Concepts of C++ Programming

Lecture 4: References, Arrays, Pointers

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

Value Categories (Simplified)

lvalue

- Can appear on *left* side of assignment
- Locates an object
- Has an address
- Examples:
  - Variable names: `var`
  - Assignment exprs: `a = b`

rvalue

- Can only appear on *right* side of assignment
- Might not have address
- lvalue can be converted implicitly to rvalue
- Examples:
  - Literals: `42`
  - Most exprs: `a + b`, `a < b`

Reference Declarations (1)[44]

- Declare an alias to an existing object or function
- Lvalue reference: `type& declarator`

- Definitions must be initialized to refer to a valid object/function
- Declarations don't need initializer, e.g. parameters

- Peculiarities:
  - References are immutable, i.e. can't change which object is aliased
  - References are not objects
  - ⇒ No references to references

[44] https://en.cppreference.com/w/cpp/language/reference

Lvalue References: Example (Alias)

```
unsigned i = 10;
unsigned j = 20;
unsigned& r = i; // r is now an alias for i

r = 15; // modifies i to 15
r = j; // modifies i to 20

i = 42;
j = r; // modifies j to 42
```

Lvalue References: Example (Pass By Reference)

- References are used to implement pass-by-reference semantics

```
#include <print>
void computeAnswer(int& result) {
  result = 42;
}

int main() {
  int theAnswer = -1;
  computeAnswer(theAnswer); // theAnswer is now 42
}
```

Lvalue References: Example (Returning Reference)

- Function calls returning lvalue references are lvalues

```
int global1 = 0;
int global2 = 0;

int& getGlobal(int num) {
  if (num == 1)
    return global1;
  return global2;
}

int main() {
  getGlobal(1) = 10; // global1 is now 10
  getGlobal(2)--; // global2 is now -1
}
```

References and cv-Qualifiers

- References themselves cannot be cv-qualified
- But the referenced type can be
  - Reference can be initialized by less cv-qualified type
    e.g. `const int&` can be initialized from `int&`

```
#include <print>

void printAnswer(const int& answer) {
  std::println("{}", answer);
}

int main() {
  int theAnswer = 42;
  printAnswer(theAnswer); // cannot modify theAnswer
}
```

Pass-By-Reference

Quiz: What is the output of the program?

```cpp
#include <print>
void foo(const int& a, int& b, const int& c) {
  b += a;
  b += c;
}

int main() {
  int x = 1;
  foo(x, x, x);
  std::println("{}", x);
}
```

A. (undefined behavior) B. 1 C. 2 D. 3 E. 4

Dangling References[45]

- Lifetime of object can end while references still exist

⇝ dangling reference, when used: undefined behavior

```
int& foo() {
  int i = 123;
  return i; // DANGER: returns dangling reference
}
int bar() {
  int& res = foo();
  return res; // object used outside its lifetime => UB
}
```

[45] https://en.cppreference.com/w/cpp/language/reference#Dangling_references

Rvalue References

- Extend the lifetime of temporary objects
  - NB: const lvalue references can also extend lifetime of temporaries
- Rvalue reference: `type&& declarator`
- Cannot bind directly to lvalues

```
int i = 10;
int&& j = i; // ERROR: cannot bind lvalue
int&& r = 42; // OK

int&& k = i + i; // OK, k == 20
k += 22; // OK, k == 42

const int& l = i * i; // OK, l == 100
l += 10; // ERROR: cannot modify constant reference
```

Passing Rvalues

Quiz: What is the output of the program?

```
#include <print>
int foo(const int& a, const int& b, int&& c) {
  c += b;
  return c + a;
}

int main() {
  int x = 1;
  int r = foo(x, x, x);
  std::println("{}", r);
}
```

A. (compile error) B. 1 C. 2 D. 3 E. 4

Passing Rvalues

Quiz: What is the output of the program?

```
#include <print>
int foo(const int& a, const int& b, int&& c) {
  c += b;
  return c + a;
}

int main() {
  int x = 1;
  int r = foo(x, x * 2, x + 10);
  std::println("{}", r);
}
```

A. (compile error) B. (undefined behavior) C. 13 D. 14 E. 26

Reference Declaration Syntax

- & and && syntactically belong to the declarator!

```
int i = 10;
int& a = i, k = 2; // a is int&, k is int
```

⇒ Only declare one identifier at a time!

