Lec-4: Arrays, Pointers and References

Bernard Nongpoh

Array Declaration (1)

- An array declaration declares an object of array type
  - `type declarator[expression]`
  - Expression must be an expression which evaluates to integral constant at **compile time**

- For example: `T a[N];` for some type `T` and compile-time constant `N`
  - `a` consists of `N` contiguously allocated elements of type `T`
  - Elements are numbered **0,...,N-1**
  - Elements can accessed with the subscript operator [ ]
    - **a[0]...a[N-1]**
  - Without an initializer, every element of a is uninitialized

Array Declaration (2)

```
int a[10];
for(int i=0;i<10;i++){
a[i]=i+1;
}
```

- Array objects are lvalues, but they cannot be assigned to

```
int a[10];
int b[10];
a=b; //  error: array type 'int[10]' is not assignable
```

- Arrays cannot be returned from functions

```
int[] foo(); // Error
```

Source

Array Initialization (1)

Array can be default-initialized, in which case every element is default-initialized

```
unsigned short a[10] ={}; // a contains 10 zeros
```

Array can be list-initialized, in which case the size may be omitted

```
unsigned short a[] = {1,2,3,4,5,6,7,8,9,10};
```

Source

Array Initialization (2)

Multi-dimensional arrays may also be list-initialized, but only the first dimension may have unknown bound

```
int b[][2] = {
{0,1},
{2,3},
{4,5}
};
```

Source

size_t

C++ has a designated type for indexes and `std::size_t` from `<cstddef>`

- `size_t` is an unsigned integer type that is large enough to represent sizes and all possible array indexes on the target architecture
- The C++ language and standard library use `size_t` when handling indexes or sizes
- Generally, use `size_t` for array indexes and sizes

```
int arr[5] = {10, 20, 30, 40, 50};

for (std::size_t i = 0; i < 5; ++i) {
    std::cout << arr[i] << " ";
}
```

Source

std::array (1)

- C-Style arrays should be avoided whenever possible
- Use the std::array type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking
- `std::array` is a template type with two template parameters (the element type and count

```
#include <array>
int main(){
    std::array<unsigned short,10> a; // Garbage values
    for (std::size_t i = 0; i < a.size(); ++i) {
        std::cout << a[i] << " "; // No bound checking
    }
}
```

Source

std::array (2)

- C-Style arrays should be avoided whenever possible
- Use the std::array type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking
- `std::array` is a template type with two template parameters (the element type and count

```
#include <array>
int main(){
    std::array<unsigned short,10> a={}; // All zeros
    for (std::size_t i = 0; i < a.size(); ++i) {
        std::cout << a[i] << " ";       }
}
```

Source

std::array (3)

- C-Style arrays should be avoided whenever possible
- Use the std::array type defined in the `<array>` standard header instead
- Same semantics as a C-style array
- Optional bounds-checking
- `std::array` is a template type with two template parameters (the element type and count

```
#include <array>
int main(){
    std::array<unsigned short,10> a={1,2};
    // a[0] = 1, a[1] = 2, remaining elements are value-initialized to 0
    for (std::size_t i = 0; i < a.size(); ++i) {
        std::cout << a[i] << " ";
     }
}
```

Source

std::vector (1)

- std::array is inflexible due to compile-time fixed size
- The std::vector type defined in the <vector> standard header provides dynamic-sized arrays
- Storage is automatically expanded and contracted as needed
- Elements are still stored contiguously in memory

```
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
    std::vector<int> v = {1, 2, 3};

    for (std::size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
```

Source

std::vector (2)

- Initialization

```
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
    std::vector<int> v(5); // size = 5, All elements are value-initialized to 0

    for (std::size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
```

Source

std::vector (3)

- Initialization: Define size and initial value

```
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
    std::vector<int> v(5,10); Size = 5. All elements initialized to 10

    for (std::size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
```

Source

`std::vector` (3)

- Initialization: Define size and initial value

```
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
    std::vector<int> v(5,10);  Size = 5. All elements initialized to 10

    for (std::size_t i = 0; i < v.size(); ++i) {
        std::cout << v[i] << " ";
    }
}
```

Source

`std::vector`: useful functions (1)

- `push_back` - insert an element at the end of the vector
- `size` - queries the current size
- `clear` - clears the contents
- `resize` - change the number of stored elements
- The subscript operator can be used with similar semantics as for C-style arrays

Source

std::vector: useful functions (2)

```
#include <stddef.h>
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t
int main() {
     std::vector<int> a;
     for(size_t i=0;i<10;i++)
        a.push_back(i);
     std::cout<<a.size()<<std::endl; // prints 10
     a.clear();
     std::cout<<a.size()<<std::endl; // prints 0
     a.resize(10); // a now contains 10 zeros
     std::cout<<a.size()<<std::endl; // prints 10
}
```

Source

Range-For (1)

- Execute a for-loop over a range

```
for (init-statement; range-declaration:range-expression)
     loop statement
```

