CS100
Introduction to Programming

Lecture 19. Memory management

Today’s learning objectives

- Learning about scopes and the different types of memory
- Learning about the problems resulting from lots of freedom to manipulate memory
  - Memory leaks
  - Segmentation faults
- Dynamic sizing

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Constructors

- Method that is called when an instance is created

```
class Integer {
public:
  int m_val;
  Integer() {
    m_val = 0; printf("default constructor\n");
  }
};


int main() {
  Integer i;
}
```

Output:
default constructor

Constructors

- When making an array of objects, default constructor is invoked on each

```
class Integer {
public:
  int m_val;
  Integer() {
    m_val = 0; printf("default constructor\n");
  }
};


int main() {
  Integer arr[3];
}
```

Output:
default constructor
default constructor
default constructor

Constructors

- When making a class instance, the default constructor of its fields are invoked

```
class Integer {
public:
  int m_val;
  Integer() {
    m_val = 0; printf("Integer default constructor\n");
  }
};
class IntegerWrapper {
public:
  Integer m_val;
  IntegerWrapper() {
    printf("IntegerWrapper default constructor\n");
  }
};

int main() {
  IntegerWrapper q;
}
```

Output:
Integer default constructor
IntegerWrapper default constructor

Constructors

- Constructors can accept parameters

```
class Integer {
public:
  int m_val;
  Integer(int v) {
    m_val = v; printf("constructor with arg %d\n", v);
  }
};

int main() {
  Integer i(3);
}
```

Output:
constructor with arg 3

Constructors

- Constructors can accept parameters
  - Can invoke single-parameter constructor via assignment to the appropriate type

```
class Integer {
public:
  int m_val;
  Integer( int v ) {
    m_val = v; printf("constructor with arg %d\n");
  }
};

int main() {
  Integer i(3);
  Integer j = 5;
}
```

Output:
constructor with arg 3
constructor with arg 5

Constructors

- If a constructor with parameters is defined, the default constructor is no longer available

```
class Integer {
public:
  int m_val;
  Integer(int v) {
    m_val = v; printf("constructor with arg %d\n");
  }
};

int main() {
  Integer i(3); // ok
  Integer j;
}
```

Error: No default constructor available for Integer

Constructors

- If a constructor with parameters is defined, the default constructor is no longer available
  - Without a default constructor, can't declare arrays without initializing

```
class Integer {
public:
  int m_val;
  Integer(int v) {
    m_val = v; printf("constructor with arg %d\n");
  }
};


int main() {
  Integer i(3); // ok
  Integer b[2];
}
```

Error: No default constructor available for Integer

Constructors

- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor

```
class Integer {
public:
  int m_val;
  Integer() {
    m_val = 0;
  }
  Integer(int v) {
    m_val = v;
  }
};

int main() {
  Integer i;    // ok
  Integer j(3); // ok
}
```

Constructors

- If a constructor with parameters is defined, the default constructor is no longer available
  - Can create a separate 0-argument constructor
  - Or, use default arguments

```
class Integer {
public:
  int m_val;
  Integer(int v = 0) {
    m_val = v;
  }
};

int main() {
  Integer i;    // ok
  Integer j(3); // ok
}
```

Constructors

- How do I refer to a field when a method argument has the same name?
- **this**: a pointer to the current instance

```
class Integer {
public:
  int val;
  Integer(int val = 0) {
    this->val = val;
  }
};
```

this->val is a shorthand for (*this).val

Constructors

- How do I refer to a field when a method argument has the same name?
- **this**: a pointer to the current instance

```
class Integer {
public:
  int val;
  Integer(int val = 0) {
    this->val = val;
  }
  void setVal(int val) {
    this->val = val;
  }
};
```

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Scoping and Memory

- Whenever we declare a new variable (**`int x`**), memory is allocated
- When can this memory be freed up (so it can be used to store other variables)?
  - When the variable goes out of scope

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  if (true) {
    int x = 5;
  }
  // x now out of scope, memory it used to occupy can be reused
}
```

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable’s value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```

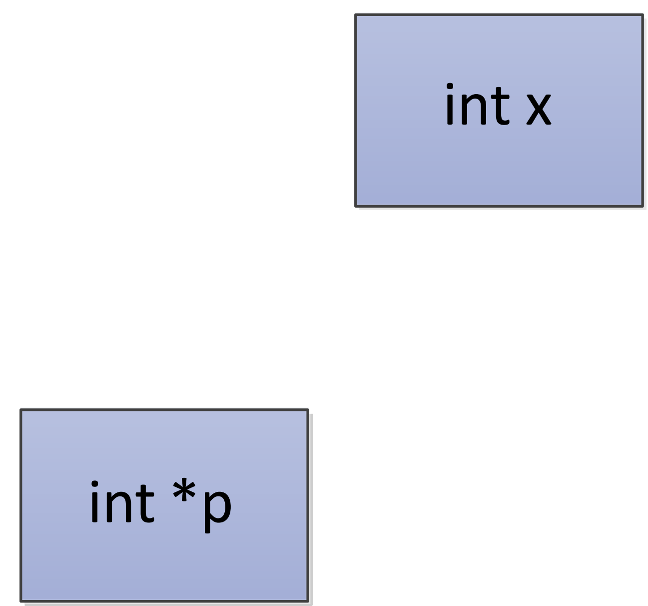
here

int *p

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable’s value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```

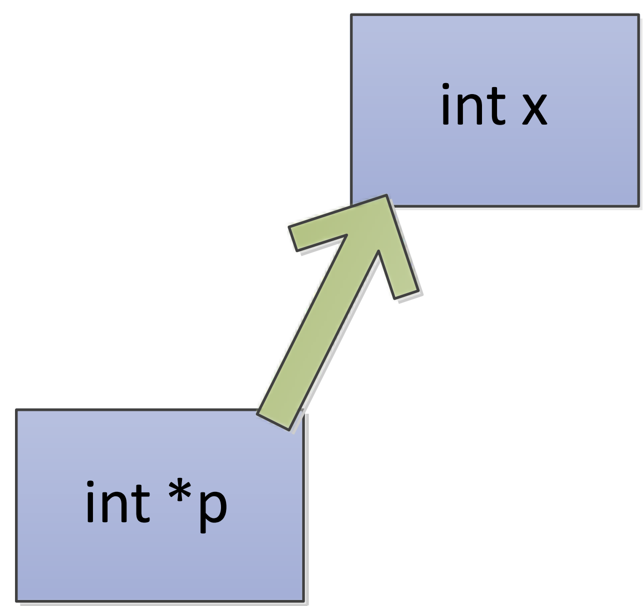


Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```

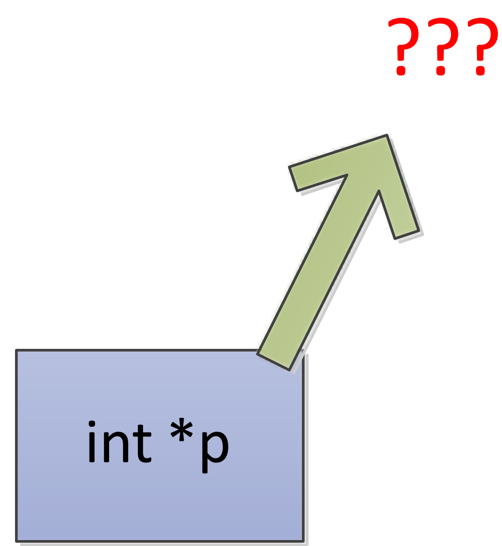
here



Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value
  - Here, p has become a **dangling pointer** (points to memory whose contents are undefined)