- `int& j = 1;` and `int &j = 1;` are valid, follow code style

Rvalue References: Overload Resolution

```
void foo(int& x);
void foo(const int& x);
void foo(int&& x);

int& bar();
int baz();

int main() {
  int i = 42;
  const int j = 84;

  foo(i); // calls foo(int&)
  foo(j); // calls foo(const int&)
  foo(123); // calls foo(int&&)

  foo(bar()) // calls foo(int&)
  foo(baz()) // calls foo(int&&)
}
```

Arrays[46]

- Syntax (C-style arrays): `type declarator[expression];`
- *expression* must be an integer constant **at compile-time**
- Elements can be accessed with `[]` with index $0 \cdots < N$
- Arrays cannot be assigned or returned

```
unsigned short arr[10];
for (unsigned i = 0; i < 10; ++i)
  arr[i] = i * i;

unsigned a[10];
unsigned b[10];
a = b; // ERROR: cannot assign arrays
```

[46] https://en.cppreference.com/w/cpp/language/array

Array Initialization

- Without an initializer, elements are default-initialized
  - Remember: for local variables, this means uninitialized
- Zero-initializer:

```
unsigned short arr[10] = {}; // 10 zeroes
```

- List-initializer:

```
unsigned short arr[] = {1, 2, 3, 4, 5, 6}; // 6 elements
```

Array Memory Layout

Elements of an array are allocated **contiguously** in memory

- Given `unsigned short a[10];` containing the integers 1 through 10
- Assuming a 2-byte `unsigned short` type
- Assuming little-endian byte ordering

a[0]		a[1]		a[2]		a[3]		a[4]		a[5]		a[6]		a[7]		a[8]		a[9]	
01	00	02	00	03	00	04	00	05	00	06	00	07	00	08	00	09	00	0a	00
00		02		04		06		08		0a		0c		0e		10		12	

Address

Arrays are just dumb chunks of memory

- Out-of-bounds accesses are not detected
- May lead to rather weird bugs, not necessarily crashes
- Exist mainly due to compatibility requirements with C

sizeof Array

- Like for other types: `sizeof` return array size **in bytes**
- Divide by size of an element to determine array length

```
unsigned short a[10];

for (unsigned i = 0; i < sizeof(a) / sizeof(a[0]); ++i)
  a[i] = i * i;
```

(Don't do this in C++)

Multi-Dimensional Arrays

- Array elements can be arrays themselves

```
unsigned md[3][2]; // array with 3 elements of (array of 2 unsigned int)
for (unsigned i = 0; i < 3; ++i)
  for (unsigned j = 0; j < 2; ++j)
    md[i][j] = 3 * i + j;

unsigned b[][2] = { // only the outermost dimension can be omitted
  {0, 1},
  {2, 3},
  {4, 5},
};
```

- Elements still allocated contiguously in memory

size_t[47]

- Designated types for indexed and sizes: `std::size_t` (`<cstddef>`)
- Unsigned integer type large enough to represent all possible array sizes and indices on the target architecture
- Used throughout the standard library for indices/sizes
- Generally use `size_t` for indexes and array sizes
  - For small arrays, `unsigned` might be sufficient
  - Do not use `int`

[47] https://en.cppreference.com/w/cpp/types/size_t

std::array[48]

C-style arrays should be avoided whenever possible

- Use the std::array type defined in the <array> standard header instead
- Similar semantics as a C-style array
- Optional bounds-checking and other useful features
- *template type* with two parameters (element type and count)

```
#include <array>
int main() {
  std::array<unsigned short, 10> a;
  for (size_t i = 0; i < a.size(); ++i)
    a[i] = i + 1; // no bounds checking
}
```

[48] https://en.cppreference.com/w/cpp/container/array

std::array

- ... can be returned (unlike C-style arrays)

```
std::array<int, 10> squares() {
  std::array<int, 10> res = {}; // zero-initialize all elements
  for (size_t i = 0; i < a.size(); ++i)
    res[i] = i * i;
  return res;
}
```

