Copying Objects

• Last time, introduced a Complex class
  
  ```cpp
class Complex {
    double re, im;
  public:
    Complex(double re, double im);  
    ...  
  };
``` 

• What if we want to make a copy of a specific object? i.e. initialize a Complex from another Complex
  
  ```cpp
  Complex c1{5, 2};
  Complex c2{c1};  // Makes a copy!
  ``` 

• C++ automatically generates a **copy constructor** for every class
Copying Objects (2)

• The copy constructor is used when objects are passed by-value
  
  double magnitude(Complex c);
  
  • c is passed by value
  
  • A copy of c is made, and magnitude operates on the copy
  
  • The copy constructor is used
  
  • (This is why we want to pass objects by const-reference; to avoid the overhead of unnecessary copying)

• The default copy constructor generated by C++ simply copies the values of all members into the new object
  
  • Sometimes this causes problems...

• To write our own version of the copy constructor, implement this constructor:
  
  Complex(const Complex &c); // Must pass by reference
Assigning Objects

• Similarly, can use assignment on objects without any extra code
  ```cpp
  Complex c1{5, 2};
  Complex c2;
  ...
  c2 = c1;
  ```

• This is called the copy-assignment operator

• The default copy-assignment operator generated by C++ simply copies the values of all members from the RHS into the LHS

• To write our own version of the copy-assignment operator, implement this member operator-overload function:
  ```cpp
  Complex & Complex::operator=(const Complex &c);
  ```
  • Must return a non-const reference to the LHS of the assignment, in order to support operator-chaining, e.g. `c3 = c2 = c1;`
Allocating an Object on the Heap

• When you need a large chunk of memory, or you need to create objects that live beyond the lifetime of a specific function call, you can allocate memory from the heap

  Complex *p = new Complex{3, 5};
  • p points to a Complex object allocated on the heap

• To access members of the object pointed to by p, must use -> operator

  cout << p.real() << "", " << p.imag();         // ERROR
  cout << p->real() << "", " << p->imag();       // OK!

• If your program allocates memory from the heap, your program must also take care to release it! Otherwise you will have a memory leak.

  delete p;
  • p will still contain an address; don’t use it after deleting the object!
Heap-Allocating Arrays of Objects

• Can also allocate arrays of objects on the heap
  ```cpp
  Complex *p = new Complex[1000];
  ```
  • p points to an array of 1000 Complex objects, allocated on the heap

• Each element is initialized with the class’ default constructor
  • Not possible to call a different constructor during array initialization
  • If your element type doesn’t have default initialization, not possible to use in array allocations

• Can access array elements as usual, e.g. `p[0].real()`
  • Each element is a Complex object, so use . instead -> for member access

• Freeing arrays is slightly more complicated:
  ```cpp
  delete[] p;
  ```
  • NOTE: Must use delete[] with new[], and delete with new! Do not mix!!!
  • The compiler will not stop you from mixing the two. The types do not indicate whether the allocation is an array or a single object.
Heap-Allocating Arrays of Primitives

• Can also allocate arrays of primitive values
  
  ```cpp
double *array = new double[numValues];
  ```

• Primitive types do not have constructors or destructors. The values are unitialized.
  • If there are random values in the memory area used for the allocation, the new array may contain garbage
  • *This doesn’t always happen, but it will eventually!*  

• **Always** initialize arrays of primitive values after allocating
  ```cpp
  for (int i = 0; i < numValues; i++)
    array[i] = 0;
  ```

• When finished, free with `delete[]` as usual
  ```cpp
  delete[] array;
  ```
Managing Heap-Allocated Memory

• Managing heap-allocated memory in C++ programs is difficult and bug-prone, particularly as program size grows

• Simple solution: Don’t heap-allocate memory at all! 😊
  • When possible, use std::vector<T>, std::array<T>, std::string, etc.

• When you must heap-allocate memory, use the C++ class lifecycle to make memory management easier

• When an object goes out of scope, its destructor is called automatically...

• Strategy:
  • Heap-allocate memory in class constructor (and in a very few other places)
  • Free memory in destructor
  • The object manages memory for you – abstraction / encapsulation

• Pattern is called Resource Acquisition Is Initialization (RAII)
Array of Floats

• A class to manage an array of floats:

```cpp
class FloatArray {
    int count;
    float *elems;

public:
    FloatArray(int n);
    ~FloatArray();
    ...;
};
```
Array of Floats (2)

- Constructor:
  ```cpp
  FloatArray::FloatArray(int n) {
    count = n;
    elems = new float[count];
    for (int i = 0; i < count; i++)
      elems[i] = 0;
  }
  ```

- Destructor:
  ```cpp
  FloatArray::~FloatArray() {
    delete[] elems;
  }
  ```
Array of Floats (3)

