	Overview of the Lecture	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers
Resource Ownership in C++	 Part 1 – RAII Principle (in C++) Acquisition-Release Pattern in C/C++ 	
Jan Faigl	RAII – Resource Acquisition is Initialization	Part I
·	RAII Threading	Part 1 – RAII Principle (in C++)
Department of Computer Science Faculty of Electrical Engineering	Smart Pointers	Fart I = I(Arr Frinciple (In C++))
Czech Technical University in Prague	Part 2 – Move and Copy Semantics (in C++)	
Lecture 13	Assignment of Objects Holding Resources	
B3B36PRG – Programming in C	lvalues & rvalues	
	Move and Copy Semantics	
David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 1 / 48		David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 3 / 48
Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers
Acquisition-Release Pattern in C	Acquisition-Release Pattern in C	Acquisition-Release Pattern in C
<pre>int main(void) { int *array = malloc(SIZE * sizeof(int)); /* ACQUISITION */</pre>	<pre>int main(void) { FILE *in_file = fopen(FILE_NAME, "r"); /* ACQUISITION */</pre>	<pre>int main(void) { pthread_mutex_init(&mtx, NULL);</pre>
		<pre>pthread_mutex_lock(&mtx); /* ACQUISITION */</pre>
/* do work */	/* do work */	<pre>/* do work in critical section */</pre>
<pre>free(array); /* RELEASE */ return 0;</pre>	<pre>fclose(in_file); /* RELEASE */ </pre>	<pre>pthread_mutex_unlock(&mtx); /* RELEASE */</pre>
<pre>return 0; }</pre>	return 0; }	return 0;
		}
David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 5 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 6 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 7 / 48
Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers
Acquisition-Release Pattern in C	Acquisition-Release Pattern in C++	But what if something goes wrong?
<pre>int main(void) </pre>	<pre>int main(void) {</pre>	<pre>int main(void) {</pre>
<pre>t pthread_create(&thread, NULL, foo, NULL); /* ACQUISITION */</pre>	<pre>MyClass* c = new MyClass(); /* ACQUISITION */ int* array = new int[SIZE];</pre>	<pre>int *array = malloc(SIZE * sizeof(int)); /* ACQUISITION */</pre>
• · · · · · ·		<pre>if(!everithing_ok) {</pre>
/* do work */	/* do work */	<pre>return 100; /* !!! Resource is not released */ }</pre>
<pre>pthread_join(&thread, NULL); /* RELEASE */ return 0;</pre>	<pre>delete[] array;</pre>	
return 0; }	<pre>delete c; /* RELEASE */ return 0;</pre>	<pre>free(array); /* RELEASE */ return 0;</pre>
	}	}
lavid Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 8 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 9 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 10 / 48

Automatic Deductor Call • Obstacts scaled at the of all learned the main(value) • Obstacts scaled at the of all learned the main(value) • Obstacts is called at the of all learned the main(value) • Obstacts is called at the of all learned the main(value) • Obstact is called at the od all learned the main(value) • Obstact is called at the od all learned the main(value) • Obstact is called at the od all learned the main(value) • Obstact is called at the od all learned the main(value) • Obstact is called at the od all learned the main(value) • Obstact is call						
 Description for an output of the and of th	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers			
<pre> • Detector solids to the out of leased is matrices is matrixes is matrix</pre>	Automatic Destructor Call		Resources Acquisition is Initialization			
<pre>int minifered f writes of f writes of f f writes of f f f writes of f f f writes of f f f write</pre>		Destructor is called at the end of life-time!				
<pre>// do wax +/ // do wax +/ // "for constructions MyClass : /* Construct</pre>						
<pre>inductive if (induces intervention); /* de sour */ if (induces intervention); /* (induces interve</pre>	int main(void)					
<pre>/* do your */ /* "ryClass() /* UNE END! */ /* "ryClass() /*</pre>	MyClass c; /* Constructor MyClass() is called */					
<pre>// "Polace() /= NUCH HERE! */ // "TypElace() /= NUCH HERE! */ // "TyPEITER */ // "TyPEITE</pre>	/* do work */		Resource release is handled by the destructor.			
