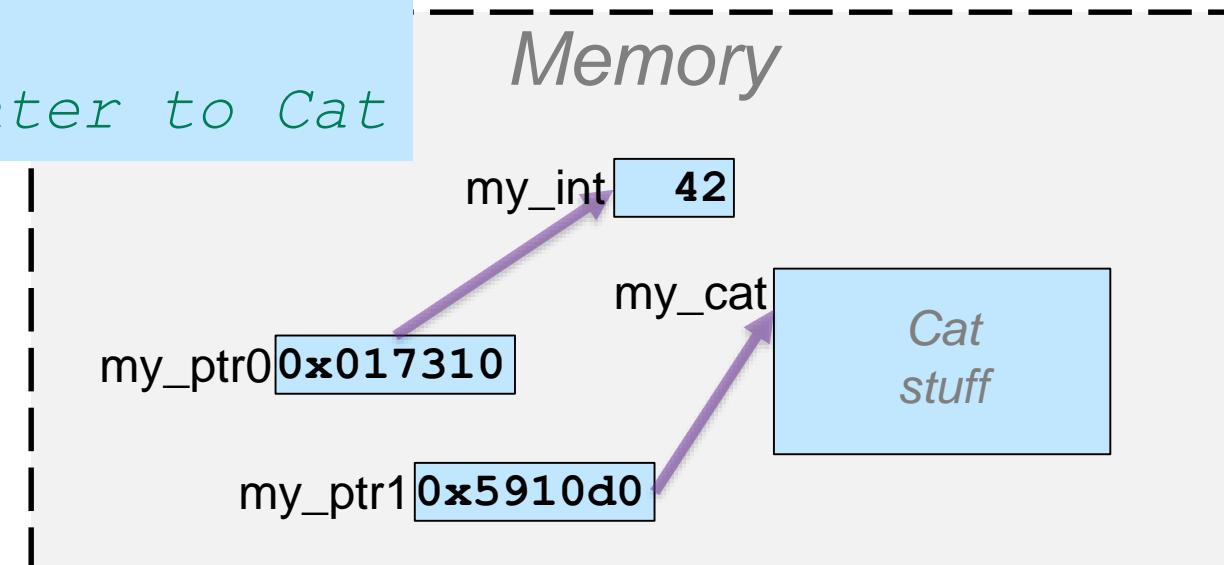
- “The Address Of” operator: &

```
int my_int = 42;  
int* my_ptr0 = &my_int; // pointer to int  
Cat my_cat;  
Cat* my_ptr2 = &my_cat; // pointer to Cat
```

- Now, we have *two ways* to work with the object:

- The object “name” identifies the object
- “Indirection”: A pointer identifies the object (*it holds the address of the object*)

RECALL: A pointer holds an address



Initializing Your Pointer

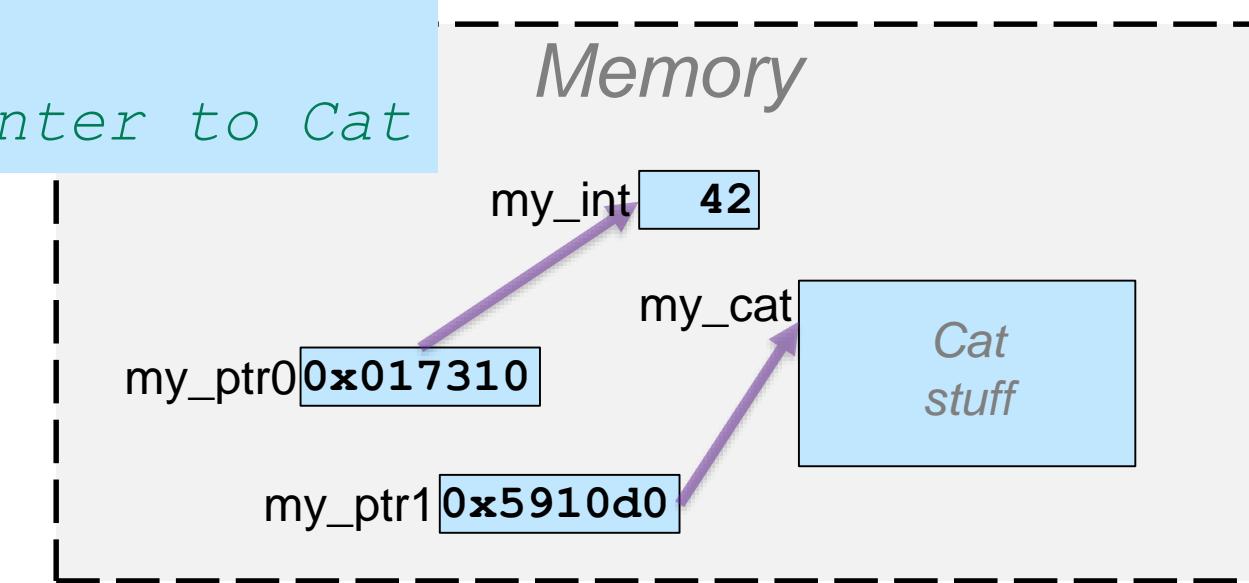
- “The Address Of” operator: &

```
int my_int = 42;  
int* my_ptr0 = &my_int; // pointer to int  
Cat my_cat;  
Cat* my_ptr2 = &my_cat; // pointer to Cat
```

- Now, we have *two ways* to work with the object:

- The object “name” identifies the object
- “Indirection”: A pointer identifies the object (*it holds the address of the object*)

RECALL: A pointer holds an address

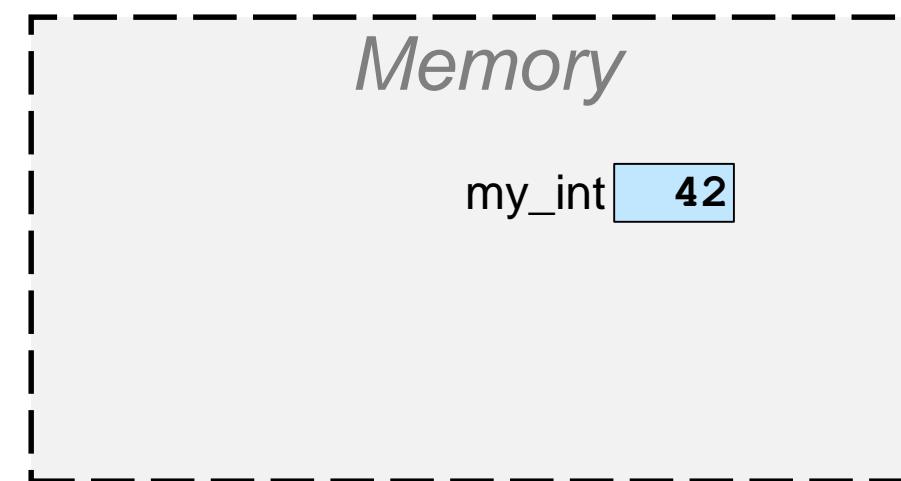


“Reference Semantics”:
Many Computer Science data structures
demand implementation through pointers

Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;
```

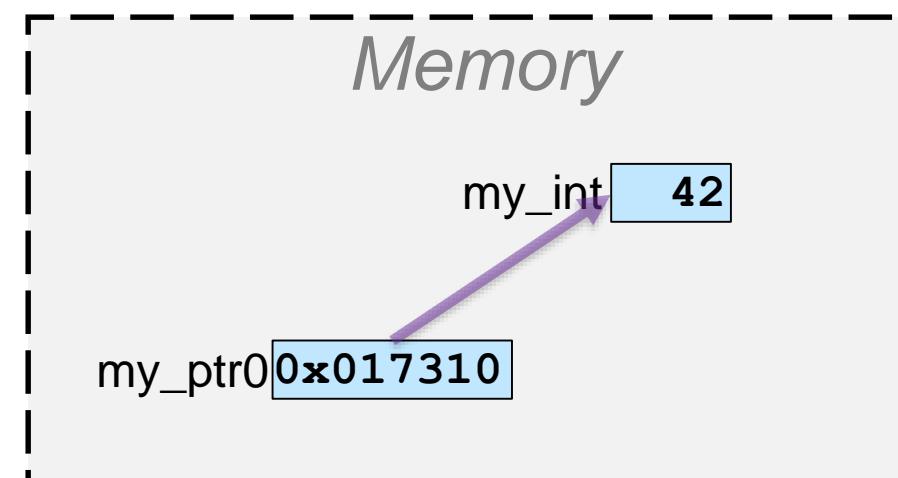


Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;
```

Reference Semantics: The pointer “references” the object at the target-address

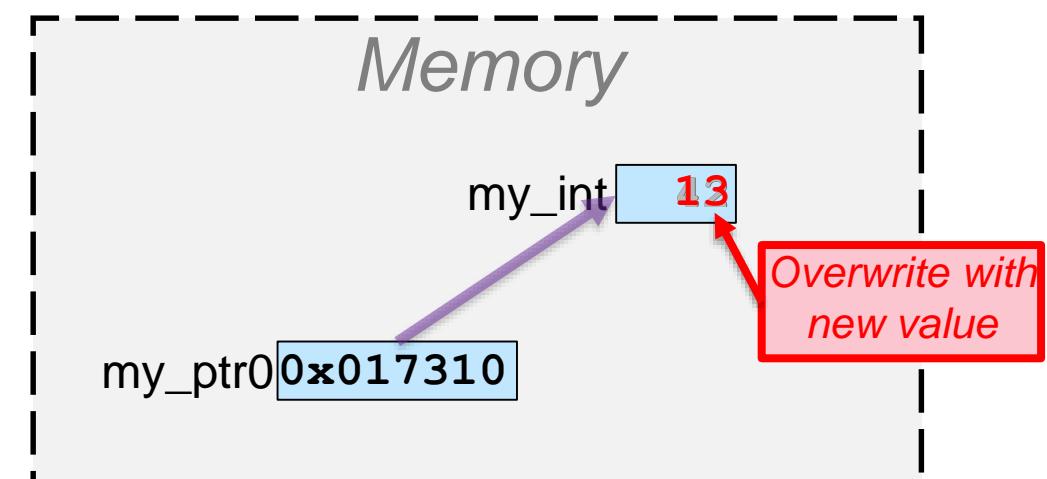


Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;
```

Reference Semantics: The pointer “references” the object at the target-address

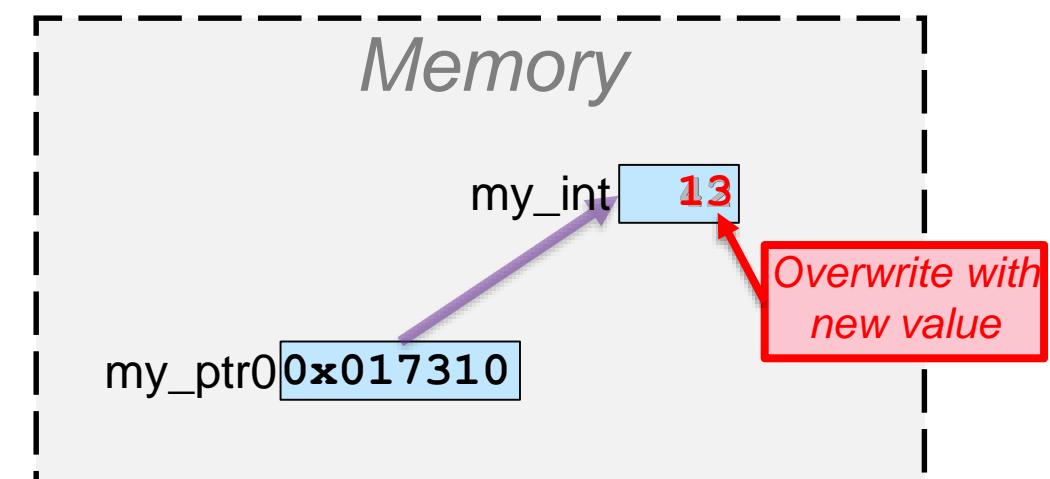


Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;  
*my_ptr0 = 13; // same thing
```

Reference Semantics: The pointer “references” the object at the target-address



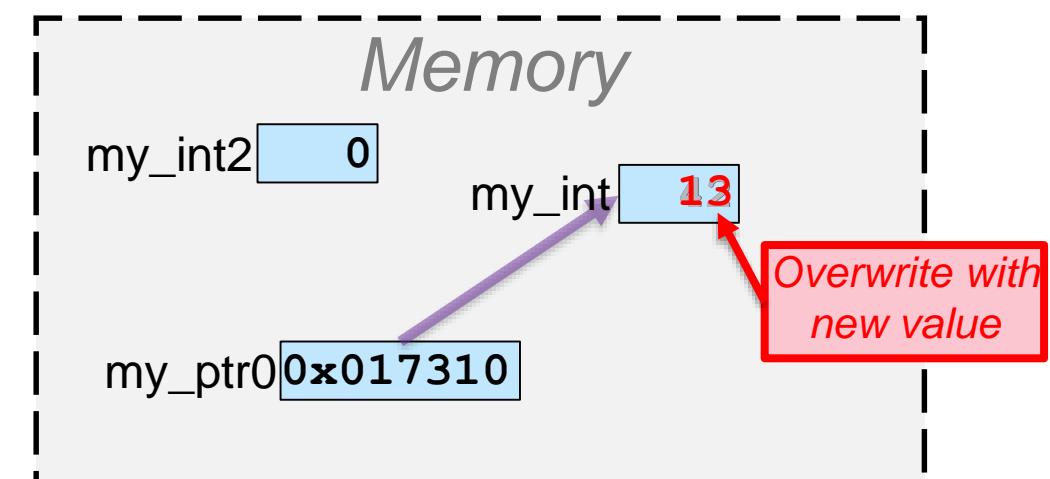
Indirection: “Going-to”
the target-address

Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;  
*my_ptr0 = 13; // same thing  
  
int my_int2 = 0;
```

Reference Semantics: The pointer “references” the object at the target-address



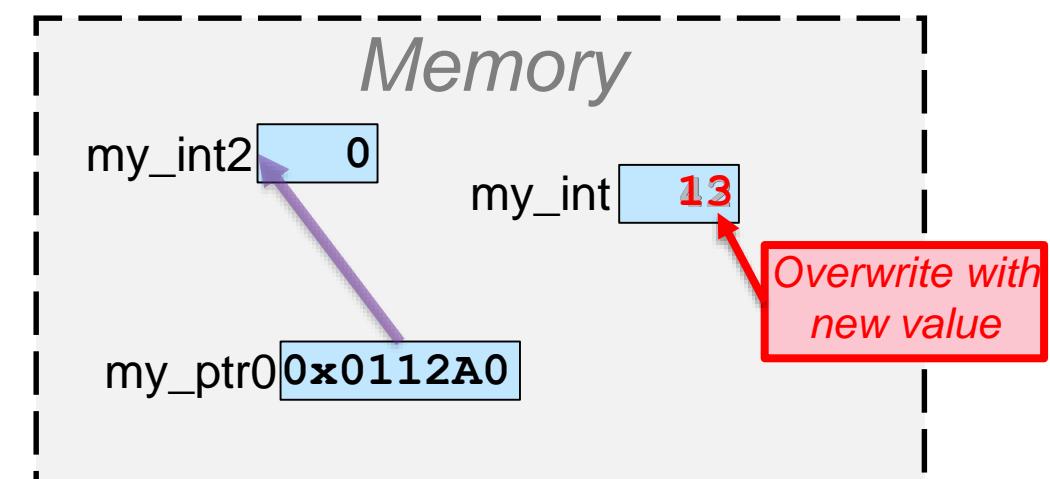
Indirection: “Going-to”
the target-address

Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;  
*my_ptr0 = 13; // same thing  
  
int my_int2 = 0;  
// Change pointer value  
my_ptr0 = &my_int2;
```

Reference Semantics: The pointer “references” the object at the target-address



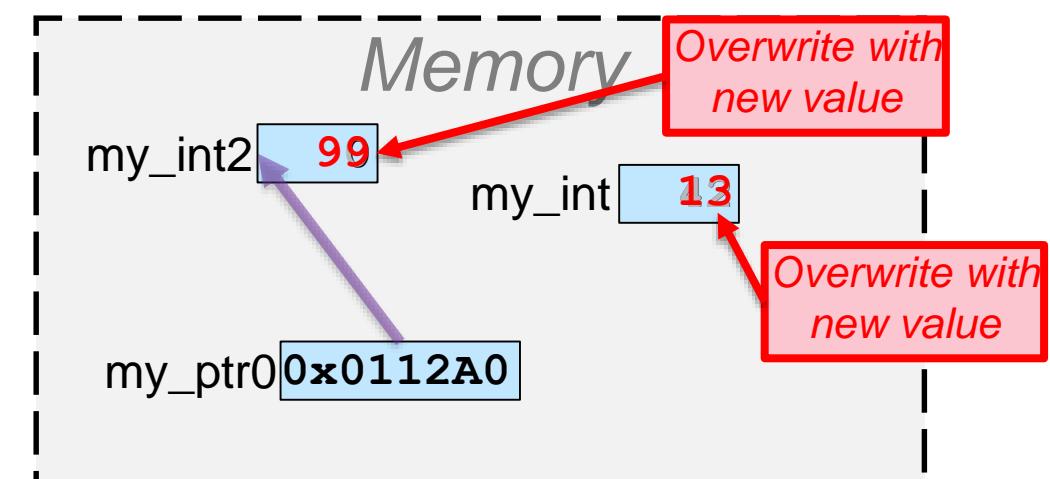
Indirection: “Going-to”
the target-address

Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;  
*my_ptr0 = 13; // same thing  
  
int my_int2 = 0;  
// Change pointer value  
my_ptr0 = &my_int2;  
  
my_int2 = 99;
```

Reference Semantics: The pointer “references” the object at the target-address



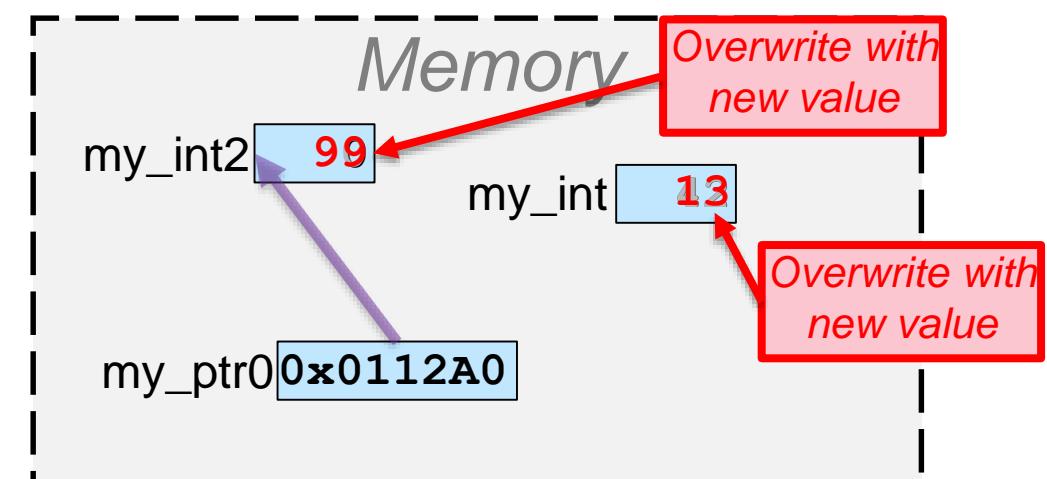
Indirection: “Going-to”
the target-address

Indirection: Using Your Pointer

- “The Indirection” operator: *

