Arrays and Dynamic Memory

James Daly
CSE 232
Michigan State University
Good old-fashioned C arrays

- Just like Mom used to program with
An array is a **contiguous, fixed-size** piece of memory

- Cannot grow, cannot change size
- A sequence of elements

The values within the contiguous chunk can be addressed individually

Worth remembering: just one big chunk of memory, larger than an individual typed variable
Not objects, no methods

- As a big ole chunk of memory, these are **not C++ objects**
  - No internal structure
    - For example, no size information
  - No method calls
- We can do some C++ things to an array, but it takes some work
C++11 array vs good ole C array

- It is worth noting that C++11 has an object called `array` with equivalent functionality to a C-array
  - It is in fact an object, a fixed size sequence
  - It has some internal structure
    - It knows its size

- We study C-arrays here
  - The concept of a C-array is so pervasive it is worth studying
  - One time we don’t follow the latest stuff in the C++11 standard
Array

- Example 15.1
C-style array

- **Syntax**
  - `type array_name[capacity];`
    - `type` is any type (predefined or programmer-defined)
    - `array_name` is an identifier
    - `capacity` is the number of slots (indexing starts at 0)
      - The size of the array is the `type_size * capacity`
Declarations

const size_t num = 3;
int int_ary[num]; // array of 3 integers
double dbl_ary[num]; // array of 3 doubles
string str_ary[num]; // array of 3 strings

- Storage, e.g. 4-bytes per int
size_t

- Just as every STL object has a size type, there is a generic size type (an unsigned integer) that can be used for non-object array sizes.

size_t ary_size = 100;
const for capacity

- Good programming practice
  - Use const for capacity of c-style arrays
  
  ```
  const size_t max=5;
  int fred[max];
  for (size_t i = 0; i < max; i++) {} 
  ```
- If size needs to be changed, only the capacity max needs to be changed
Operations

- `int ary[3]; // array of 3 ints`
- **Subscript**
  - **Assignment**: `ary[0] = 6;`
- **Input / Output**
  - The elements are handled as their types
  - *e.g.* `cout << ary[0] << endl; // int 6`
- **Arithmetic:**
  - `ary[1] = ary[0] + 5;`
Initialization

- **Syntax:** `int ary[4] = {2, 4};`
- **Behavior:** Initialize elements starting with leftmost, i.e. element 0.
  - Remaining elements are initialized to zero.
  
  ```
  ary[0] = 2; ary[1] = 4; ary[2] = 0; ary[3] = 0;
  ```

- **Compiler can also determine size**
  - `int ary[] = {0, 1, 2}; // size 3`
Type is important

- First, each array needs a type so the size of memory requested can be calculated (number of elements * size of type)
- Because of this, each array can only hold elements of the same type
  - Mostly. There always a way around these things 😊
Arrays and Pointers

▪ “degrading” an array
▪ Example 15.2
Array vs pointer

- When you have a big chunk of memory of some fixed size, there are really two ways to look at it:
  - As an array with some fixed size
    - Not stored in the array remember!
  - As a pointer to the beginning of the memory chunk
Array pointers

- You could view ary as an int* pointer to the first element of the array chunk
- That is
  - *ary == ary[0];
  - *(ary + 1) == ary[1];
  - ary++; // Don’t do that, why?
Mostly equivalent way to express index

- One could view the subscript index as an address offset from the beginning pointer to the array.
- Remember pointer arithmetic is based on “element” math.
  - `ptr + 1` points to the next `value`.
  - Address goes up by the size of the type to get to the next value.
Array type vs Pointer type

- C++ is sensitive to knowing the size of the array:
  - If the compiler knows the size then it allows you to do things like for-each
  - If the compiler cannot know the size, it treats it like a pointer and C++ things won’t work
- We say **degrading** the array to a pointer
const size_t size = 5;
int ary1[size]{8, 5, 6, 7, 5};
ary1[1] = 25;

for (auto element : ary1)
    cout << "Element:" << element << endl;

char ary2[]{‘a’, ‘b’, ‘c’, ‘d’};
for (auto element : ary2)
    cout << "Element:" << element << ‘,’;
cout << endl;

Compiler knows, or can infer the sizes so we can do range stuff like a for-each loop
const size_t size = 5;
int ary1[size] = {8, 5, 6, 7, 5};
int *ptr = ary1;

for (int *p = ary1; p < (ary1 + size); p++)
    cout << "Element:" << *p << endl;

for (auto e : ptr)
    cout << *e << endl;