<pre>// Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the end of scope. */ // Typican() /* Descrutter is called at the</pre>			Resource is bound to lifetime object instance.			
<pre> function so function so</pre>		}				
<pre>b / 0 / 0 / 0 / 0 / 0 / 0 / 0 / 0 / 0 /</pre>	}					
<pre>static Main State Prior S</pre>						
Example Array Implementation struct Malleckooption : atd;:exception { count char what Count nearcopt { return "Mallec error"; } ; class MyArray (index grad return (Malleckooption : atd;:exception { data sp = (int+loalIacGine, pixe(int); if (data sp = (int+loaIIACGine, pixe(int); if						
<pre>strut: Milodiception : std:sreeption { cost char + that() cost neercept { return "fallec error"; } }; class Mytray { ulong size_p; int* data_p; public: MVaray(ulong size): size_p(size) { data_p = nullptp { data_p =</pre>						
<pre>coast char* what() coast nexcept { return "Malloc error"; } } // // // // // // // // //</pre>		Example Array Implementation	Implementation of RAII in Standard Library			
<pre> p; class MyArray { ulong size_p; int data_p; (int)class_class_ficies, size_f(size) { data_p = (int)call(class_e, size_f(size)); if(data_p = millptx) { duthows Mail Constraints int data_p; data data data data data da</pre>						
<pre>class MyArray { ilit Gata_p; ilit Gata_p;</pre>	•					
<pre>tures for any for the set of the set of</pre>	close Muterou (Dynamic array - std::vector			
<pre>public:</pre>			File - std::ifstream / std::ofstream			
<pre>public. HyArray(); int size() const; ; int size() const; ; ; int size() const; ; ; ; int size() const; ; ; ; ; ; ; ; ; ; ; ; ; ;</pre>	•	}	3			
TWyArray(); // free(data_p); init& operator[](ulong index); // free(data_p); uint size() const; } >: BBBMPG - Letter 13 Converds in C+1 est3/nymrsy-GP i/ / BC Acquisite/Balan Atoms NCCL+ BBBMPG - Letter 13 Converds in C+1 est3/nymrsy-GP i/ / BC Acquisite/Balan Atoms NCCL+ BBBMPG - Letter 13 Converds in C+1 est3/nymrsy-GP i/ / BC Acquisite/Balan Atoms NCCL+ BBBMPG - Letter 13 Converds in C+1 est3/nymrsy-GP i/ / BC Acquisite/Balan Atoms NCCL+ RAI - Reserve Acquisite is balance Acquisite/Balan Atoms NCCL+ RAI - Reserve Acquisite is balance I Generic wrapper for dynamic array. int main(void) { (std::ofstream outFile("out.txt"); outFile <		}				
<pre>intk operator[[(ulong index);</pre>			Pointer to heap - std::unique_pointer / std::shared_pointer			
<pre>uint size() const; }; uint size() const; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;</pre>		free(data_p);				
<pre>}; but verture 12 but verture 13 but verture 14 but verture</pre>		ſ				
<pre>Used Vacues, 202 EXEMPNG - Lecture 11 Ownership in C++ 11 / all [Veed Vacues, 202 EXEMPNG - Lecture 13 Ownership in C++ 15 / di [Veed Vacues, 202 EXEMPNG - Lecture 13 Ownership in C++ 15 / di [Veed Vacues, 202 EXEMPNG - Lecture 14 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 14 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 15 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 16 Ownership in C++ 16 / di [Veed Vacues, 202 EXEMPNG - Lecture 16 / dived Vacues, 202 EXEMPNG - Lecture 16 / differences, 202 EXEMPNG - Lecture 16 / d</pre>						
<pre>std::vector Generic wrapper for dynamic array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic reallocation of the underlying array. Generic wrapper for dynamic matching dynamic reallocation of the underlying ar</pre>	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 14 / 48					
<pre>e Generic wrapper for dynamic array. More general version of MyArray. Other useful features: such as push_back() with dynamic reallocation of the underlying array. int main() f std::vectorsint> v = { 7, 5, 16, 8 }; v.push_back(25); v</pre>	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers				
