

Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 4 – Heap Memory Management

Thomas Corbat / Felix Morgner
Rapperswil, 21.03.2023
FS2023



OST

Ostschweizer
Fachhochschule

- **Topics:**

- Recap Week 3
- Pointers
- Reading Declarations Correctly
- Heap Memory (De)Allocation

- You understand how raw pointers work in C++
- You can read declarations correctly
- You can use basic heap memory management when necessary

Recap Week 3



- **ParamType is a value/pointer type**

- <expr> is a reference type: ignore the reference
- Ignore const of <expr> type (outermost)
- Pattern match <expr>'s type against ParamType to figure out T

```
template <typename T>  
auto f(ParamType param) -> void;
```

- **ParamType is a reference**

- <expr> is a reference type: ignore the reference
- Pattern match <expr>'s type against ParamType to figure out T

- **ParamType is a forwarding reference (T&& / auto&&)**

- <expr> is an lvalue: T and ParamType become lvalue references!
- Otherwise (if <expr> is an rvalue): Rules for pointer/references apply

- We need something that is aware of the actual template parameter type
- Recap from Forwarding References: We know whether param was an lvalue or an rvalue

```
int      x    = 23;  
int const cx  = x;  
int const& crx = x;
```

```
log_and_do(x)           -> T = int&  
log_and_do(cx)          -> T = int const&  
log_and_do(crx)         -> T = int const&  
log_and_do(27)          -> T = int  
log_and_do(std::move(x)) -> T = int
```

} lvalues
} rvalues

- If T is of reference type we need to pass an lvalue otherwise we need to pass an rvalue
- How can we do it?

■ std::forward

```
template <typename T>  
auto log_and_do(T&& param) -> void {  
    //log  
    do_something(std::forward<T>(param));  
}
```

- **What does `std::forward` do?**

- It's a "conditional" cast to an rvalue reference...
- This allows arguments to be treated as what they originally were (lvalue or rvalue references)

- **Implementation is similar to the following (there is also an overload for rvalue references):**

```
template<typename T>
decltype(auto) forward(std::remove_reference_t<T>& param) {
    return static_cast<T&&>(param);
}
```

- **If T is of value type, T && is an rvalue reference in the return expression**
- **If T is of lvalue reference type, the resulting type is an rvalue reference to an lvalue reference**
 - Example: if T = int & then T && would mean int & &&
- **What is <Type> & && supposed to mean?**
 - As you know from reference collapsing: <Type> &

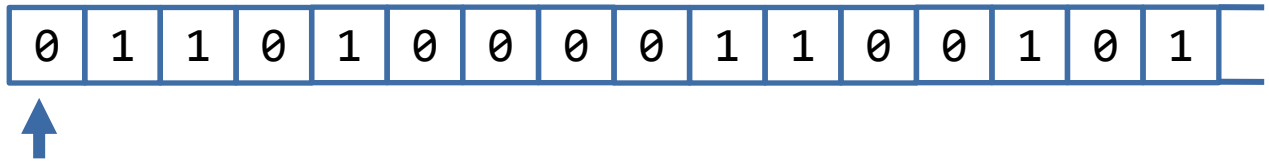
Pointers

The pointers, memory locations and sizes in this lecture are freely invented!



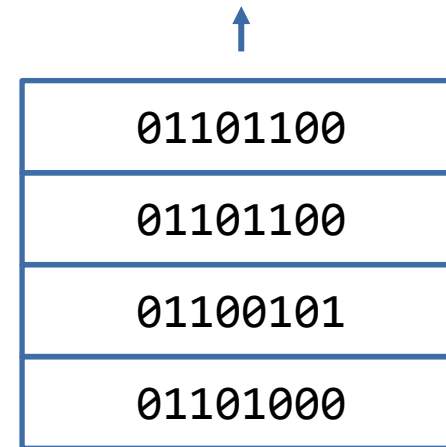
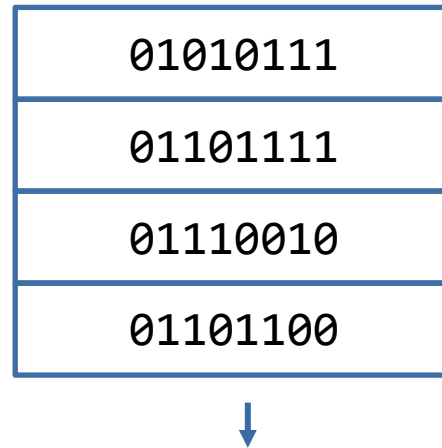
- **Turing Machine (Model)**

- Unlimited Space



- **Stack / Heap (Reality)**

- Limited Space

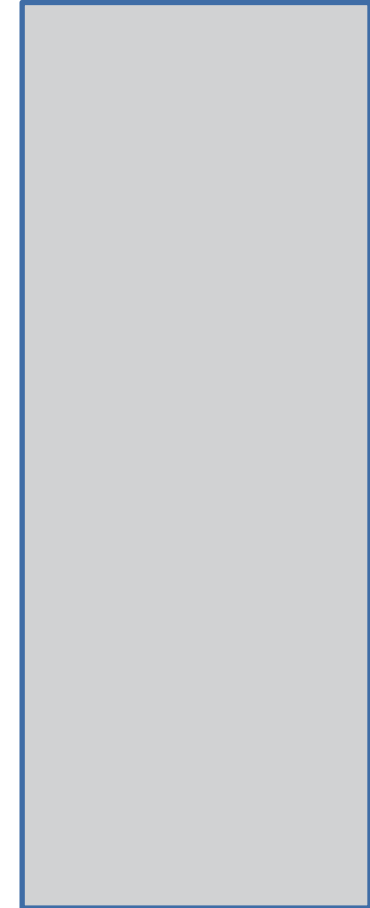


- **Deterministic**
- **Local variables get deleted automatically upon leaving their scope**

```
auto foo() -> void {  
    int i{5};  
    {  
        double d = 23.0;  
    }  
}  
  
auto main() -> int {  
    ...  
    foo();  
    ...  
}
```

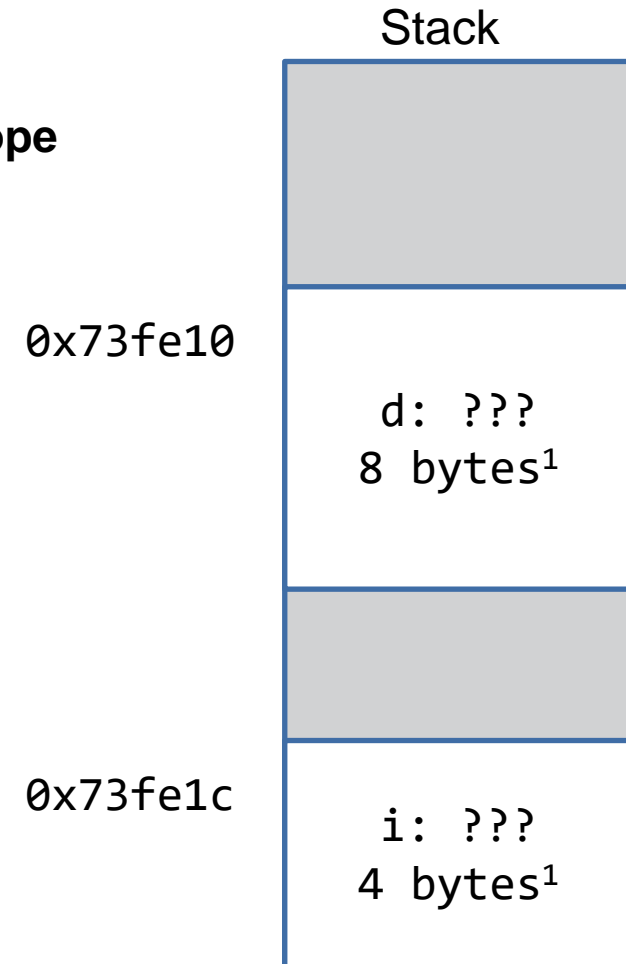


Stack



- Deterministic
- Local variables get deleted automatically upon leaving their scope