```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```



A Problematic Task

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:

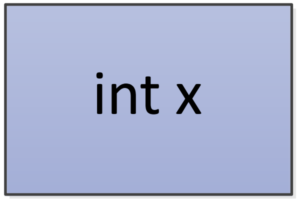
```
int* getPtrToFive() {
  int x = 5;
  return &x;
}
```

A Problematic Task

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope

```
int* getPtrToFive() {
  int x = 5;
  return &x;
}
int main() {
  int *p = getPtrToFive();
  printf("%d\n", *p); // ???
}
```

here

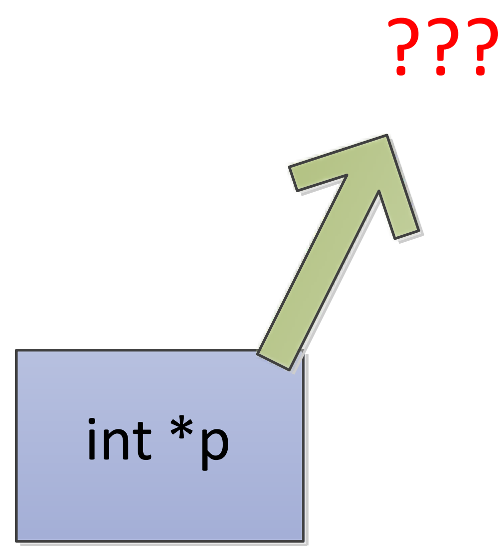


A Problematic Task

- Implement a function which returns a pointer to some memory containing the integer 5
- Incorrect implementation:
  - x is declared in the function scope
  - As getPtrToFive() returns, x goes out of scope. So a dangling pointer is returned

```c
int* getPtrToFive() {
  int x = 5;
  return &x;
}
int main() {
  int *p = getPtrToFive();
  printf("%d\n", *p); // ???
}
```

here

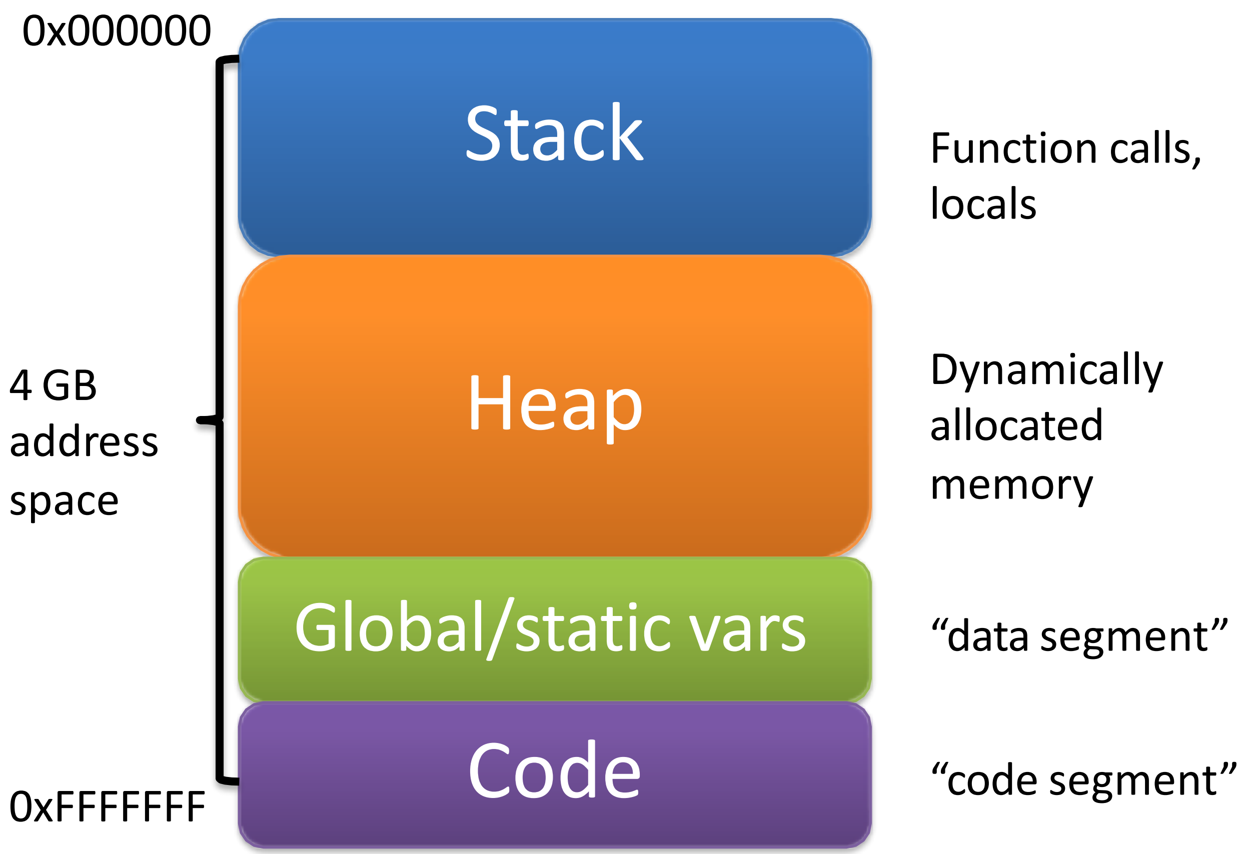


Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Memory Types

- each process gets its own memory chunk, or ***address space***



Stack Allocation

- memory allocated by the program as it runs
  - local variables
  - function calls

- fixed at compile time

Heap Allocation

- dynamic memory allocation
  - memory allocated at run-time

Heap

- Function for allocating memory:
  - **malloc()**
    - Requires **#include <stdlib.h>** to work

malloc()

```
void* malloc ( <size to be allocated> )
```

```
char *letters;
letters = (char*) malloc(userVariable *
                                        sizeof(char));
```

- malloc returns a pointer to a ***contiguous*** block memory of the size requested

Casting Allocated Memory

- **`malloc()`** return a pointer of type `void`, so you must cast the memory to match the given type

```
letters = (char*) malloc(userVariable *
                                    sizeof(char));
```

Handling Allocated Memory

- **<u>IMPORTANT</u>**: before using allocated memory make sure it's <u>*actually been allocated*</u>
- if memory wasn't correctly allocated, the address that is returned will be **`null`**
  - this means there isn't a contiguous block of memory large enough to handle request

Exiting in Case of NULL

- if the address returned is **`null`**, your program should exit
  - **`exit()`** takes an integer value
  - non-zero values are used as error codes

```
if (grades == NULL) {
  printf("Memory not allocated,
              exiting.\n");
  exit(-1);
}
```

Managing Your Memory

- ***stack*** allocated memory is automatically freed when functions **`return`**
  - including **`main()`**

Stack

- memory on the ***heap*** was allocated by you – so it must also be freed by you

Heap

Freeing Memory

- done using the **`free()`** function
  - free takes a pointer as an argument: **`free(grades);`** **`free(letters);`**

- **`free()`** does not work recursively
  - for each individual allocation, there must be an individual call to free that allocated memory
  - called in a sensible order

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Back to C++: The new operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

The new operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory

```
int *x = new int;
```