- ... can be passed as parameter (unlike C-style arrays)

```
// NB: src is copied by value, might be expensive!
// Prefer const std::array<int, 10>& src instead. (btw, don't write this code)
void copy(std::array<int, 10>& dst, std::array<int, 10> src) {
  assert(dst.size() == src.size() && "size␣mismatch!");
  for (size_t i = 0; i < dst.size(); ++i)
    dst[i] = src[i];
}
```

For-Range Loop

- Syntax: `for (range-declaration : range-expression) loop-statement`
- Execute loop body for every element in range expression

```
std::array<int, 3> a = {1, 2, 3};
for (int& elem : a)
  elem *= 2;
// a is now {2, 4, 6}

for (const int& elem : a)
  std::println("{}", elem);
```

Special Case: String Literals

- String literals are immutable null-terminated character arrays
- Type of literal with N characters is `const char[N+1]`

- Artifact of C compatibility
- Generally avoid, use `std::string_view` or `std::string` instead
- Occasionally needed for interfacing with C APIs

String Literals

Quiz: What does the function `f` return?

```
size_t f() { return sizeof("Hello!"); }
```

A. (compile error) B. impl.-defined C. 5 D. 6 E. 7

Pointers[49]

- Syntax: `type* cv declarator`
  - As for references/arrays/functions, the `*` is part of the declarator
- No pointers to references, cv qualifies the pointer itself

- Points to an object, stores *address* of first object byte in memory
- Pointers are objects (unlike references)
- Like reference, pointers can dangle

```
int* a; // pointer to (mutable) int
const int* a; // pointer to const int
int* const a; // const pointer to (mutable) int
const int* const a; // const pointer to const int

int** e; // pointer to pointer to int
```

[49] https://en.cppreference.com/w/cpp/language/pointer

Address-Of Operator[50]

- Operator &: obtain pointer to object
- Opeand must be an lvalue expression, cv-qualification is retained

```
int a = 10;
int* ap = &a;
const int c = 20;
const int* cp = &c;
int* cp2 = &c; // ERROR: cannot convert const int* to int*

int& r = a; // Reference to a
int* rp = &r; // Pointer to a
```

[50] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_address-of_operator

Indirection Operator[51]

- Operator $*$: obtain lvalue reference to pointed-to object
- Operand must be a pointer, cv-qualification is retained
- Also referred to as *pointer dereference*

```
int a = 10;
int* ap = &a;
int& ar = *ap;
ar = 20; // a is now 20
*ap = 4; // a is now 4
```

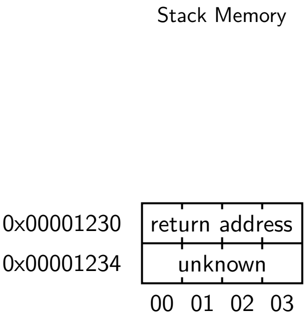
[51] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_indirection_operator

What is Happening? (1)

```c
int main() {



}
```

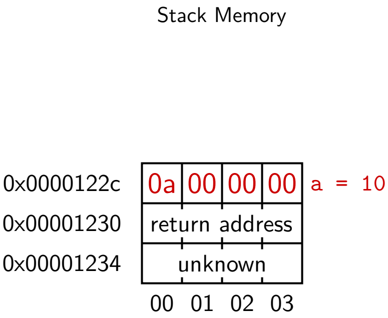


What is Happening? (2)

```
int main() {
    int a = 10;



}
```



What is Happening? (3)

```
int main() {
    int a = 10;
    int b = 123;



}
```

Stack Memory

0x00001228	7b	00	00	00	b = 123
0x0000122c	0a	00	00	00	a = 10
0x00001230	return address				
0x00001234	unknown				
	00	01	02	03	

What is Happening? (4)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;

}
```

Stack Memory

0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	0a	00	00	00	a = 10
0x00001230	return address				
0x00001234	unknown				
	00	01	02	03	

What is Happening? (5)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;



}
```

Stack Memory

0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	2a	00	00	00	a = 42
0x00001230	return address				
0x00001234	unknown				
	00	01	02	03	

What is Happening? (6)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;


}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	2a	00	00	00	a = 42
0x00001230	return address				
0x00001234	unknown				

What is Happening? (7)

```c
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;



}
```