- Executes ***init-statement*** once, then executes ***loop-statement*** once for each element in the range defined by range-expression
- ***range-expression*** may be an expression that represents a sequence (e.g. an array or an object for which begin and end functions are defined, such as `std::vector`)
- ***range-declaration*** should declare a named variable of the element type of the sequence, or a reference to that type
- ***init-statement*** may be omitted

Source

Range-For (2)

```
#include <stddef.h>
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
   std::vector<unsigned short> a={1,2,3};
   for(const unsigned short v:a){
       std::cout<<v<<std::endl;
   }

return 0;
}
```

Source

Range-For (2)

```
#include <stddef.h>
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
   std::vector<unsigned short> a={1,2,3};
   for(const unsigned short v:a){
       std::cout<<v<<std::endl;
   }

return 0;
}
```

```
#include <stddef.h>
#include <vector>
#include <iostream>
#include <cstddef>   // std::size_t

int main() {
   std::vector<unsigned short> a={1,2,3};
// C++ 20
for(int sum=0;const unsigned short v:a){
       sum+=v;
       std::cout<<"Running sum = "<<sum<<std::endl;
}
   return 0;
}
```

Source

References (1)

- A **reference** is an **alias (another name)** for an existing object
- It does **not** create a new object and does **not** own memory

```
int x = 10;
int& r = x;    // r is an alias for x
```

- `x: variable`
- `10: Memory content`
- `r: alias or reference`
- x and r refer to the **same memory location**
- Any modification through one is visible through the other

Source

References (2)

```cpp
#include <iostream>
int main() {
   int x = 10;
   int& r = x;   // reference (alias)
   std::cout<<"&x="<<&x<<std::endl;
   std::cout<<"&r="<<&r<<std::endl;
   r = 20;       // modify via reference
   std::cout << x << std::endl;  // prints 20
   x=10;
   std::cout << x << std::endl;  // prints 10
   return 0;
}
```

```
&x=0x16ee5b338
&r=0x16ee5b338
20
10
```

- References **do not have their own storage**
- &r is the same as &x
- Modifying via reference modifies the original variable
- Modifying the original variable reflects in the reference

Source

References (3)

```
int &j;
int &i = 10;
int &k = i + j;
```

```
test.cpp:3:7: error: declaration of reference variable 'j' requires an initializer
    3 |  int &j;
      |       ^
test.cpp:4:6: error: non-const lvalue reference to type 'int' cannot bind to a temporary of type 'int'
    4 | int &i = 10;
      |      ^   ~~
test.cpp:5:6: error: non-const lvalue reference to type 'int' cannot bind to a temporary of type 'int'
    5 | int &k = i + j;
      |      ^   ~~~~~
3 errors generated.
```

Source

References (4)

```
int &j;
int &i = 10;
int &k = i + j;
```

```
test.cpp:3:7: error: declaration of reference variable 'j' requires an initializer
    3 |  int &j;
      |       ^
test.cpp:4:6: error: non-const lvalue reference to type 'int' cannot bind to a temporary of type 'int'
    4 | int &i = 10;
      |      ^   ~~
test.cpp:5:6: error: non-const lvalue reference to type 'int' cannot bind to a temporary of type 'int'
    5 | int &k = i + j;
      |      ^   ~~~~~
3 errors generated.
```

- Uninitialized Reference
  - A reference **must be initialized**
  - A reference is an **alias**, not a variable
- Reference to Temporary
  - `5` is a **temporary (rvalue)**
  - Non-Const references **cannot bind to temporaries**
- Reference to Expression Result
  - `i + j` creates a **temporary**
  - Temporary has **no stable lifetime**

[Source]

References (5)

```
int a;
int &j=a;
const int &i = 10;
const int &k = i + j;
```

- References **must be initialized**
- Non-const references **cannot bind to temporaries**
- Expressions produce **rvalues**
- `const T&` can bind to temporaries

A reference must always bind to a real object with a valid lifetime, never to a temporary or uninitialized value

Source

References (6)

```cpp
#include <iostream>
int main() {
      int a;
      int &j=a;
      const int &i = 10;
      const int &k = i + j;
      std::cout<<"&a"<<&a<<std::endl;
      std::cout<<"&j"<<&j<<std::endl;
      std::cout<<"&i"<<&i<<std::endl;
      std::cout<<"&k"<<&k<<std::endl;
return 0;
}
```

- Const tells compiler to allocate a memory with a value 10, similarly for reference k.

```
&a=0x16ba57338
&j=0x16ba57338
&i=0x16ba57324
&k=0x16ba57314
```

Source

Call-by-Reference (1)

```
void foo(int &a, int b) {
    std::cout << "&a=" << &a << std::endl;
    a = 7;
    b = 8;
    std::cout << "&b=" << &b << std::endl;
}

int main() {
    int a = 10;
    foo(a, a);
    std::cout << "a=" << a << std::endl;
}
```

```
&a=0x16f68f338
&b=0x16f68f314
a=7
```

Source

Actual Parameters (Caller Side)

- a in `main` is the **actual argument**
- Both arguments in `foo(a, a)` refer to the **same variable**
- Value of a before call: `10`
- Address of a is fixed in `main`