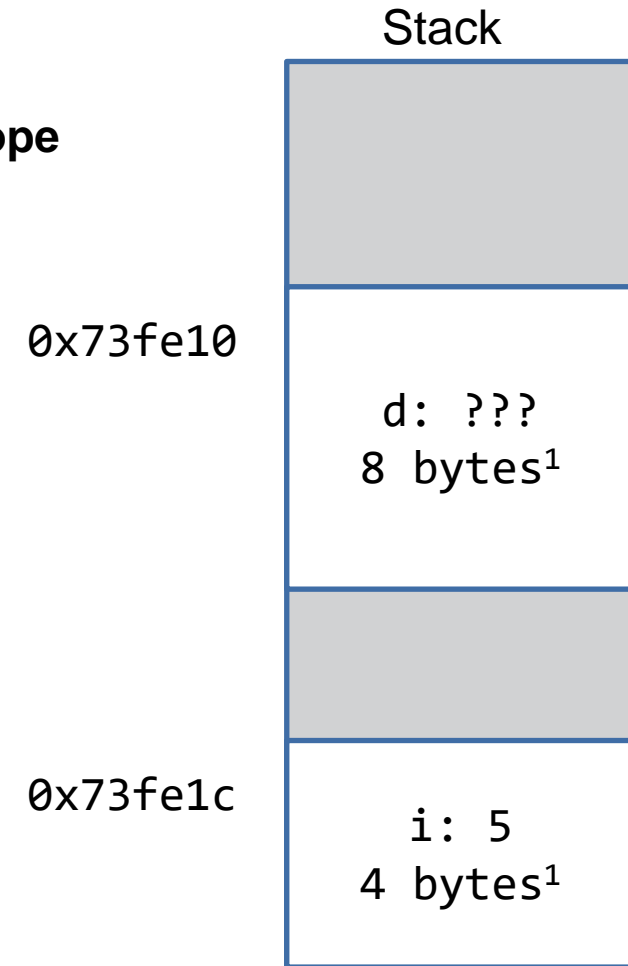
```
→ auto foo() -> void {  
    int i{5};  
    {  
        double d = 23.0;  
    }  
}  
  
auto main() -> int {  
    ...  
    foo();  
    ...  
}
```



¹ Sizes depend on the platform

- Deterministic
- Local variables get deleted automatically upon leaving their scope

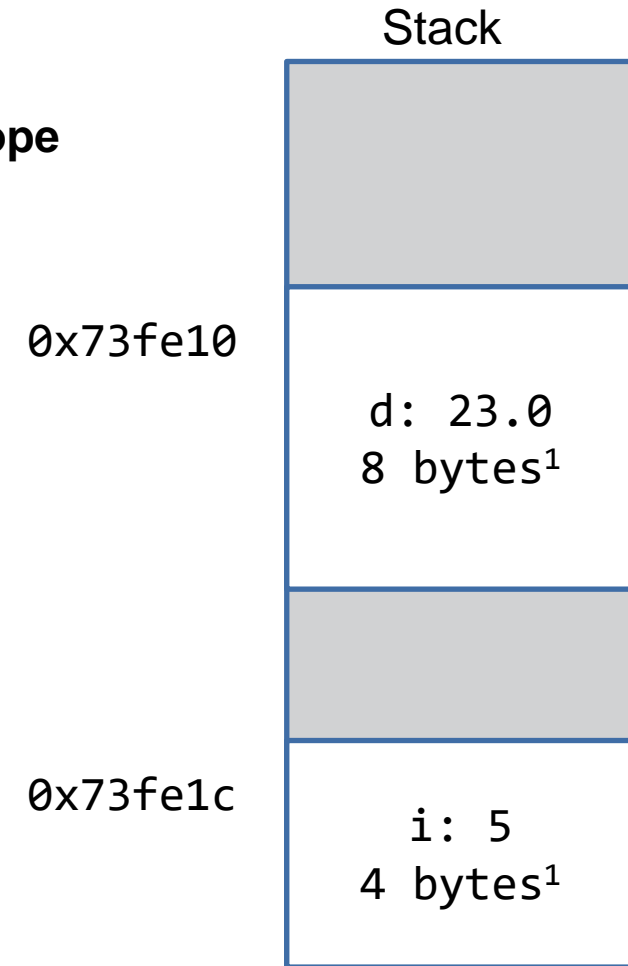
```
→ auto foo() -> void {  
    int i{5};  
    {  
        double d = 23.0;  
    }  
}  
  
auto main() -> int {  
    ...  
    foo();  
    ...  
}
```



¹ Sizes depend on the platform

- Deterministic
- Local variables get deleted automatically upon leaving their scope

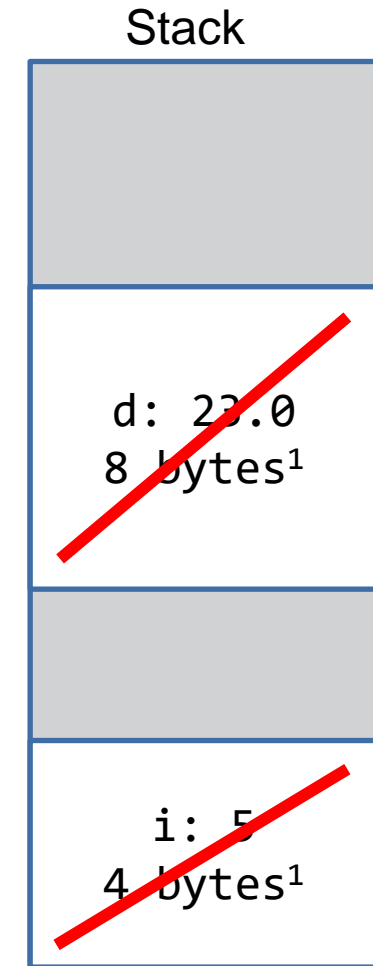
```
→ auto foo() -> void {  
    int i{5};  
    {  
        double d = 23.0;  
    }  
}  
  
auto main() -> int {  
    ...  
    foo();  
    ...  
}
```



¹ Sizes depend on the platform

- Deterministic
- Local variables get deleted automatically upon leaving their scope

```
→ auto foo() -> void {  
    int i{5};  
    {  
        double d = 23.0;  
    }  
}  
  
auto main() -> int {  
    ...  
    foo();  
    ...  
}
```



¹ Sizes depend on the platform

- Deterministic (too)
- Creation and deletion happens explicitly



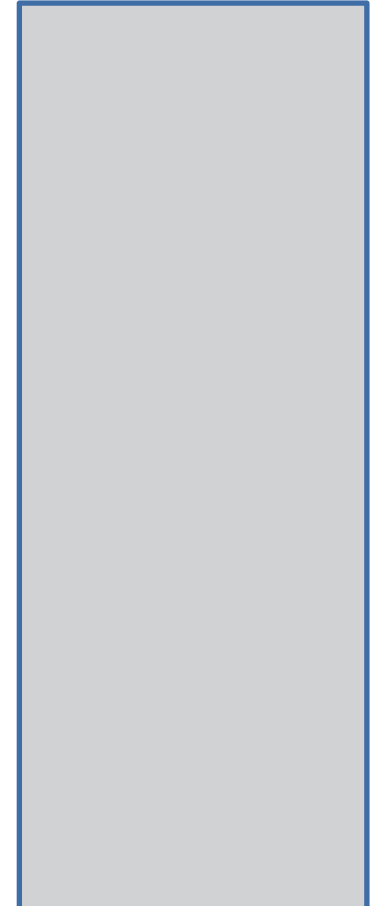
```
auto foo() -> void {  
    auto ip = new int{5};  
    ...  
    delete ip;  
}
```