Stack Memory

	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	2c	12	00	00	c = 0x122c
0x00001228	7b	00	00	00	b = 123
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				

What is Happening? (8)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;


}
```

Stack Memory

Address	00	01	02	03	
0x00001220	24	12	00	00	d = 0x1224
0x00001224	28	12	00	00	c = 0x1228
0x00001228	7b	00	00	00	b = 123
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				

What is Happening? (9)

```c
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;


}
```

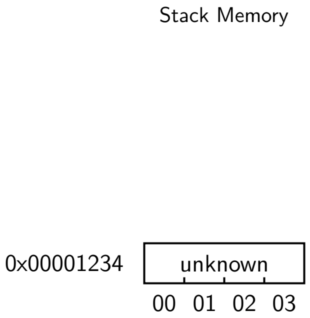
Stack Memory

0x00001220	24	12	00	00	d = 0x1224
0x00001224	28	12	00	00	c = 0x1228
0x00001228	18	00	00	00	b = 24
0x0000122c	41	01	00	00	a = 321
0x00001230	return address				
0x00001234	unknown				
	00	01	02	03	

What is Happening? (10)

```
int main() {
    int a = 10;
    int b = 123;
    int* c = &a;
    *c = 42;
    int** d = &c;
    **d = 321;
    *d = &b;
    **d = 24;

    return 0;
}
```



Pointers to References?

Quiz: Why are pointers to references impossible?

A. References are not objects and thus have no address.
B. Would be redundant to pointers to pointers.
C. Taking the address of the referenced object

Null Pointers[52]

- Pointer can point to object, or nowhere (null pointer)
- Null pointer has special value `nullptr`
- Null pointers of same type are considered as equal
- Dereferencing null pointers is **undefined behavior**

```
int safe_deref(const int* x) { // just as an example
  if (x == nullptr)
    return 0;
  return *x;
}
```

[52] https://en.cppreference.com/w/cpp/language/pointer#Null_pointers

Null Pointers

Quiz: Which answer is NOT correct?

```
int safe_deref2(const int* x) {
  int v = *x;
  if (x == nullptr)
    return 0;
  return v;
}
```

A. The compiler can simply remove the null check.
B. The program might crash when `nullptr` is passed.
C. The program might return zero.
D. The null check prevents an invalid pointer dereference.

Subscript Operator[53]

- Treat pointer as pointer to first element of an array
- Follow the same semantics as the array subscript

```
std::array<int, 3> arr = {12, 34, 45};
const int* ptr = &arr[0]; // pointer to first element, no dereference

for (unsigned i = 0; i < 3; i++)
  std::println("{}", ptr[i]);
```

- C-style arrays often implicitly decay to pointers to the first element

```
int arr[] = {12, 34, 45};
const int* ptr = arr; // pointer to first element
```

[53] https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator

Pointer Arithmetic: Addition[54]

- `ptr + idx`/`ptr - idx`: move pointer `idx` *elements* to left/right
  - Moves underlying address by `idx * sizeof(*ptr)`
- `ptr[idx]` equals `*(ptr + idx)`; `&ptr[idx]` equals7 `ptr + idx`

```
std::array<int, 3> arr = {12, 34, 45};
const int* ptr = &arr[1]; // pointer to second element

// prints: 12 45
std::println("{}␣{}", *(ptr - 1), *(ptr + 1));
```

[54] https://en.cppreference.com/w/cpp/language/operator_arithmetic#Additive_operators

Pointer Arithmetic: Past-The-End Pointers

- Only valid pointers are allowed to be dereferenced
- Pointers shall point to valid objects or be `nullptr`
- Exception: pointer past the end of the last element is allowed

⇝ Constructing out-of-bounds pointers is **undefined behavior**

```
std::array<int, 3> arr = {12, 34, 45};
const int* begin = &arr[0]; // OK, points to first element
const int* end = &arr[arr.size()]; // OK, past-the-end pointer

for (const int* p = begin; p != end; ++p) // OK
  std::println("{}", p);

int v = *end; // NOT OK: dereferencing past-the-end pointer
int* oobPtr = begin + 4; // NOT OK: pointer out of bounds
```

Pointer Arithmetic: Subtraction

- Assuming two pointers ptr1 and ptr2 point into the same array
- ptr1 - ptr2 is the number of elements between the pointers

```
#include <cstddef>
int main() {
  int array[3] = {123, 456, 789};
  const int* ptr1 = &array[0];
  const int* ptr2 = &array[3]; // past-the-end pointer

  std::ptrdiff_t diff1 = ptr2 - ptr1; // 3
  std::ptrdiff_t diff2 = ptr1 - ptr2; // -3
}
```

String Literals Quiz

Quiz: What is the output of the program?