Formal Parameters (Callee Side)

- `int& a` → **reference parameter**
  - Becomes an **alias** to caller's a
  - &a inside `foo` equals address of `main`'s a
- `int b` → **value parameter**
  - A **new local copy** is created
  - &b is a **different address**

Swapping Values in C++: Value vs Address vs Reference

Call-by -Value

```
Void swap(int x, int y);
void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
}
int main(){
int a = 10, b = 20;
swap(a, b);
std::cout<<"a="<<a<<std::endl; // a=10
std::cout<<"b="<<b<<std::endl; // b=20
}
```

- x and y are **copies** of a and b
- Swapping affects only local copies
- Original variables remain unchanged

Source

Swapping Values in C++: Value vs Address vs Reference

Call-by-Address

```
void swap(int* x, int* y);
void swap(int* x, int* y) {
   int temp = *x;
   *x = *y;
   *y = temp;
}
int main(){
int a = 10, b = 20;
swap(&a, &b); // Unnatural call
std::cout<<"a="<<a<<std::endl;
std::cout<<"b="<<b<<std::endl;
}
```

- Addresses of a and b are passed
- Dereferencing allows modification of originals
- Requires explicit * and &

Source

Swapping Values in C++: Value vs Address vs Reference

Call-by-Reference

```
void swap(int& x, int& y);
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
int main(){
     int a = 10, b = 20;
     swap(a, b); // Natural call
     std::cout<<"a="<<a<<std::endl;
     std::cout<<"b="<<b<<std::endl;
}
```

- x and y are **aliases** of a and b
- No copies, no pointers
- Clean and safe syntax

Source

Const Reference Parameters (1)

- A **`const` reference parameter** (`const T&`) also aliases the argument
- However, the function is **not permitted to modify** the argument through this reference

```
void g(const int& x) {
    // x = 10;    // compile-time error
}
```

- `const` applies to **access through the reference**, not to the object itself
- The object may still be modified **through other aliases**

Const Reference Parameters (2)

Without `const` Reference

```cpp
#include<iostream>
int increment(int& x);
int increment(int& x) {
      x++;
      return x;
}
int main(){
      int a = 5;
      int b = increment(a);
      // a = 6, b = 6
      std::cout<<"a="<<a<<std::endl;
      std::cout<<"b="<<b<<std::endl;
}
```

- `x` is a **non-const reference**
- Function **modifies the caller's variable**
- Side effect is visible outside the function

Const Reference Parameters (2)

`With const` Reference

```
#include<iostream>
int increment(const int& x);
int increment(const int& x) {
      return x+1;
}
int main(){
      int a = 5;
      int b = increment(a);
      // a = 6, b = 6
      std::cout<<"a="<<a<<std::endl;
      std::cout<<"b="<<b<<std::endl;
}
```

- `x` is a **read-only alias**
- Function does **not modify** the caller
- Pure computation

Return by Reference (1)

```
int& max(int& a, int& b) {
    return (a > b) ? a : b;
}

// main func
int main(){
int x = 10, y = 20;
int l=max(x,y);
std::cout<<l<<std::endl;
max(x, y) = 100;
std::cout<<y<<std::endl;

}
```

- The function returns an **alias** to an existing object
- **No copy** is made
- The returned reference refers to an object that **must outlive the function**
- `y` becomes `100`
- Returned reference aliases either `x` or `y`

Return by Reference (2)

```
int& bad() {
    int x = 10;
    return x;   // dangling reference
}
```

- x is destroyed when function returns
- Reference becomes invalid
- **Undefined behavior**

warning: reference to stack memory associated with local variable 'x' returned [-Wreturn-stack-address]

Return by Reference (3)

```
static int& ok() {
    static int x = 10;
    return x;       // lifetime is static
}
```

Pointers vs References (1)

<table>
<tr><th>Pointers</th><th>References</th></tr>
<tr><td>Refers to an address (exposed)</td><td>Refers to an address (hidden)</td></tr>
<tr><td>Pointers can point to NULL / nullptr</td><td>References cannot be NULL</td></tr>
<tr><td><code>int* p = NULL;</code>
<code>// p is not pointing to any object</code></td><td><code>int& j; // ERROR</code>
<code>// wrong: reference must be initialized</code></td></tr>
<tr><td>Can point to different variables at different times</td><td>Referent is fixed for the lifetime</td></tr>
<tr><td><code>int a, b; int* p;</code>
<code>p = &a;</code>
<code>...</code>
<code>p = &b;</code></td><td><code>int a, c; int& b = a;</code> // OK
<code>...</code>
<code>b = c;</code> // (assignment, not reseating)</td></tr>
<tr><td>NULL checking is required</td><td>NULL checking not required</td></tr>
</table>

Source

Pointers vs References (2)

Pointers	References
Allows users to **operate on the address**	Does **not allow operating on the address**
Supports pointer arithmetic (`p++`, `p+1`)	No arithmetic on references
Array of pointers allowed	**Array of references not allowed**
Syntax uses `*` and `->`	Syntax looks like normal variable
More flexible but error-prone	Safer, clearer intent

Source