```
int my_int = 42;  
int* my_ptr0 = &my_int;  
  
my_int = 13;  
*my_ptr0 = 13; // same thing  
  
int my_int2 = 0;  
// Change pointer value  
my_ptr0 = &my_int2;  
  
my_int2 = 99;  
*my_ptr0 = 99; // same thing
```

Reference Semantics: The pointer “references” the object at the target-address



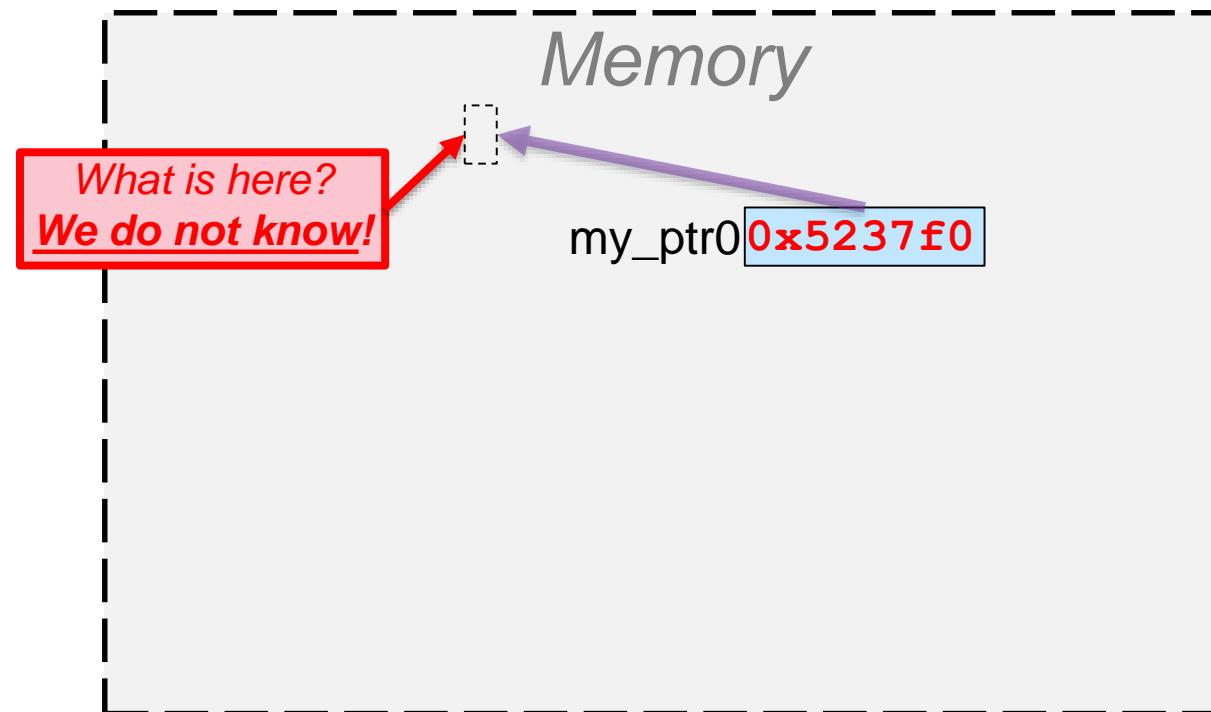
Indirection: “Going-to”
the target-address

The “Rogue Pointer”

```
void*
```

```
my_ptr0; // pointer to ??
```

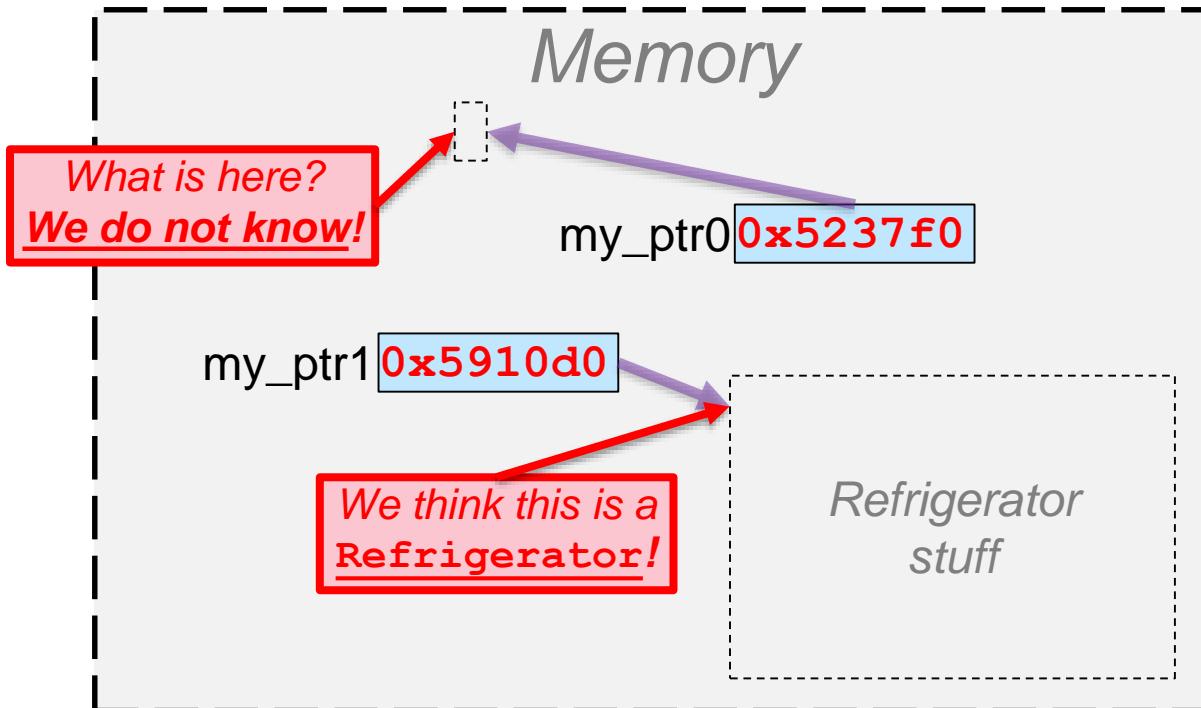
RECALL: A pointer holds an **address**



The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an **address**

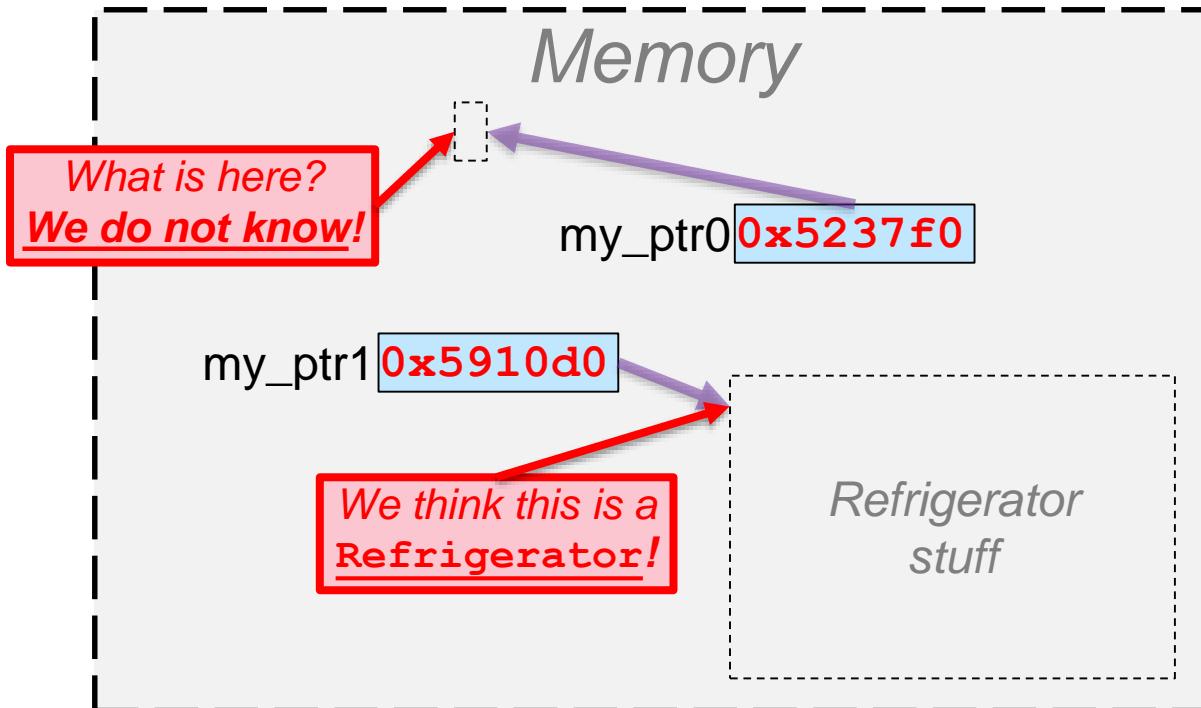


The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an **address**

1. A pointer holds an address

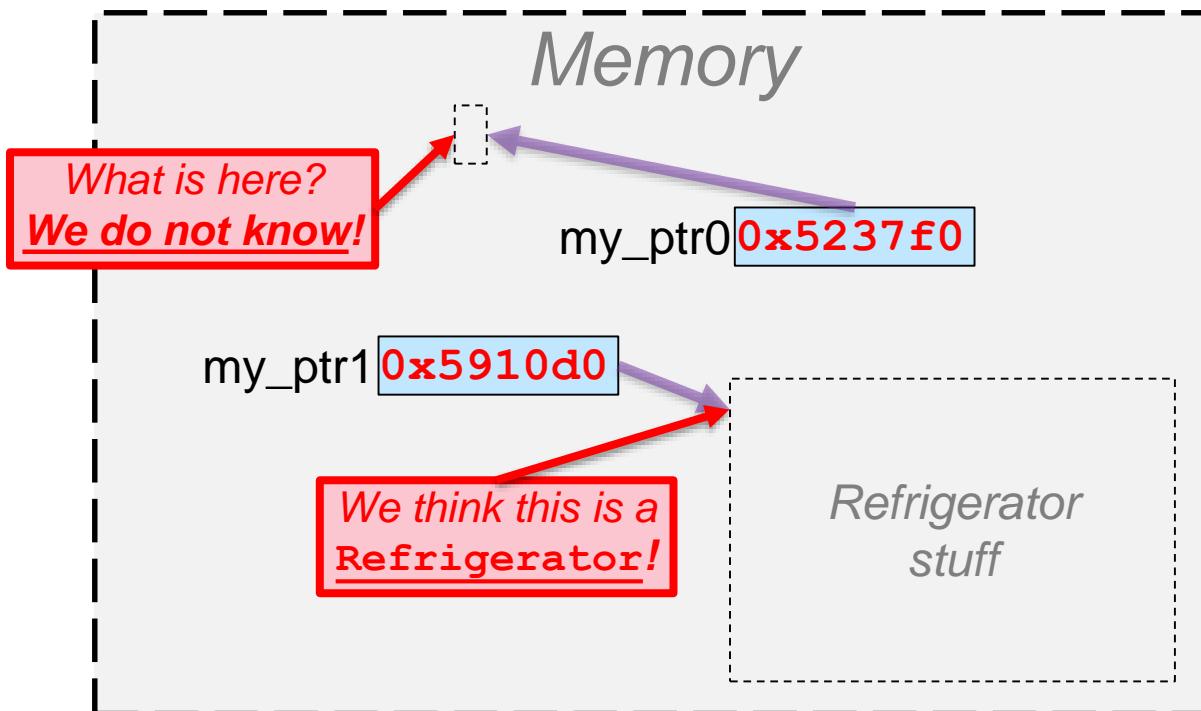


The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an **address**

1. A pointer holds an address
2. You create pointers

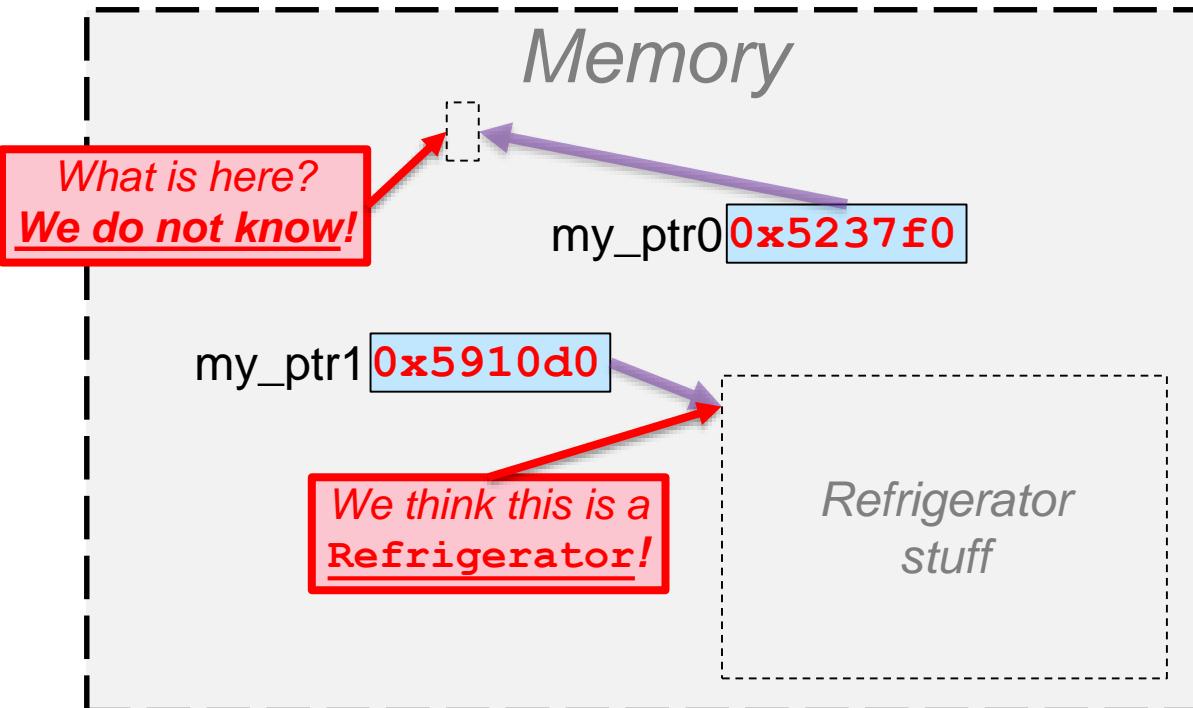


The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an **address**

1. A pointer holds an address
2. You create pointers
3. Uninitialized objects have “garbage” values

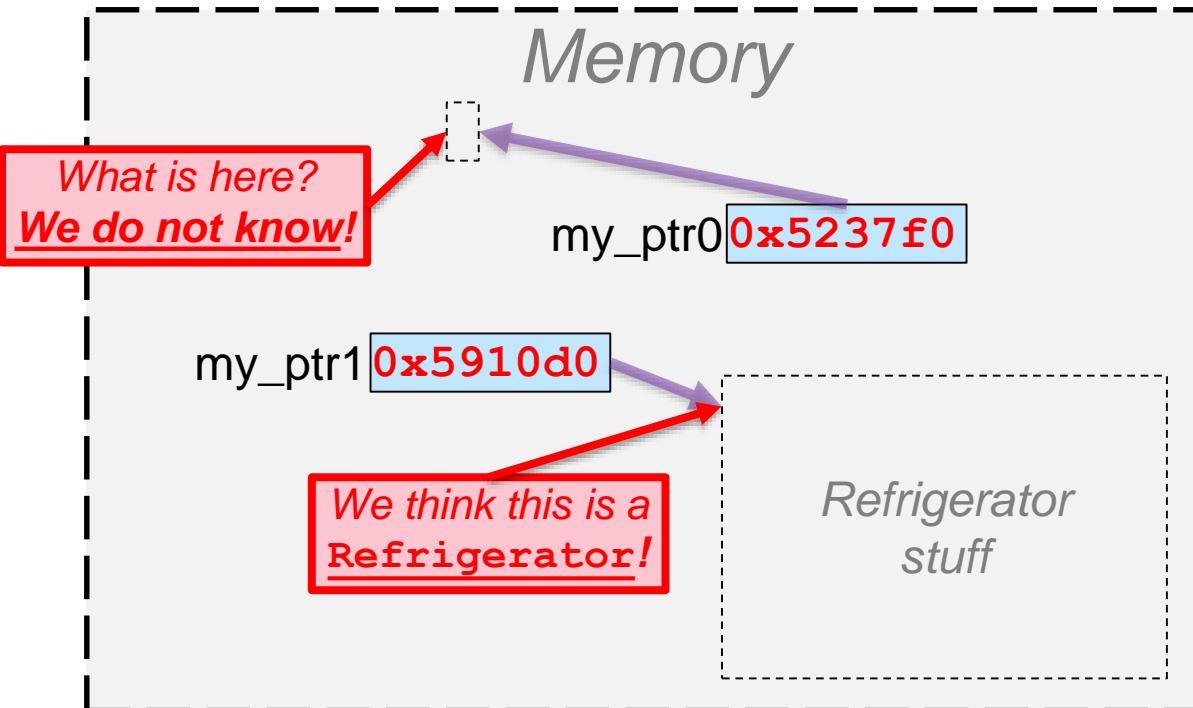


The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an address

1. A pointer holds an address
2. You create pointers
3. Uninitialized objects have “garbage” values
4. Your uninitialized pointer is “Rogue”
(it could point to anything, you cannot reason about it)

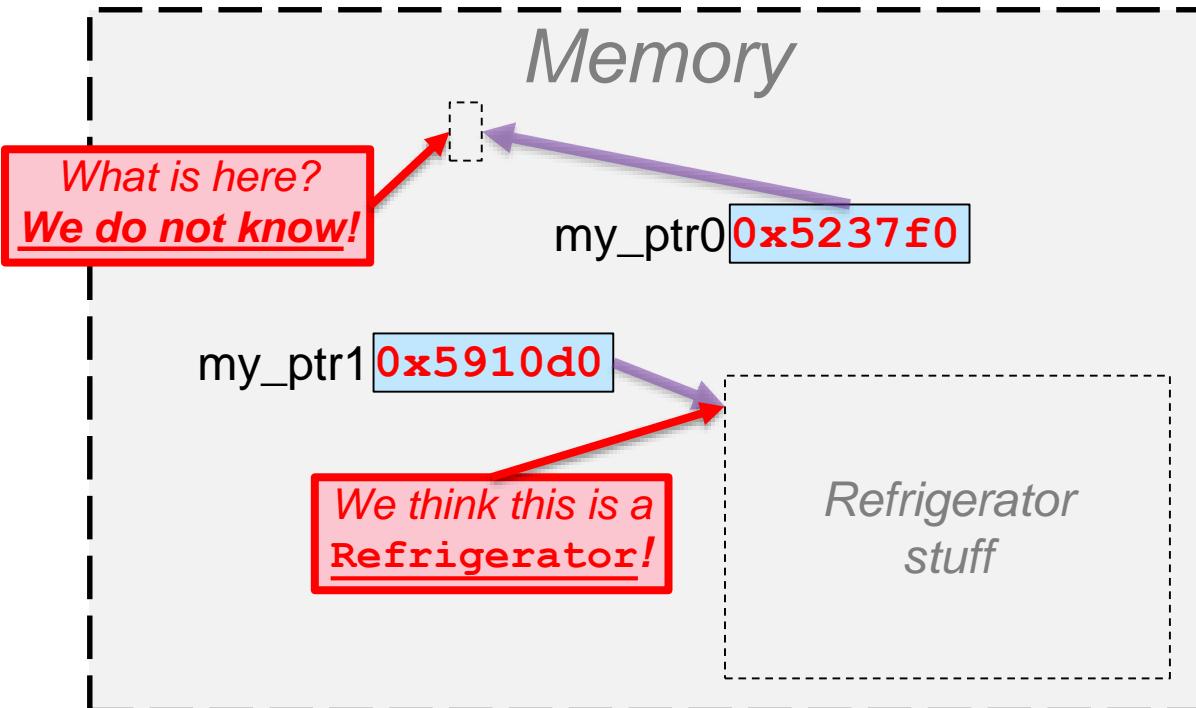


The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an address

1. A pointer holds an address
2. You create pointers
3. Uninitialized objects have “garbage” values
4. Your uninitialized pointer is “Rogue”
(it could point to anything, you cannot reason about it)



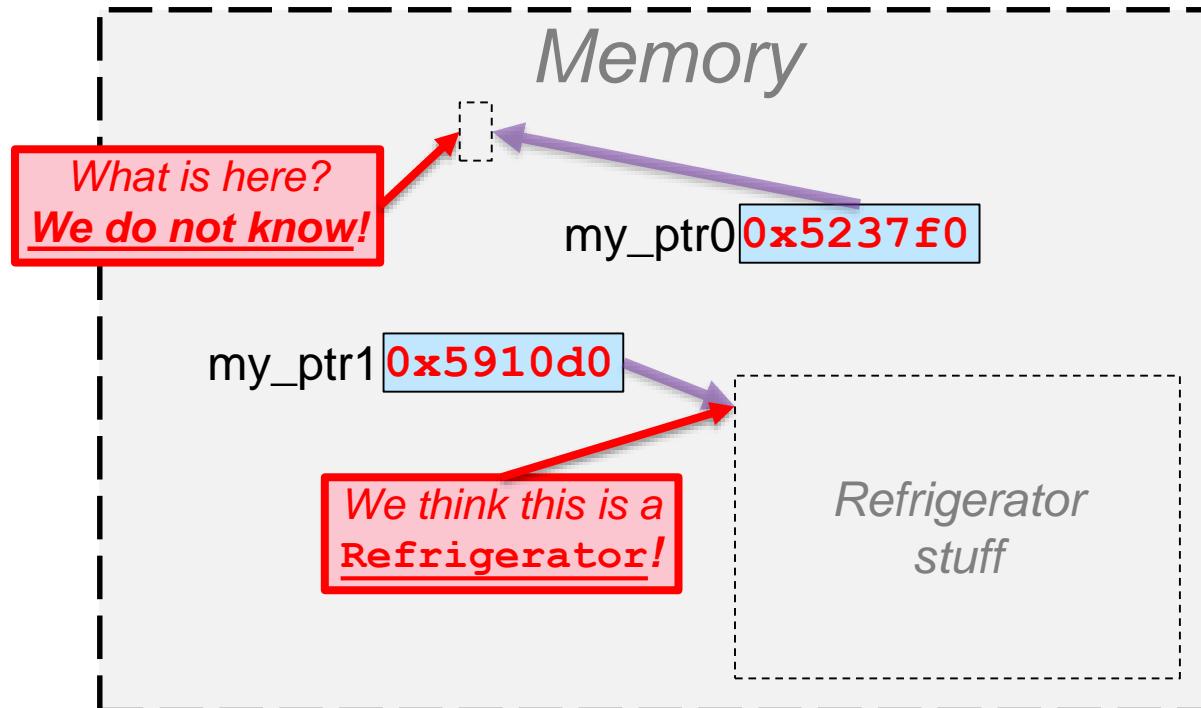
Using a rogue pointer
leads to **Undefined Behavior**

The “Rogue Pointer”