- FloatArray takes care of memory management, so we don’t have to!

```cpp
float getAverage() {
    int numFloats;
    cin >> numFloats;
    FloatArray f{numFloats};
    for (int i = 0; i < numFloats; i++) {
        float value;
        cin >> value;
        f.set(i, value);
    }
    return f.average();
}
```

- When f goes out of scope, its destructor is called automatically
- Heap memory allocated within f is freed automatically
Copying Arrays of Floats

• What does this code do?
  ```
  void f(int n) {
    FloatArray fa1{n};
    ... // populate fa1

    FloatArray fa2{fa1}; // Make a copy!
    ...
  }
  ```
• Recall:
  • The default copy constructor generated by C++ simply copies the values of all members into the new object
• Hmmm....
Copying Arrays of Floats (2)

• What does this code do?

```c
void f(int n) {
    FloatArray fa1{n};
    ... // populate fa1

    FloatArray fa2{fa1};  // Make a copy!
    ... 
}
```

• The default copy-constructor performs a **shallow copy**

• This code has several issues
  • Changes through fa1 will be visible through fa2, and vice versa
  • The code will likely crash with a double-free of the memory block
Custom Copy Constructors

• If your class dynamically allocates memory, you usually need to implement a custom copy-constructor that performs a deep copy
  • The object being initialized needs its own memory region!

• Updated code for FloatArray:
  ```cpp
  FloatArray::FloatArray(const FloatArray &f) {
    count = f.count;
    // Make a deep copy
    elems = new float[count];
    for (int i = 0; i < count; i++)
      elems[i] = f.elems[i];
  }
  
  • Note: Can directly access private members of f because we are still in the FloatArray code
    • Makes the implementation short and clean
Assigning Arrays of Floats

• What does this code do?
  ```
  void f(int n) {
    FloatArray fa1{n};
    ... // populate fa1
    
    FloatArray fa2{10};
    ... 
    fa2 = fa1;
  }
  ```

• This doesn’t invoke the copy-constructor, because it isn’t part of a variable-initialization statement

• Rather, it invokes the **copy-assignment operator**
  ```
  FloatArray & FloatArray::operator=(const FloatArray &f)
  ```
Assigning Arrays of Floats (2)

• What does this code do?
  ```
  void f(int n) {
    FloatArray fa1{n};
    ... // populate fa1
    FloatArray fa2{10};
    ...
    fa2 = fa1;
  }
  ```
• C++ also generates a default copy-assignment operator for you
• The default copy-assignment operator generated by C++ simply copies the values of all members from the RHS into the LHS
• *We have the same problems as before, but we also leak memory!*
Custom Copy-Assignment Operators

• Previous observation:
  • If your class dynamically allocates memory, you usually need to implement a custom copy-constructor that performs a deep copy
  • The object being initialized needs its own memory region

• Similarly:
  • If your class dynamically allocates memory, you usually need to implement a custom copy-assignment operator that cleans up any existing allocation, and also performs a deep copy
  • The object being assigned to may already hold some memory, which needs to be freed
  • The object being assigned to needs its own memory region
The Rule of Three

- **The Rule Of Three:** If your class defines any of the following:
  - A destructor
  - A copy-constructor
  - A copy-assignment operator
- It probably needs to define **all three**.
- (There is also a **Rule of Five** – we will discuss in a future lecture)

- Aside: We would avoid needing to do this if we simply used a `std::vector<T>` or `std::array<T>`!
  - These classes already manage heap-allocated memory properly for us
- Gives rise to our favorite rule: **The Rule of Zero**
  - Write classes in such a way that you can rely on the default behavior of operations like the destructor, copy-constructor, copy-assignment, etc.
Custom Copy-Assignment Operator

• Copy-assignment operator must follow specific rules
  • Make sure to release any dynamically-allocated resources, then allocate new resources to receive the values from the RHS (i.e. do a deep copy)
  • Return a non-const reference to the LHS of the assignment

• Example FloatArray implementation, take 1:

  FloatArray & FloatArray::operator=(const FloatArray &f) {
    delete[] elems;               // Release old memory
    count = f.count;
    elems = new float[count];    // Allocate new memory
    for (int i = 0; i < count; i++)
      elems[i] = f.elems[i];

    // Return non-const reference to myself
    return *this;
  }
Custom Copy-Assignment Operator (2)

• Example FloatArray implementation, take 1:

```cpp
FloatArray & FloatArray::operator=(const FloatArray &f) {
    delete[] elems; // Release old memory
    count = f.count;
    elems = new float[count]; // Allocate new memory
    for (int i = 0; i < count; i++)
        elems[i] = f.elems[i];
    return *this;
}
```

• What happens if we write this code?