0x73fe08

Stack



Heap

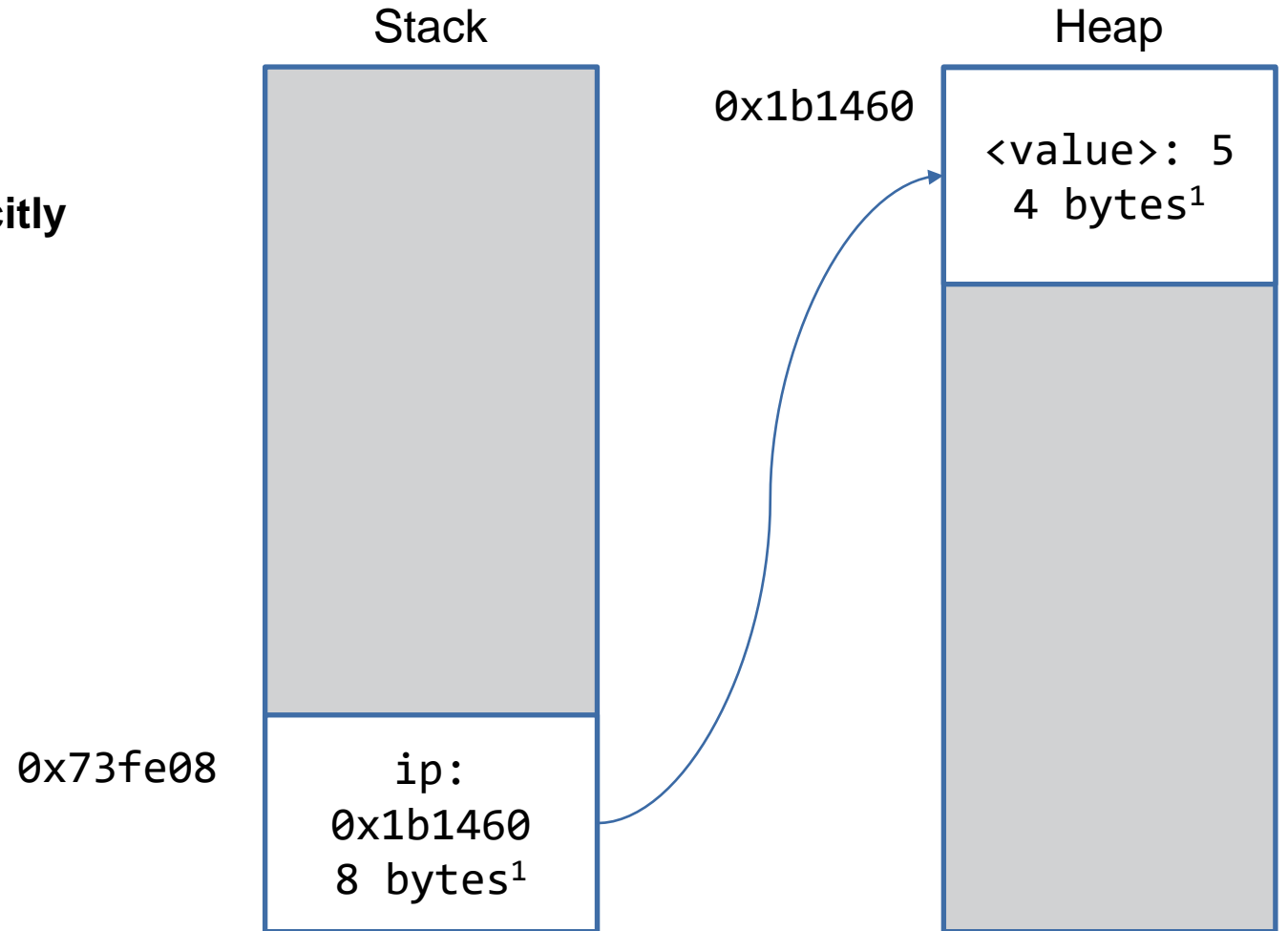


¹ Sizes depend on the platform

- Deterministic (too)
- Creation and deletion happens explicitly



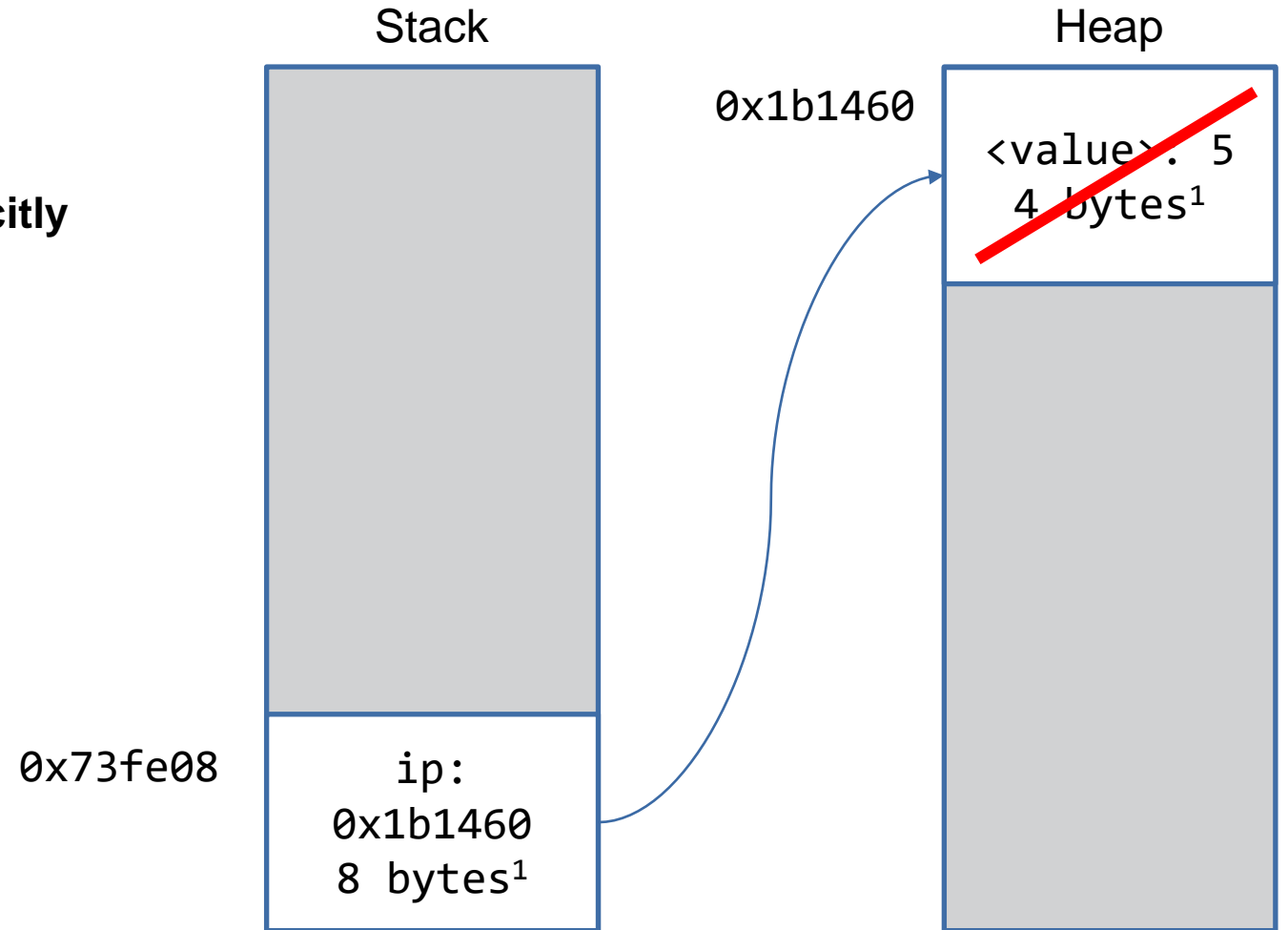
```
auto foo() -> void {  
    auto ip = new int{5};  
    ...  
    delete ip;  
}
```



¹ Sizes depend on the platform

- Deterministic (too)
- Creation and deletion happens explicitly

```
auto foo() -> void {  
    auto ip = new int{5};  
    ...  
    delete ip;  
}
```