In the first loop, we use a regular for to iterate through the pointers

In the second, the pointer is not an array type, for-each won’t work

C++ can’t infer the size anymore
Pointers and Iterators

- For the most part, you can treat a pointer as an iterator if you want to run generic algorithms on an array
  - However no `.begin()` or `.end()` operators, not C++ objects
  - Remember, C++ wants the end to be **one past** the last element you care about
const size_t size = 5;
int ary1[size] {8, 5, 6, 7, 5};

int *ptr_ary1_front = ary1;
int *ptr_ary1_back = ary1 + size;

sort(ptr_ary1_front, ptr_ary1_back);
copy(ptr_ary1_front, ptr_ary1_back,
     ostream_iterator<int>(cout, "\n");

Set up the pointer the way you want (by hand) and you can run generic algorithms on your array.
Begin and end functions

- Objects have methods `.begin()` and `.end()` to provide iterators to their respective start and finish+1.
- Arrays have no methods. C++ provides functions `begin()` and `end()` to give us the start and finish+1 as pointers if the compiler knows the array size.
int ary1[5] = {1, 2, 3, 4, 5};

copy(begin(ary1), end(ary1),
    ostream_iterator<int>(cout, " "));

Size is fixed so compiler knows size

With known size, compiler can figure the begin and end address
Arrays as parameters

- Example 15.3
3 ways

- 3 ways to pass an array to a function
- Note, it is always a pointer or a reference
  - Never a copy
- 2 ways to degrade the array to a pointer
- 1 way passes as a reference with size info maintaining the full array type
First way

- **Syntax:** `int sum(int ary[])`
  - `[]` indicates the parameter is an array
  - No size info in that array
  - Is implicitly a pointer!
  - No info on the size of the array
    - Size is required to be passed separately
    - `int sum(int ary[], size_t size)`
Second way, directly as a pointer

- Syntax: `int sum(int *ary, size_t size)`
  - Indicates the parameter is a pointer
  - You can still do subscripting on the array in the function
  - No size info again
int sum(long *ary, size_t size) {
    int sum = 0;
    cout << "Size:" << sizeof(ary) << endl;
    for (size_t i = 0; i < size; i++) {
        sum += ary[i];
        // sum += *(ary + i); // equivalent
    }
    return sum;
}

Either phrasing results in the same thing
- pointer to a chunk of memory
- fixed size, no size available in the array
- a degraded array type
- sizeof(ary1) yields the size of a single pointer
Third way

- If you set the size (somehow) in the function call itself, then the compiler can figure out how to do thing like for-each
  - Array type is preserved and the array is not degraded
long prod(const long (&ary)[3]) {
    long result = 1;
    cout << "Size:" << sizeof(ary) << endl;
    for (auto &element : ary) {
        result = result * element;
    }
    return result;
}

int main() {
    long ary1{1, 2, 3};
    prod(ary1)
}
template<typename Type, size_t Size>
long squares(const Type(&ary)[Size]) {
    long result = 0;
    cout << "Size of info:" << sizeof(ary) << endl;
    for (auto element : ary)
        result = result + (element * element);
    return result;
}

Very nice. Allows the compiler to deduce the size (without us setting it explicitly as before) via template, and instantiate the temple to new size of each array. Again, some challenging syntax here.
Dynamic memory

- Memory on demand
- Example 15.4
- Example 15.5
Compile time vs run time

- Good to remind ourselves
  - Compile time: what is known at the time of running the compiler to make an executable
  - Run time: what is known when the user actually runs the executable
STL objects vs us

- STL objects know how to get more memory during **runtime**
  - We love them for this. Vectors, maps, etc. can get bigger when we ask them to as they run
- For things like arrays, fixed-size non-object:
  - They are a fixed size at compile time!
How does the STL do it?

