CS 142
Dynamic Memory

*Adapted from OpenCourseWare Lecture Notes by Geza Kovacs

Announcements

- Program 7 has been assigned - due Sunday, April 19th by 11:55pm

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;
  }
  cout << *p << endl; // ???
}
```


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;
  }
  cout << *p << endl; // ???
}
```

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;
  }
  cout << *p << endl; // ???
}
```

here

int x

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;
  }
  cout << *p << endl; // ???
}
```

here

int x

int *p

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;
  }
  cout << *p << endl; // ???
}
```

here

???

int *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;
}
```


- 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();
  cout << *p << endl; // ???
}
```

here

int x

- 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

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

int main() {
  int *p = getPtrToFive();
  cout << *p << endl; // ???
}
```

here

???

int *p

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
- Terminology note:
  - If using **int x;** the allocation occurs on a region of memory called **the stack**
  - If using **new int;** the allocation occurs on a region of memory called **the heap**

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;
```


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

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

- Implement a function which returns a pointer to some memory containing the integer 5
  - Allocate memory 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();
  cout << *p << endl; // 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();
    cout << *p << endl;
  }
}
```

incorrect

- 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();
    cout << *p << endl;
  }
}
```

int *p

- 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();
    cout << *p << endl;
  }
}
```

1st iteration

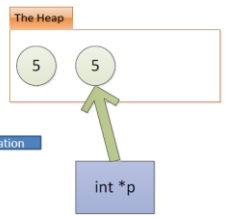
The Heap

5

int *p

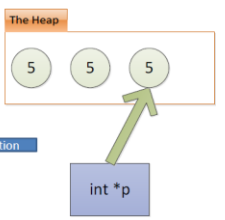
- If you don't use de-allocate memory using **delete**, your application will waste memory

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

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```



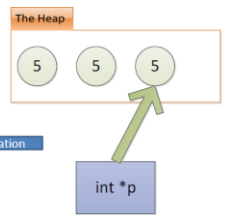
- If you don't use de-allocate memory using **delete**, your application will waste memory
- When your program allocates memory but is unable to de-allocate it, this is a **memory leak**

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

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
}
```



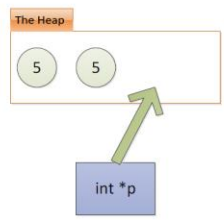
- Does adding a delete after the loop fix this memory leak?

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

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    cout << *p << endl;
  }
  delete p;
}
```



- Does adding a delete after the loop fix this memory leak?
  - No; only the memory that was allocated on the last iteration gets 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();
    cout << *p << endl;
  }
  delete p;
}
```



- 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();
    cout << *p << endl;
    delete p;
  }
}
```


- 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();
    cout << *p << endl;
    delete p;
  }
}
```

int *p

- 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();
    cout << *p << endl;
    delete p;
  }
}
```

1st iteration

The Heap

5

int *p

- 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();
    cout << *p << endl;
    delete p;
  }
}
```

1st iteration

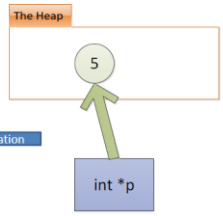
The Heap

int *p

- 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();
    cout << *p << endl;
    delete p;
  }
}
```

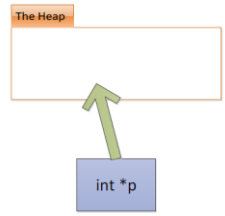
2nd iteration



- 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();
    cout << *p << endl;
    delete p;
  }
}
```

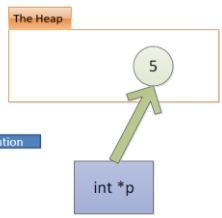
2nd iteration



- 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();
    cout << *p << endl;
    delete p;
  }
}
```

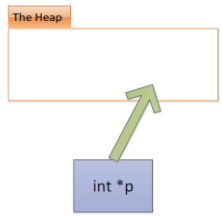
3rd iteration



- 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();
    cout << *p << endl;
    delete p;
  }
}
```

3rd iteration



Don't Use Memory After Deletion

incorrect

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

int main() {
  int *x = getPtrToFive();
  delete x;
  cout << *x << endl; // ???
}
```

correct

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

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
}
```


Don't Delete Memory Twice

incorrect

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

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
  delete x;
}
```

correct

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

int main() {
  int *x = getPtrToFive();
  cout << *x << endl; // 5
  delete x;
}
```


Only `delete` if memory was allocated by new

incorrect

```
int main() {
  int x = 5;
  int *xPtr = &x;
  cout << *xPtr << endl;
  delete xPtr;
}
```

correct

```
int main() {
  int x = 5;
  int *xPtr = &x;
  cout << *xPtr << endl;
}
```


Try This

- Allocate two new ints on the heap (dynamically).
- Set them equal to 10 and 20 and print them.
- Switch the pointers so each pointer now points to the opposite int.
- Print them again.
- Deallocate the integers.

Allocating Arrays

- When allocating arrays on the stack (using "int arr[SIZE]"), SIZE must be a constant

```
int numItems;
cout << "how many items?";
cin >> numItems;
int arr[numItems]; // not allowed
```


Allocating Arrays

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

```
int numItems;
cout << "how many items?";
cin >> 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;
cout << "how many items?";
cin >> numItems;
int *arr = new int[numItems];
delete[] arr;
```


Ex: Storing values input by user

```
int main() {
  int numItems;
  cout << "how many items? ";
  cin >> numItems;
  int *arr = new int[numItems];
  for (int i = 0; i < numItems; ++i) {
    cout << "enter item " << i << ": ";
    cin >> arr[i];
  }
  for (int i = 0; i < numItems; ++i) {
    cout << arr[i] << endl;
  }
  delete[] arr;
}
```

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

Variables that grow and/or shrink

- Using new type[num] still doesn't make the dynamic memory grow or shrink.
- So how do vectors work?
  - A vector starts off by allocating (using new) a "default" amount of space for items in the vector.
  - If we add too many things to a vector, it will allocate more space, copy everything in the vector into the new space, then delete[] the old space.

Try This

- Allocate (on the heap) an array of 5 doubles.
- Assign some numbers to the array.
- [Pretend that we want to add more numbers.]
- Allocate (on the heap) a second array of 10 doubles.
- Copy the doubles from the old array into the new one.
- delete[] the old array.
- Print the new array.
- delete[] the new array.