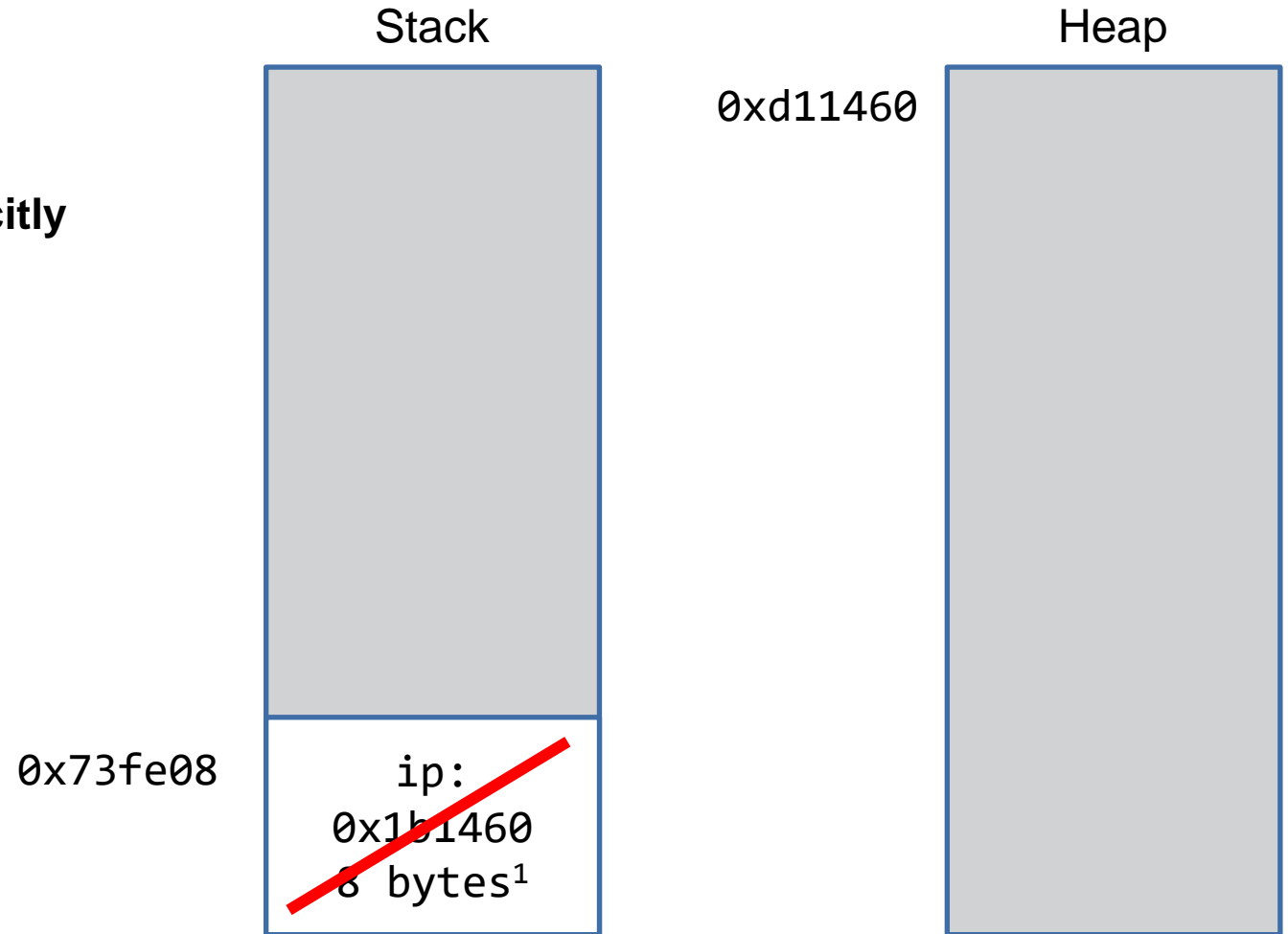


¹ Sizes depend on the platform

- Deterministic (too)
- Creation and deletion happens explicitly

→

```
auto foo() -> void {  
    auto ip = new int{5};  
    ...  
    delete ip;  
}
```



¹ Sizes depend on the platform

- What is the type of ip in the example?

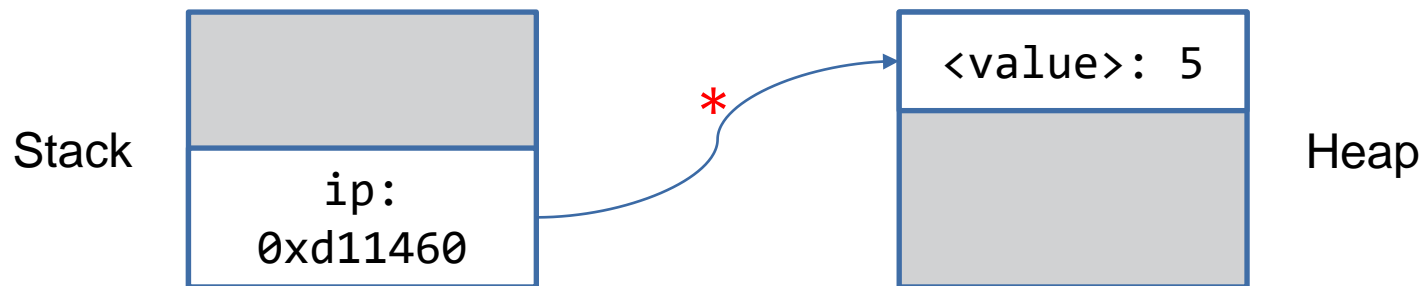
```
auto ip = new int{5};
```

- Explicit declaration

```
int * ip = new int{5};
```

- Accessing the value at a pointer

```
int v = * ip; //v == 5
```



- What is the type of arr in the example?

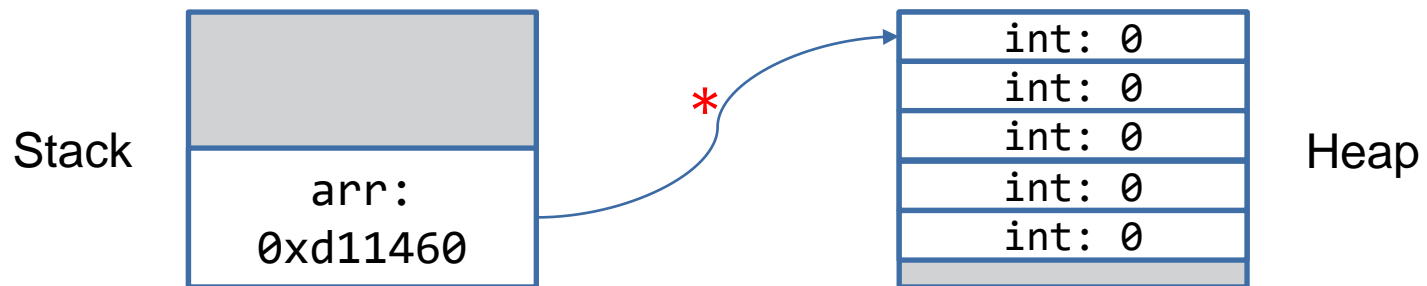
```
auto arr = new int[5]{};
```

- Explicit declaration

```
int * arr = new int[5]{};
```

- Accessing elements with index operator []

```
int v = arr[4];
```



- Direct Member Access (->)
- this is a pointer

```
struct S {  
    auto member() -> void {  
        this->value = ...;  
    }  
    int value;  
};  
  
auto foo() -> void {  
    S* sp = new S{};  
    sp->member();  
  
    //same as  
    (*sp).member();  
}
```