```
void* my_ptr0; // pointer to ??  
Refrigerator* my_ptr1; // pointer to Refrigerator
```

RECALL: A pointer holds an address

1. A pointer holds an address
2. You create pointers
3. Uninitialized objects have “garbage” values
4. Your uninitialized pointer is “Rogue”
(it could point to anything, you cannot reason about it)



Using a rogue pointer leads to **Undefined Behavior**

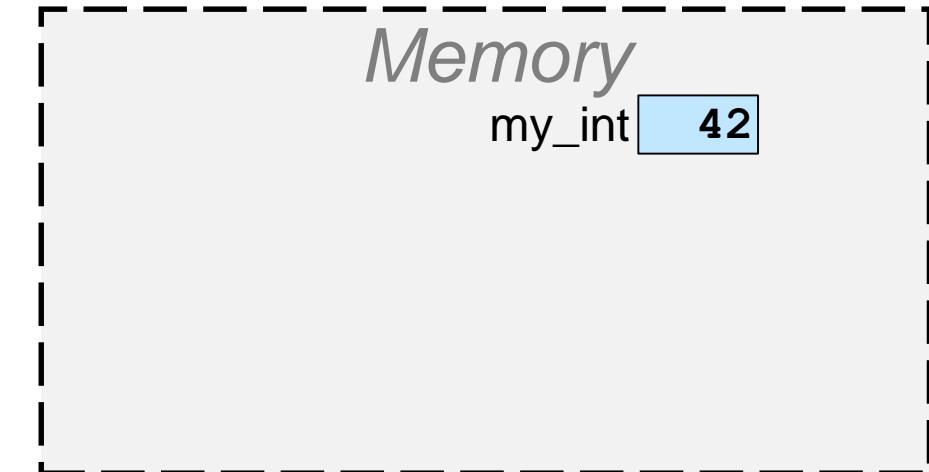
Best Practice: Initialize your pointers to your objects

Safer: Reference

Reference Semantics: The pointer
“references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;
```



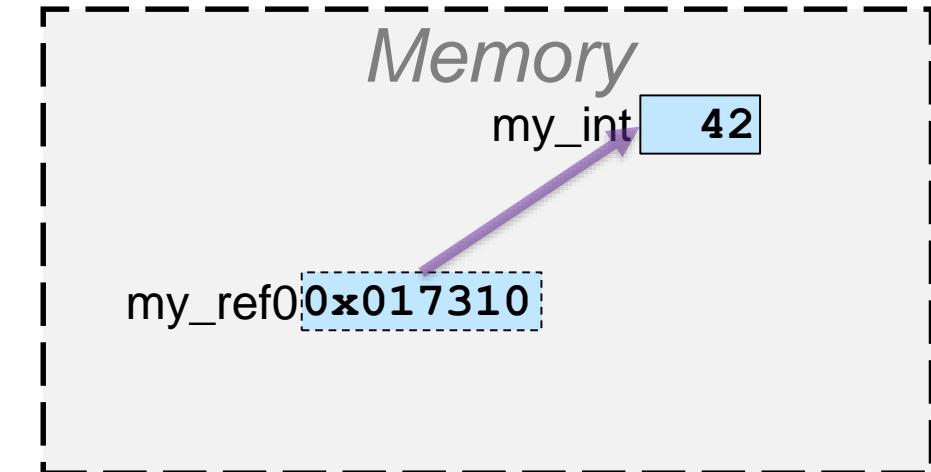
Safer: Reference

Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;
```

Create a
Reference

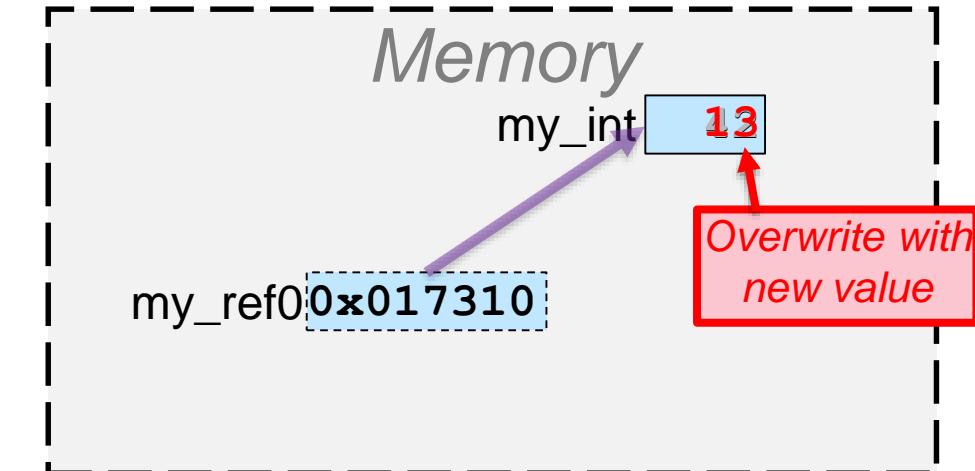


Safer: Reference

Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;  
  
Create a  
Reference  
my_int = 13;
```



Safer: Reference

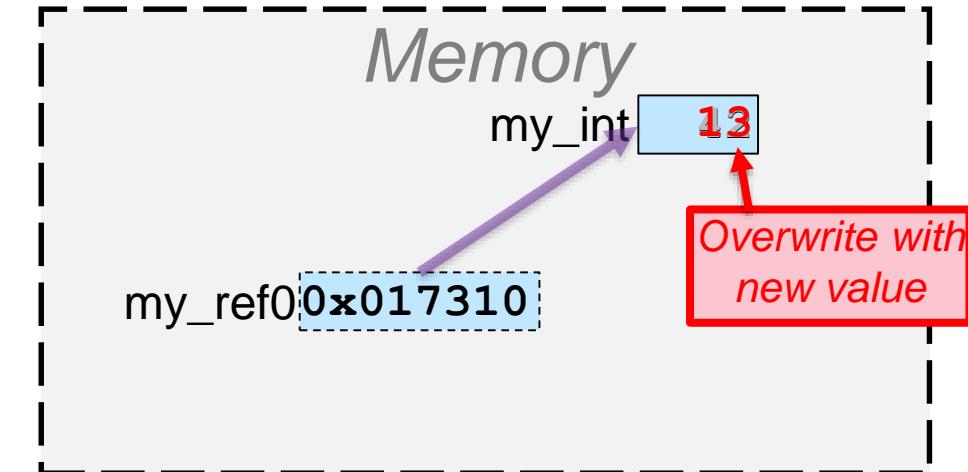
Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;  
  
my_int = 13;  
my_ref0 = 13; // same thing
```

Create a Reference

Use the reference “as an alias” for the referenced object



Safer: Reference

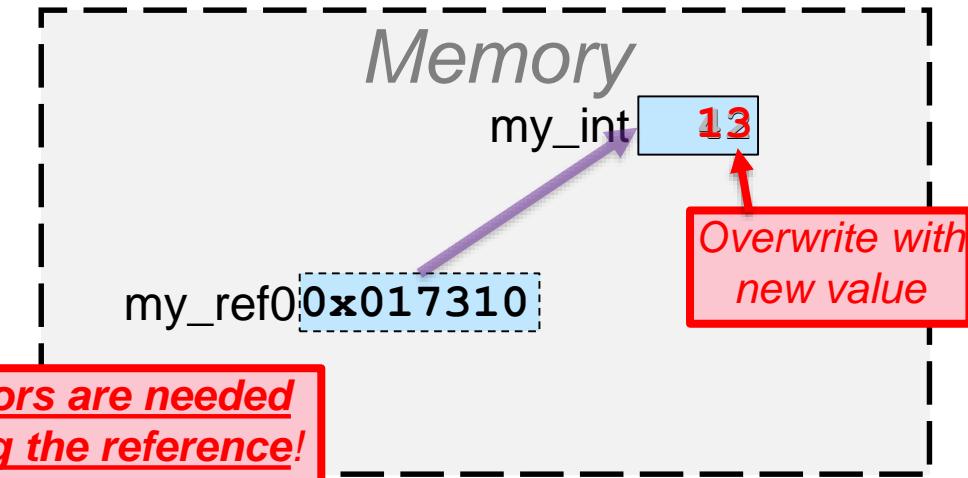
Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;  
  
my_int = 13;  
my_ref0 = 13; // same thing
```

Create a Reference

Use the reference “as an alias” for the referenced object



No operators are needed when using the reference!

It's just like using the referenced-object (because the reference is an “alias” for the referenced object)

Safer: Reference

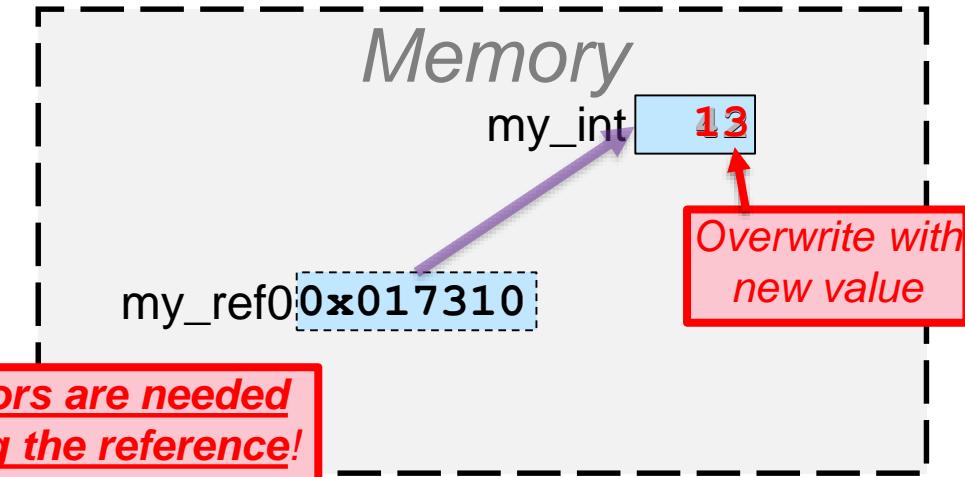
Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;  
  
my_int = 13;  
my_ref0 = 13; // same thing
```

Create a Reference

Use the reference “as an alias” for the referenced object



No operators are needed when using the reference!

It's just like using the referenced-object (because the reference is an “alias” for the referenced object)

A “Reference” is:

- Logically an “alias” for the referenced-instance
- Physically implemented as a pointer to the referenced-object

Safer: Reference

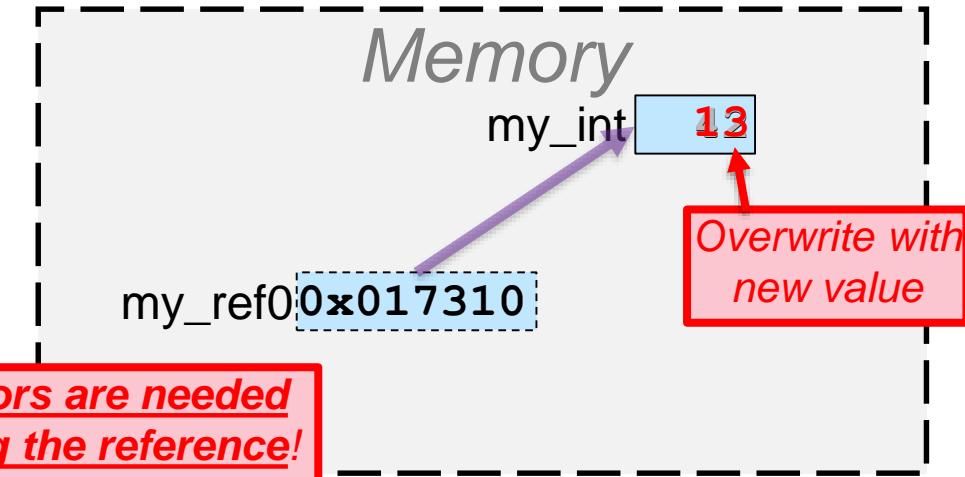
Reference Semantics: The pointer “references” the object at the target-address

- Create a pointer that *must* be initialized at creation-time (and which will never point to anything else)

```
int my_int = 42;  
int& my_ref0 = my_int;  
  