```cpp
FloatArray f{1000};
... // Populate f
f = f;
```

f is both LHS and RHS of the assignment. First step is to delete the internal array of data... 😞
Custom Copy-Assignment Operator (3)

• Copy-assignment operator must follow specific rules
  • Make sure to release any dynamically allocated resources, then allocate new resources to receive the values from the RHS (i.e. do a deep copy)
  • Return a non-const reference to the LHS of the assignment
  • Properly identify and handle self-assignment!

• An easy way to detect self-assignment: compare the address of the LHS and RHS of the assignment
  • If they are the same address, can safely assume it’s self-assignment
A correct FloatArray implementation of copy-assignment:

```cpp
FloatArray & FloatArray::operator=(const FloatArray &f) {
    // Detect and handle self-assignment
    if (this == &f)
        return *this;

    delete[] elems; // Release old memory
    count = f.count;
    elems = new float[count]; // Allocate new memory
    for (int i = 0; i < count; i++)
        elems[i] = f.elems[i];

    // Return non-const reference to myself
    return *this;
}
```
The **bool** Type and Comparisons

• C++ has a **bool** type to use for representing Boolean values
  • Two values: **true** and **false**

• If you write code that keeps track of flags, or returns true/false based on a condition, use the **bool** type, not **int**!

• Example: Comparison operators

  ```cpp
  bool operator==(const MyClass &c1, const MyClass &c2) {
    ...
  }
  ```

• Easiest to implement != in terms of ==

  ```cpp
  bool operator!=(const MyClass &c1, const MyClass &c2) {
    return !(c1 == c2);
  }
  ```

• Ensures that != is truly the inverse of ==
C++ Inline Functions

- In C++, can provide the definition of functions as part of the declaration
  ```cpp
class Complex {
    double re, im;

  public:
    ...
    double real() const {
      return re;
    }
    
    double imag() const {
      return im;
    }
  }
```
- These are called **inline functions**
C++ Inline Functions (2)

• Due to its object-oriented nature, C++ encourages a high level of encapsulation and modularity in code
  • Make data-members private, and provide public member functions to access this state

• Problem: Function-invocations aren’t free
  • Must pass arguments, set up stack frame, jump to function code, jump back
  • The approach of the language encourages a lot of extra function invocations

• Solution: If a function is short and simple, the compiler can simply replace the function-invocation with the function’s body

• Example:
  
  ```cpp
  complex c = ...;
  cout << c.real() << "", " << c.imag();
  // Compiles into: cout << c.re << ", " << c.im;
  ```
C++ Inline Functions (3)

- Any function you define (i.e. write code for) in a class declaration is a candidate to be inlined...

- **The compiler will not blindly inline functions!** It will evaluate whether it makes sense to do so, or not
  - If a function is recursive, it usually won’t be inlined
  - If a function is large and complex, and will cause significant bloat in the binary file, it usually won’t be inlined
  - Inlining is primarily for short, simple functions

- asdf
C++ Inline Functions (4)

- Providing the definition of member-functions inline, *inside of* a class declaration, requires no additional syntax.

- Example: a file `complex.h`

```cpp
class Complex {
    double re, im;
public:
    ...
    double real() const {
        return re;
    }

    double imag() const {
        return im;
    }
};
```

- No need to define `Complex::real()` or `Complex::imag()` in the `complex.cpp` file if they are defined in the `complex.h` file.
If you wish to define a top-level function (i.e. not a member-function in a class) in the header file, you must use the `inline` keyword.

Example: still inside the file `complex.h`

```cpp
inline bool operator==(const Complex& c1, const Complex & c2) {
    return c1.real() == c2.real() &&
           c1.imag() == c2.imag();
}
```

Without the `inline` keyword, you will likely encounter “multiple definition” errors at compilation and link time 😞
This Week’s Assignment

• This week’s assignment will be to implement a 2D integer Matrix class whose dimensions can be specified to the constructor

• In C/C++, best approach to represent a 2D matrix/array is to map the 2D (row, column) coordinates into a 1D array
  • Numerous reasons for this, including performance, ease of maintenance, etc.

• Given a matrix of size rows x cols, how to map a given 2D (r, c) coordinate into the corresponding 1D cell?
  • index = r * cols + c (row-major order)
  • index = c * rows + r (column-major order)

• **Row-major order** means that column-values in the same row are physically adjacent to each other in memory

• C/C++ multidimensional arrays use row-major order
  • A few other languages (e.g. Fortran, MATLAB, R) use column-major order
This Week’s Assignment (2)

• Because the **Matrix** class dynamically allocates memory, it needs a destructor, a copy-constructor, and a copy-assignment operator
  • Follow the **Rule of Three**!

• As usual, write Doxygen-style comments, and write a Makefile

• Tests are provided! 😊