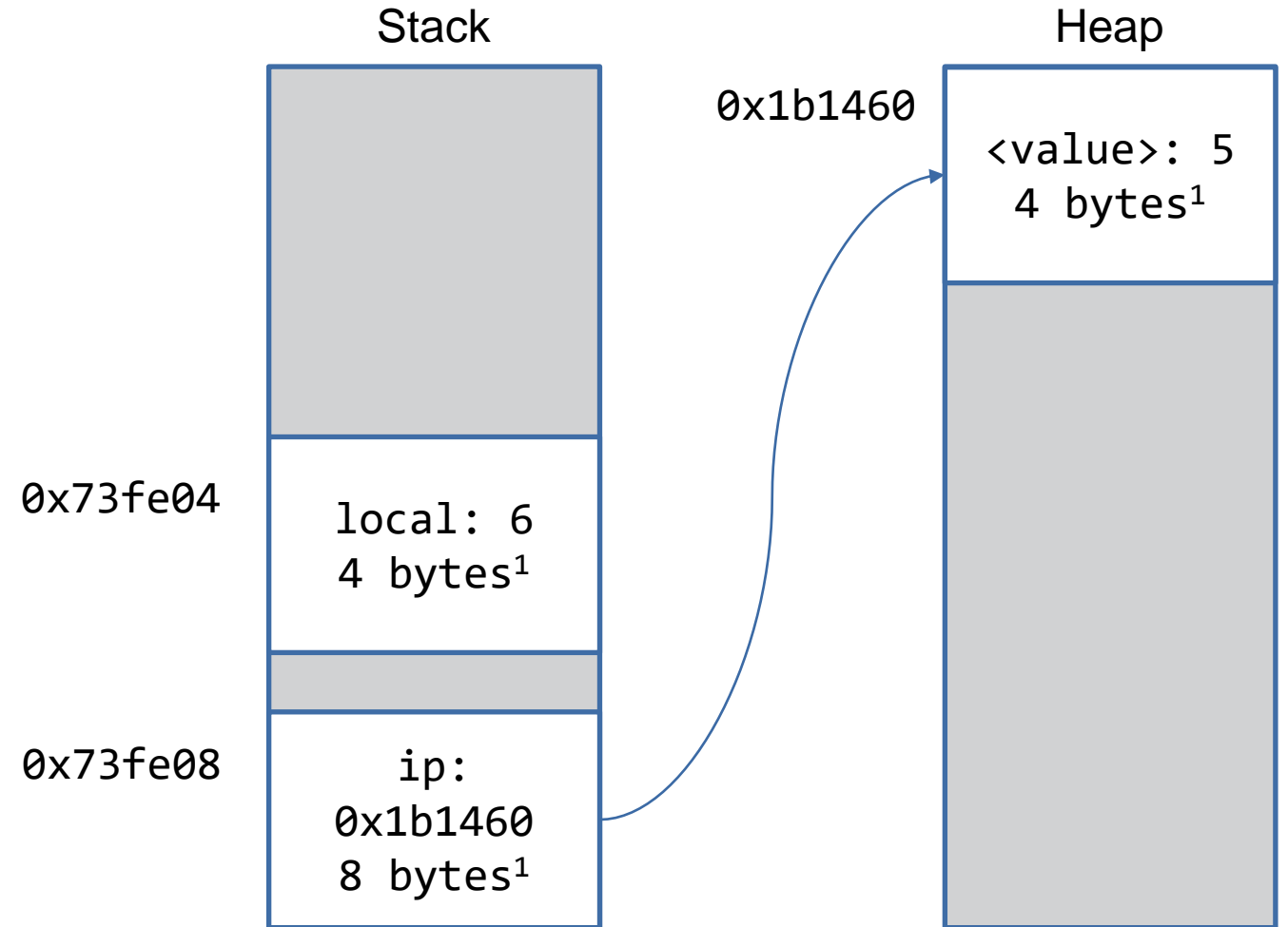
- Pointers can be used as parameters

- Addresses can be taken with (&)

■ Or `std::addressof()`

```
auto foo(int* p) -> void {
}

auto bar() -> void {
    int* ip = new int{5};
    int local = 6;
    foo(ip);
    foo(&local);
}
```

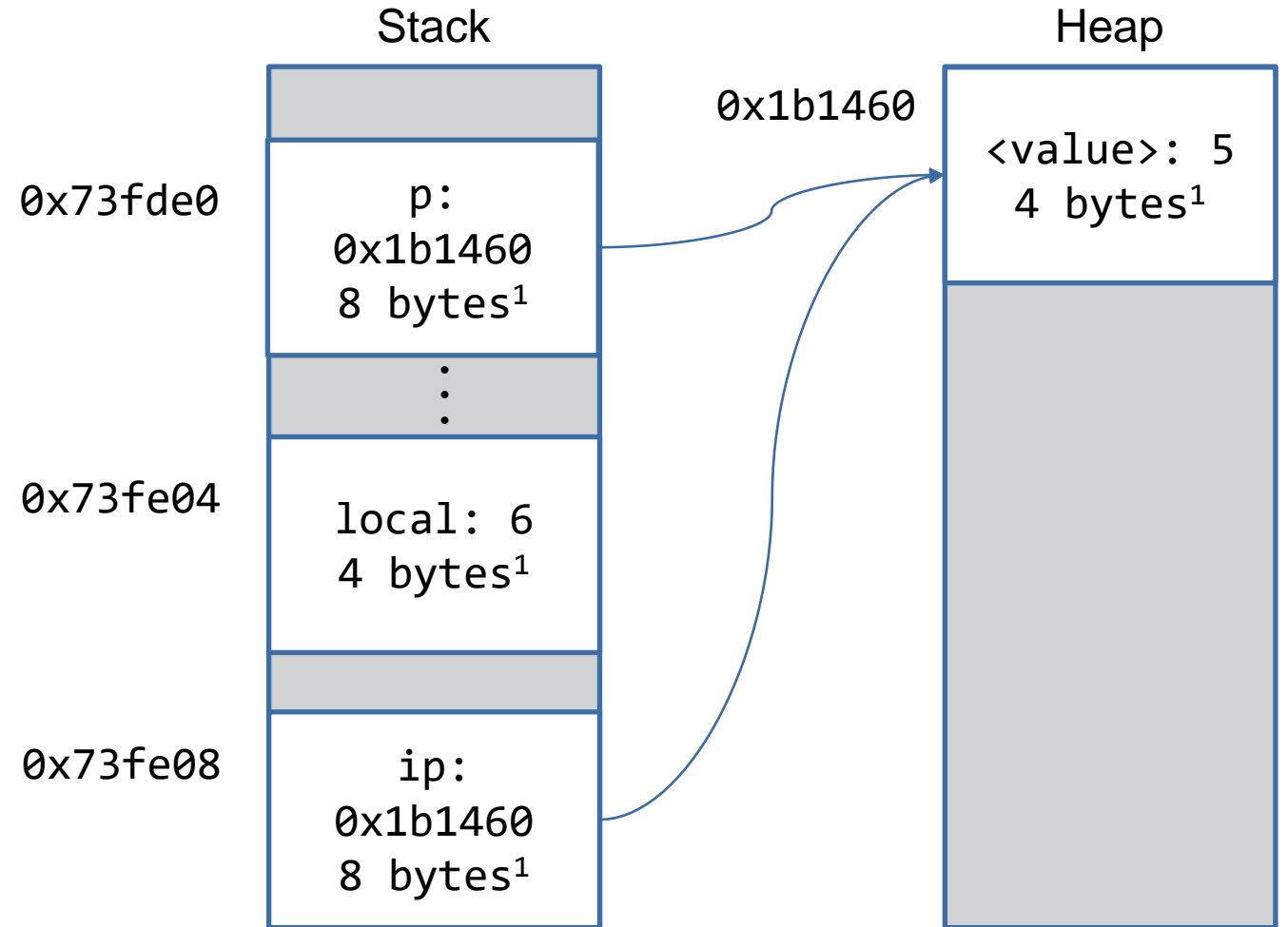


¹ Sizes depend on the platform

```

auto foo(int* p) -> void {
}

auto bar() -> void {
    int* ip = new int{5};
    int local = 6;
    foo(ip);
    foo(&local);
}
    
```