Type parameter needed to determine how much memory to allocate

The new operator

- Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory:
  - If using **int x;** the allocation occurs on **the stack**
  - If using **new int;** the allocation occurs on **the heap**

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

The delete operator

- De-allocates memory that was previously allocated using **new**
- Takes a pointer to the memory location

```
int *x = new int;
// use memory allocated by new
delete x;
```

The delete operator

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate using **new** to ensure it remains allocated

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}
```

The delete operator

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate using **new** to ensure it remains allocated
  - When done, de-allocate the memory using **delete**

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}
int main() {
  int *p = getPtrToFive();
  printf("%d\n", *p); // 5
  delete p;
}
```

Delete Memory When Done Using It

- If you don’t use de-allocate memory using **delete**, your application will waste memory

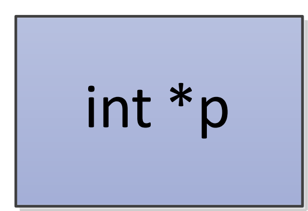
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```

incorrect

Delete Memory When Done Using It

- If you don't use de-allocate memory using **delete**, your application will waste memory

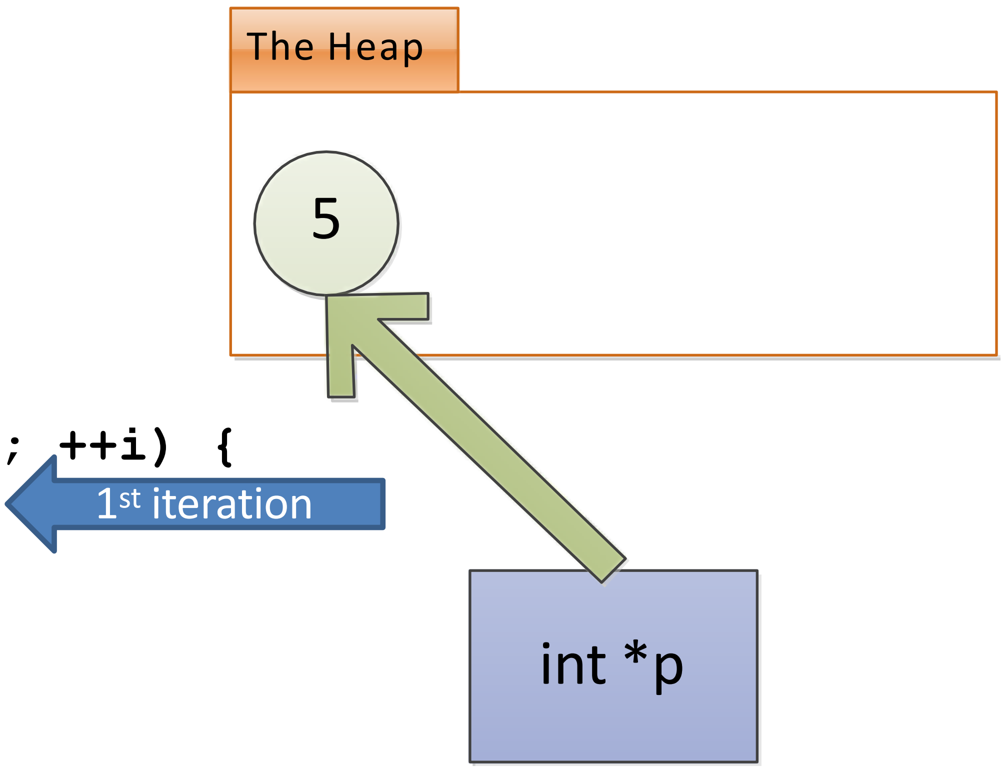
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```



Delete Memory When Done Using It

- If you don't use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```

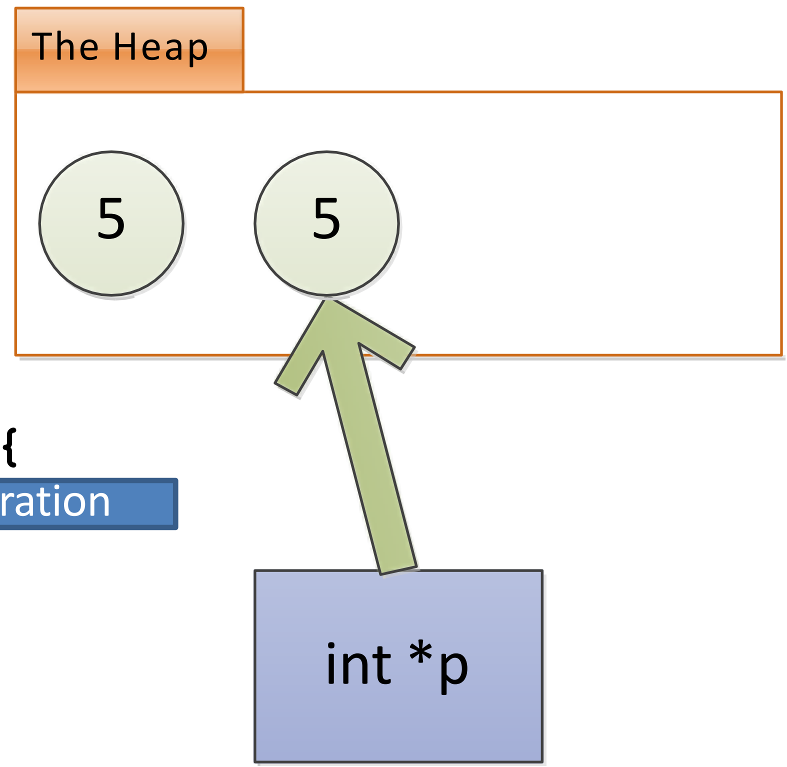


Delete Memory When Done Using It

- If you don’t use de-allocate memory using **delete**, your application will waste memory

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```

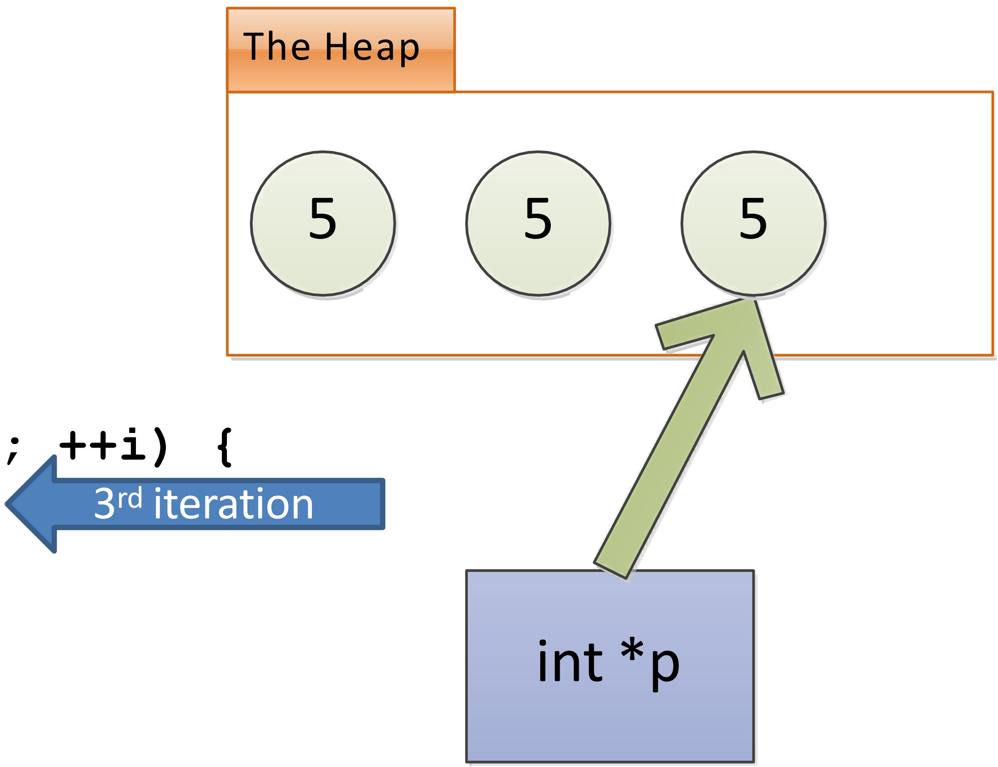
2nd iteration



Delete Memory When Done Using It

- When your program allocates memory but is unable to de-allocate it, this is a **memory leak**