my_int = 13;  
my_ref0 = 13; // same thing
```

Create a Reference

Use the reference “as an alias” for the referenced object



No operators are needed when using the reference!

It's just like using the referenced-object (because the reference is an “alias” for the referenced object)

Tip: References are easier and safer than pointers (*but are still just pointers*)

A “Reference” is:

- Logically an “alias” for the referenced-instance
- Physically implemented as a pointer to the referenced-object

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. Value-semantics
 2. Reference Semantics (*i.e.*, “pointer-indirection”)

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. Value-semantics
 2. Reference Semantics (*i.e., “pointer-indirection”*)
- Is how everything really, physically, actually happens
 - In your software (*guarantees by the C++ Language Standard*)
 - In your hardware (*what the hardware is actually doing*)

This is how all systems
ACTUALLY RUN!

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. Value-semantics
 2. Reference Semantics (*i.e.*, “pointer-indirection”)
- Is how everything really, physically, actually happens
 - In your software (*guarantees by the C++ Language Standard*)
 - In your hardware (*what the hardware is actually doing*)

This is how all systems
ACTUALLY RUN!

*With everything you do, think about:
Is indirection occurring, or not?*

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. Value-semantics
 2. Reference Semantics (*i.e., “pointer-indirection”*)
- Is how everything really, physically, actually happens
 - In your software (*guarantees by the C++ Language Standard*)
 - In your hardware (*what the hardware is actually doing*)
- Thinking about indirection:
 - Is a “design” activity

*This is how all systems
ACTUALLY RUN!*

*With everything you do, think about:
Is indirection occurring, or not?*

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. Value-semantics
 2. Reference Semantics (*i.e., “pointer-indirection”*)
- Is how everything really, physically, actually happens
 - In your software (*guarantees by the C++ Language Standard*)
 - In your hardware (*what the hardware is actually doing*)
- Thinking about indirection:
 - Is a “design” activity
 - Is an “implementation” activity

*This is how all systems
ACTUALLY RUN!*

*With everything you do, think about:
Is indirection occurring, or not?*

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. **Value-semantics**
 2. **Reference Semantics** (*i.e., “pointer-indirection”*)
- Is how everything really, physically, actually happens
 - **In your software** (*guarantees by the C++ Language Standard*)
 - **In your hardware** (*what the hardware is actually doing*)
- Thinking about indirection:
 - Is a “**design**” activity
 - Is an “**implementation**” activity
 - **Helps us reason about behavior**, we can:
 - **Plan** for intended behavior
 - **Understand** edge cases and surprising behavior

*This is how all systems
ACTUALLY RUN!*

*With everything you do, think about:
Is indirection occurring, or not?*

Always: Discern Between Value and Reference

- Everything in your system is one of:
 1. **Value-semantics**
 2. **Reference Semantics** (*i.e., “pointer-indirection”*)
- Is how everything really, physically, actually happens
 - **In your software** (*guarantees by the C++ Language Standard*)
 - **In your hardware** (*what the hardware is actually doing*)
- Thinking about indirection:
 - Is a “**design**” activity
 - Is an “**implementation**” activity
 - **Helps us reason about behavior**, we can:
 - **Plan** for intended behavior
 - **Understand** edge cases and surprising behavior

*This is how all systems
ACTUALLY RUN!*

*With everything you do, think about:
Is indirection occurring, or not?*

*Skill: **Understanding Indirection**
Why? It explains **The Object Lifecycle***

What Is C++ ?



General-Purpose, Multi-Paradigm, Systems-Level

C++ is a
1 ↗ **general-purpose,**
2 ↗ **multi-paradigm,**
3 ↗ **systems-level** language

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming
 - Is designed to be **extended through libraries**

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming
 - Is designed to be **extended through libraries**
 - Through: **User-defined types**, **operator overloading**

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming
 - Is designed to be **extended through libraries**
 - Through: **User-defined types**, **operator overloading**
 - Can implement (efficiently!) **any conceivable Computer Science artifact**

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming
 - Is designed to be **extended through libraries**
 - Through: **User-defined types**, **operator overloading**
 - Can implement (*efficiently!*) **any conceivable Computer Science artifact**
 - To address:
 - **Domain-specific concerns** (*resource models, execution models*)
 - **Special-purpose problems** (*modeling domain-specific constraints*)

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

C++ Is General Purpose

- C++ is a **General Purpose Language**:
 - For general-purpose software programming
 - Is designed to be **extended through libraries**
 - Through: **User-defined types**, **operator overloading**
 - Can implement (*efficiently!*) **any conceivable Computer Science artifact**
 - To address:
 - **Domain-specific concerns** (*resource models, execution models*)
 - **Special-purpose problems** (*modeling domain-specific constraints*)

In Contrast:

A **Special Purpose Language** provides semantics to address problems for a **specific domain** examples:

- Prolog, Lisp: Designed for AI (*Artificial Intelligence*)
- SQL: Designed for database manipulation

Extending C++ Through Libraries is very powerful!

- Used to implement **other programming languages**
- Used to implement **Domain-Specific Languages (DSLs)**

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “*How To Think*”

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “*How To Think*”
 - Makes some concepts simple and obvious
 - Makes other concepts inconsistent, or impossible

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “*How To Think*”
 - Makes some concepts simple and obvious
 - Makes other concepts inconsistent, or impossible
- C++ is a Multi-Paradigm Language (*it does not tell you “How To Think”*)

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “***How To Think***”
 - Makes some concepts simple and obvious
 - Makes other concepts inconsistent, or impossible
- C++ is a Multi-Paradigm Language (*it does not tell you “How To Think”*)

You can *do*
<insert-paradigm-here>
with C++

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “*How To Think*”
 - Makes some concepts simple and obvious
 - Makes other concepts inconsistent, or impossible
- C++ is a Multi-Paradigm Language (*it does not tell you “How To Think”*)

You can *do*
<insert-paradigm-here>
with C++

C++ Is Not...

- C++ is not an Object Oriented Language
 - Rather, you can *do* OO with C++
- C++ is not a functional language
 - Rather, you can *do* functional programming with C++
- C++ is not an imperative language
 - Rather, you can *do* imperative programming with C++
- ...*insert your paradigm here...*

C++ Is Multi-Paradigm

Programming Paradigm:
How to think

- A Programming Paradigm defines, “*How To Think*”
 - Makes some concepts simple and obvious
 - Makes other concepts inconsistent, or impossible
- C++ is a Multi-Paradigm Language (*it does not tell you “How To Think”*)

You can *do*
<insert-paradigm-here>
with C++

All paradigms ultimately map to the underlying hardware (*for execution*)

C++ is that mapping

C++ Is Not...

- C++ is not an Object Oriented Language
 - Rather, you can *do* OO with C++
- C++ is not a functional language
 - Rather, you can *do* functional programming with C++
- C++ is not an imperative language
 - Rather, you can *do* imperative programming with C++
- ...*insert your paradigm here...*

Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You

In Practice:

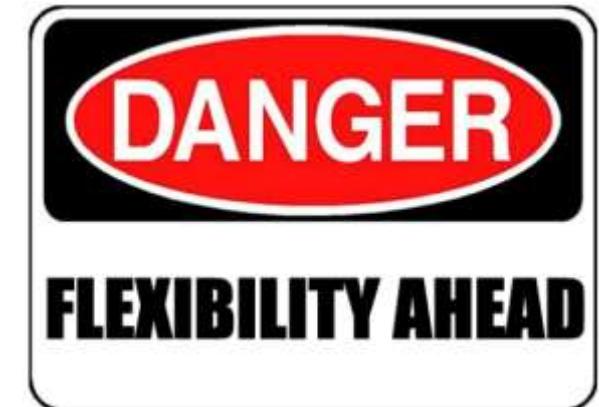
Most systems use more than one paradigm
(for pragmatic reasons, “best tool for the job”)

Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*

In Practice:

Most systems use more than one paradigm
(for pragmatic reasons, “best tool for the job”)



Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*
 - Other languages constrain your choices to a smaller set of options

In Practice:

Most systems use more than one paradigm
(for pragmatic reasons,
“best tool for the job”)

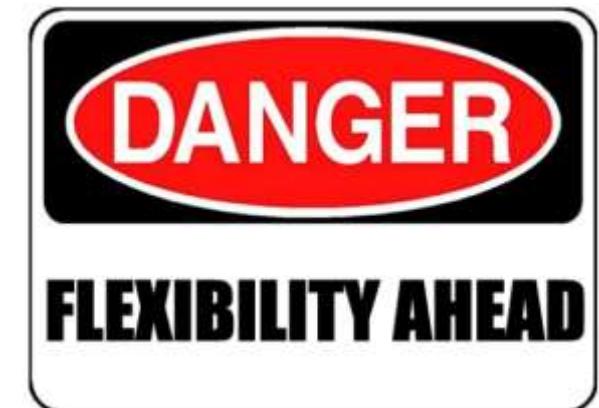


Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*
 - Other languages constrain your choices to a smaller set of options
 - Other languages promote “One Best Way To Do Something”
...but, what if that “Best Way” does not work for you?

In Practice:

Most systems use more than one paradigm
(for pragmatic reasons,
“best tool for the job”)

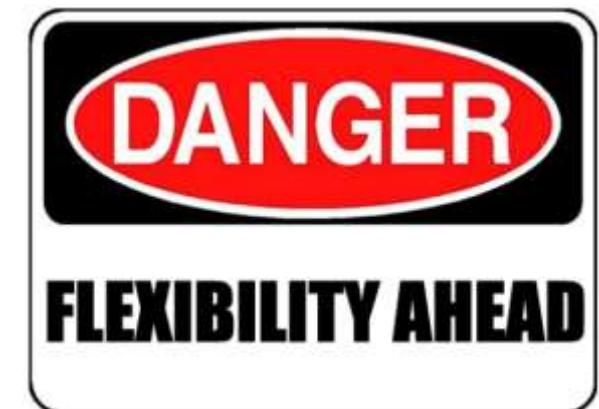


Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*
 - Other languages constrain your choices to a smaller set of options
 - Other languages promote “One Best Way To Do Something”
...but, what if that “Best Way” does not work for you?
 - Other languages do not allow for Turing-Complete compile-time logic execution and code-generation based on static types
(i.e., *template metaprogramming*)

In Practice:

Most systems use more than one paradigm
(for pragmatic reasons,
“best tool for the job”)



Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*
 - Other languages constrain your choices to a smaller set of options
 - Other languages promote “One Best Way To Do Something”
...but, what if that “Best Way” does not work for you?
 - Other languages do not allow for Turing-Complete compile-time logic execution and code-generation based on static types
(i.e., *template metaprogramming*)

Mixing all the paradigms in your program **is complicated** (*that's crazy!*)

*C++ itself is not complicated;
Rather, it is the nightmare that you create which is complicated*

In Practice:

Most systems use more than one paradigm (for pragmatic reasons, “best tool for the job”)



Which Paradigm For Me?

- If you want to know “*Which Paradigm?*” should you use for your program, C++ Will Not Help You
- C++ is worse than most languages: *Too Flexible!*
 - Other languages constrain your choices to a smaller set of options
 - Other languages promote “One Best Way To Do Something”
...but, what if that “Best Way” does not work for you?
 - Other languages do not allow for Turing-Complete compile-time logic execution and code-generation based on static types
(i.e., template metaprogramming)

Mixing all the paradigms in your program **is complicated** (*that's crazy!*)

C++ itself is not complicated;
Rather, it is the nightmare that you create which is complicated

In Practice:

Most systems use more than one paradigm (for pragmatic reasons, “best tool for the job”)



Overheard in the hallway:

“I must learn all the paradigms so I can mix all the paradigms in my C++ program.” --Anonymous

A Silly Thought!
Do Not Let This Happen To You

- Action
- Agent-oriented
- Array-oriented
- Automata-based
- Concurrent computing
 - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
 - Functional
 - Functional logic
 - Purely functional
 - Logic
 - Abductive logic
 - Answer set
 - Concurrent logic
 - Functional logic
 - Inductive logic
 - Constraint
 - Constraint logic
 - Concurrent constraint logic
 - Dataflow
 - Flow-based
 - Reactive
 - Ontology
- Differentiable
- Dynamic/scripting
- Event-driven
- Function-level (contrast: Value-level)
 - Point-free style
 - Concatenative
- Generic
- Imperative (contrast: Declarative)
 - Procedural
 - Object-oriented
 - Polymorphic
- Intentional
- Language-oriented
 - Domain-specific
- Literate
- Natural-language programming
- Metaprogramming
 - Automatic
 - Inductive programming
 - Reflective
 - Attribute-oriented
 - Macro
 - Template
- Non-structured (contrast: Structured)
 - Array
- Nondeterministic
- Parallel computing
 - Process-oriented
- Probabilistic
- Quantum
- Stack-based
- Structured (contrast: Non-structured)
 - Block-structured
 - Object-oriented
 - Actor-based
 - Class-based
 - Concurrent
 - Prototype-based
 - By separation of concerns:
 - Aspect-oriented
 - Role-oriented
 - Subject-oriented
 - Recursive
 - Symbolic
 - Value-level (contrast: Function-level)

**"Lots of ways
to think"**

Programming Paradigms

C++ Is A Systems-Level Language

System-Level Language: Provides (very) fine-grained control over the language semantics

C++ Is A Systems-Level Language

System-Level Language: Provides (very) fine-grained control over the language semantics

- C++ is suitable for systems-level programming
 - Managing resources (*including all memory, buses, and controllers*)
 - Managing hard-constraints (*including time/clock*)
 - Interfacing with hardware (*including OS and device driver implementation*)

C++ Is A Systems-Level Language

System-Level Language: Provides (very) fine-grained control over the language semantics

- C++ is suitable for systems-level programming
 - Managing resources (*including all memory, buses, and controllers*)
 - Managing hard-constraints (*including time/clock*)
 - Interfacing with hardware (*including OS and device driver implementation*)
- Superior to other programming languages, in C++:

1

You have the **tools**
to be efficient

Easier to express
elegant solutions!

C++ Is A Systems-Level Language

System-Level Language: Provides (very) fine-grained control over the language semantics

- C++ is suitable for systems-level programming
 - Managing resources (*including all memory, buses, and controllers*)
 - Managing hard-constraints (*including time/clock*)
 - Interfacing with hardware (*including OS and device driver implementation*)
- Superior to other programming languages, in C++:

1

You have the tools to be efficient

Easier to express elegant solutions!

2

You make explicit choices to spend resources

Easier to reason about what is going on!

Systems-Level Programming

Low-Level Languages
Programming model defined
by hardware architecture

Systems-Level Programming

Middle-Level Languages

Provide direct access to bits,
bytes, and addresses

Low-Level Languages

Programming model defined
by hardware architecture

Systems-Level Programming

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits,
bytes, and addresses

Low-Level Languages
Programming model defined
by hardware architecture

Systems-Level Programming

*Systems-Level
Programming*

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits,
bytes, and addresses

Low-Level Languages
Programming model defined
by hardware architecture

Systems-Level Programming

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Systems-Level Programming

Example C++ Systems-Level Features

- Direct mapping to target hardware
 - Defined in terms of, “*The Abstract Machine*”
 - Directly support hardware ISAs
(Instruction Set Architectures)

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Systems-Level Programming

Example C++ Systems-Level Features

- Direct mapping to target hardware
 - Defined in terms of, “The Abstract Machine”
 - Directly support hardware ISAs
(Instruction Set Architectures)
- Resource Idioms
(example, deterministic ctor...dtor, RAII)

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Systems-Level Programming

Example C++ Systems-Level Features

- Direct mapping to target hardware
 - Defined in terms of, “The Abstract Machine”
 - Directly support hardware ISAs
(Instruction Set Architectures)
- Resource Idioms
(example, deterministic ctor...dtor, RAII)
- Zero-Cost Abstractions
(compile-time optimization based on programmer’s intent)

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Systems-Level Programming

Example C++ Systems-Level Features

- Direct mapping to target hardware
 - Defined in terms of, “The Abstract Machine”
 - Directly support hardware ISAs
(Instruction Set Architectures)
- Resource Idioms
(example, deterministic ctor...dtor, RAII)
- Zero-Cost Abstractions
(compile-time optimization based on programmer’s intent)
- Template Metaprogramming
(compile-time code generation)

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Systems-Level Programming

Example C++ Systems-Level Features

- Direct mapping to target hardware
 - Defined in terms of, “The Abstract Machine”
 - Directly support hardware ISAs
(Instruction Set Architectures)
- Resource Idioms
(example, deterministic ctor...dtor, RAII)
- Zero-Cost Abstractions
(compile-time optimization based on programmer’s intent)
- Template Metaprogramming
(compile-time code generation)
- constexpr and related
(compile-time execution and evaluation)

“Boxing” and “Unboxing” is a phenomenon in high-level languages for runtime mapping of running-execution model (virtual or guest machine) to the actual system resources (host machine)

High-Level Languages
Abstract underlying hardware

Middle-Level Languages
Provide direct access to bits, bytes, and addresses

Low-Level Languages
Programming model defined by hardware architecture

Systems-Level
Programming

Abstractions: What Is The Cost?

Abstraction (*n*): The wrapping of a low-level concept into a higher-level concept

Abstractions: What Is The Cost?

- Example: Wrap a function

Abstraction (*n*): The wrapping of a low-level concept into a higher-level concept

“A function that is more complex than a simple composition of two functions (i.e., `f(g(x))`) should be given a name.” -- Sean Parent

Abstractions: What Is The Cost?

- Example: Wrap a function

Abstraction (*n*): The wrapping of a low-level concept into a higher-level concept

“A function that is more complex than a simple composition of two functions (i.e., `f(g(x))`) should be given a name.” -- Sean Parent

- This “name” is an abstraction

Abstractions: What Is The Cost?

- Example: Wrap a function

Abstraction (*n*): The wrapping of a low-level concept into a higher-level concept

"A function that is more complex than a simple composition of two functions (i.e., `f(g(x))`) should be given a name." -- Sean Parent

- This “name” is an abstraction

The C++ Language Standard:

- Is defined based on your Intent
- Is described through an “**Abstract Machine**”
(which models directly to the underlying hardware)

*Provides guarantees usable by
hardware manufacturers
(Is why hardware manufacturers
target their Instruction Set
Architectures (ISAs) to C and C++)*

Abstractions: What Is The Cost?

- Example: Wrap a function

Abstraction (*n*): The wrapping of a low-level concept into a higher-level concept

"A function that is more complex than a simple composition of two functions (i.e., `f(g(x))`) should be given a name." -- Sean Parent

- This “name” is an abstraction

The C++ Language Standard:

- Is defined based on your Intent
- Is described through an “**Abstract Machine**”
(which models directly to the underlying hardware)

*Provides guarantees usable by hardware manufacturers
(Is why hardware manufacturers target their Instruction Set Architectures (ISAs) to C and C++)*

In C++, this has Zero Cost

Abstraction costs in various languages:

- Python: Every level is slower than the level below
- C#, Java: Not designed for “zero cost”, examples:
 - Invoking a lambda always involves a “virtual” call
 - “JIT” compilers cannot afford many optimizations like inlining
- Rust: Some zero-cost abstractions, some with runtime cost
- C++: Many zero-cost abstractions (*user-defined types, `constexpr`, templates, all things available through compiler optimizations...*)

C++: Zero Cost Abstractions

*Thinking about levels of languages has become a mistake. Think of C++ as a language with an unsurpassed ability to manipulate hardware directly plus a set of abstraction mechanisms that allows us to raise the level of abstraction when needed. [...] **C++ enables zero-overhead abstraction** to get us away from the hardware **without adding cost**. By “zero-abstraction” I mean **not a byte and not a cycle wasted compared to hand-crafted low-level alternatives**. [...] Offering both hardware access and abstraction is the basis of C++. Doing it efficiently is what distinguishes it from other languages.*

-- Bjarne Stroustrup

“Thoughts On C++17”, 2015

Spending Your Resources

Software Development:
Deciding how to “spend” resources



Spending Your Resources

Software Development: Deciding how to “spend” resources

- With everything you do, you are “spending” your system resources
 - If you do nothing, then no system resources are used
 - If you do something, you are allocating system resources for that something



Spending Your Resources

Software Development: Deciding how to “spend” resources

- With everything you do, you are “spending” your system resources
 - If you do nothing, then no system resources are used
 - If you do something, you are allocating system resources for that something
- With C++, these decisions are EXPLICIT!
 - It's never WORSE than what you asked for
 - It's usually BETTER than what you asked for



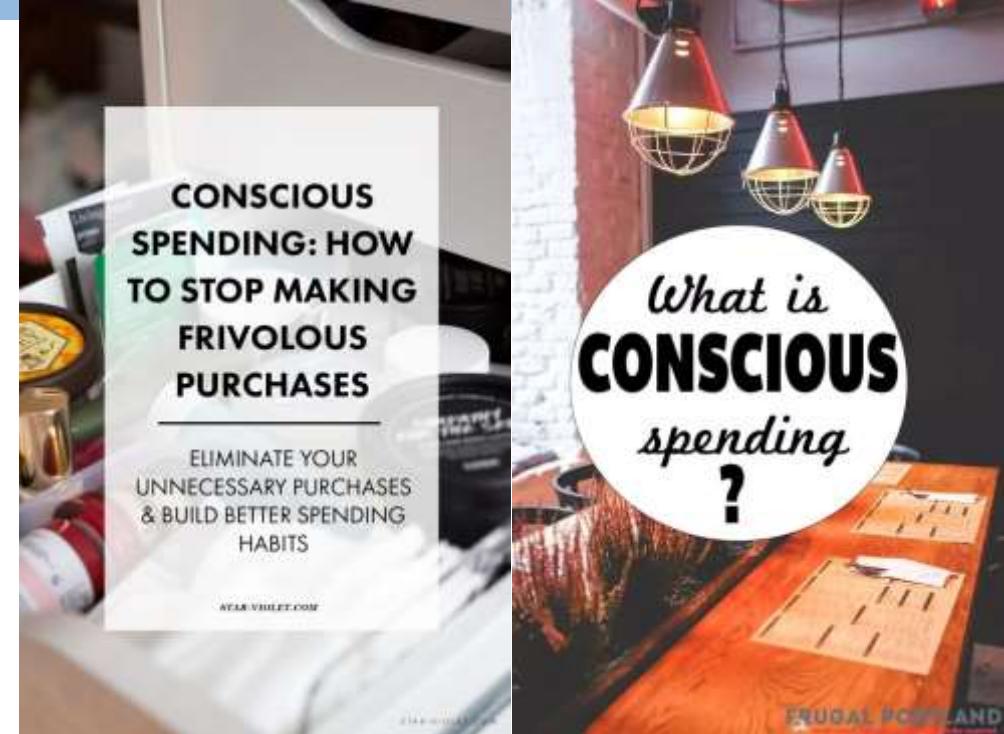
Spending Your Resources

Software Development: Deciding how to “spend” resources

- With everything you do, you are “spending” your system resources
 - If you do nothing, then no system resources are used
 - If you do something, you are allocating system resources for that something
- With C++, these decisions are EXPLICIT!
 - It's never WORSE than what you asked for
 - It's usually BETTER than what you asked for

C++ Optimization: Based on your intent

Results in MUCH greater efficiency!



Spending Your Resources

Software Development: Deciding how to “spend” resources

- With everything you do, **you are “spending” your system resources**
 - If you do nothing, then no system resources are used
 - If you do something, you are allocating system resources for that something
- With C++, **these decisions are *EXPLICIT!***
 - It's never WORSE** than what you asked for
 - It's usually BETTER** than what you asked for

C++ Optimization: Based on your *intent*

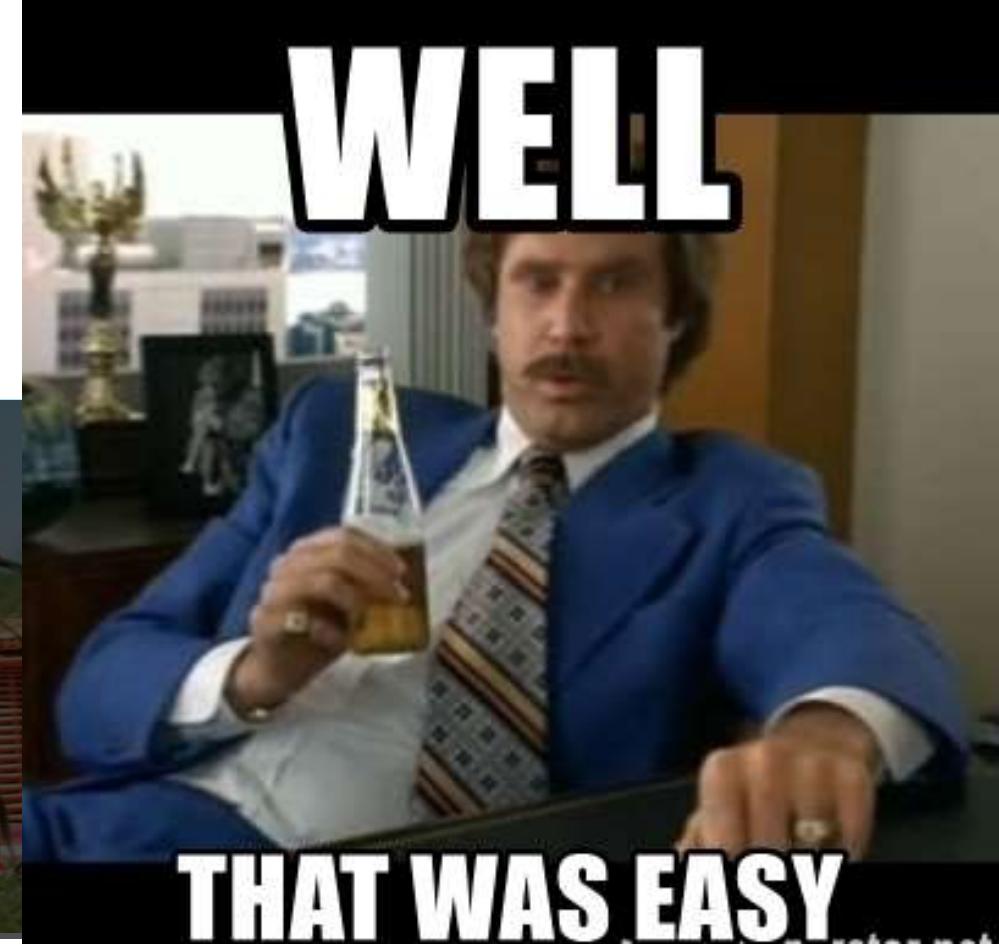
Results in **MUCH greater efficiency!**



BONUS:

The Compiler and CPU will lie, cheat, and steal to give you what you asked for, **WITHOUT** spending the resources you thought you were spending

C++ Is Strongly Typed



Compile-Time static type checking provides Guarantees

C++ Is Strongly Typed

- The Type System: One of the Most *Powerful* things in C++

C++ Is Strongly Typed

- The Type System: One of the Most Powerful things in C++

Types are *Defined*:

You associate “rules”
with payload (*state*)

*Empty payload
is allowed*

C++ Is Strongly Typed

- The Type System: One of the Most Powerful things in C++
- You should use the Type System whenever possible
 - In design, and implementation
 - To represent state, and implement algorithms
 - *Net Effect:* Invariants are maintained

Types are *Defined*:

You associate “rules”
with payload (*state*)

Empty payload
is allowed

C++ Is Strongly Typed

- The Type System: One of the Most Powerful things in C++
- You should use the Type System whenever possible
 - In design, and implementation
 - To represent state, and implement algorithms
 - Net Effect: Invariants are maintained

Types are *Defined*:

You associate “rules”
with payload (*state*)

Empty payload
is allowed

GUARANTEED!

- Can Rely upon design principles
- Can Reason about behavior
- Can Scale System with confidence!

C++ Is Strongly Typed

- The Type System: One of the Most Powerful things in C++
- You should use the Type System whenever possible
 - In design, and implementation
 - To represent state, and implement algorithms
 - Net Effect: Invariants are maintained
- How: User-Defined Types
 - You define a “key concept” (a “type”)
 - Includes “state” (can be empty, or has lots of stuff)
 - Includes “behavior” (properties, algorithms, and relationships that describe invariants on that type)

Types are Defined:

You associate “rules”
with payload (*state*)

Empty payload
is allowed

GUARANTEED!

- Can Rely upon design principles
- Can Reason about behavior
- Can Scale System with confidence!

C++ Is Strongly Typed

- The Type System: One of the Most Powerful things in C++
- You should use the Type System whenever possible
 - In design, and implementation
 - To represent state, and implement algorithms
 - Net Effect: Invariants are maintained
- How: User-Defined Types
 - You define a “key concept” (a “type”)
 - Includes “state” (can be empty, or has lots of stuff)
 - Includes “behavior” (properties, algorithms, and relationships that describe invariants on that type)

Types are *Defined*:

You associate “rules” with payload (*state*)

Empty payload
is allowed

GUARANTEED!

- Can Rely upon design principles
- Can Reason about behavior
- Can Scale System with confidence!

Types are *Composed*:

You aggregate types into “bigger” types, and apply rules to the aggregate

Your Domain: Rules and Types