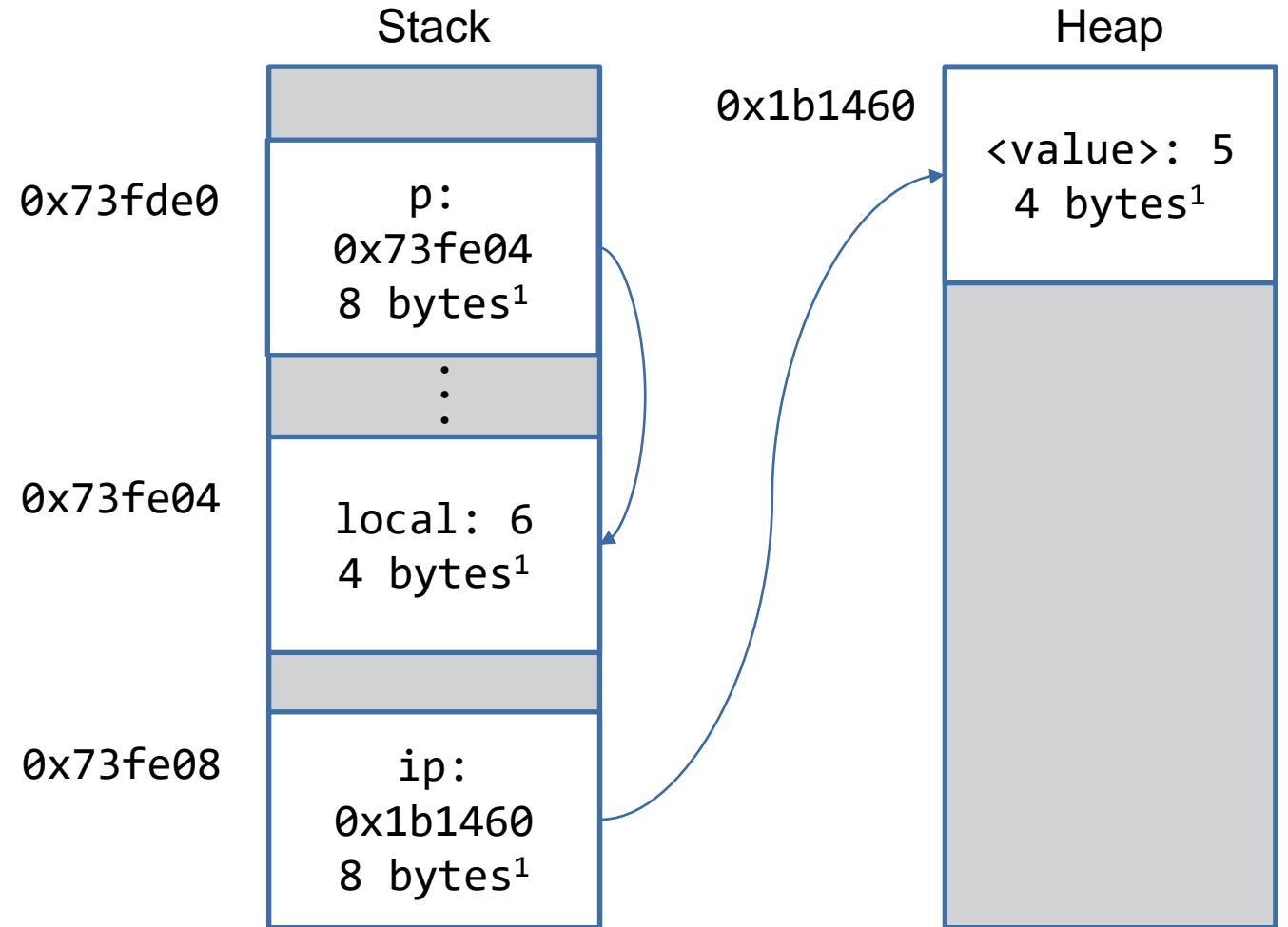


¹ Sizes depend on the platform

```

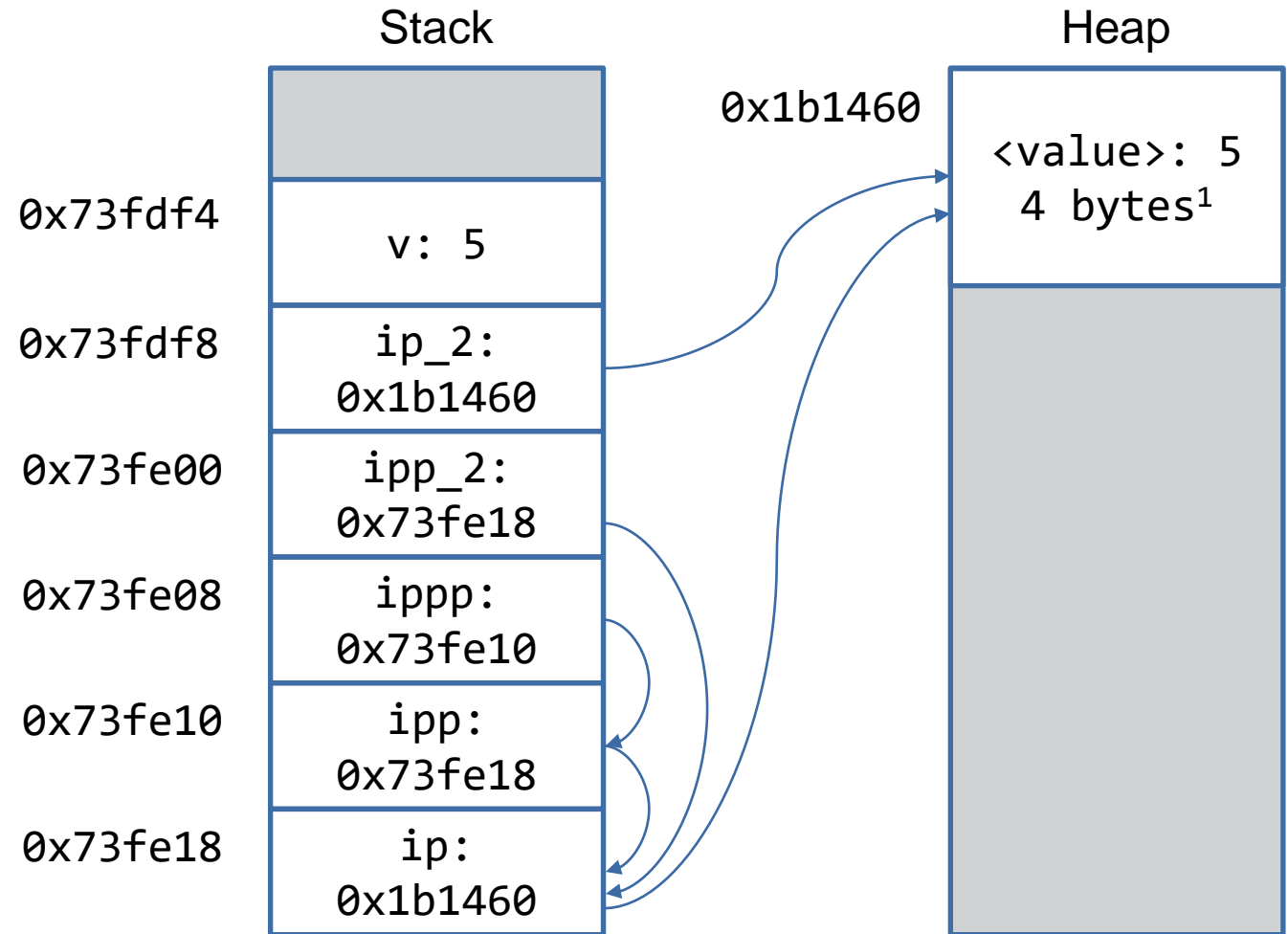
auto foo(int* p) -> void {
}

auto bar() -> void {
    int* ip = new int{5};
    int local = 6;
    foo(ip);
    foo(&local);
}
    
```



¹ Sizes depend on the platform

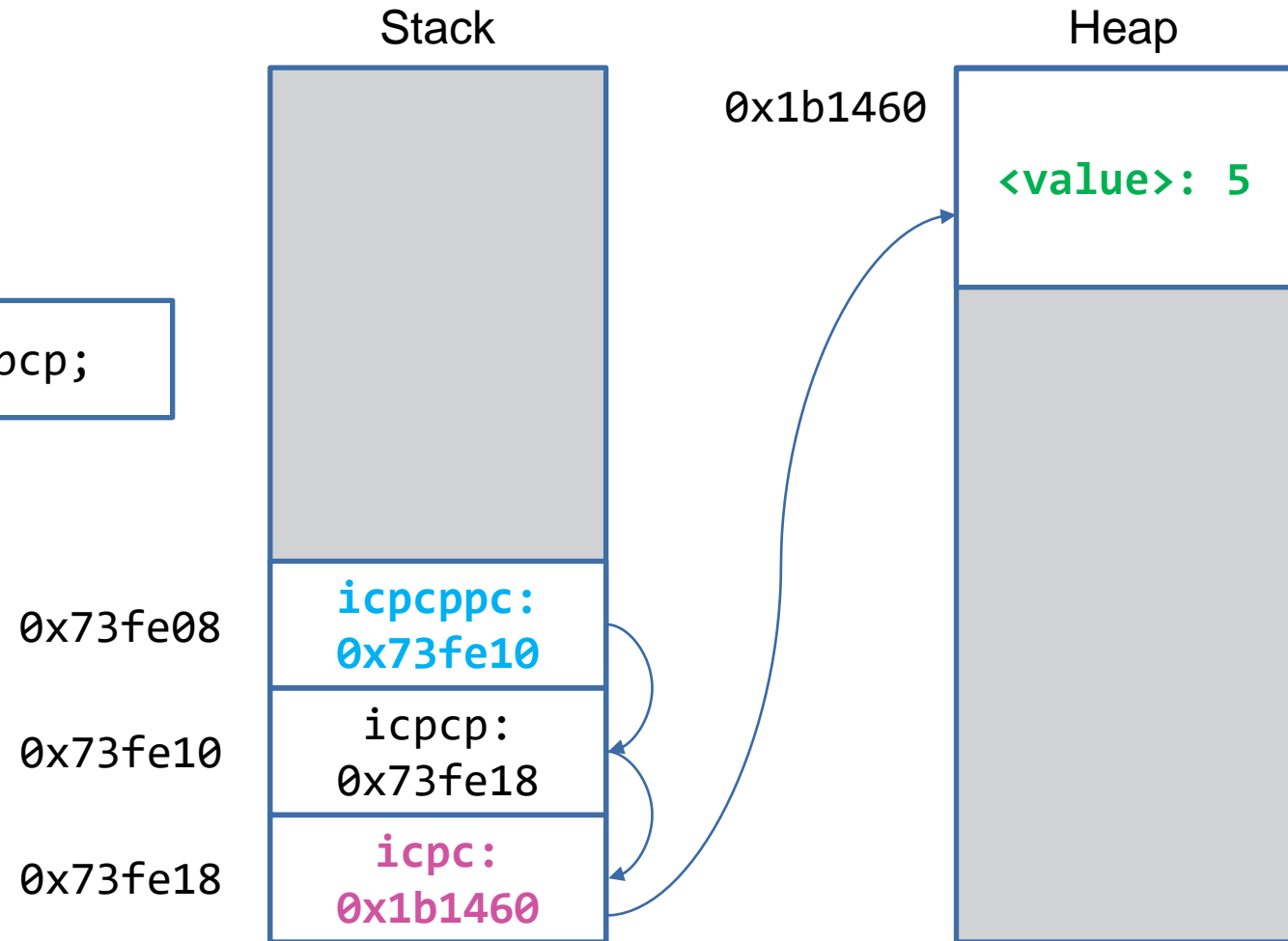

```
auto bar() -> void {
    int * ip = new int{5};
    int ** ipp = &ip;
    int *** ipp2 = &ipp;
    int ** ipp_2 = ipp;
    int * ip_2 = *ipp;
    int v = ***ipp;
}
```