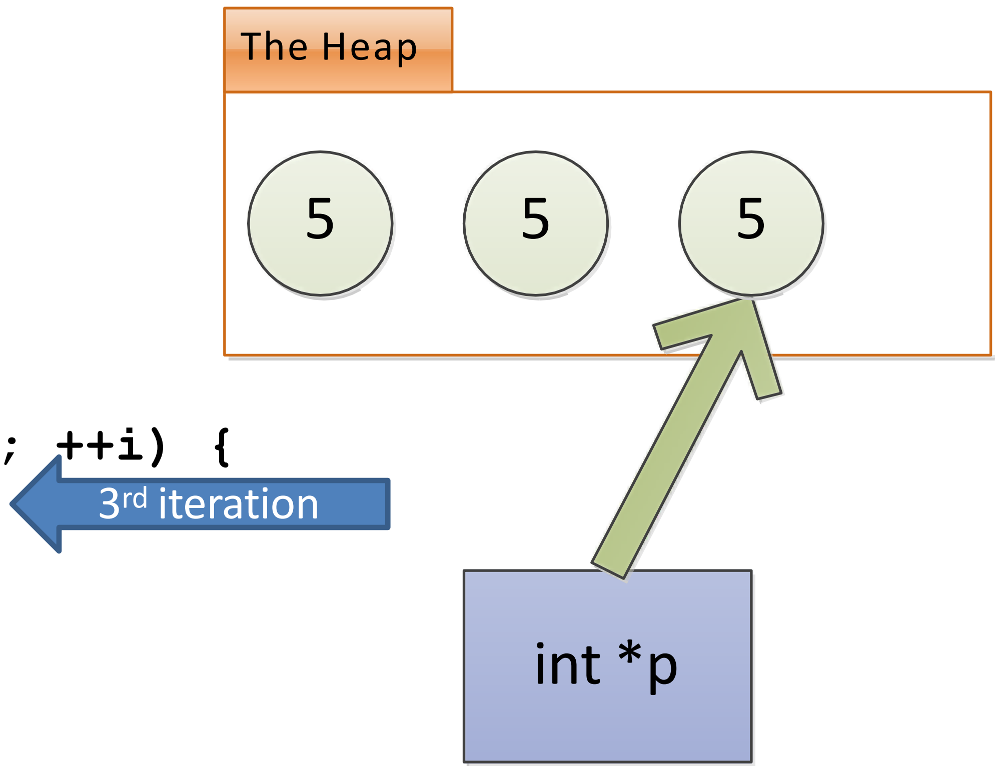
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```



Delete Memory When Done Using It

- Does adding delete after loop fix memory leak?

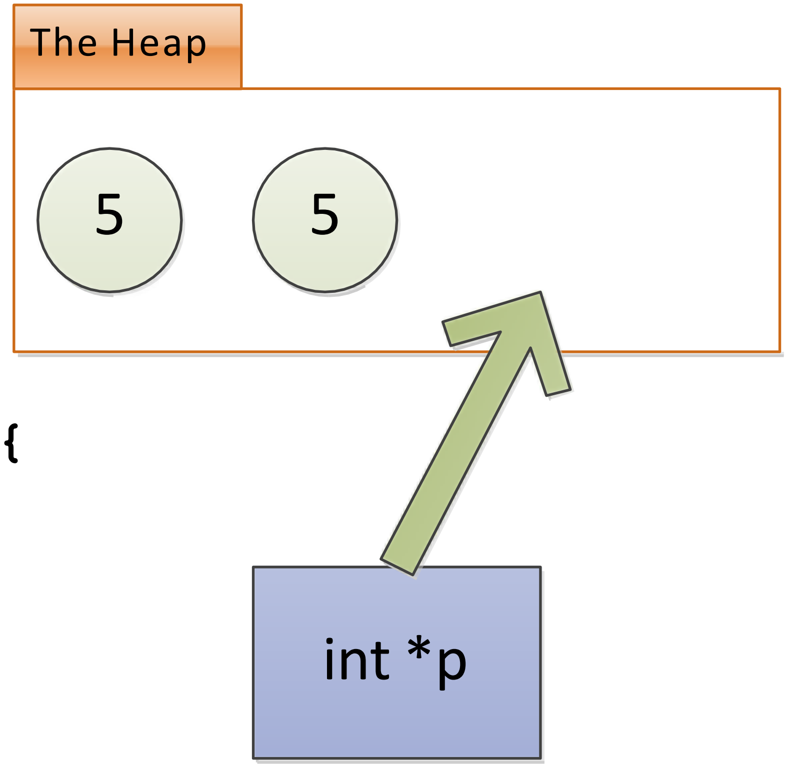
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
  delete p;
}
```



Delete Memory When Done Using It

- Does adding delete after loop fix memory leak?
  - –Only memory allocated on last iteration is de-allocated

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
  delete p;
}
```



Delete Memory When Done Using It

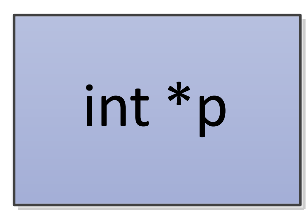
- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```

Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

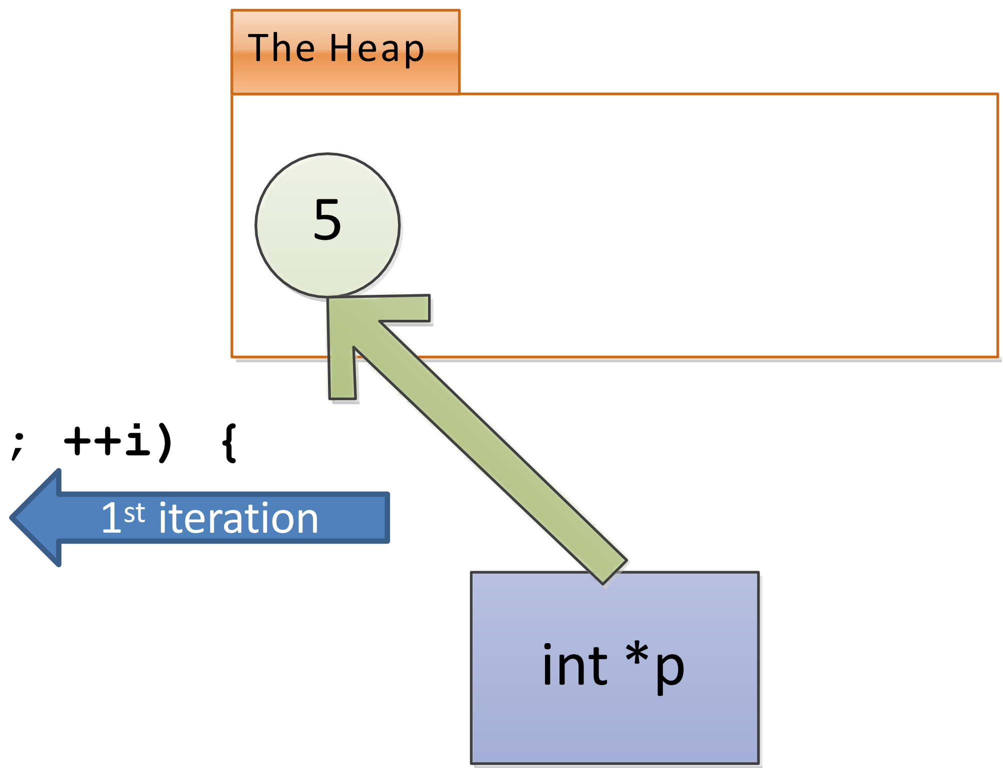
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```



Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

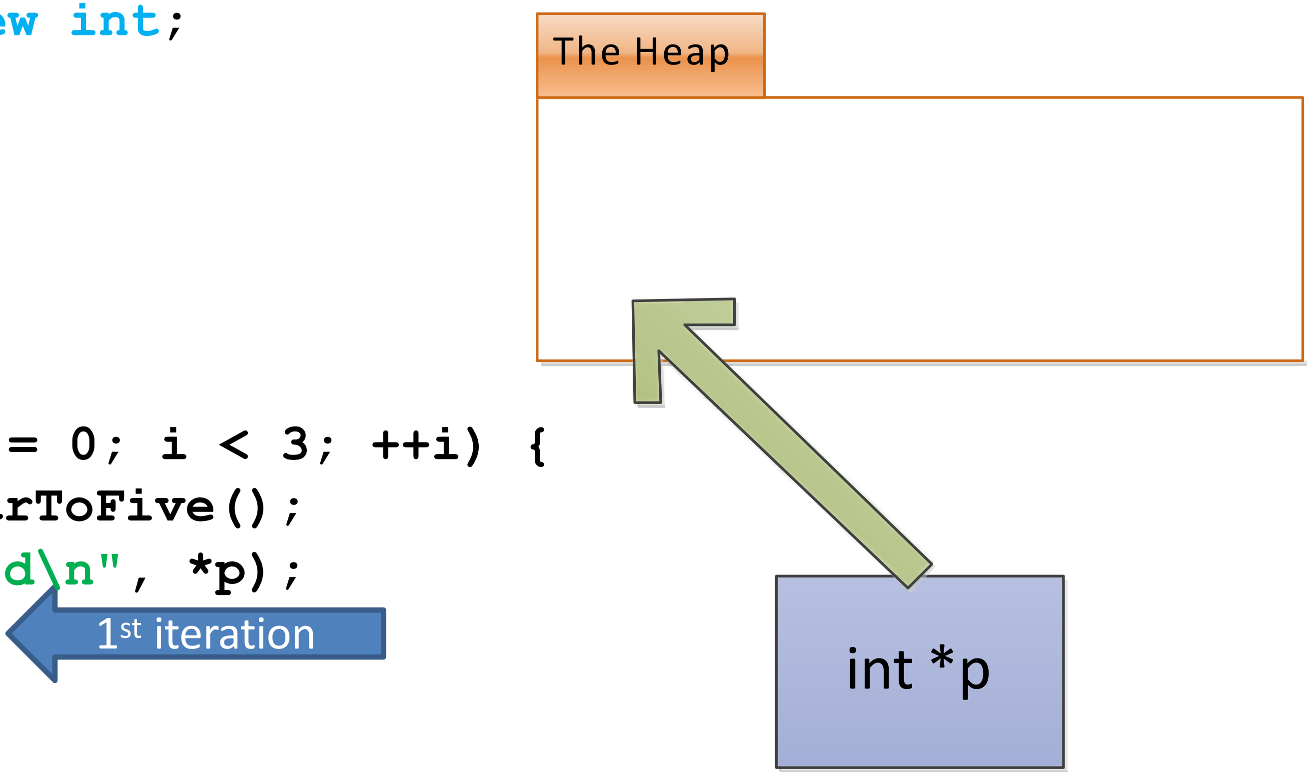
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```



Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

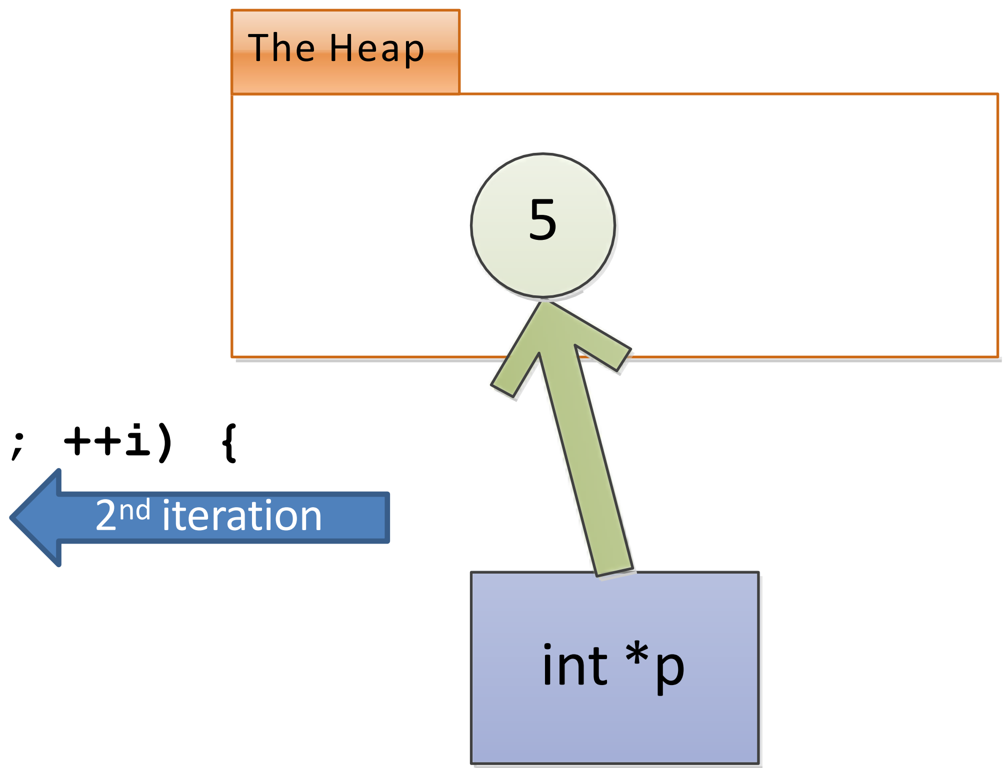
```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```



Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```

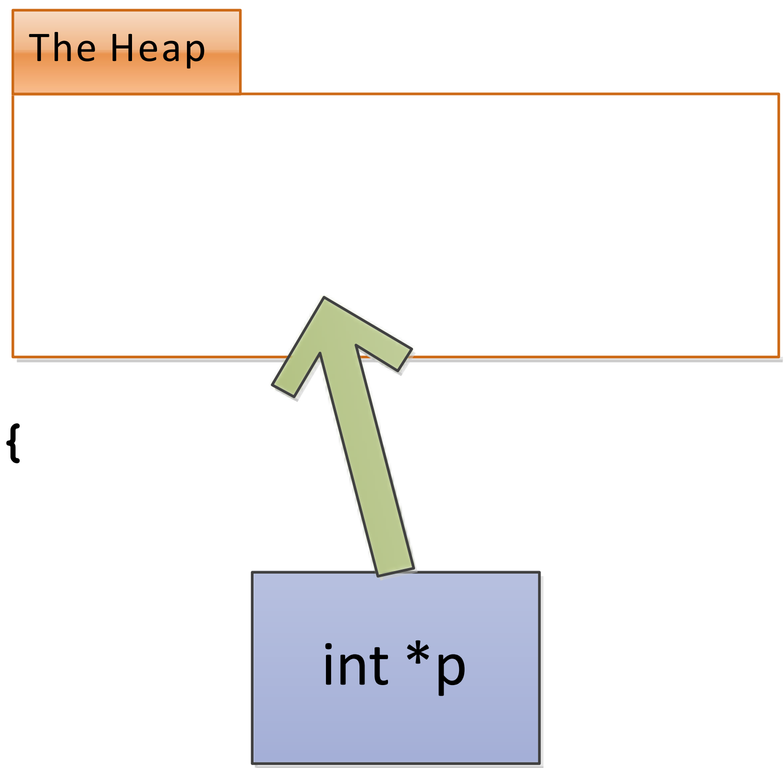


Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```

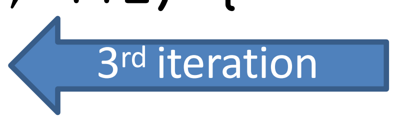
2^{nd} iteration

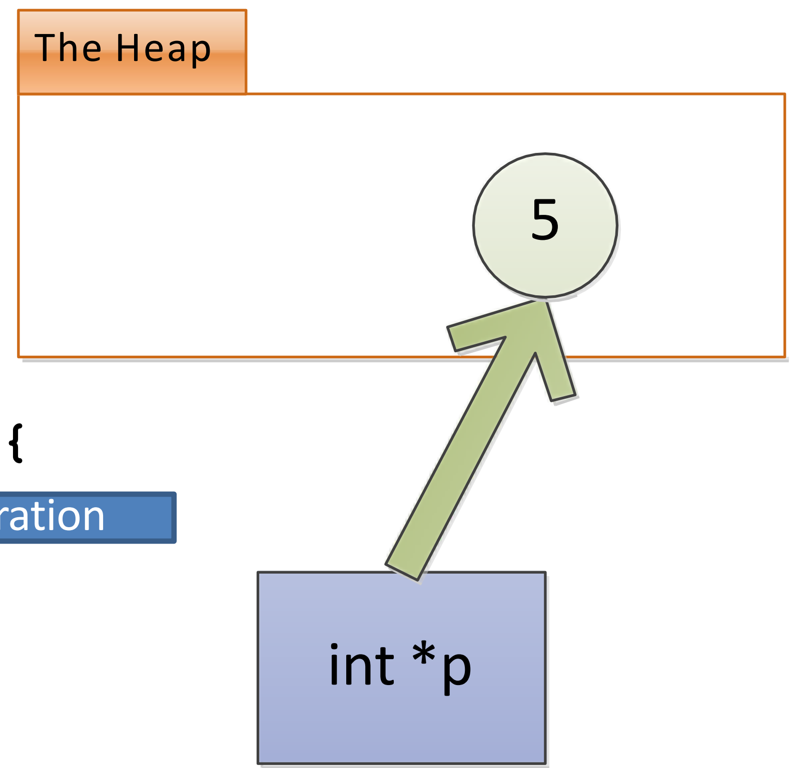


Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```



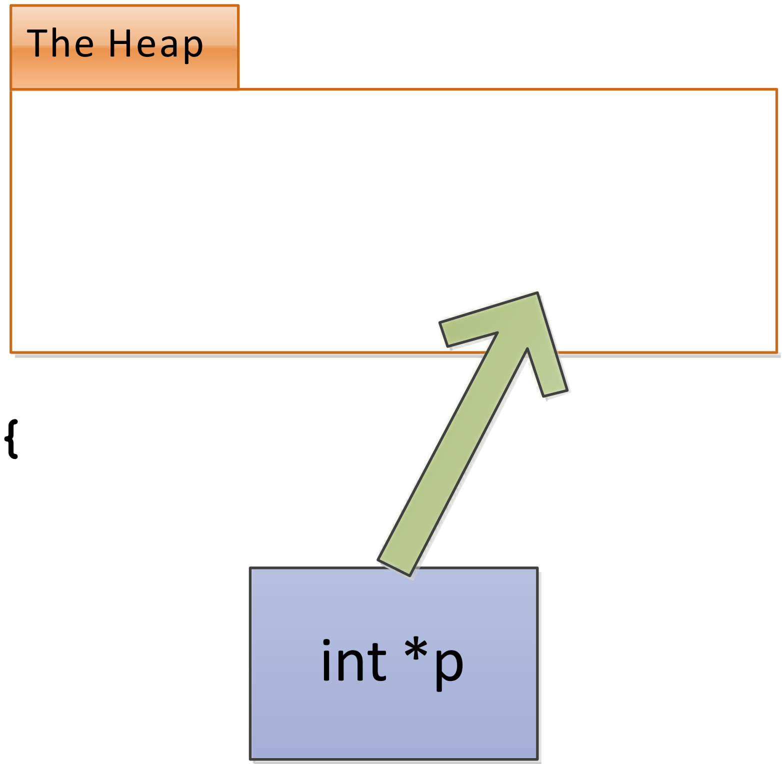


Delete Memory When Done Using It

- To fix the memory leak, de-allocate memory within the loop

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
    delete p;
  }
}
```

3rd iteration



Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Don’t Use Memory After Deletion

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  delete x;
  printf("%d\n", *x); // ???
}
```

Don’t Use Memory After Deletion

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  delete x;
  printf("%d\n", *x); // ???
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
}
```

Don’t delete memory twice

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
  delete x;
}
```

Don’t delete memory twice

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
  delete x;
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
}
```

Only delete if memory was allocated by new

incorrect

```
int main() {
   int x = 5;
   int *xPtr = &x;
   printf("%d\n", *xPtr);
   delete xPtr;
}
```

Only delete if memory was allocated by new

incorrect

```
int main() {
  int x = 5;
  int *xPtr = &x;
  printf("%d\n",    *xPtr);
  delete xPtr;
}
```

correct

```
int main() {
  int x = 5;
  int *xPtr = &x;
  printf("%d\n", *xPtr);
}
```

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Allocating Arrays

- When allocating arrays on the stack (using "int arr[SIZE]"), size must be a constant
  (Note: C99 standard may allow this, to have C++ standard imposed strictly, use compiler option **-pedantic**!)

```
int numItems;
printf("how many items?\n");
scanf("%d", &numItems);
int arr[numItems]; // not allowed
```

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

```
int numItems;
printf("how many items?\n");
scanf("%d", &numItems);
int *arr = new int[numItems];
```

Type of items in array

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size

```
int numItems;
printf("how many items?\n");
scanf("%d", &numItems);
int *arr = new int[numItems];
```