```
enum class MeasureType {  
    DASH, TEASPOON, TABLESPOON,  
    CUP, PINT, QUART, GALLON  
};
```

Your Domain: Rules and Types

```
class KitchenMeasure {  
    MeasureType measure_type_;  
    int count_num_;  
};
```

```
enum class MeasureType {  
    DASH, TEASPOON, TABLESPOON,  
    CUP, PINT, QUART, GALLON  
};
```

*Define type **KitchenMeasure**
...which is composed of types
MeasureType and an **int**!*

Your Domain: Rules and Types

```
class KitchenMeasure {  
    MeasureType measure_type_;  
    int count_num_;  
};
```

```
class Rating {  
    // 0...5 stars  
    int num_stars_;  
};
```

```
enum class MeasureType {  
    DASH, TEASPOON, TABLESPOON,  
    CUP, PINT, QUART, GALLON  
};
```

Define type *KitchenMeasure*
...which is composed of types
MeasureType and an ***int***!

**Define type
*Rating***

Your Domain: Rules and Types

```
class KitchenMeasure {  
    MeasureType measure_type_;  
    int count_num_;  
};
```

```
class Rating {  
    // 0...5 stars  
    int num_stars_;  
};
```

```
enum class MeasureType {  
    DASH, TEASPOON, TABLESPOON,  
    CUP, PINT, QUART, GALLON  
};
```

```
class Recipe {  
    std::vector<KitchenMeasure> measures_;  
    Rating rating_;  
};
```

Define type
Rating

Define type *Recipe*
...which is composed
of types *Rating* and
std::vector<KitchenMeasure>

Your Domain: Rules and Types

```
class KitchenMeasure {  
    MeasureType measure_type_;  
    int count_num_;  
};
```

```
class Rating {  
    // 0...5 stars  
    int num_stars_;  
};
```

**Define type
Rating**

```
class Recipe {  
    std::vector<KitchenMeasure> measures_;  
    Rating rating_;  
};
```

**Define type Recipe
...which is composed
of types Rating and
std::vector<KitchenMeasure>**

```
class RecipeBox {  
    std::vector<Recipe> recipes_;  
public:  
    void addRecipe(const Recipe& recipe);  
    int getNumRecipes() const;  
};
```

Establish properties and rules

- Can add *Recipe* to *RecipeBox*
- Can ask *RecipeBox* for how many *Recipe* instances it contains

Leveraging The Type System

*"You can create a `KitchenMeasure`,
as long as you provide a
`MeasureType` and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure (  
        MeasureType measure_type,  
        int count_num);  
};
```

Leveraging The Type System

*"You can create a **KitchenMeasure**,
as long as you provide a
MeasureType and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure(  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0; // COMPILE-ERROR
```

Leveraging The Type System

*"You can create a **KitchenMeasure**,
as long as you provide a
MeasureType and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure(  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0;      // COMPILE-ERROR  
KitchenMeasure my_km1(5);   // COMPILE-ERROR
```

Leveraging The Type System

*"You can create a **KitchenMeasure**,
as long as you provide a
MeasureType and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure (  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0;      // COMPILE-ERROR  
KitchenMeasure my_km1(5);   // COMPILE-ERROR  
KitchenMeasure my_km2(MeasureType::TEASPOON, 2); //OK
```

Leveraging The Type System

*"You can create a **KitchenMeasure**,
as long as you provide a
MeasureType and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure(  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0;      // COMPILE-ERROR  
KitchenMeasure my_km1(5);   // COMPILE-ERROR  
KitchenMeasure my_km2(MeasureType::TEASPOON, 2); //OK
```

1 **Types** describe your domain

Leveraging The Type System

*"You can create a `KitchenMeasure`,
as long as you provide a
`MeasureType` and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure(  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0; // COMPILE-ERROR  
KitchenMeasure my_km1(5); // COMPILE-ERROR  
KitchenMeasure my_km2(MeasureType::TEASPOON, 2); //OK
```

1 **Types** describe your domain

2 **Rules** enforce invariants among types

Leveraging The Type System

*"You can create a `KitchenMeasure`,
as long as you provide a
`MeasureType` and a count"*

```
class KitchenMeasure {  
public:  
    KitchenMeasure(  
        MeasureType measure_type,  
        int count_num);  
};
```

```
KitchenMeasure my_km0; // COMPILE-ERROR  
KitchenMeasure my_km1(5); // COMPILE-ERROR  
KitchenMeasure my_km2(MeasureType::TEASPOON, 2); //OK
```

- 1 **Types** describe your domain
- 2 **Rules** enforce invariants among types
- 3 **Invariants** ensure your system Does The Right Thing™

Type Checking

Type Checking:
The process of verifying and
enforcing type constraints

- Type Checking is a (*limited*) form of Program Verification
 - Many types of errors are caught early in the development cycle
 - Also called “Type Safety”

Type Checking

Type Checking:
The process of verifying and
enforcing type constraints

- Type Checking is a (*limited*) form of Program Verification
 - Many types of errors are caught early in the development cycle
 - Also called “Type Safety”
- Static type checkers evaluate only the type information that can be determined at compile-time

Type Checking

Type Checking:
The process of verifying and
enforcing type constraints

- Type Checking is a (*limited*) form of Program Verification
 - Many types of errors are caught early in the development cycle
 - Also called “Type Safety”
- Static type checkers evaluate only the type information that can be determined at compile-time
- (*For Static Typing*): Execution can be made more efficient (*because the compiler can verify that the checked conditions hold for all possible executions*)

Type Checking

Type Checking:
The process of verifying and
enforcing type constraints

- Type Checking is a (*limited*) form of Program Verification
 - Many types of errors are caught early in the development cycle
 - Also called “Type Safety”
- Static type checkers evaluate only the type information that can be determined at compile-time
- (*For Static Typing*): Execution can be made more efficient (*because the compiler can verify that the checked conditions hold for all possible executions*)

Dynamic Typing: Type checking is performed at run-time

Static Typing: Type checking is performed before runtime

Example: Performed
“at compile-time”

Static vs. Dynamic Typing

Example: Performed
at compile-time

Dynamic Typing: Type checking is performed at runtime

Static Typing: Type checking is performed before runtime

Static vs. Dynamic Typing

Example: Performed at compile-time

Dynamic Typing: Type checking is performed at runtime

Static Typing: Type checking is performed before runtime

1. A variable may be re-bound to a different type during its lifecycle
2. Type definition may be mutable at runtime

Static vs. Dynamic Typing

Example: Performed at compile-time

Dynamic Typing: Type checking is performed at runtime

1. A variable may be re-bound to a different type during its lifecycle
2. Type definition may be mutable at runtime

Static Typing: Type checking is performed before runtime

1. A variable is bound to its “type” for its entire scope
2. Type definition is not mutable at runtime

Static vs. Dynamic Typing

Example: Performed at compile-time

Dynamic Typing: Type checking is performed at runtime

1. A variable may be re-bound to a different type during its lifecycle
2. Type definition may be mutable at runtime

- **Very flexible!**
(Can redefine types at runtime)
- **Hard to reason about!**
 - Anything can be redefined
(guarantees do not exist)
 - May result in runtime errors
(for discovered nonsensical expressions)

Examples: Perl, Python, Ruby, Javascript, Lisp, Lua, PHP

Static Typing: Type checking is performed before runtime

1. A variable is bound to its “type” for its entire scope
2. Type definition is not mutable at runtime

Static vs. Dynamic Typing

Example: Performed at compile-time

Dynamic Typing: Type checking is performed at runtime

1. A variable may be re-bound to a different type during its lifecycle
2. Type definition may be mutable at runtime

- **Very flexible!**
(Can redefine types at runtime)
- **Hard to reason about!**
 - Anything can be redefined
(guarantees do not exist)
 - May result in runtime errors
(for discovered nonsensical expressions)

Examples: Perl, Python, Ruby, Javascript, Lisp, Lua, PHP

Static Typing: Type checking is performed before runtime

1. A variable is bound to its “type” for its entire scope
2. Type definition is not mutable at runtime

- **Fixed rules established at compile-time!**
(Cannot be changed at runtime)
- **Can reason about these invariants** at runtime!
 - Because rules are unchanging
(guarantees exist)
 - No runtime errors for type constraints that can be verified at compile-time

Examples: C, C++, Java, Rust, Go, Haskell, Fortran

Scale Better, And Safer!

The Type System enforces
your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

Scale Better, And Safer!

The Type System enforces your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

Possible exceptions:

- Impedance mismatch in APIs and adapter layers
- Issues with threading and concurrent models (is a design issue)

Scale Better, And Safer!

The Type System enforces your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

- Computer Science Today: Well-Established that “strong typing”:
 - Is beneficial for system design and development
 - Significantly reduces bugs

Without strong typing, you must rely on runtime tests

Possible exceptions:

- Impedance mismatch in APIs and adapter layers
- Issues with threading and concurrent models (is a design issue)

Scale Better, And Safer!

The Type System enforces your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

- Computer Science Today: Well-Established that **“strong typing”**:
 - Is beneficial for system design and development
 - Significantly reduces bugs  *Without strong typing, you must rely on runtime tests*
- *Example:* TypeScript (stronger typing) was invented to enable better scaling and reduced bug count over Javascript (*not strongly typed*)

Possible exceptions:

- Impedance mismatch in APIs and adapter layers
- Issues with threading and concurrent models (is a design issue)

Scale Better, And Safer!

The Type System enforces your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

- Computer Science Today: Well-Established that “strong typing”:
 - Is beneficial for system design and development
 - Significantly reduces bugs  Without strong typing, you must rely on runtime tests
- Example: Typescript (stronger typing) was invented to enable better scaling and reduced bug count over Javascript (*not strongly typed*)

Possible exceptions:

- Impedance mismatch in APIs and adapter layers
- Issues with threading and concurrent models (is a design issue)

Compile-time Static Type Checking:
C++ Has the Strongest Type system of any language

Scale Better, And Safer!

The Type System enforces your design invariants

The majority of bugs in almost any system can *almost always* be resolved through better use of the type system

- Computer Science Today: Well-Established that “strong typing”:
 - Is beneficial for system design and development
 - Significantly reduces bugs *Without strong typing, you must rely on runtime tests*
- Example: TypeScript (stronger typing) was invented to enable better scaling and reduced bug count over Javascript (*not strongly typed*)

Possible exceptions:

- Impedance mismatch in APIs and adapter layers
- Issues with threading and concurrent models (is a design issue)

Compile-time Static Type Checking:
C++ Has the Strongest Type system of any language

Also Enables:
Turing-Complete
Compile-Time
Metaprogramming
(i.e., “C++ templates”)

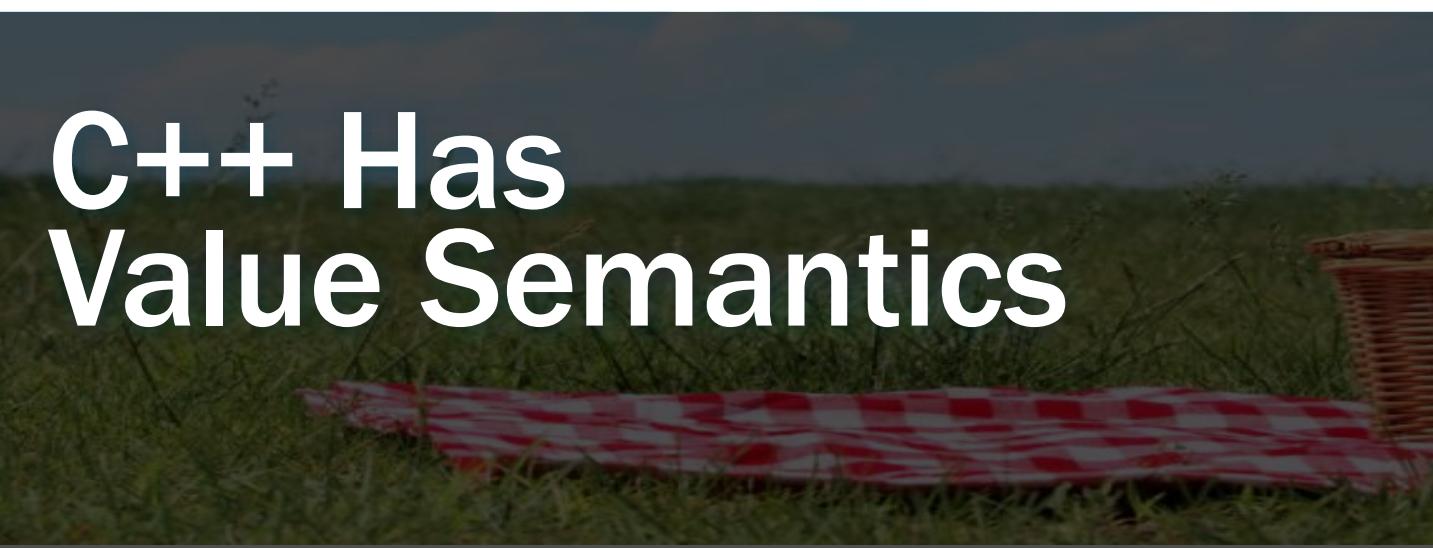
C++ is a

1 → **general-purpose,**

2 → **multi-paradigm,**

4 → **strongly-typed,**

3 → **systems-level** language



C++ Has Value Semantics

Easier (and Safer! and more Correct!)

To Reason About Values



How Does Your Program Run?

- Your software program is on disk. What happens?

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 2. Operating System “executes” your program

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - How? It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - **How?** It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - **How?** After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - **How?** It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - **How?** After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.
 - **How?** Data and instructions are loaded (*copied*) onto the CPU bus, and then (*copied*) into CPU caches, and then loaded (*copied*) into CPU registers.

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - **How?** It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - **How?** After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.
 - **How?** Data and instructions are loaded (*copied*) onto the CPU bus, and then (*copied*) into CPU caches, and then loaded (*copied*) into CPU registers.
 - Additional **copies exist separate for each processor core**, and these copies are kept “in-sync” (*re-copied*) across cores as individual bit-patterns (*values*) are overwritten with new values

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - **How?** It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - **How?** After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.
 - **How?** Data and instructions are loaded (*copied*) onto the CPU bus, and then (*copied*) into CPU caches, and then loaded (*copied*) into CPU registers.
 - Additional **copies exist separate for each processor core**, and these copies are kept “in-sync” (*re-copied*) across cores as individual bit-patterns (*values*) are overwritten with new values
 - **Mutated values are eventually copied back** from the CPU register, to the CPU caches, across the CPU bus, to main memory, and perhaps copied onto the data bus for write to disk.

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - How? It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - How? After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.
 - How? Data and instructions are loaded (*copied*) onto the CPU bus, and then (*copied*) into CPU caches, and then loaded (*copied*) into CPU registers.
 - Additional **copies exist separate for each processor core**, and these **copies** are kept “in-sync” (*re-copied*) across cores as individual bit-patterns (**values**) are overwritten with new values
 - **Mutated values are eventually copied back** from the CPU register, to the CPU caches, across the CPU bus, to main memory, and perhaps *copied* onto the data bus for write to disk.

How Does Your Program Run?

- Your software program is on disk. What happens?
 1. Software image is loaded (*copied*) into memory
 - How? It is loaded (*copied*) onto the data-bus, and then (*copied*) to main memory
 2. Operating System “executes” your program
 - How? After setting up system-resources, an instruction sequence from your program is fed (*copied*) into the CPU, which begins instruction execution.
 - How? Data and instructions are loaded (*copied*) onto the CPU bus, and then (*copied*) into CPU caches, and then loaded (*copied*) into CPU registers.
 - Additional **copies exist separate for each processor core**, and these **copies** are kept “in-sync” (*re-copied*) across cores as individual bit-patterns (**values**) are overwritten with new values
 - **Mutated values are eventually copied back** from the CPU register, to the CPU caches, across the CPU bus, to main memory, and perhaps *copied* onto the data bus for write to disk.

No “original thing” exists.

Everything is a copy, or a mutation, of some other thing.

Values Are All That Exist

The only value that can possibly exist:

A bit-pattern, which is
interpreted as a primitive type

Your CPU only
processes
primitive types

**It's Values
all the way
down!**



Values Are All That Exist

The only value that can possibly exist:

A bit-pattern, which is
interpreted as a primitive type

Your CPU only
processes
primitive types

**It's Values
all the way
down!**

Your computer moves
bit-patterns around.
These are called “values”



The Data Object

- All-copies everywhere is chaos. How can we reason about anything?

The Data Object

Data Object: A term to identify
a conceptual entity

- All-copies everywhere is chaos. How can we reason about anything?
- We think in terms of a “Data Object”

*Data Object is
“a useful fiction”*

The Data Object

Data Object: A term to identify a conceptual entity

- All-copies everywhere is chaos. How can we reason about anything?
- We think in terms of a “Data Object”
- “Data Object” Allows us to reason about “the current value” for that entity, examples:
 - Our program creates a “data object”, and it may have different values over its lifetime, but logically we reason that it is the same “entity”
 - That data object is copied many times, perhaps simultaneously into caches for many processor cores; but the CPU will synchronize all updates to ensure each copy is logically “the correct current-value”

The Data Object

Data Object: A term to identify a conceptual entity

- All-copies everywhere is chaos. How can we reason about anything?
- We think in terms of a “Data Object”
- “Data Object” Allows us to reason about “the current value” for that entity, examples:
 - Our program creates a “data object”, and it may have different values over its lifetime, but logically we reason that it is the same “entity”
 - That data object is copied many times, perhaps simultaneously into caches for many processor cores; but the CPU will synchronize all updates to ensure each copy is logically “the correct current-value”

*Data Object is
“a useful fiction”*

The C++ Standard, the compiler, and the CPU all conspire to provide guarantees for data object value consistency
(so we can reason about synchronization among the many copies of that data object that simultaneously exist)

A Useful Fiction: Data Object

- Thinking in terms of a “data object” is a convenient lie (*that we tell to our self*)

A Useful Fiction: Data Object

- Thinking in terms of a “data object” is a convenient lie (*that we tell to our self*)
- Technically, a “data object” does not exist

A Useful Fiction: Data Object

- Thinking in terms of a “data object” is a **convenient lie** (*that we tell to our self*)
- Technically, a “**data object**” **does not exist**
- In reality, we have a “**data object value**”, which:
 1. Is **physically a bit-pattern**
 2. Is **interpreted as a primitive type** (*the CPU only processes primitive types*)
 3. Is **relocated** (*the value is moved around*)
 4. Is **copied** (*multiple instances simultaneously exist for that data object value*)
 5. Is **versioned** (*each copied value is associated with a version of that data object over its mutating lifecycle*)

A Useful Fiction: Data Object

- Thinking in terms of a “data object” is a **convenient lie** (*that we tell to our self*)
- Technically, a “**data object**” **does not exist**
- In reality, we have a “**data object value**”, which:
 1. Is **physically a bit-pattern**
 2. Is **interpreted as a primitive type** (*the CPU only processes primitive types*)
 3. Is **relocated** (*the value is moved around*)
 4. Is **copied** (*multiple instances simultaneously exist for that data object value*)
 5. Is **versioned** (*each copied value is associated with a version of that data object over its mutating lifecycle*)

Data Object

A “*lie*” we tell
to our self

Data Object Value

Exists!

A Useful Fiction: Data Object

- Thinking in terms of a “data object” is a convenient lie (*that we tell to our self*)
- Technically, a “data object” **does not exist**
- In reality, we have a “data object value”, which:
 1. Is physically a bit-pattern
 2. Is interpreted as a primitive type (*the CPU only processes primitive types*)
 3. Is relocated (*the value is moved around*)
 4. Is copied (*multiple instances simultaneously exist for that data object value*)
 5. Is versioned (*each copied value is associated with a version of that data object over its mutating lifecycle*)

*The two hardest things
in Computer Science:*

1. Naming Things
2. Cache Invalidation
3. Off-By-One errors

Data Object

A “lie” we tell
to our self

Data Object Value

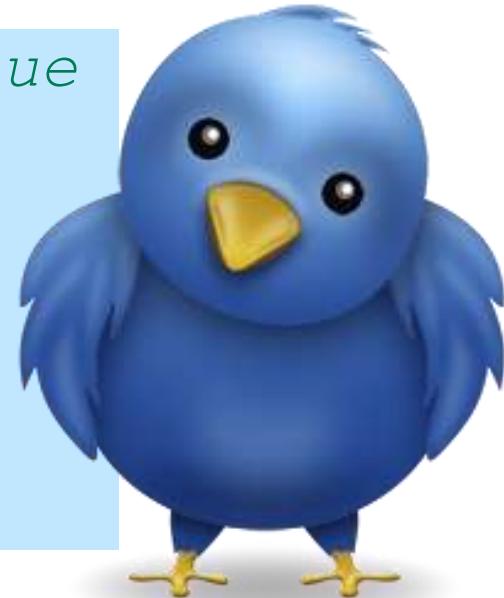
Exists!

Data Object In Practice

?