<pre>More general version of MyArray. int main(void) for (int n: v) { std::cout < "," = { "; std::co</pre>	std::vector	File streams	RAII Thread and Mutex			
<pre>Mote general version of nyArray. Mote general version of nyArray. Other useful features: such as push_back() with dynamic reallocation of the underlying array. int main() { std::vector<int> v = { 7, 5, 16, 8 }; v.push_back(25); v.push_back(13); std::cout << "y = { "; for (int n : v) { std::cout << "y = { "; std::cout << "; "; "; std::cout << "; "; std::cout << "; "; std::cout << "; "; ";</int></pre>		int main (maid)				
<pre>such as push_back() with dynamic reallocation of the underlying array. std::ottream outFile("out.txt"); outFile << "Hello World\n"; std::ottream inFile("in.txt"); std::ottream inFile("in.txt"); std::ottream inFile("in.txt"); int a; inFile >> a; 'rmy_jthread(Function&& f, Args&& args) : thread(f, args) { for (int n : v) { std::out << "y = { "; for (int n : v) { std::out << ", "; } std::out << "; \n"; } } </pre>	· · ·					
<pre>int main() { int main() { std::vector<int> v = { 7, 5, 16, 8 };</int></pre>						
<pre>std::vector<int> v = { 7, 5, 16, 8 }; std::vector<int> v = { 7, 5, 16, 8 }; v.push_back(25); v.push_back(13); std::cout << "v = { "; for (int n : v) { std::cout << "}; } </int></int></pre>	<pre>int main()</pre>	outFile << "Hello World\n";	•			
<pre>v.push_back(25); v.push_back(25); v.push_back(23); std::cout < "v = { "; for (int n : v) { std::cout < ", "; } std::cout < "}; \n"; } </pre> inFile >> a; /* Destructor of outFile/inFile automatically closes the files. */ return 0; } } ;	<pre>{ std::vector<int> v = { 7, 5, 16, 8 };</int></pre>		<pre>my_jthread(Function&& f, Args&& args) : thread(f, args) {};</pre>			
<pre>v.pum.pack(13); std::cout << "y = { "; for (int n: v) { std::cout << ", "; } std::cout << "}; \n"; } </pre> /* Destructor of outFile/inFile automatically closes the files. */ return 0; } } ;			~mv ithread() {			
<pre>for (int n : v) { std::cout << n << ", "; } std::cout << "}; int = distribution of outfile/infile distribution of outfile distribution of o</pre>			<pre>if(thread.joinable()) {</pre>			
<pre>} } std::cout << "}; \n"; }</pre>	for (int n : v) {		thread.join();			
}; };	}		}			
Tanvets/ Char	<pre>std::cout << "}; \n"; }</pre>		};			
David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 B3836PRG - Lecture 13: Ownership in C++ 1 / 4 David Valouch, 2022 David Valouch,	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 17 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 18 / 48	David Valouch, 2022 B3B36PRG - Lecture 13: Ownership in C++ lec13/thread.cpp 20 / 48			

<pre>structure is a structure is a s</pre>		Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers	Acquisition-Release Pattern in C/C++ RAII – Resource Acquisition is Initialization RAII Threading Smart Pointers			
class my.loc_gunt { std::mirror std:: partie: my.loc_gunt(s): my.loc_g	RAII Thread and Mutex	RAII Thread and Mutex	RAII Thread and Mutex			
<pre>styled_gesd() { y_j); y_j y_j</pre>	<pre>std::mutex* mtx; public: my_lock_guard(std::mutex& mtx) : mtx(&mtx) { mtx.lock();</pre>	<pre>for(int i = 0; i < n; ++i) { my_lock_guard guard(*mtx); int tmp = *a; std::this_thread::sleep_for(std::chrono::microseconds(1)); *a = tmp + 1; }</pre>	<pre>{ int counter = 0; countTwice2(&counter, 10);</pre>			
$\frac{1}{1 + 1 + 1 + 1} = \frac{1}{1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 +$	<pre>mtx->unlock(); };</pre>	<pre>std::mutex counterMutex; my_jthread thrd1(coutnWorker, val, counter, &counterMutex); my_jthread thrd2(coutnWorker, val, counter, &counterMutex);</pre>	return 0;			
$ \frac{1}{10000000000000000000000000000000000$	David Valouch. 2022 B3B36PRG – Lecture 13: Ownership in C++ 21 / 48		David Valouch. 2022 B3B36PRG – Lecture 13: Ownership in C++ 23 / 48			
