Dynamic memory management using Pointers

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: March 21, 2025

Pointers and New

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
  - error-prone and might be difficult to understand
  - could have avoided by using arrays and references

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand
    - could have avoided by using arrays and references
- But what if we don’t know the size of required memory at compile-time

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
  - error-prone and might be difficult to understand
  - could have avoided by using arrays and references
- But what if we don't know the size of required memory at compile-time
  - we can allocate memory of size MAX (a constant)

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand
    - could have avoided by using arrays and references
- But what if we don't know the size of required memory at compile-time
    - we can allocate memory of size MAX (a constant)
- Problems with this model:

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand
    - could have avoided by using arrays and references
- But what if we don’t know the size of required memory at compile-time
    - we can allocate memory of size MAX (a constant)
- Problems with this model:
    - might run out of space (despite having a lot of unused memory)

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand
    - could have avoided by using arrays and references
- But what if we don’t know the size of required memory at compile-time
    - we can allocate memory of size MAX (a constant)
- Problems with this model:
    - might run out of space (despite having a lot of unused memory)
    - can be under-utilized (e.g. if we only use a small part of MAX)

Static vs Dynamic allocation

- So far used pointers for allocating space during compile-time only
    - error-prone and might be difficult to understand
    - could have avoided by using arrays and references
- But what if we don't know the size of required memory at compile-time
    - we can allocate memory of size MAX (a constant)
- Problems with this model:
    - might run out of space (despite having a lot of unused memory)
    - can be under-utilized (e.g. if we only use a small part of MAX)
- Solution: allocate memory on demand at run-time!

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

```
int *px = new int;
*px = 777;
cout << *px;
```

- Must free-up space when done, using keyword delete (otherwise memory leak can happen!)

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

```
int *px = new int;
*px = 777;
cout << *px;
```

- Must free-up space when done, using keyword delete (otherwise memory leak can happen!)
    - also clean up any dangling pointers using nullptr

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

```
int *px = new int;
*px = 777;
cout << *px;
```

- Must free-up space when done, using keyword delete (otherwise memory leak can happen!)
  - also clean up any dangling pointers using nullptr

Run-time Allocation

- Allocates memory in *heap* (in contrast to .text, .data, and stack)
- Uses keyword `new` to allocate memory

```
int *px = new int;
*px = 777;
cout << *px;
```

- Must free-up space when done, using keyword delete (otherwise memory leak can happen!)
    - also clean up any dangling pointers using nullptr

```
...
delete px;
px = nullptr;
```

Simple example: Book Record

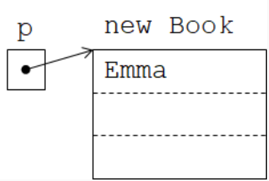
Simple example: Book Record

```cpp
struct Book {
  string title;
  string author;
  string call_number;
};
void printBook(const Book *pBook){
  cout << "title: " << pBook->title << endl;
  cout << "author: " << pBook->author << endl;
  cout << "call number: " << pBook->call_number;
  cout << endl;
}
int main(){
  // allocate a Book from heap
  Book *pb = new Book;
  pb->title = "Security";
  pb->author = "Matt Bishop";
  pb->call_number = "QA.420";
  printBook(pb);
  delete pb; // explicit deallocation
  return 0;}
```

Pictorial representation: Run-time Allocation

Pictorial representation: Run-time Allocation

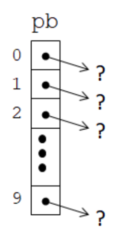
```
Book *p;
p = new Book;
p -> title = "Emma";
```



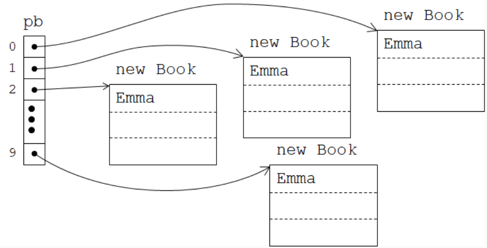
Safely Allocating Memory

Safely Allocating Memory

```
Book *pb[10];

for (int i = 0; i < 10; I++){
  pb[i] = new Book;
  pb[i] -> title = "Emma";
 }
```



...



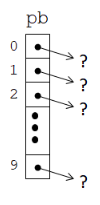
Safely Deallocating Memory

Safely Deallocating Memory

Safely Deallocating Memory

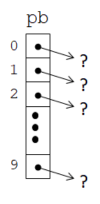
Safely Deallocating Memory

```
for (int i = 0; i < 10; i++)
  delete pb[i];
```



Safely Deallocating Memory

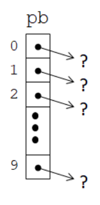
```
for (int i = 0; i < 10; i++)
  delete pb[i];
```



- Dangling pointers!