```
int x, y; // 'x' has "garbage" value
```

'x' version 0

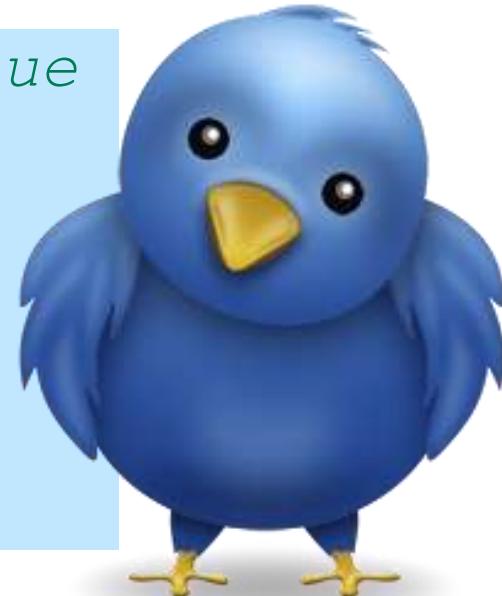


Which version of 'x'?

Data Object In Practice

?

```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1
```

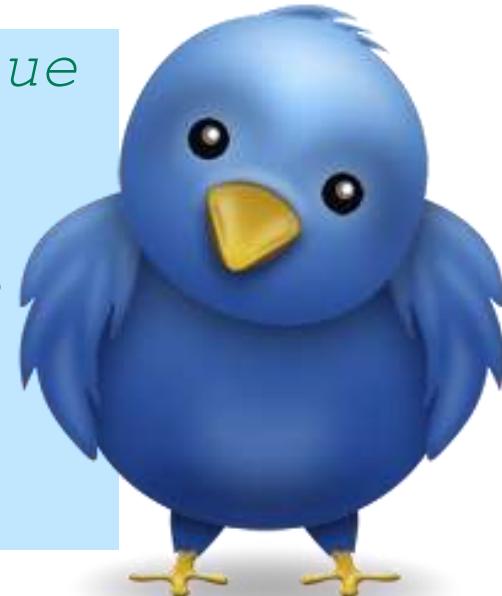


Which version of 'x'?

Data Object In Practice



```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?
```

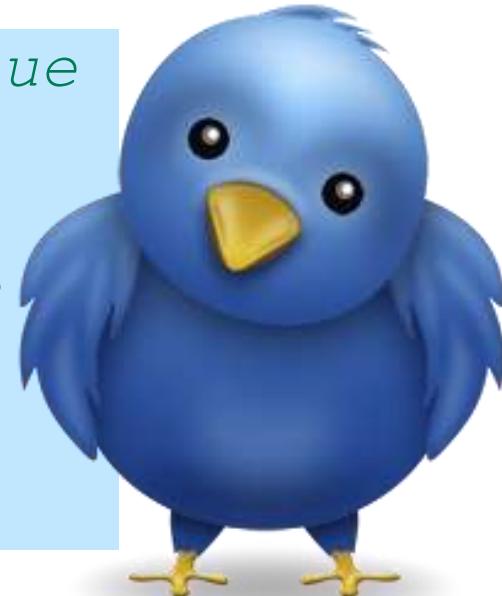


Which version of 'x'?

Data Object In Practice



```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2
```

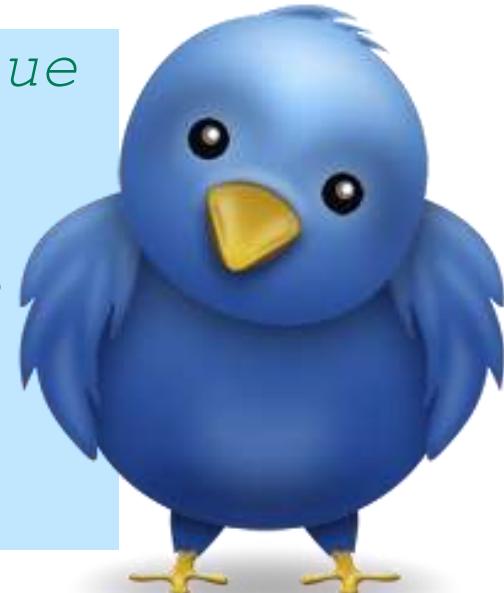


Which version of 'x'?

Data Object In Practice

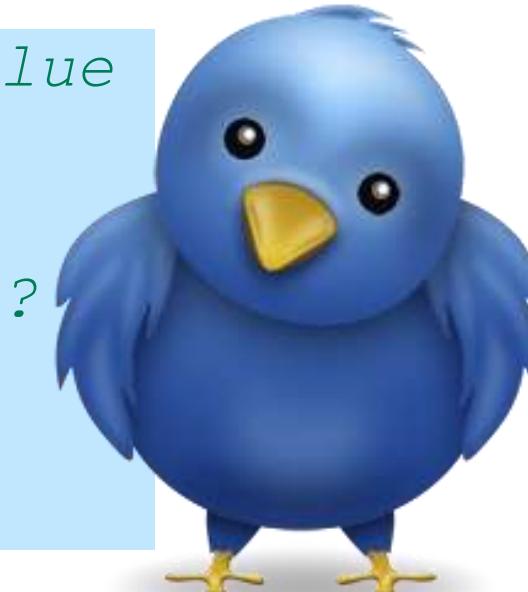


```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n
```



Which version of 'x'?

Data Object In Practice



```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n
```

Should use 'x'
version 1!

Which version of 'x'?

Data Object In Practice



```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n  
  
Should use 'x'  
version 1!
```

Which version of 'x'?

- A data object always has a “current” value

Data Object In Practice

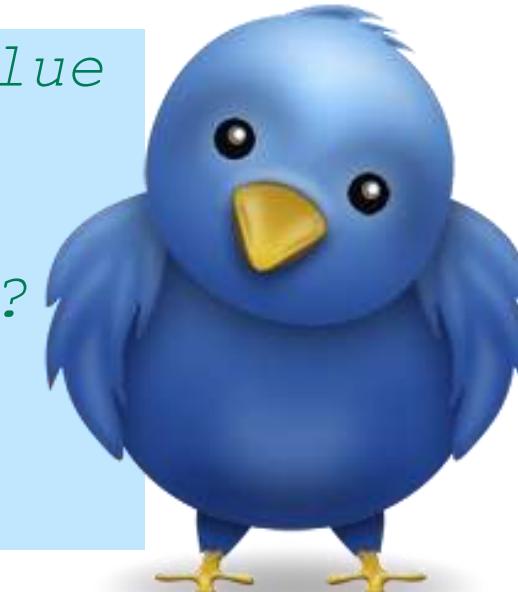


```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n  
  
Should use 'x'  
version 1!
```

Which version of 'x'?

- A data object always has a “current” value
 - For const data objects, this value never changes
(easier to reason about!) ← **Always prefer const when possible**

Data Object In Practice



Which version of 'x'?

```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n
```

Should use 'x' version 1!

- A data object always has a “current” value
 - For const data objects, this value never changes
(easier to reason about!) ← **Always prefer const when possible**
 - For non-const data objects, this value can mutate over time
(each mutation identifies a new version of that data object)

Data Object In Practice



```
int x, y; // 'x' has "garbage" value  
'x' version 0  
  
x = 1;      // 'x' has value 1  
'x' version 1  
y = x + 3; // which version of 'x'?  
x = 5;      // 'x' has value 5  
'x' version 2  
...  
x = ...;    // 'x' has value ...  
'x' version n
```

Should use 'x'
version 1!



Which version of 'x'?

- A data object always has a “current” value
 - For const data objects, this value never changes
(easier to reason about!) *Always prefer const when possible*
 - For non-const data objects, this value can mutate over time
(each mutation identifies a new version of that data object)

Trivia:

Compilers implement this through
Static Single Assignment (SSA)

Value vs. Reference Semantics

Value Semantics:

Each object is a value
(independent of everything else)

Reference Semantics:

Objects are identified by address
(which coexist within a monolithic model)

Value vs. Reference Semantics

Value Semantics:

Each object is a value
(independent of everything else)

Reference Semantics:

Objects are identified by address
(which coexist within a monolithic model)

- Value Semantics: **Each data object is a value**
 - Each value is independent of all other values (*value of a data object, or address*)
 - Easier to reason about (*values are uncoupled from other values*)
 - Easier to optimize (*by compiler, by CPU*)
 - Easier to scale (*on local core, or multi-core, or distributed*)

Value vs. Reference Semantics

Value Semantics:

Each object is a value
(independent of everything else)

Reference Semantics:

Objects are identified by address
(which coexist within a monolithic model)

- **Value Semantics:** **Each data object is a value**
 - Each value is independent of all other values (*value of a data object, or address*)
 - Easier to reason about (*values are uncoupled from other values*)
 - Easier to optimize (*by compiler, by CPU*)
 - Easier to scale (*on local core, or multi-core, or distributed*)
- **Reference Semantics:** We **identify a data object by its address**
 - Indirection: We “go-to” the data object identified by an address
 - **“Reference”:** We create an “alias” to a data object (*How?*):
 - We create a pointer with a value of the referenced object
 - We automatically do “indirection” (*to “pretend” the pointer is the referenced object*)

Quiz Time

Value Semantics is a tool for:

- A. Simpler Reasoning
- B. Uncoupled Concerns
- C. Better Optimization
- D. Better Scaling
- E. All of the above

I HAVE NO IDEA



WHAT I'M DOING

Quiz Time

Value Semantics is a tool for:

- A. Simpler Reasoning
- B. Uncoupled Concerns
- C. Better Optimization
- D. Better Scaling
- E. All of the above

I HAVE NO IDEA



WHAT I'M DOING

Quiz Time

Value Semantics is a tool for:

- A. Simpler Reasoning
- B. Uncoupled Concerns
- C. Better Optimization
- D. Better Scaling
- E. All of the above

Reference Semantics is a tool for:

- A. Blaming someone
- B. Promoting speed
- C. Computing a bonus structure
- D. Ensuring no copies are made
- E. All of the above

I HAVE NO IDEA



WHAT I'M DOING

Quiz Time

Value Semantics is a tool for:

- A. Simpler Reasoning
- B. Uncoupled Concerns
- C. Better Optimization
- D. Better Scaling
- E. All of the above

Reference Semantics is a tool for:

- A. Blaming someone
- B. Promoting speed
- C. Computing a bonus structure
- D. Ensuring no copies are made
- E. All of the above

I HAVE NO IDEA



WHAT I'M DOING

Value-Semantics In Practice

- In C++, value semantics is “the default”

```
int foo(Dog some_dog)
{ // A "Dog" value is
// is passed-in
...
}
```

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
// is passed-in
...
}
```

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
// is passed-in
...
}
```

The default:
“By Value”



Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
// is passed-in
...
}
```

The default:
“By Value”

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

The SAFEST
one!

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

The SAFEST
one!

The MOST
DANGEROUS one!

Value-Semantics In Practice

- In C++, value semantics is “the default”
- Explicit syntax tells the compiler you want reference semantics

```
int foo(const Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog& some_dog)
{ // A "never-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

```
int foo(const Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  ...
}
```

```
int foo(Dog* some_dog)
{ // A "maybe-null" pointer
  // to a "Dog" is passed-in
  // AND we can mutate the
  // referenced object!
  ...
}
```

In C++, you must “opt-in”
for Reference Semantics

```
int foo(Dog some_dog)
{ // A "Dog" value is
  // is passed-in
  ...
}
```

The default:
“By Value”

The SAFEST
one!

The MOST
DANGEROUS one!

Java supports
only this version

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”
- } ← “*False Dichotomy*” between
“value-types” and “reference-types”
can make reasoning more complex

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”
- “Boxing” and “Unboxing” is the transition between a “reference” and a “value” (such as *to/from a primitive type*)

“False Dichotomy” between
“value-types” and “reference-types”
can make reasoning more complex

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”
 - **“Boxing” and “Unboxing” is the transition** between a “reference” and a “value” (such as *to/from a primitive type*)
 - Sometimes it is “unclear” whether you have a value-type or reference type
- “False Dichotomy” between
“value-types” and “reference-types”
can make reasoning more complex

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”
- “Boxing” and “Unboxing” is the transition between a “reference” and a “value” (such as *to/from a primitive type*)
- Sometimes it is “unclear” whether you have a value-type or reference type

“False Dichotomy” between
“value-types” and “reference-types”
can make reasoning more complex

In C++, value or reference is
ALWAYS clear (through syntax)

Reference Semantic Languages

- Most languages support some level of both value-semantics and reference-semantics, example:
 - “big” or “user-defined” types are “reference-types”
 - “small” or “primitive” types are “value-types”
- “Boxing” and “Unboxing” is the transition between a “reference” and a “value” (such as *to/from a primitive type*)
- Sometimes it is “unclear” whether you have a value-type or reference type

“False Dichotomy” between
“value-types” and “reference-types”
can make reasoning more complex

In C++, value or reference is
ALWAYS clear (through syntax)

Value Semantic Languages:

C
C++
Rust

Reference Semantic Languages:

C#, Cobol, D, Dart, Go, Groovy, Haskell, Java, Javascript, Kotlin, Lisp, Lua, Nim, Perl, Python, R, Ruby, Scala, Scheme, Simula, Smalltalk, SQL, Tcl, Typescript

False Dichotomy: “Value” and “Reference” Type

- A “Reference-Semantic” language introduces **separate semantics** for:
 - “**Value-types**”: Usually introduced for performance reasons, such as for primitive types
 - “**Reference-types**”: Usually introduced for language-semantic reasons, such as for garbage-collected or user-defined types

False Dichotomy: “Value” and “Reference” Type

- A “Reference-Semantic” language introduces **separate semantics** for:
 - **“Value-types”**: Usually introduced for performance reasons, such as for primitive types
 - **“Reference-types”**: Usually introduced for language-semantic reasons, such as for garbage-collected or user-defined types

Recall: Each thing that (*actually*) exists is one of:

1. **Data object value**
2. **Pointer value**

Using these values directly is “elegant”

False Dichotomy: “Value” and “Reference” Type

- A “Reference-Semantic” language introduces **separate semantics** for:
 - “**Value-types**”: Usually introduced for performance reasons, such as for primitive types
 - “**Reference-types**”: Usually introduced for language-semantic reasons, such as for garbage-collected or user-defined types

Recall: Each thing that (*actually*) exists is one of:

1. **Data object value**
2. **Pointer value**

Using these values directly is “elegant”

ISSUE:

Reference-languages introduce **(implicit) rules** (*specific to that language*) regarding **differing default behaviors** for types (*which are inconsistent, unclear, and do not translate to other reference languages*)

False Dichotomy: “Value” and “Reference” Type

- A “Reference-Semantic” language introduces **separate semantics** for:
 - “**Value-types**”: Usually introduced for performance reasons, such as for primitive types
 - “**Reference-types**”: Usually introduced for language-semantic reasons, such as for garbage-collected or user-defined types

Recall: Each thing that (*actually*) exists is one of:

1. **Data object value**
2. **Pointer value**

Using these values directly is “elegant”

ISSUE:

Reference-languages introduce **(implicit) rules** (*specific to that language*) regarding **differing default behaviors** for types (*which are inconsistent, unclear, and do not translate to other reference languages*)