```
#include <print>
int main() {
  std::println("{}", "Hello!" + 3);
}
```

A. (compile error) B. (undefined behavior) C. `"Hello!3"` D. `"lo!"` E. (an address)

Don't use the preprocessor like this, this is primarily for illustration.

Void Pointer[55]

- Pointer to `void` is allowed
- Pointers can be implicitly converted to void pointer (retaining cv-quals)
- To use void pointer, it must be casted to a different type

- Used to pass object of unknown type
- Often used in C interfaces (e.g., `malloc`)
- Tentatively avoid in C++

[55] https://en.cppreference.com/w/cpp/language/pointer#Pointers_to_void

static_cast[56]

- `static_cast<new type>(expression)`
- Cast expression to "related" type, must be at least as cv-qual'ed
  - E.g., cast from void pointer to pointer of different type
  - Many more cases, see reference

```
int i = 42;
void* vp = &i; // OK, no cast required
int* ip = static_cast<int*>(vp); // OK
long* lp = static_cast<long*>(ip); // ERROR
long* lp = static_cast<long*>(vp); // Undefined behavior!

double d = static_cast<double>(i);
```

[56] https://en.cppreference.com/w/cpp/language/static_cast

reinterpret_cast[57]

- `reinterpret_cast<new type>(expression)`
- Cast expression to “unrelated” type, reinterpreting bit pattern
- Very limited set of allowed conversions
  - E.g., converting pointer to object to pointer to `char` or `std::byte`
- Invalid conversions usually lead to **undefined behavior**
- Only use when strictly required! Also avoid C-style casts

[57] https://en.cppreference.com/w/cpp/language/reinterpret_cast

Strict Aliasing Rule

- Object access with an expression of a different type is **undefined behavior**

⇒ Accessing an `int` through a `float*` is not allowed (pointer aliasing)

⇒ Compilers assume that pointers of different types have different values

- (There are few exceptions)

```
float f = 42.0f;
// Undefined behavior!
int i = *reinterpret_cast<int*>(&f);
```

Pointers are Actually Complex

- Pointers generally consist of the address of the pointed-to object
- But: pointers have more semantic information (provenance[58])
  - Pointers have "information" about the underlying object
  - Used for compiler optimization
- Some hardware platforms have unusual addressing schemes
  - E.g., CHERI with 128-bit capabilities, basically pointer with bounds and permissions

[58] `https://www.ralfj.de/blog/2020/12/14/provenance.html`

Pointers vs. References

	Reference	Pointer
Usable for passing-by-reference?	Yes	Yes
Guaranteed non-null?	Yes	No
Is an object itself?	No	Yes
Can change which object is referred to?	No	Yes
Supports pointer arithmetic?	No	Yes

Recommendation (we will revisit this later):

- Prefer references for pass-by-references
- Use pointer for: optional references (`nullptr`), pointer changes object, pointer arithmetic required, storing references in an array

References, Arrays, Pointers – Summary

- Value classes lvalues (locations) and rvalues
- References are aliases to other objects
- Rvalue references extend lifetime of temporary objects
- Arrays contiguously store multiple elements of same type
- String literals are a special case of an array
- Pointers are objects that point to other objects, or `nullptr`
- Pointers support arithmetic
- Pointer casts are possible, but are often invalid

References, Arrays, Pointers – Questions

- Why are arrays of references impossible?
- How can the object referenced by a reference be changed?
- How to pass an object by-reference in C++?
- What is the difference between lvalue and rvalue references?
- What is different between const-lvalue and rvalue references?
- What is the relation between arrays and pointers?
- Which operations on pointers are undefined behavior?
- When is using pointer advisable over using a reference?