Safely Deallocating Memory

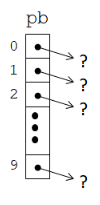
```
for (int i = 0; i < 10; i++)
  delete pb[i];
```



- Dangling pointers!
- assign nullptr to indicate this

Safely Deallocating Memory

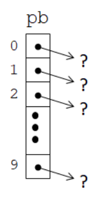
```
for (int i = 0; i < 10; i++)
  delete pb[i];
```



- Dangling pointers!
- assign nullptr to indicate this

Safely Deallocating Memory

```
for (int i = 0; i < 10; i++)
  delete pb[i];
```



- Dangling pointers!
- assign nullptr to indicate this

```
for (int i = 0; i < 10; i++)
  pb[i] = nullptr;
```

The Null Pointer

- If you ever try to dereference `nullptr`, your program will immediately crash

The Null Pointer

- If you ever try to dereference `nullptr`, your program will immediately crash
    - This is *good!*

The Null Pointer

- If you ever try to dereference `nullptr`, your program will immediately crash
    - This is *good!*
    - Crashes right at the point of failure

The Null Pointer

- If you ever try to dereference `nullptr`, your program will immediately crash
  - This is *good!*
  - Crashes right at the point of failure
  - Doesn’t silently fail and access garbage memory

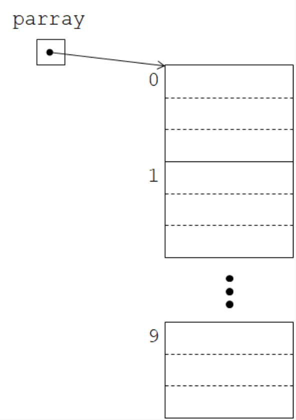
Pointer to dynamic array: Allocation

Pointer to dynamic array: Allocation

Pointer to dynamic array: Allocation

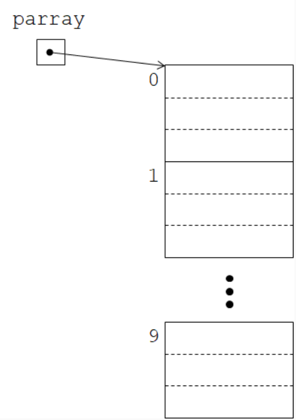
Pointer to dynamic array: Allocation

```
Book *parray = new Book[10];
```



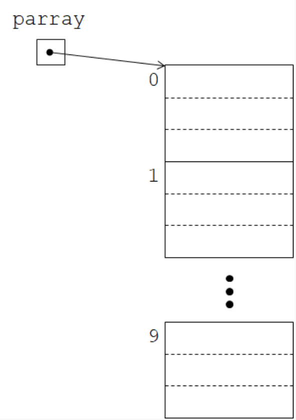
Pointer to dynamic array: Allocation

```
Book *parray = new Book[10];
```

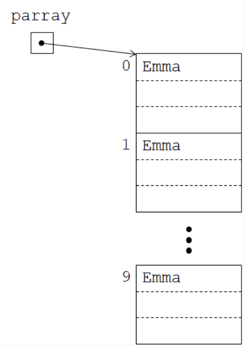


Pointer to dynamic array: Allocation

```
Book *parray = new Book[10];
```



```
for (int i = 0; i < 10; i++)
  parray[i].title = "Emma";

// could have also used:
// (parray+i)->title = "Emma";
// (*(parray+i)).title = "Emma";
```



Pointer to dynamic array: Deallocation

Pointer to dynamic array: Deallocation

```
Book *parray = new Book[10];

for (int i = 0; i < 10; i++)
  parray[i].title = "Emma";

...

delete [ ] parray;

parray = nullptr;
```

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory
    - This is why arrays are always pass-by-reference

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory
    - This is why arrays are always pass-by-reference
- `A[i]` is the same as `*(A + i)`

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory
  - This is why arrays are always pass-by-reference
- `A[i]` is the same as `*(A + i)`
  - Get the value `i` places after the start of the array

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory
    - This is why arrays are always pass-by-reference
- `A[i]` is the same as `*(A + i)`
    - Get the value `i` places after the start of the array
- Why we always need to pass the length of the array