```
new Integer(100) == new Integer(100)
// ...is false in Java
// ...is true in Scala (which is implemented in Java)
```

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

- Results in:
 1. **Inability to reason about logic** (e.g., *introduces ambiguities*)
 - What is the meaning of “=” and “==” ?
 - Is identity based on pointers (*like in Java?*)
 - Are arrays copied?
 - Are string values copied? (*If so, why not lists?*)
 - Are lists treated differently than strings? (*and why different?*)

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

- Results in:
 1. **Inability to reason about logic** (e.g., *introduces ambiguities*)
 - What is the meaning of “=” and “==” ?
 - Is identity based on pointers (*like in Java?*)
 - Are arrays copied?
 - Are string values copied? (*If so, why not lists?*)
 - Are lists treated differently than strings? (*and why different?*)
 2. **Inability to write consistent code**
 - Can you write a generic “swap” function that works for both value-types and reference types?
 - In Java, there is no such thing as “pointer to a primitive” (*some algorithms cannot be expressed*)

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

- Results in:
 1. **Inability to reason about logic** (e.g., *introduces ambiguities*)
 - What is the meaning of “=” and “==” ?
 - Is identity based on pointers (*like in Java?*)
 - Are arrays copied?
 - Are string values copied? (*If so, why not lists?*)
 - Are lists treated differently than strings? (*and why different?*)
 2. **Inability to write consistent code**
 - Can you write a generic “swap” function that works for both value-types and reference types?
 - In Java, there is no such thing as “pointer to a primitive” (*some algorithms cannot be expressed*)
 3. **Inability to reason about behavior, and performance**
 - C# **class** types live “in-the-heap”, but **struct** types live “on-the-stack”
 - Java: No choice, all classes are “in-the-heap”

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

- Results in:

1. **Inability to reason about logic** (e.g., *introduces ambiguities*)

- What is the meaning of “=” and “==” ?
- Is identity based on pointers (*like in Java?*)
- Are arrays copied?
- Are string values copied? (*If so, why not lists?*)
- Are lists treated differently than strings? (*and why different?*)

This is why **Reference Languages**
tend to not support “const”
(meaning is ambiguous)

2. **Inability to write consistent code**

- Can you write a generic “swap” function that works for both value-types and reference types?
- In Java, there is no such thing as “pointer to a primitive” (*some algorithms cannot be expressed*)

3. **Inability to reason about behavior, and performance**

- C# **class** types live “in-the-heap”, but **struct** types live “on-the-stack”
- Java: No choice, all classes are “in-the-heap”

Reference Semantics: Loss Of Ability

False Dichotomy between “Value” and “Reference” types introduces:
Differences based on language semantics (*not based on underlying reality*)

- Results in:

1. **Inability to reason about logic** (e.g., *introduces ambiguities*)

- What is the meaning of “=” and “==” ?
- Is identity based on pointers (*like in Java?*)
- Are arrays copied?
- Are string values copied? (*If so, why not lists?*)
- Are lists treated differently than strings? (*and why different?*)

This is why **Reference Languages**
tend to not support “const”
(meaning is ambiguous)

2. **Inability to write consistent code**

- Can you write a generic “swap” function that works for both value-types and reference types?
- In Java, there is no such thing as “pointer to a primitive” (*some algorithms cannot be expressed*)

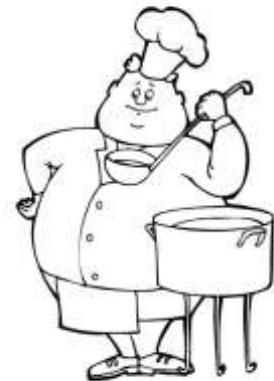
3. **Inability to reason about behavior, and performance**

- C# **class** types live “in-the-heap”, but **struct** types live “on-the-stack”
- Java: No choice, all classes are “in-the-heap”

Concurrency reasoning
can become very difficult!

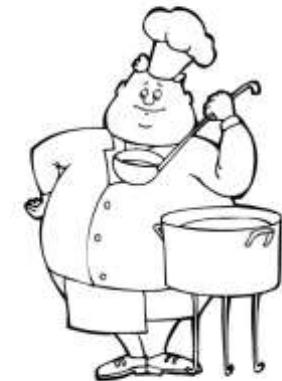
Side-Effects: Beware!

- Reference Semantics can increase computational density
 - You are not creating-and-destroying many objects, you are “re-using” the same objects
 - These objects are incrementally mutating
(*system is more complex, computational density increases*)
 - Mutated objects (*by definition*) are “side-effects”



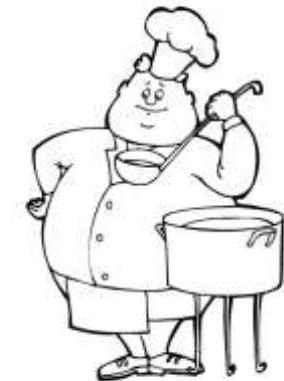
Side-Effects: Beware!

- Reference Semantics can increase computational density
 - You are not creating-and-destroying many objects, you are “re-using” the same objects
 - These objects are incrementally mutating
(*system is more complex, computational density increases*)
 - Mutated objects (*by definition*) are “side-effects”
- Motivation for Functional Programming: **No Side Effects!**
 - *Why?* Side-effects are hard to reason about



Side-Effects: Beware!

- Reference Semantics can increase computational density
 - You are not creating-and-destroying many objects, you are “re-using” the same objects
 - These objects are incrementally mutating
(*system is more complex, computational density increases*)
 - Mutated objects (*by definition*) are “side-effects”
- Motivation for Functional Programming: **No Side Effects!**
 - *Why?* Side-effects are hard to reason about



“Referential Transparency”: References to your objects are obvious, so you can reason about access and mutation

Ensures No Surprises
*from data objects being modified
“behind the scenes”*

Side-Effects: Beware!

- Reference Semantics can increase computational density
 - You are not creating-and-destroying many objects, you are “re-using” the same objects
 - These objects are incrementally mutating
(*system is more complex, computational density increases*)
 - Mutated objects (*by definition*) are “side-effects”
- Motivation for Functional Programming: **No Side Effects!**
 - *Why?* Side-effects are hard to reason about



“Referential Transparency”: References to your objects are obvious, so you can reason about access and mutation

Ensures No Surprises
from data objects being modified
“behind the scenes”

Due to (*lack of*) **Referential Transparency** (*concerns about “side-effects”*), Reference-Semantic Languages are harder to reason about than:

- Value-Semantic Languages
- Functional Programming Styles

C++ is a

1 → **general-purpose,**

2 → **multi-paradigm,**

3 → **systems-level**

4 → **strongly-typed,**

5 → **value-semantic**

language

C++ Uses Lexical Scoping

Deterministic creation, and destruction

**COOKING
WITH
YODA**

Cake mix leads to oven.
Oven leads to icing.
Icing leads to cake.

Most Object Lifetimes Are Short



- In practice, most object lifetimes are short
 - We form expressions, and author algorithms, that compute “temporary” values

Most Object Lifetimes Are Short



- In practice, **most object lifetimes are short**
 - We form expressions, and author algorithms, that compute “temporary” values
- We **tend to perform localized reasoning** because:
 - **It is “easier”** (*humans understand it*)
 - **It is “safer”** (*edge cases are more obvious*)
 - **It “composes better”** (*algorithms with localized context compose nicely into larger algorithms*)
 - **It “scales better”** (*resource contention is minimized or removed when context is explicit and minimal for accessing values; scarce resources can be allocated with knowledge of possible contention*)

Most Object Lifetimes Are Short



- In practice, **most object lifetimes are short**
 - We form expressions, and author algorithms, that compute “temporary” values
- We **tend to perform localized reasoning** because:
 - **It is “easier”** (*humans understand it*)
 - **It is “safer”** (*edge cases are more obvious*)
 - **It “composes better”** (*algorithms with localized context compose nicely into larger algorithms*)
 - **It “scales better”** (*resource contention is minimized or removed when context is explicit and minimal for accessing values; scarce resources can be allocated with knowledge of possible contention*)
- As a *BONUS*, the **compiler and CPU optimize and schedule execution** based on localized contexts
 - *Example:* Localized context defines the “**Load**” and “**Store**” of values to/from CPU registers to perform computation (**scope ends when value is evicted**)

Most Object Lifetimes Are Short



- In practice, **most object lifetimes are short**
 - We form expressions, and author algorithms, that compute “temporary” values
- We **tend to perform localized reasoning** because:
 - **It is “easier”** (*humans understand it*)
 - **It is “safer”** (*edge cases are more obvious*)
 - **It “composes better”** (*algorithms with localized context compose nicely into larger algorithms*)
 - **It “scales better”** (*resource contention is minimized or removed when context is explicit and minimal for accessing values; scarce resources can be allocated with knowledge of possible contention*)
- As a *BONUS*, the **compiler and CPU optimize and schedule execution** based on localized contexts
 - *Example:* Localized context defines the “**Load**” and “**Store**” of values to/from CPU registers to perform computation (**scope ends when value is evicted**)

Best Practice (Computer Science):

We encourage local reasoning of local values

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope: Data Object Availability

Scope: The binding of a name to a data object

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Scope: Data Object Availability

Scope: The binding of a name to a data object

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope must result in binding to **a valid object**
(or badness occurs, in any language!)

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

- Scope includes aspects of:
 - **What is the lifetime of the data object?**
 - **How is access to the data object resolved?**

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope must result in binding to **a valid object**
(or badness occurs, in any language!)

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

- Scope includes aspects of:
 - **What is the lifetime of the data object?**
 - **How is access to the data object resolved?**

... Dynamic Scope

```
{  
    float f = r * 3.14;  
}
```

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope must result in binding to **a valid object** (or badness occurs, in any language!)

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

- Scope includes aspects of:
 - What is the lifetime of the data object?
 - How is access to the data object resolved?

... Dynamic Scope

```
{  
    float f = r * 3.14;  
}
```

What is 'r'?

Dynamically resolve 'r'!

Example: In high-level language, look through “call-stack-of-named-objects” for a value named ‘r’!

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope must result in binding to **a valid object** (or badness occurs, in any language!)

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

- Scope includes aspects of:
 - **What is the lifetime of the data object?**
 - **How is access to the data object resolved?**

... Dynamic Scope

```
{  
    float f = r * 3.14;  
}
```

What is 'r'?

Dynamically resolve 'r'!

Example: In high-level language, look through “call-stack-of-named-objects” for a value named ‘r’!

... Static Scope

```
{ // outer-scope  
float r = 12.5;  
{ // inner-scope  
    float f = r * 3.14;  
}  
}
```

Scope: Data Object Availability

Scope: The binding of a name to a data object

Scope must result in binding to **a valid object** (or badness occurs, in any language!)

1

Dynamic Scope: Names are resolved dynamically when the function executes

2

Static Scope: Inner functions contain the scope of parent functions

Also called:
Lexical Scope

- Scope includes aspects of:
 - **What is the lifetime of the data object?**
 - **How is access to the data object resolved?**

... Dynamic Scope

```
{  
    float f = r * 3.14;  
}
```

What is 'r'?

Dynamically resolve 'r'!

Example: In high-level language, look through “call-stack-of-named-objects” for a value named ‘r’!

... Static Scope

```
{ // outer-scope  
float r = 12.5;  
{ // inner-scope  
    float f = r * 3.14;  
}
```

**'r' is defined
in an outer-scope**

**Inner-scope “sees” the ‘r’
defined in the outer-scope**

Lexical Scope: The Local (*Lexical*) Environment

- In Lexical Scoping, a name refers to a data object in the local (*lexical*) environment

Lexical Scope: The Local (*Lexical*) Environment

- In Lexical Scoping, a name refers to a data object in the local (*lexical*) environment
 - Also called “Static Scoping”: Data object is bound through analysis of static program text

Lexical Scope: The Local (*Lexical*) Environment

- In Lexical Scoping, a name refers to a data object in the local (*lexical*) environment
 - Also called “Static Scoping”: Data object is bound through analysis of static program text



Static scoping allows the programmer to reason about object references such as parameters, variables, constants, types, functions, etc. as simple name substitutions. This makes it much easier to make modular code and reason about it, since the local naming structure can be understood in isolation. In contrast, dynamic scope forces the programmer to anticipate all possible dynamic contexts in which the module's code may be invoked.

[https://en.wikipedia.org/wiki/Scope_\(computer_science\)#Lexical_scoping](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping)
– Wikipedia

Dynamic Scoping
Cannot scale!



Lexical Scope: The Local (*Lexical*) Environment

- In Lexical Scoping, a name refers to a data object in the local (*lexical*) environment
 - Also called “Static Scoping”: Data object is bound through analysis of static program text

Static scoping allows the programmer to reason about object references such as parameters, variables, constants, types, functions, etc. as simple name substitutions. **This makes it much easier to make modular code and reason about it**, since the local naming structure can be understood in isolation. In contrast, dynamic scope forces the programmer to anticipate all possible dynamic contexts in which the module's code may be invoked.

[https://en.wikipedia.org/wiki/Scope_\(computer_science\)#Lexical_scoping](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping)
– Wikipedia

Dynamic Scoping
Cannot scale!

Lexical Scoping is standard in:

- All ALGOL-based languages (*Pascal, Modula2, Ada*)
- Modern functional languages (*ML, Haskell*)
- C-style languages (*C, C++*)

Powerful C++ Tool: Lexical Scope

- C++ Lexical Scoping binds data objects at compile-time
(through analysis of static program text)

With C++ Lexical Scope,
object lifetime access
is never undefined

Powerful C++ Tool: Lexical Scope

- C++ Lexical Scoping binds data objects at compile-time (*through analysis of static program text*)
- Lots of languages rely upon Lexical Scoping. *HOWEVER*, very few languages have deterministic destructors (dtors) to provide VERY POWERFUL idioms for resource management
 - Destructor is (*deterministically!*) invoked upon exiting scope
 - Example C++ Idiom: RAII (*Resource Acquisition Is Initialization*)

With C++ Lexical Scope,
object lifetime access
is never undefined

Powerful C++ Tool: Lexical Scope

- C++ Lexical Scoping binds data objects at compile-time (*through analysis of static program text*)
- Lots of languages rely upon Lexical Scoping. *HOWEVER*, very few languages have deterministic destructors (dtors) to provide VERY POWERFUL idioms for resource management
 - Destructor is (*deterministically!*) invoked upon exiting scope
 - Example C++ Idiom: RAII (*Resource Acquisition Is Initialization*)

With C++ Lexical Scope,
object lifetime access
is never undefined

```
...
// 'my_lock' NEVER exists here
{
    MyResourceLock my_lock; // Create my_lock
                           // in this scope
    { // 'my_lock' is still visible
        ... // do stuff safely (lock in-place)
    }
}
// 'my_lock' NEVER exists here
```

Scope for **my_lock** { }

Powerful C++ Tool: Lexical Scope

- C++ Lexical Scoping binds data objects at compile-time (*through analysis of static program text*)
- Lots of languages rely upon Lexical Scoping. *HOWEVER*, very few languages have deterministic destructors (dtors) to provide VERY POWERFUL idioms for resource management
 - Destructor is (*deterministically!*) invoked upon exiting scope
 - Example C++ Idiom: RAII (*Resource Acquisition Is Initialization*)

With C++ Lexical Scope,
object lifetime access
is never undefined

The diagram shows a code snippet with annotations:

```
...  
// 'my_lock' NEVER exists here  
{  
    MyResourceLock my_lock; // Create my_lock in this scope  
    { // 'my_lock' is still visible  
        ... // do stuff safely (lock in-place)  
    }  
}  
// 'my_lock' NEVER exists here
```

A red bracket on the left labeled "Scope for `my_lock`" spans from the opening brace of the innermost block to the closing brace of the outermost block. A red circle highlights the identifier `my_lock` in the declaration `MyResourceLock my_lock;`. A red callout box points to this circle with the text "Create `my_lock` in this scope". Another red callout box at the bottom right points to the closing brace of the outermost block with the text "`my_lock` dtor DETERMINISTICALLY invoked! (Is C++ Guarantee, Can Reason About This!)".

Powerful C++ Tool: Lexical Scope

- C++ Lexical Scoping binds data objects at compile-time (*through analysis of static program text*)
- Lots of languages rely upon Lexical Scoping. *HOWEVER*, very few languages have deterministic destructors (dtors) to provide VERY POWERFUL idioms for resource management
 - Destructor is (*deterministically!*) invoked upon exiting scope
 - Example C++ Idiom: RAII (*Resource Acquisition Is Initialization*)

With C++ Lexical Scope,
object lifetime access
is never undefined