Number of items to allocate

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size
- De-allocate arrays with **delete[]**

```
int numItems;
printf("how many items?\n");
scanf("%d", &numItems);
int *arr = new int[numItems];
delete[] arr;
```

Ex: Storing values input by the user

```
int main() {
  int numItems;
  printf("how many items?\n");
  scanf("%d", &numItems);
  int *arr = new int[numItems];
  for (int i = 0; i < numItems;  ++i) {
    printf("enter item %d: ", i);
    scanf("%d", &arr[i]);
  }
  for (int i = 0; i < numItems; ++i) {
    printf("%d\n", arr[i]);
  }
  delete[] arr;
}
```

```
how many items? 3
enter item 0: 7
enter item 1: 4
enter item 2: 9
7
4
9
```

Outline

- Constructors
- Scoping and Memory
- Memory Types
- Back to C++: The **new** operator
- Memory leaks: The **delete** operator
- Segmentation faults
- Dynamically sized arrays
- New & delete with classes

Allocating Class Instances using new

- **new** can also be used to allocate a class instance

```
class Point {
public:
  int m_x, m_y;
};

int main() {
  Point *p = new Point;
  delete p;
}
```

Allocating Class Instances using new

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {
public:
  int m_x, m_y;
  Point() {
    m_x = 0; m_y = 0; printf("default constructor\n");
  }
};

int main() {
  Point *p = new Point;
  delete p;
}
```

Output:
default constructor

Allocating Class Instances using new

- **new** can also be used to allocate a class instance
- The appropriate constructor will be invoked

```
class Point {
public:
  int m_x, m_y;
  Point( int nx,  int ny) {
     m_x=ny; m_y= ny; printf("2-arg constructor\n");
  }
};

int main() {
  Point *p = new Point(2, 4);
  delete p;
}
```

Output:
2-arg constructor

Destructor

- Destructor is called when the class instance gets de-allocated

```
class Point {
public:
  int m_x, m_y;
  Point() {
    printf("constructor invoked\n");
  }
  ~Point() {
    printf("destructor invoked\n");
  }
}
```

- Destructor is called when the class instance gets de-allocated
  - If allocated with **new**, when **delete** is called

```
class Point {
public:
  int m_x, m_y;
  Point() {
    printf("constructor invoked\n");
  }
  ~Point() {
    printf("destructor invoked\n");
  }
};
int main() {
  Point *p = new Point;
  delete p;
}
```

Output:
constructor invoked
destructor invoked

- Destructor is called when the class instance gets de-allocated
  - If allocated with **new**, when **delete** is called
  - If stack-allocated, when it goes out of scope

```
class Point {
public:
  int m_x, m_y;
  Point() {
    printf("constructor invoked\n");
  }
  ~Point() {
    printf("destructor invoked\n");
  }
};
int main() {
  if (true) {
    Point p;
  }
  printf("p out of scope\n");
}
```

Output:

constructor invoked
destructor invoked
p out of scope

Example: Representing an Array of Integers

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let's make them fields in a class

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
};
```

Pointer to the first element

- When representing an array, often pass around both the pointer to the first element and the number of elements
  - Let’s make them fields in a class

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
};
```

Number of elements in the array

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
};

int main() {
  IntegerArray arr;
  arr.m_size = 2;
  arr.m_data = new int[arr.m_size];
  arr.m_data[0] = 4; arr.m_data[1] = 5;
  delete[] a.m_data;
}
```

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
};

int main() {
  IntegerArray arr;
  arr.m_size = 2;
  arr.m_data = new int[arr.m_size];
  arr.m_data[0] = 4; arr.m_data[1] = 5;
  delete[] a.m_data;
}
```

Can move this into a constructor

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
};

int main() {
  IntegerArray arr(2);
  arr.m_data[0] = 4; arr.m_data[1] = 5;
  delete[] arr.m_data;
}
```

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
};

int main() {
  IntegerArray arr(2);
  arr.m_data[0] = 4; arr.m_data[1] = 5;
  delete[] arr.m_data;
}
```

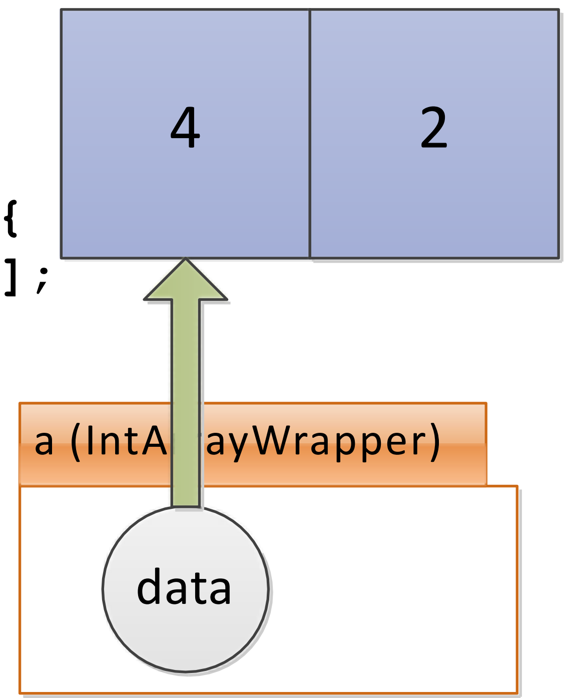
Can move this into a destructor

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray () {
    delete[] m_data
  }
};

int main() {
  IntegerArray arr(2);
  arr.m_data[0] = 4; arr.m_data[1] = 5;
}
```

De-allocate memory used by fields in destructor incorrect

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};

int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // not 4!
}
```

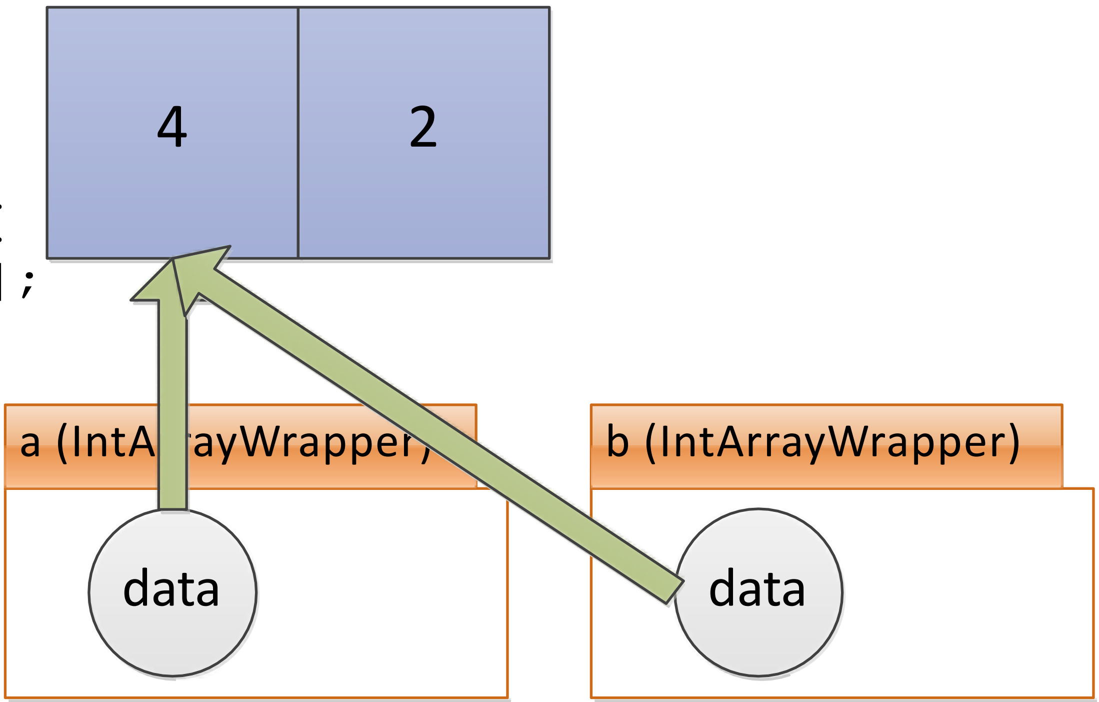
```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};