Arrays are just pointers

- An array A is actually just a pointer to the start of the array in memory
  - This is why arrays are always pass-by-reference
- `A[i]` is the same as `*(A + i)`
  - Get the value `i` places after the start of the array
- Why we always need to pass the length of the array
  - Address tells us where it starts, not where it ends

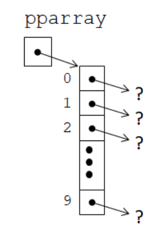
Pointer to array of pointers (double pointer): Allocating

Pointer to array of pointers (double pointer): Allocating

Pointer to array of pointers (double pointer): Allocating

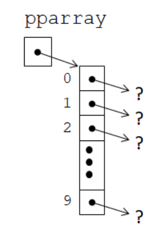
Pointer to array of pointers (double pointer): Allocating

```
Book **pparray = new Book*[10];
```



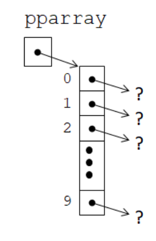
Pointer to array of pointers (double pointer): Allocating

```
Book **pparray = new Book*[10];
```

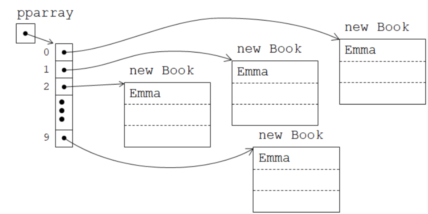


Pointer to array of pointers (double pointer): Allocating

```
Book **pparray = new Book*[10];
```



```
for (int i = 0; i < 10; i++){
  parray[i] = new Book;
  parray[i] -> title = "Emma";
 }
```



Pointer to array of pointers (double pointer): Deallocating

Pointer to array of pointers (double pointer): Deallocating

```
Book **pparray = new Book * [10];

for (int i = 0; i < 10; i++){
  pparray[i] = new Book;
  pparray[i] -> title = "Emma";
 }
```

- Deallocation must be done in the reverse order of allocation

Pointer to array of pointers (double pointer): Deallocating

```
Book **pparray = new Book * [10];

for (int i = 0; i < 10; i++){
  pparray[i] = new Book;
  pparray[i] -> title = "Emma";
 }
```

- Deallocation must be done in the reverse order of allocation

Pointer to array of pointers (double pointer): Deallocating

```
Book **pparray = new Book * [10];

for (int i = 0; i < 10; i++){
  pparray[i] = new Book;
  pparray[i] -> title = "Emma";
 }
```

- Deallocation must be done in the reverse order of allocation

```
for (int i = 0; i < 10; i++){
  delete pparray[i];
  // following is redundant, since we
  // are about to delete parray
  pparray[i] = nullptr;
 }
delete [] pparray;
pparray = nullptr;
```

Collection data structures with maximum capacity

- Example: print in reverse order

Collection data structures with maximum capacity

- Example: print in reverse order

Collection data structures with maximum capacity

- Example: print in reverse order

```
const int CAPACITY = 1000;

int main(){
  int A[CAPACITY];
  int length;

  cin >> length;

  for (int i = 0; i < length; i++)
    cin >> A[i];


  for (int i = length - 1; i >= 0; i--)
    cout << A[i] << endl;

  return 0;
}
```

Collection data structures w/o maximum capacity

- Example: print in reverse order

Collection data structures w/o maximum capacity

- Example: print in reverse order

Collection data structures w/o maximum capacity

- Example: print in reverse order

```
int main(){
  int length;
  int *A;

  // Read length of sequence
  cin >> length;

  // Allocate enough memory to hold
  // sequence
  A = new int[length];

  for (int i = 0; i < length; i++)
    cin >> A[i];

  // Write sequence in rev. order
  for (int i = length - 1; i >= 0; i--)
    cout << A[i] << endl;

  // Deallocate memory
  delete [] A;
```

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:
- allocate a bigger array

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:
- allocate a bigger array
- copy the contents of the old array to the new one

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:
- allocate a bigger array
- copy the contents of the old array to the new one
- deallocate the old array

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:
- allocate a bigger array
- copy the contents of the old array to the new one
- deallocate the old array
- use the new array to store incoming integers until it is filled up again

Dynamically Expanding and Shrinking: IDea

- Initialize the array with some arbitrary capacity
- Insert integers into the array as usual
- When the array is filled up, expand the capacity of the array as follows:
- allocate a bigger array
- copy the contents of the old array to the new one
- deallocate the old array
- use the new array to store incoming integers until it is filled up again
- Deallocate the array when it is no longer needed