 Wrapper sound hap pointer. • Wrapper sound hap pointer. • estic indige_ptr • Free the memory of delton. • Only our subject pointing to a souling code date so may exist. • Only our subject pointing to a souling code date so may exist. • Water and a point of the same address may exist. • Water address m						
 Wrapper around heap points: • Wrapper arou	Smart Pointers	Shared Pointer	Shared Pointer			
Addgement of CQC++ RAM-Preserves Adaptations is Individuality (NAU Threading Smart Palates) Shared Pointer template <class t=""> my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_ptr<t>::my_shared_p</t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></class>	 std::unique_ptr Frees the memory on deletion. Only one unique_ptr pointing to a specific address may exist. May not be <i>copied</i> only <i>moved</i>. std::shared_ptr Keeps reference counter. Last shared pointer frees the memory. Multiple shared_ptrs pointing to the same address may exist. 	pointee_ pRefCount pointee_ pRefCount pointee_ pRefCount image source: https://stackoverflow.com/questions/9200664/how-is-the-stdr1shared-ptr-implemented	<pre>class my_shared_ptr { T* ptr; int* ref_counter; public: my_shared_ptr(T* ptr); my_shared_ptr(my_shared_ptr<t>& other);</t></pre>			
<pre>template<class t=""> my_shared_ptr<t>:my_shared_ptr(T* ptr) : ptr(ptr), ref_counter(new int(1)) {} template<class t=""> my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_shared_ptr<t>:my_sha</t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></t></class></t></class></pre>						
	<pre>template<class t=""> my_shared_ptr<t>::my_shared_ptr(T* ptr) : ptr(ptr), ref_counter(new int(1)) {} template<class t=""> my_shared_ptr<t>::my_shared_ptr(my_shared_ptr<t>& other) : ptr(other.ptr), ref_counter(other.ref_counter) { *ref_counter += 1; } template<class t=""> my_shared_ptr<t>::~my_shared_ptr() { if (*ref_counter > 1) { *ref_counter -= 1; } else { delete ref_counter; delete ptr; } </t></class></t></t></class></t></class></pre>		 Recall MyArray What should the following code do? MyArray array1(10); MyArray array2 = array1; Remember MyArray structure class MyArray { ulong size_p; int* data_p; 			
		David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 29 / 48	David Valouch, 2022 B3B36PRG – Lecture 13: Ownership in C++ 31 / 48			

Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	
Assignment of Objects Holding R	esources		Assignment of Objects Hold	ing Resources		Assignment of Objects Ho	lding Resources		
<pre>class MyArray { ulong size_p; int* data_p; }; More specifically:</pre>			 Copy the pointer. array1 			 Allocate new array and copy array1 			
 What should happen to data_p? Multiple options: Copy the pointer. Allocate new array and copy data. Copy the pointer, but invalidate or 	iginal data.		 PROBLEM: Which object har This is simmilar to the behavior 	•		 PROBLEM: Possible redunc (e.g. returning from functio 	lancy if array1 is about do be deleted n).		
David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	32 / 48 Move and Copy Semantics	David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	33 / 48 Move and Copy Semantics	David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	34 / 48 Move and Copy Semantics	
Assignment of Objects Holding R		note and copy semantics	Move and Copy Semantics • Copy:		more and copy semanaci	Value Categories		more and copy committee	
■ Copy the pointer, but invalidate origi	nal data.	array1 array2				 Each expression in C++ has a type and value category. Ivalue - 'left value' (L = r) An expression whose evaluation determines the identity of an object or function ¹ - glvalue Is not xvalue. 			