int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // not 4!
}
```



here

- Default copy constructor copies fields

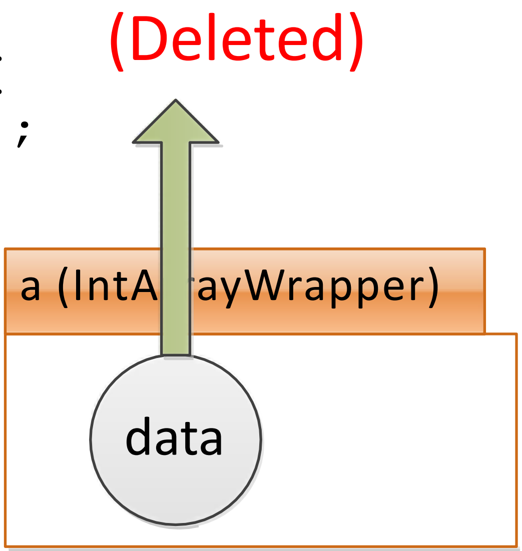
```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};

int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // not 4!
}
```



This call uses the default copy constructor, which simply copies all fields of the object

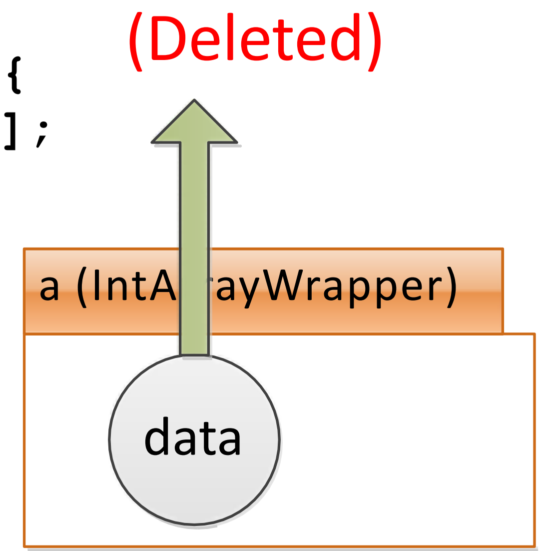
- When b goes out of scope, destructor is called (de-allocates array), a.data now a dangling pointer

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};

int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // not 4!
}
```



- 2nd bug: when a goes out of scope, its destructor tries to delete the (already-deleted) array

```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};

int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // not 4!
}
```

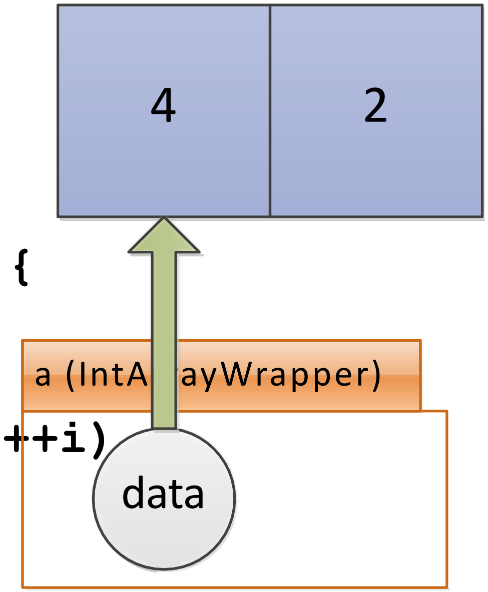


Program crashes as it terminates

- Write your own copy constructor to fix these bugs

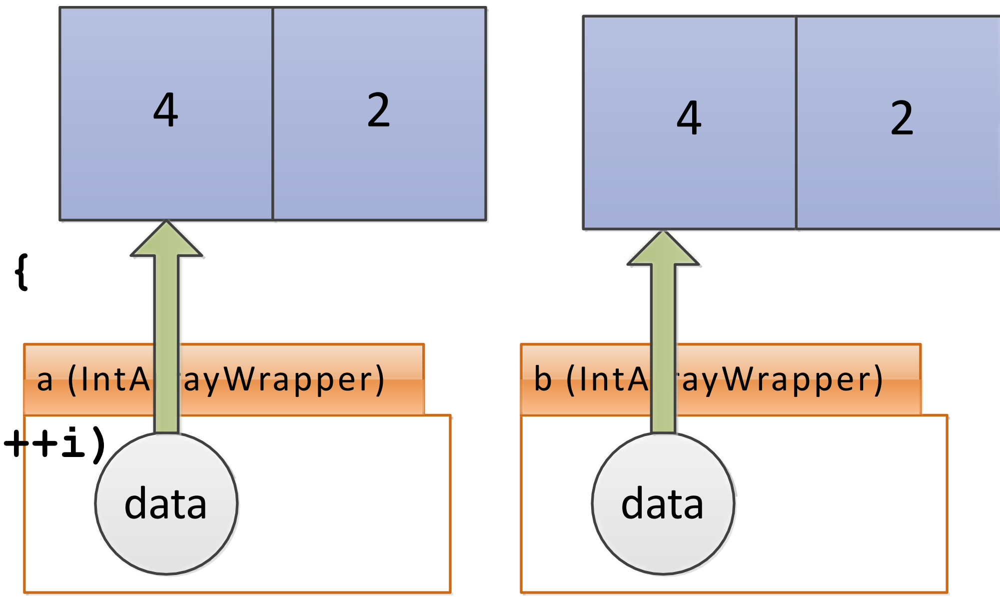
```
class IntegerArray {
public:
  int *m_data;
  int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  IntegerArray(IntegerArray &o) {
    m_data = new int[o.m_size];
    m_size = o.m_size;
    for (int i = 0; i < m_size; ++i)
      m_data[i] = o.m_data[i];
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};
```

```
class IntegerArray {
public:
  int *m_data; int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  IntegerArray(IntegerArray &o) {
    m_data = new int[o.m_size];
    m_size = o.m_size;
    for (int i = 0; i < m_size; ++i)
      m_data[i] = o.m_data[i];
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};
int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // 4
}
```



here

```
class IntegerArray {
public:
  int *m_data; int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  IntegerArray(IntegerArray &o) {
    m_data = new int[o.m_size];
    m_size = o.m_size;
    for (int i = 0; i < m_size; ++i)
      m_data[i] = o.m_data[i];
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};
int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // 4
}
```



```
class IntegerArray {
public:
  int *m_data; int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  IntegerArray(IntegerArray &o) {
    m_data = new int[o.m_size];
    m_size = o.m_size;
    for (int i = 0; i < m_size; ++i)
      m_data[i] = o.m_data[i];
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};
int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // 4
}
```