Expand/Shrink main function

Expand/Shrink main function

```cpp
int main(){
  // Initialize encapsulated array
  init();

  // Read sequence
  int x;
  cin >> x;
  while (cin){
    append(x);
    cin >> x;
  }

  // Write sequence in reverse order
  for (unsigned int i = length(); i > 0; i--)
    cout << retrieve(i - 1) << endl;

  // Deallocate encapsulated array
  cleanup();

  return 0;
}
```

Initializing

Initializing

```
// Amount of memory available
unsigned int array_capacity = 0;
// Amount of memory used
unsigned int array_length = 0;
// Actual memory resource
int *array = nullptr;

bool isInitialized(){
  return (array != nullptr);
}

void init(){
  assert(! isInitialized());
  // Default initial capacity
  array_capacity = 4;
  // Array is empty initially
  array_length = 0;
  // Allocate array
  array = new int[array_capacity];
  assert(isInitialized());
} // end init()
```

Example: Append for shrinking/growing

Example: Append for shrinking/growing

```
void append(int x) {
  assert(isInitialized());

  // Expand capacity if full
  if (array_length == array_capacity)
    expand();

  // Append to the end
  array[array_length] = x;
  // Update array length
  array_length++;
}
```

Expand for Shrinking/Growing

Expand for Shrinking/Growing

```
void expand() {
  assert(isInitialized());
  assert(array_capacity > 0);
  assert(array_length == array_capacity);

  // Calculate new capacity
  int new_array_capacity = array_capacity * 2;
  // Allocate bigger array
  int *new_array = new int[new_array_capacity];

  // Copy contents
  for (unsigned int i = 0; i < array_length; i++)
    new_array[i] = array[i];
  // Deallocate old array
  delete[] array;

  // Use new array and update capacity
  array = new_array;
  array_capacity = new_array_capacity;
  assert(array_length < array_capacity);
}
```

Shrinking/Growing: Retrieve and Cleanup

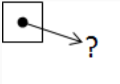
Shrinking/Growing: Retrieve and Cleanup

```
unsigned int length(){
  assert(isInitialized());
  return array_length;
}

int retrieve(unsigned int i){
  assert(isInitialized());
  assert(i < length());
  return array[i];
}

void cleanup(){
  assert(isInitialized());
  // Deallocate memory resource
  delete [] array;
  // Establish postconditions
  array = nullptr;
  array_capacity = 0;
  array_length = 0;
  assert(! isInitialized());
}
```

Dynamically allocated 2d arrays

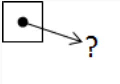
Dynamically allocated 2d arrays

```
// allocate the 2D array
int** pparray;
```

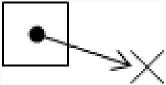


Dynamically allocated 2d arrays

```
// allocate the 2D array
int** pparray;
```

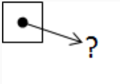


```
pparray = nullptr;
```

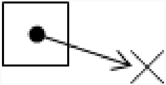


Dynamically allocated 2d arrays

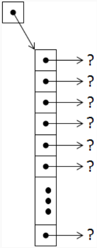
```
// allocate the 2D array
int** pparray;
```



```
pparray = nullptr;
```



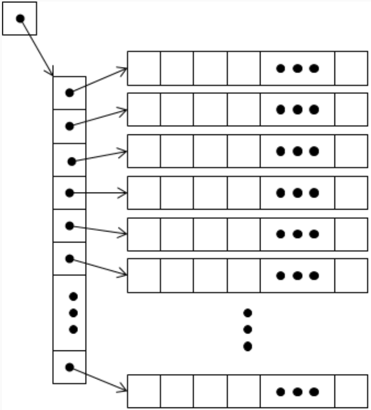
```
pparray = new int*[10];
```



Allocating the Second Dimension

Allocating the Second Dimension

```
// allocate the 2D array
int** pparray = nullptr;
pparray = new int*[10];

for (unsigned int i = 0; i < 10; i++){
  pparray[i] = new int[20];
 }
```



Using Dynamic 2D arrays

Using Dynamic 2D arrays

```
// store 7 in position 6 of row 2
pparray[2][6] = 7;
```

- How about using pointers?

Using Dynamic 2D arrays

```
// store 7 in position 6 of row 2
pparray[2][6] = 7;
```

- How about using pointers?

Using Dynamic 2D arrays

```
// store 7 in position 6 of row 2
pparray[2][6] = 7;
```

- How about using pointers?

```
 *(*(parray + 2) + 6) = 7;
// when done:
// deallocate in reverse order
for (unsigned int i = 0; i < 10; i++)
  delete [] pparray[i];
delete [] pparray;
```