- **Pointers can be const**

- **const** is on the RIGHT side of the *

```
int const * const * * const icpcppc = &icpcp;
```



- Represents a null-Pointer
- Literal (prvalue)
- Type: `nullptr_t`
- Implicit conversion to any pointer type: `T *`
- Prefer `nullptr` over `0` and `NULL`
 - No implicit conversion to integral type
 - No ambiguity on overload with integral type (discouraged)
 - Significant for template type deduction

```
auto bar(int i) -> void;
auto bar(S* ps) -> void;

//calls
bar(0);           //bar(int)
bar(NULL);        //surprising
bar(nullptr);     //bar(S*)
```

Reading Declarations



- Declarations are read starting by the declarator

- First read to the right until a closing parenthesis is encountered
- Second read to the left until an opening parenthesis is encountered
- Third jump out of the parentheses and start over



int const	* const	*	* const	icpcppc;
(5)	(4)	(3)	(2)	(1)

- icpcppc is the declarator (name)

- there is nothing to the right
- to the left: const pointer to a pointer to a const pointer to a const int

- Reading the example

- icpcppc is a const pointer to a pointer to a const pointer to a const int

(1)

(2)

(3)

(4)

(5)

- **Modifiers right to the declarator**

- **Array Declarator:** []
- **Function Parameter List:** (<parameter declarations>)

- **Modifiers left to the declarator**

- **References:** &&, &
- **Pointers:** *
- **Type:** e.g. int

void	(*	f)	(int	&	,	double)
(4)	(2)	(1)					(3)

(1) f is)

(2) (a pointer to

(3) a function, taking a reference to int and a double, returning

(4) void

- Array example

int	const	*	(*	f	[2]	[3])	[5];
-----	-------	---	---	---	---	-----	-----	---	------

f is an array of 2 elements of arrays of 3 elements of pointers to arrays of 5 elements of pointers to const int

- In extremis

int	(*	f	(int	(*)	(int))	(int)	;
-----	---	---	---	---	-----	---	---	---	---	-----	---	---	---	-----	---	---

```
int (*f(int*)(int)) (int);
```

f is a function

**this function takes a pointer to a function as argument (1)
and returns a pointer to a function (2)**

(1) this function takes an int and returns an int

(2) this function takes an int and returns an int

- **Never do something like this without proper type aliases!**

```
using alias = int(*)(int);  
alias f(alias);
```


- **const** might be written to the left of the type it qualifies

```
int const i; //both declarations  
const int i; //are the same
```

const applies to its left neighbor; only if there is no left neighbor it applies to its right neighbor



- **const** might be written to the left of the type it qualifies

```
int const i; //both declarations  
const int i; //are the same
```

- **Let's add an alias**

```
using alias = int;  
alias const i; //both declarations  
const alias i; //are the same
```

- What about more complex types

```
int const * const icpc; //both declarations  
const int * const cipc; //are the same
```

- Aliases worked well before

```
//Extract the int const * part  
using alias = int const *;  
alias const icpc; //works well
```

```
//Extract the int * const part  
using alias = int * const;  
const alias cipc; //not good
```

The type is int * const const now

- Always write const to the right of the type

- It's consistent with every other placement of const (e.g. pointers)
- It avoids surprises with type aliases

- The mutable specifier refers to the declared name
- What does the following mean?



```
mutable const int * mutable_const_int_pointer;
```

The diagram illustrates the scope of the `mutable` and `const` specifiers in the declaration `mutable const int * mutable_const_int_pointer;`. A green arrow points from the `mutable` keyword to the pointer name `mutable_const_int_pointer`, indicating that `mutable` applies to the pointer itself. Another green arrow points from the `const` keyword to the `int` type, indicating that `const` applies to the data being pointed to.

- `const` makes the pointee `int` immutable
- `mutable` makes the pointer mutable
- When does this make sense? Shouldn't the non-const `mutable_const_int_pointer` always be mutable?

HEAP Memory (De)Allocation



- **Explicit heap memory allocation**

- new expression

- **Syntax**

`new <type> <initializer>`

- **Allocates memory for an instance of <type>**

- **Returns a pointer to the object or array created (on the heap)**

- of type <type> *

- **The arguments in the <initializer> are passed to the constructor of <type>**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int x, y;  
};  
  
auto createPoint(int x, int y) -> Point* {  
    return new Point{x, y}; //constructor  
}  
  
auto createCorners(int x, int y) -> Point* {  
    return new Point[2]{{0, 0}, {x, y}};  
}
```

- **Explicit heap memory deallocation**

- delete expression

- **Syntax**

`delete <pointer>`

- **Deallocates the memory (of a single object) pointed to by the <pointer>**

- **Calls the Destructor of the destroyed type**

- **delete nullptr is well defined**

- it does nothing

- **Deleting the same object twice is Undefined Behavior!**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int x, y;  
}  
  
auto funWithPoint(int x, int y) -> void {  
    Point * pp = new Point{x, y};  
    //pp member access with pp->  
    //pp is the pointer value  
    delete pp; //destructor  
}
```



- **Explicit heap memory deallocation**

- `delete[]` expression

- **Syntax**

`delete[]` <pointer-to-array>

- **Deallocates the memory (of an array) pointed to by the <pointer-to-array>**
- **Calls the Destructor of the destroyed objects**
- **Also deletes multidimensional arrays**

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int x, y;  
}  
  
auto funWithPoint(int x, int y) -> void {  
    Point * arr = new Point[2]{{0, 0},  
                                {x, y}};  
    //element access with [], e.g. arr[1]  
    //arr points to the first element  
    delete[] arr; //destructors  
}
```


- Construction of an object at an already allocated memory location

- Syntax

```
new (<location>) <type> <initializer>
```

- Does **NOT** allocate new memory
 - You need to make sure that the memory at <location> is suitable for construction of a new object and any element there must be destroyed properly before (or be uninitialized)
- Calls the Constructor for creating the object at the given location (<location>)
- Returns the given memory location

```
struct Point {  
    Point(int x, int y) :  
        x {x}, y {y} {}  
    int x, y;  
};  
  
auto funWithPoint() -> void {  
    auto ptr = new Point{9, 8};  
    //must release Point{9, 8}  
    new (ptr) Point{7, 6};  
    delete ptr;  
}
```

- Does not exist, but a destructor can be called explicitly

- Syntax

- Call like any other member function

```
S * ptr = ...;  
ptr->~S();
```

- Destroys the object, but does not free its memory
- You cannot destroy an array in this way as a whole
 - You need to create your own infrastructure, which keeps track of arrays to achieve this
- Better use `std::destroy_at(ptr);`

```
struct Resource {  
    Resource() {  
        /*allocate resource*/  
    }  
    ~Resource() {  
        /*deallocate resource*/  
    }  
};  
  
auto funWithPoint() -> void {  
    auto ptr = new Resource{};  
    ptr->~Resource();  
    new (ptr) Resource{};  
    delete ptr;  
}
```

- **Point is not default constructible**

- We cannot allocate arrays of default-constructed Points