```
...  
// 'my_lock' NEVER exists here  
{  
    MyResourceLock my_lock; // 'my_lock' NEVER exists here  
    { // 'my_lock' is still visible  
        ... // do stuff safely (lock in-place)  
    }  
}  
// 'my_lock' NEVER exists here
```

Scope for **my_lock**

Create **my_lock** in this scope

my_lock dtor DETERMINISTICALLY invoked!
(Is C++ Guarantee, Can Reason About This!)

C++ Best Practice:
Leverage Lexical Scope
(and the destructor)
whenever possible!

Safe!
Deterministic!
Never Undefined!

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

Destructor Is Invoked Upon Scope-exit

- Is Deterministic (C++ Guarantee! Can Reason About This!)
- Provides powerful idioms (example: RAII)

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

Destructor Is Invoked Upon Scope-exit

- Is Deterministic (C++ Guarantee! Can Reason About This!)
- Provides powerful idioms (example: RAII)

NO OTHER LANGUAGE
provides this guarantee!

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

Destructor Is Invoked Upon Scope-exit

- Is Deterministic (C++ Guarantee! Can Reason About This!)
- Provides powerful idioms (example: RAII)

NO OTHER LANGUAGE
provides this guarantee!

Better Local Reasoning

- Consistent, predictable, and obvious runtime behavior (*all is resolved at compile-time*)
- Programmer understands local context in isolation from external context
- Referential Transparency (*no surprises from data objects being modified behind-the-scenes*)
- Improved concurrency (*local context is independent from other threads*)

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

Destructor Is Invoked Upon Scope-exit

- Is Deterministic (C++ Guarantee! Can Reason About This!)
- Provides powerful idioms (example: RAII)

NO OTHER LANGUAGE
provides this guarantee!

Better Local Reasoning

- Consistent, predictable, and obvious runtime behavior (*all is resolved at compile-time*)
- Programmer understands local context in isolation from external context
- Referential Transparency (*no surprises from data objects being modified behind-the-scenes*)
- Improved concurrency (*local context is independent from other threads*)

Better Optimization

- Resolved at compile-time (*no runtime overhead*)
- Lexical scope keeps everything on the stack (*almost zero overhead for allocation and deletion*)
- Efficient scheduling (*compiler optimizes knowing exact lifetime of resources for the computation*)
- No referencing aliasing problems (*no dependency ambiguities to defeat the compiler/CPU optimizer*)

Benefits Of Lexical Scope In C++

Is Totally Safe™

- No memory management issues
- Never a dangling reference
- Cannot refer to an object not in scope

A tool **SO POWERFUL**,
you should use it **EVERY**
CHANCE YOU GET

Destructor Is Invoked Upon Scope-exit

- Is Deterministic (C++ Guarantee! Can Reason About This!)
- Provides powerful idioms (example: RAII)

NO OTHER LANGUAGE
provides this guarantee!

Better Local Reasoning

- Consistent, predictable, and obvious runtime behavior (*all is resolved at compile-time*)
- Programmer understands local context in isolation from external context
- Referential Transparency (*no surprises from data objects being modified behind-the-scenes*)
- Improved concurrency (*local context is independent from other threads*)

Better Optimization

- Resolved at compile-time (*no runtime overhead*)
- Lexical scope keeps everything on the stack (*almost zero overhead for allocation and deletion*)
- Efficient scheduling (*compiler optimizes knowing exact lifetime of resources for the computation*)
- No referencing aliasing problems (*no dependency ambiguities to defeat the compiler/CPU optimizer*)

C++ is a

1 → **general-purpose,**

2 → **multi-paradigm,**

4 → **strongly-typed,**

5 → **value-semantic,**

3 → **systems-level language**

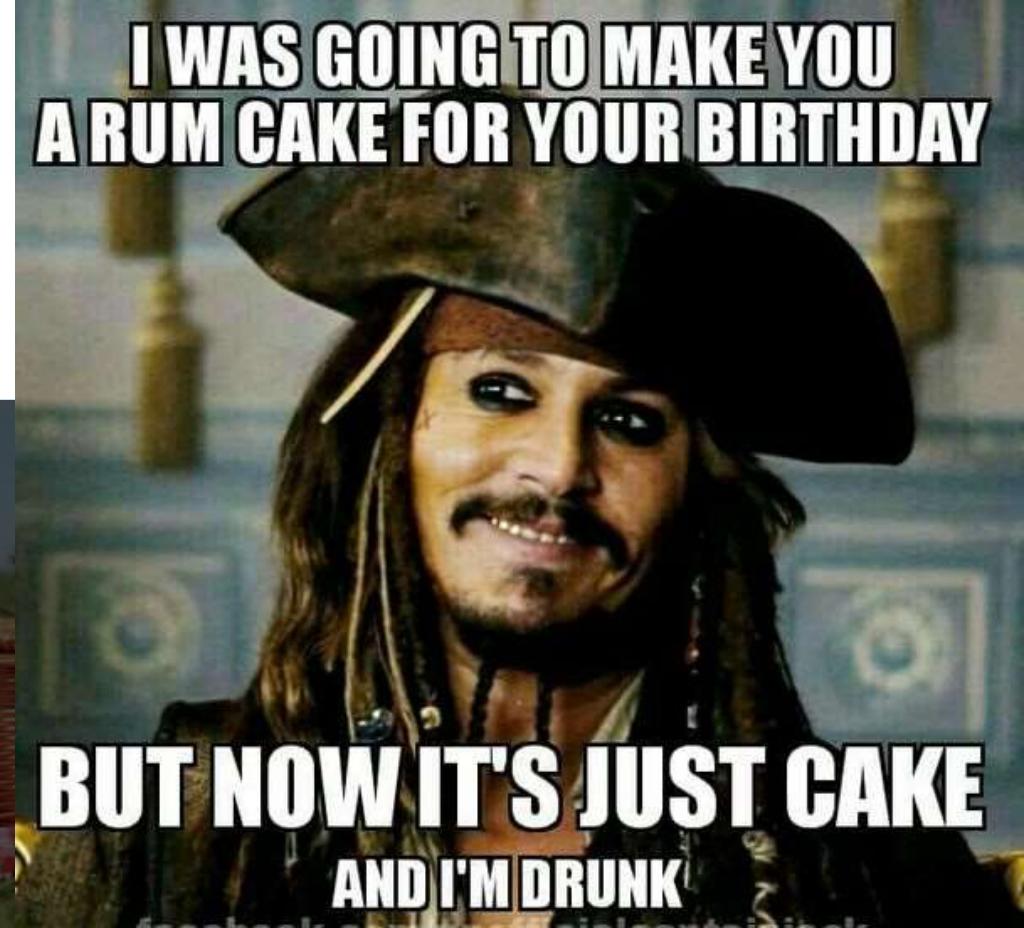
with **lexical scoping** 6

C++ is a
general-purpose,
multi-paradigm,
strongly-typed,
value-semantic,
systems-level language
with lexical scoping
and deterministic destruction

The diagram consists of seven red arrows pointing from numbers 1 through 7 to specific words in the text. 1 points to 'general-purpose'. 2 points to 'multi-paradigm'. 3 points to 'systems-level'. 4 points to 'strongly-typed'. 5 points to 'value-semantic'. 6 points to 'lexical scoping'. 7 points to 'deterministic destruction'.



C++ Uses A Single-Pass Compiler



C++ Supports Incremental Compilation

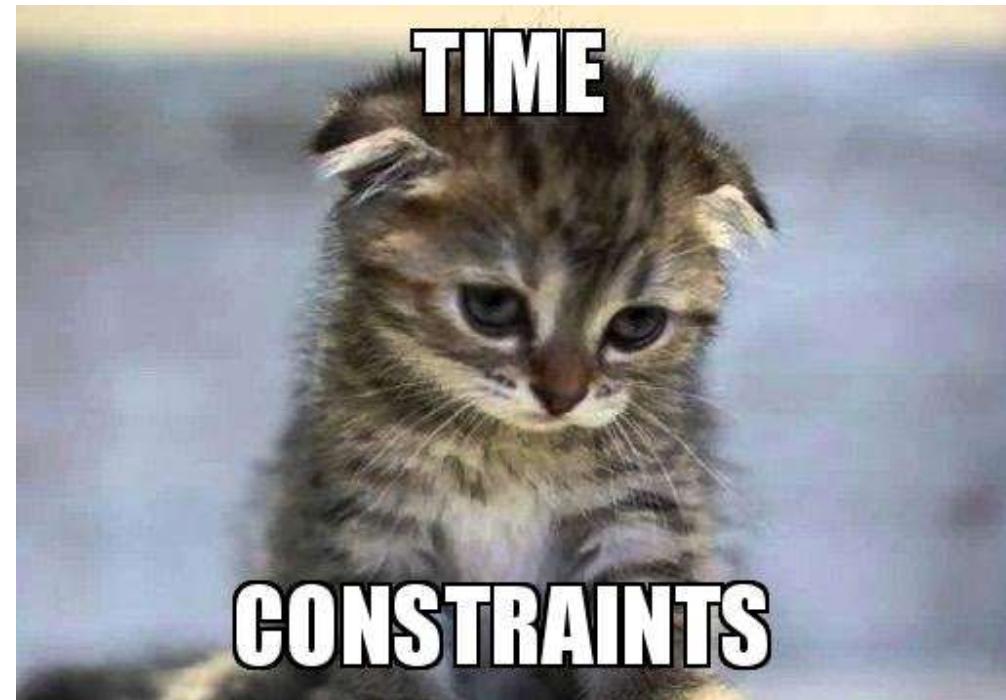
C++ Uses A Single-Pass Compiler

- *Supported: Incremental Compilation*

C++ Uses A Single-Pass Compiler

- Supported: Incremental Compilation

*Content
Removed
Due To
Time Constraints*



C++ is a

1 general-purpose,
2 multi-paradigm,
4 strongly-typed,
5 value-semantic,
3 systems-level language
with lexical scoping,
6 deterministic destruction,
7 and a single-pass compiler
8

C++ is a

1 → general-purpose,
2 → multi-paradigm,
4 → strongly-typed,
5 → value-semantic,
3 → systems-level language
with lexical scoping,
deterministic destruction,
and a single-pass compiler

This is **ALL GOOD STUFF**
that we *KNOW* to be
Computer Science
Best Practice!
(We would not want it any other way!)

Cooking With C++



Serving Something Good, And Great!

Do You Eat?

- What are the *requirements* for what you consume?

Do You Eat?

- What are the *requirements* for what you consume?
 - Minimal: Does not kill you



Do You Eat?

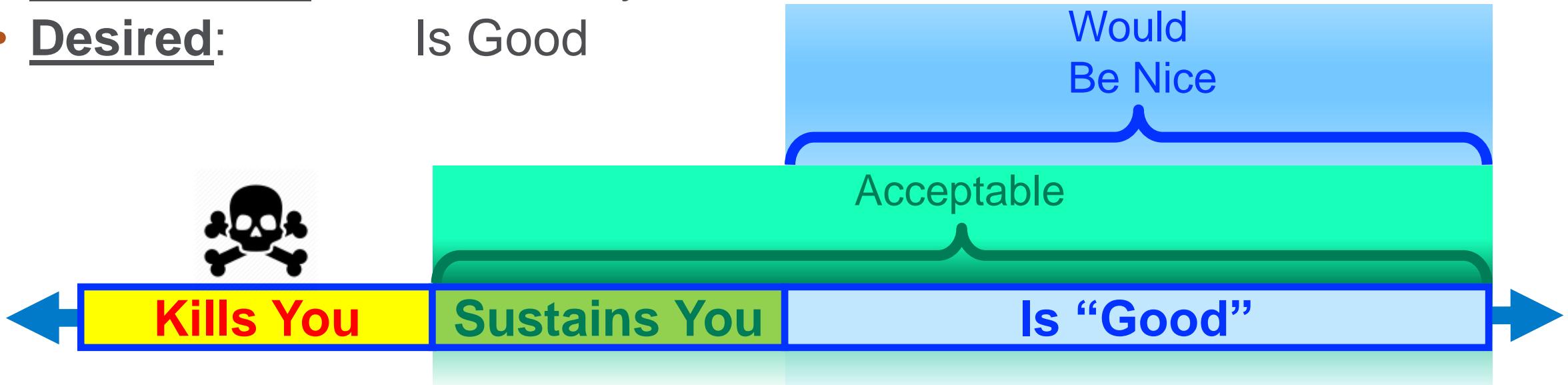
- What are the *requirements* for what you consume?
 - **Minimal**: Does not kill you
 - **Reasonable**: Sustains you



Do You Eat?

- What are the *requirements* for what you consume?

- Minimal: Does not kill you
- Reasonable: Sustains you
- Desired: Is Good



Do You Eat?

- What are the *requirements* for what you consume?
 - Minimal: Does not kill you
 - Reasonable: Sustains you
 - Desired: Is Good
 - Would-be-nice: Is Great



Do You Eat?

- What are the *requirements* for what you consume?
 - Minimal: Does not kill you
 - Reasonable: Sustains you
 - Desired: Is Good
 - Would-be-nice: Is Great



A “Cook” gives us what we need.
A “Chef” may get “*fancy with the spices*”
...sometimes nice, but usually not essential.

Cooking With C++

Cook

Chef



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)

Chef



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)

Chef



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans

Chef



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Chef



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Chef

Software Metaphor:
“Re-Uses” existing libraries



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Chef

- May be Unconventional (*e.g., “cultural fusion”*)



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Chef

- May be Unconventional (*e.g., “cultural fusion”*)
- Tries new styles



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Chef

- May be Unconventional (*e.g., “cultural fusion”*)
- Tries new styles
- Experiments with novel or unexpected combinations



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Makes
Good Food!

Chef

- May be Unconventional (*e.g., “cultural fusion”*)
- Tries new styles
- Experiments with novel or unexpected combinations



Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Makes
Good Food!



Chef

- May be Unconventional (*e.g., “cultural fusion”*)
- Tries new styles
- Experiments with novel or unexpected combinations

Is VERY difficult!

Often:

- Is Surprising
- Is Unconventional
- Success has significant overlap with similar failed efforts

Cooking With C++

Cook

- Is Conventional (*proven styles, cuisines*)
- May follow a recipe (*perhaps with modifications and substitutions*)
- Uses stuff from cans
- May use and combine pseudo-prepared foods

Software Metaphor:
“Re-Uses” existing libraries

Makes
Good Food!

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)



Chef

- May be Unconventional (*e.g., “cultural fusion”*)
- Tries new styles
- Experiments with novel or unexpected combinations

Is VERY difficult!

Often:

- Is Surprising
- Is Unconventional
- Success has significant overlap with similar failed efforts

Cooking With C++

Cook

Chef

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook

Chef

Makes Good Food!

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>
<u>Predictable</u> : Tends to not deviate “too-far” from proven success	

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>
Predictable: Tends to not deviate “too-far” from proven success	Unpredictable: Experiments with radical new ideas

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>
<u>Predictable</u> : Tends to not deviate “too-far” from proven success	<u>Unpredictable</u> : Experiments with radical new ideas
<u>Tends to not fail</u> : Consistent adherence to proven principles	

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>
<u>Predictable:</u> Tends to not deviate “too-far” from proven success	<u>Unpredictable:</u> Experiments with radical new ideas
<u>Tends to not fail:</u> Consistent adherence to proven principles	<u>May fail SPECTACULARLY:</u> Is <i>discovering</i> the behavior envelope associated with new patterns

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Cooking With C++

Cook	Chef
Makes <u>Good Food!</u>	Can Make <u>Great Food!</u>
Predictable: Tends to not deviate “too-far” from proven success	Unpredictable: Experiments with radical new ideas
Tends to not fail: Consistent adherence to proven principles	May fail SPECTACULARLY: Is <i>discovering</i> the behavior envelope associated with new patterns

Most People Are Cooks.
(Are You A Cook, Or A Chef?)
(Probably, Both!)

Is **VERY difficult!**

- Often:
- Is **Surprising**
 - Is **Unconventional**
 - **Success has significant overlap with similar failed efforts**

You Cannot Be A Chef

Unless You
First Become A Cook

Becoming A C++ Cook:

No Matter Your Aspiration, a solid basis is first **required**:

Becoming A C++ Cook:

No Matter Your Aspiration, a solid basis is first required:

1

C++ Type System

Becoming A C++ Cook:

No Matter Your Aspiration, a solid basis is first required:

1

C++ Type System

2

Value vs. Reference Semantics

Becoming A C++ Cook:

No Matter Your Aspiration, a solid basis is first required:

1 C++ Type System

2 Value vs. Reference Semantics

3 Lexical Scoping

Becoming A C++ Cook:

No Matter Your Aspiration, a solid basis is first required:

1 C++ Type System

2 Value vs. Reference Semantics

3 Lexical Scoping

Most C++ Developers
Need Nothing Else!
(...or, “almost nothing else”)

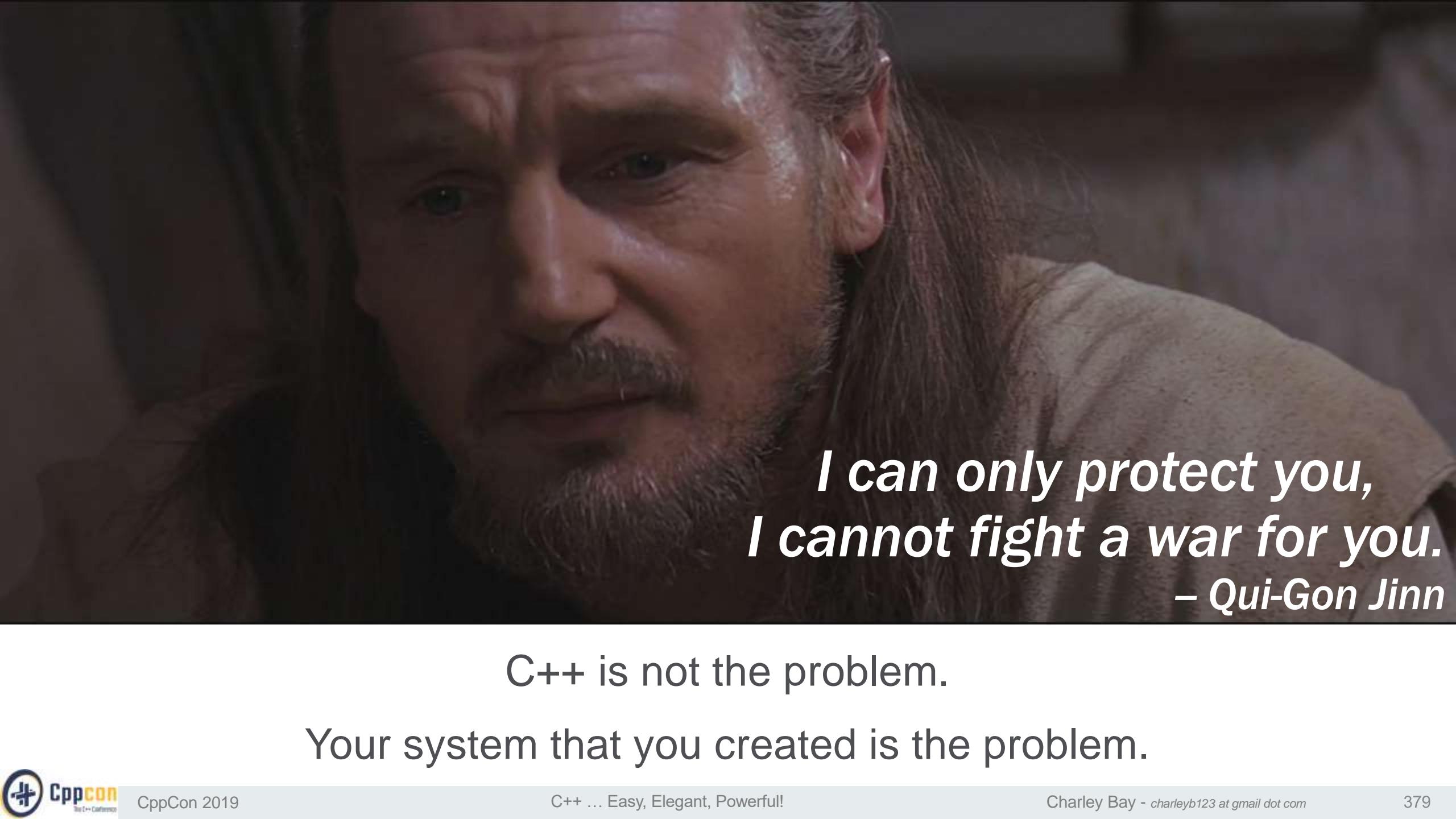
Conclusion

*The Problem is Your System
(C++ can only help you, not fix it)*



A close-up portrait of Qui-Gon Jinn from Star Wars. He has long brown hair and a full brown beard. He is looking slightly to his left with a serious expression.

*I can only protect you,
I cannot fight a war for you.*
– Qui-Gon Jinn

A close-up portrait of Qui-Gon Jinn from Star Wars. He has long brown hair and a full brown beard. He is looking slightly to the right with a serious expression.

*I can only protect you,
I cannot fight a war for you.*
– Qui-Gon Jinn

C++ is not the problem.

Your system that you created is the problem.

Taking Control ↗ software

- Taking Control in your ~~C++~~ system is usually through:

Your Object Model

1. What objects exist?

- You define the types that exist
(and rules for those types)
- You instantiate objects

2. Who owns a given object?



Your Control Flow

1. Your Object Lifecycle

- What causes objects to be created, mutated, and destroyed?

Summary

1

It's easier to think in C++

- It's how the underlying hardware works (so you can reason about what is going on)
- Surprises in behavior can be explained and understood (because your thinking is aligned with how the system is actually running)

Summary

1

It's easier to think in C++

- It's how the underlying hardware works (*so you can reason about what is going on*)
- Surprises in behavior can be explained and understood (*because your thinking is aligned with how the system is actually running*)

2

It's more efficient to think in C++

- Resource spending is explicit (*nothing is spent “behind your back”*)
- Sometimes you get stuff “for free” (*from C++ optimizations based on your “intent”*)

Summary

1

It's easier to think in C++

- It's how the underlying hardware works (*so you can reason about what is going on*)
- Surprises in behavior can be explained and understood (*because your thinking is aligned with how the system is actually running*)

2

It's more efficient to think in C++

- Resource spending is explicit (*nothing is spent “behind your back”*)
- Sometimes you get stuff “for free” (*from C++ optimizations based on your “intent”*)

3

C++ gives you the tools needed to do your job

- “Best Practice” for Computer Science (*strong typing, value semantics, lexical scope*)
- Deterministic destruction (*no other language provides this!*)

Summary

1

It's easier to think in C++

- It's how the underlying hardware works (*so you can reason about what is going on*)
- Surprises in behavior can be explained and understood (*because your thinking is aligned with how the system is actually running*)

2

It's more efficient to think in C++

- Resource spending is explicit (*nothing is spent “behind your back”*)
- Sometimes you get stuff “for free” (*from C++ optimizations based on your “intent”*)

3

C++ gives you the tools needed to do your job

- “Best Practice” for Computer Science (*strong typing, value semantics, lexical scope*)
- Deterministic destruction (*no other language provides this!*)

Other languages *interfere* with your ability to
think, reason about behavior, and express design intent.
(*How can you possibly deliver anything reliably at scale with them?*)



Cppcon
The C++ Conference

*Thank you
for coming!*