 PROBLEM: Original array becomes invalid. Similar to the behavior of unique_ptr. 			■ Move: array1 → NULL → ··· □ array2			 rvalue - 'right value' (1 = R) An expression whose evaluation computes the value of an operand of a built-in operator (such prvalue has no result object), or initializes an object.¹ - prvalue Object whose resources can be reused.¹ - xvalue 			
David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	35 / 48 Move and Copy Semantics	David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	36 / 48 Move and Copy Semantics	David Valouch, 2022 Assignment of Objects Holding Resources	B3B36PRG - Lecture 13: Ownership in C++ lvalues & rvalues	38 / 48 Move and Conv Semantics	
lvalue			rvalue			rvalue / lvalue referenc			
 Ivalue - 'left value' (L = r) Can be assigned to. Variable name Function/operator call whose value operator a = b. Pre-increment/decrement ++i, Indirection *p. Subscript a [i]. and more¹ 		signment	 rvalue - 'right value' (1 = R) Cannot be assigned to. Function/operator call who Post-increment/decrement All built in arithmetic oper. Addres-of expression &a std::move(T) And more¹ 	ose value is non-reference. i++, i		 lvalue reference T& Alias to an existing object Can be initialized by an 1 rvalue reference T&& rvalue reference T& Extend lifetime of tempo std::string s = std::string& r Can be initialized by an 1 	<pre>lvalue. rary object.¹ e.g. result of an operator "hello"; = s + s;</pre>		
¹ en.cppreference.com/w/cpp/languag David Valouch, 2022	e/value_category B3B36PRG - Lecture 13: Ownership in C++	39 / 48	¹ en.cppreference.com/w/cpp/ David Valouch, 2022	language/value_category B3B36PRG - Lecture 13: Ownership in C++	40 / 48	¹ en.cppreference.com/w/cpj David Valouch, 2022	p/language/value_category B3B36PRG - Lecture 13: Ownership in C++	41 / 48	

Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics
Copy Semantics			Move Semantics			Copy Semantics of MyArray		
 Copy Semantics Copy constructor: T(const T&) Constructs object as a copy of another object. Copy assignment: T& operator=(const T&) Copies an object in another object Frees resources previously owned by the modified object. Any resources required by an object fo a given instance must be acquired. 			 Move constructor: T(const T&&) Constructs an object using resources of another object. Move assignment: T& operator=(T&&) Moves an object into another. Ownership of resources is transferred. Frees resources previously owned by the modified object. No new resources are allocated. It is assumed the source object will be destroyed after the move. 			<pre>WyArray: MyArray (const MyArray& other) : size_p(other.size_p), data_p(new int[size_p]) { std::cout << "MyArray(&)" << '\n'; for(int i = 0; i < size_p; ++i) { data_p[i] = other.data_p[i]; } } MyArray& MyArray::operator=(const MyArray& other) { std::cout << "MyArray operator=(&)" << '\n'; delete[] data_p; size_p = other.size_p; data_p = new int[size_p]; for(int i = 0; i < size_p; ++i) { data_p[i] = other.data_p[i]; } return *this; } } </pre>		
David Valouch, 2022	B3B36PRG – Lecture 13: Ownership in C++	43 / 48	David Valouch, 2022	B3B36PRG - Lecture 13: Ownership in C+	-+ 44 / 48	David Valouch, 2022	B3B36PRG – Lecture 13: Ownership in C++	45 / 48
Assignment of Objects Holding Resources	lvalues & rvalues	Move and Copy Semantics	Topics Discussed			Topics Discussed		
<pre>Move Semantics of MyArray MyArray::MyArray(MyArray&& other)</pre>			Sun	nmary of the Lecture		Topics Discussed Resouce Acquision-Release par RAII using automatic destruct Example RAII array wrapper RAII handlig of other resource Files Mutexes Threads Smart pointers Assignment of object with reso Ivalue and rvalue Ivalue reference and rvalue Move and copy semantics	or call s	
David Valouch, 2022	B3B36PRG – Lecture 13: Ownership in C++	46 / 48	David Valouch, 2022	B3B36PRG – Lecture 13: Ownership in C+	+ 47 / 48	David Valouch, 2022	B3B36PRG - Lecture 13: Ownership in C++	48 / 48