```
new Point[2];
```

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

- **We can allocate the plain memory (std::byte[])**

```
auto memory = std::make_unique<std::byte[]>(sizeof(Point) * 2);
```

- **And initialize it later**

```
new (memory.get()) Point{1, 2};
```

- Accessing the elements is tedious. Let's add a helper function

```
auto elementAt(std::byte * memory, size_t index) -> Point& {  
    return reinterpret_cast<Point *>(memory)[index];  
}
```

- We must not use the Point if it is uninitialized

```
auto memory = std::make_unique<std::byte[]>(sizeof(Point * 2));  
Point * first = &elementAt(memory.get(), 0);  
new (first) Point{1, 2};  
Point * second = &elementAt(memory.get(), 1);  
new (second) Point{4, 5};
```

- **Before our memory is deallocated, we have to destroy the elements**

- Call the destructor explicitly or use `std::destroy_at`

```
Point * first = &elementAt(memory.get(), 0);  
new (first) Point{1, 2};  
first->~Pointer();
```

```
auto memory = std::make_unique<std::byte[]>(sizeof(Point) * 2);  
Point * first = &elementAt(memory.get(), 0);  
new (first) Point{1, 2};  
Point * second = &elementAt(memory.get(), 1);  
new (second) Point{4, 5};  
  
std::destroy_at(second);  
std::destroy_at(first);
```

- Overloading new and delete operators for a class can inhibit heap allocation

```
struct not_on_heap {  
    static auto operator new(std::size_t sz) -> void * {  
        throw std::bad_alloc{};  
    }  
    static auto operator new[](std::size_t sz) -> void * {  
        throw std::bad_alloc{};  
    }  
    static auto operator delete(void *ptr) -> void noexcept {  
        // do nothing, never called, but should come in pairs  
    }  
    static auto operator delete[](void *ptr) -> void noexcept {  
        // do nothing, never called, but should come in pairs  
    }  
};
```

- Experiment in exercises

- **Overloading new and delete operators for a class can be used to provide efficient allocation**

- useful with a memory pool for small instances
- useful if thread-local pools are used
 - heap-memory is a shared resource
- can log or limit number of heap-allocated instances
 - for testing purposes, figuring bottlenecks

- **But in general, not advisable**

- if used in standard-containers you need to adjust their Allocator template argument instead

```
struct not_on_heap {  
    //...  
};  
  
struct small_but_many {  
    //...  
};  
  
std::vector<X, PoolAllocator> vp;
```



- **Simple rules**

- Delete every object you allocated
- Do not delete an object twice
- Do not access a deleted object

- **Just don't do the following**

```
auto foo() -> void {  
    int * ip = new int{5};  
    //exit without deleting  
    //location ip points to  
}
```



```
auto foo() -> void {  
    int * ip = new int{5};  
    delete ip;  
    delete ip;  
}
```



```
auto foo() -> void {  
    int * ip = new int{5};  
    delete ip;  
    int dead = *ip;  
}
```



- More cases

```
auto bar() -> void;

auto foo() -> void {
    int * ip = new int{5};
    bar(); //exception?!
    delete ip;
}
```

```
auto foo(int * p) -> void {
    //is it up to me to
    //delete p? likely not
}
```

```
auto create() -> int * {
    int * ip = new int{5};
    return ip;
}

auto foo() -> void {
    int * ip = create();
    //My turn to delete?
    //Probably yes
}
```

- It gets even more tricky when pointers are used for shared members

```
struct Node {  
    Node(Node * parent = nullptr) :  
        parent{parent}, children{} {  
        if (parent)  
            parent->children.push_back(this);  
        }  
    Node * parent;  
    std::vector<Node *> children;  
};
```

```
auto createTree() -> Node * {  
    Node * root = new Node{};  
    new Node{root}; new Node{root};  
    return root;  
}  
auto find_child_X() -> Node * {  
    Node * root = createTree();  
    Node * child_X = ...; //find child X  
    delete root;  
    return child_X;  
}
```

- The mistakes here can be discovered with reasonable effort, but it is a much bigger issue in a large code base

- **Getting familiar with the syntax of pointers is important**
 - You will come across them in most real world C++ (legacy) applications
- **Use pointers only if necessary and only in confined code areas**
 - Reasoning about the life-time of the heap-allocated objects must be easy
- **Prefer smart pointers to make reasoning about object life-time easier**
- **You can use `emplace` to directly create objects into standard containers**

Resource Management with RAII
Container's `emplace()`
Overloading `new/delete`

Self-Study



- **Alternative to allocating and deallocating a resource explicitly**

- Wrap allocation and deallocation in a class
- Constructor for allocation
- Destructor for deallocation

- **The automatic destruction at the end of a scope will take care of the resource deallocation**

- **Works with exceptions**

- No finally required

- **STL classes for heap memory**

- `std::unique_ptr` / `std::shared_ptr`

```
struct Resource {  
    Resource() {  
        //Allocate Resource  
    }  
    ~Resource() {  
        //Deallocate Resource  
    }  
    //API for accessing the resource  
    //Don't leak the resource!  
private:  
    WrappedResource * wrappedResource;  
};
```

```
auto workWithResource() -> void {  
    Resource item{};  
    functionThatMightThrow();  
    //Resource is released automatically  
}
```

```
std::unique_ptr<char> cPtr = std::make_unique<char>('*');
```

- `std::unique_ptr<T>`

- Wraps a plain pointer
- `unique_ptr` have zero runtime overhead
- A custom deleter could be supplied if required that is called instead of the destructor

```
//Template
template<typename T, typename Deleter =
        std::default_delete<T>>
class unique_ptr;

//Specialization for unbound arrays
template<typename T, typename Deleter>
class unique_ptr<T[], Deleter>;
```

```
std::unique_ptr<char> cPtr = std::make_unique<char>('*');
```

- `std::make_unique<T>`

- Always use `make_unique` if possible

```
template<typename T, typename...Args>
std::unique_ptr<T>
make_unique(Args&&...args) {
    return std::unique_ptr<T> {
        new T(std::forward<Args>(args)...)};
}
```

- You **can** create unbound arrays
- You **cannot** create arrays of fixed size

```
std::make_unique<char[]>(42)
```

```
std::make_unique<char[42]>(...)
```

- **Putting elements into a container...**

- Copying big objects into standard containers might be inefficient
- Moving big objects into standard containers might be more efficient
- Creating big objects into the space already allocated for a standard container might be even better

- **Syntax for `emplace`, example for `std::stack`**

```
template<typename... Args>  
auto emplace(Args&&... args) -> void;
```

- **Constructs an element of type `T` from the forwarded arguments in-place**

- Can put objects into a container that can neither be copied nor moved

- **Not available for `std::array`**

- Has fixed size

- It is possible to overload the new and delete operators

```
//Global new
auto operator new      (std::size_t) -> void *
auto operator new[]    (std::size_t) -> void *

//Placement new
auto operator new      (std::size_t, void * ptr) -> void *
auto operator new[]    (std::size_t, void * ptr) -> void *

//For specific type T
auto T::operator new   (std::size_t) -> void *
auto T::operator new[] (std::size_t) -> void *

//more new operators & delete operators
```

- See <http://en.cppreference.com/w/cpp/memory/new> for more details

- **A type is non-default-constructible when...**
 - ... there is no explicit default constructor and ...
 - ... there is no implicit default constructor!
- **Why can't we use `new T[capacity]` for allocating a T array?**
 - If T is of class type it must be default-constructible to allow this call.
 - That is not the case for fundamental types!
- **Alternative**
 - Allocate a `std::byte` array and manage the objects "manually"

- **Static**

- Local
- Size known at compile-time (no_of_bytes)

```
std::array<std::byte, no_of_bytes> values_memory;
```

- **Dynamic**

- On heap
- Size known at run-time

```
std::unique_ptr<std::byte[]> values_memory;
```

- **Given:**

- An array of elements of type T is accessible through a pointer to `std::byte`.
- Declaration of data: `std::byte * data;`
- How to correctly access the fifth element in this array?

a) `data[4]`

b) `reinterpret_cast<T*>(data)[4]`

c) `*reinterpret_cast<T*>(data + 4)`

d) `*reinterpret_cast<T*>(data) + 4`

- **Note you can access the raw memory behind an `std::array` with `.data()` or for `std::unique_ptr<T[]>` with `.get()`**