- Underlying the STL is the ability to ask for and release memory during runtime
- We can do the same if we wish but
  - We must be careful. Many (many, many) programmers make mistakes at this point
  - If the STL can do it, let it. It is better at it!
C++11 or the old way

- For once, I want to talk about the “old” way to do dynamic memory, not the latest C++11 way:
  - You may rarely see the new way
  - There are some complications that are a little much
  - Nonetheless, Example 15.5 uses `shared_ptr` and `unique_ptr` (also look in the book)
So simple, two functions

- **new**
  - Gets / allocates memory from the os
  - Returns a **pointer** to it (single object or array of those objects)

- **delete**
  - de-allocates the memory, gives it back to the OS
Process Memory

- **Heap**
  - Used explicitly
  - Think malloc & new
  - Grows towards stack
  - May have “holes”
  - May overflow (run into Stack)

- **Stack**
  - Used implicitly
  - Think push and pop
  - Grows towards Heap
  - May not have “holes”
  - May overflow (run into heap)
Stack Frame

Local Variables

Previous Base Ptr
Return Address
Arg 1
Arg 2
...
Arg N

Stack Frame

Low Address

Text (code)
Data (initd vars)
BSS (uninit stuff)
Heap

Stack Frame

Stack

High Address
Ownership of memory

- The requests from `new` and `delete` do not change memory in any way, they simply mark a segment of memory as to who “owns” it
  - If you `new` some memory, the OS marks that memory in the heaps as **yours**
  - If you never `delete` it, while the program runs the OS thinks it is gone.
    - i.e. It can’t use it
Ownership

- If you delete some memory, you are simply ceding ownership back to the OS
  - The OS is now free to give the memory to some other program
  - No contents are ever changed by the OS! Until the OS gives it to another program and that program changes the memory, that memory looks like how your program left it.
new

new type(init)
- Allocate new memory of indicated type
  - can optionally provide an init, not required

new type[size]
- Make size elements of type indicated

- Both return a pointer to the new memory
delete

delete ptr;

- delete (remove ownership) of object pointed to by ptr

delete [] ptr;

- for an array, delete (remove ownership) of all the elements
  - ptr points to the beginning of the memory array to be deleted
Constructor call on new memory

- You can make a call to a constructor for the new memory and in this way you can initialize memory
  - Not required if you will fill the memory yourself
  - In general it is a good idea, otherwise the “values” stored in that memory are whatever was left over from the previous user.
Example

```c
int main() {
    // basic new
    long *lptr = new long(1234567);
    cout << "lptr:" << *lptr << endl;
    delete lptr;
}
```
Variable-length arrays

size_T size;
cout << "How big:"; cin >> size;

// Not of an array type : no begin(), end()
long *ary = new long[size];
fill(ary, size);
dump(cout, ary, size);
delete[] ary;
Memory issues
Leaking memory

```c
int main() {
    int reps = 2048;
    const size_t size = 1024;
    long temp = 0;
    for (int i = 0; i < reps; i++) {
        long *ary = new long[size]; // leak
        ary[0] = 0;
        for (size_t j = 1; j < size; j++)
            ary[j] = ary[j - 1] + temp;
        temp = ary[size];
    }
}
```
The leak

- This is **leaking memory**
  - *new* some memory
    - Get a pointer `ptr` to the memory
    - Claim ownership from OS
  - Use the memory (do something)
  - Reasssign `ptr` to some new memory
    - Didn’t delete the memory `ptr` pointed to
    - Can’t access that “orphan” memory now
    - OS doesn’t know that, still marks it as yours
Bottom line

- Your program, while it runs, accumulates ownership of memory from the OS, deleting memory resources
  - The memory footprint of your running program grows
    - Uselessly, you aren’t using that memory
    - Even if you wanted to, you lost the pointer to the memory. It is orphaned
    - OS doesn’t know, can’t reuse that memory
Morale of the story

- It is on you to free/delete memory that you have acquired
  - There are consequences to this and leaking memory is a problem
- The easiest way to avoid this:
  - **Use the STL containers**
  - Avoid the issue
Scope, what is wrong with this function?

```c
long * fn1(size_t sz, long start, long inc) {
    auto ptr = new long[sz];
    ptr[0] = start;
    for (size_t i = 0; i < sz; i++)
        ptr[i] = ptr[i - 1] + inc;
    return ptr;
}
```
Who owns the pointer?

- A ptr in the function points to memory allocated in the function
- The ptr is deleted at the end of the function
  - A copy is returned to the caller
- What happens to the memory?
Local variables are deleted, but not memory

- When the function returns, the ptr goes out of scope, but the memory it points to does not
  - It still has to be deleted. It will leak otherwise
  - Given the way this is set up, the calling program will have to delete it