Constructors, Destructors & Arrays of Objects

CS 250 - Session 2
C++ Programming II

The Plan for Tonight

- Reading - 409-441
- Class definition review [409-410]
- Properties of variables and built-in types
- Constructors and destructors [410-421]
- const member functions and this [422-429]
- Arrays of objects [429-432]
- Class scope and class enumerations [432-434]
- Abstract Data Types [ADTs] and the stack [434-439]
- Homework 3, and 4

Classes: Review

- How do you declare a class?
  - Basic structure
  - Access specifiers
  - Declaring data members & member functions
- How do you define a member function?
  - Using the scope resolution operator
  - How about inline member functions?
- How do you create an object and call methods?
Variables and Objects

- What are variables?
  - Named regions of memory that hold data
- What are objects?
  - Variables of a user-defined [class] type
- The goal of object-based programming:
  - Design new types that behave like the built-in types
  - We have to understand variable behavior to design useful user-defined types

Desired Properties I

- Built-in types have four properties:
- Property 1: Creation of variables
  - Language [or libraries] provide support for allocation of memory and for binding of a name and type to that memory.
  ```
  int a; float b;
  char * cp;
  cp = new char[1000];
  ```

Desired Variable Properties II

- Property 2: Destruction
  - Stack-based variables are automatically destroyed when they go out of scope.
  - Static variables are automatically destroyed when the program ends.
  - Dynamic variables are destroyed by the programmer, using the delete or delete [] operators
Desired Variable Properties III

- Property III: Assignment
  - You can copy the value from one variable to another using assignment
    - `a = b;` // Copies contents of `a` to `b`
    - The values of each variable should be identical
    - This is called value semantics
  - Pointers do not have value semantics
    - `char * a = "Fred", * b = "Barney";`
    - `a = b;`

Desired Variable Properties IV

- Property 4: Initialization
  - A: Initialize via value (constant or literal)
    - `int a = 32; double b = 3.3;`
    - `char name1[ ] = "Fred";` // OK?
  - B: Initialize via variable of same type
    - `int a = 32; int b = a;`
    - `char name2[ ] = name1;` // This?
  - C: Initialize via another type
    - `int a = 32; double b = a;`

Constructors I

- C++ allows classes to act like first-class variables, not like derived types
  - It does this by defining certain "built-in" methods
  - These methods are used to provide desired variable behavior to class types
  - Constructors allow variables to be created and initialized
  - Destructors allow variables to be destroyed
  - Assignment operator allows variables to be copied
Constructors II

- Let's look at a simple class called EngDist.
  - It stores distances in inches and feet
    ```cpp
    class EngDist
    {
    private:
        long feet;
        double inches;
    public:
        void read();
        void print();
        void set_distance(long f, double i);
    };
    ```

Constructors III

- Let's look at simple creation (property 1)
  - What happens when we do this?
    ```cpp
    int i;
    cout << "i = " << i;
    ```
  - How about this?
    ```cpp
    EngDist ed;
    cout << "ed = ";
    ed.print();
    ```
  - We get random values in both cases

Constructors IV

- Conclusions? For simple creation...
  - Objects work just like built-in types
    - Values in built-in types are undefined
    - Values in object types are also undefined
  - How do we handle this with primitives?
    - We must remember to initialize our variables
  - Let's try that with our EngDist object
    ```cpp
    EngDist ed2 = { 0, 0.0 };```
Constructors V

- Doesn't work, feet & inches are private
  - Can change EngDist header, make public
  - Bad idea because
    » Data can now be "ill formed" [no inches > 12]
    » Objects should keep implementation hidden
    » You don't set alarm clock by unscrewing the case
  - This core principle is called encapsulation
- Instead, every class has initialization method(s)
  - These are called constructors

Constructors VI

- Constructors make objects better than primitives
  - Constructors implicitly invoked by creating an object
  - Means all objects can be automatically initialized
- How do you define a constructor?
  - Write an initialization method [like set_distance()]
  - Method name must be same name as class
  - Method can have no return type [not even void]
    » Think of constructor as a factory that produces objects
- Often, a class has several overloaded constructors

Working Constructor I

- Normally you want to initialize to specific value
  - Can do that with primitive types like this:
    » int i = 32;
  - For object types, use a working constructor
    "A working constructor is the constructor that supplies a value for every user-modifiable field"
  - Let's add constructors to EngDist class
Working Constructor II

- Let's define a working constructor for EngDist
  - Need arguments for each field, inches & feet
  - Use EngDist as name, no return value.
  - Can use set_distance() method to initialize
  - Declaration:
    EngDist(long f, double i);
  - Definition:
    EngDist::EngDist(long f, double i)
    { set_distance(f, i); }

Working Constructor III

- How do you use a working constructor?
  - Can use the constructor name like this
    - EngDist ed = EngDist(275, 1.5);
  - Normally, add parentheses after variable name
  - Supply values in parentheses
    - EngDist ed(275, 1.5);
  - What happens when you add this code to client?
    - EngDist ed2;
  - Why did the program just stop working?

Default Constructor

- If a class has no constructor, C++ "writes" one
  - This is called the built-in default constructor
  - Works just like creation for primitives: no initialization
- Can write an overloaded default constructor
  - Declaration looks like this:
    - EngDist();
  - Definition could look like this:
    - EngDist::EngDist() { inches = feet = 0; }
Other Constructors

- You can create other overloaded constructors
  - A single-argument constructor
  - Called a conversion constructor
    - EngDist(double dist); // As feet
  - Allows you to create variables two ways
    - Explicitly: EngDist ed(23.45);
    - Implicitly: EngDist ed2 = 23.3 * 2 / 7;
    - EngDist ed3 = 75; // ????
  - Remember the argument matching rules

More Constructor Details

- Initialization syntax [NIB]
  - Can use “constructor” syntax to initialize primitives
    - int a = 3; int b(4);
- With constructors, can use initialization list
  - Following argument list, add colon, initializers
  - Preferred method of simple initialization
    - EngDist::EngDist() : feet(f), inches(i)
      {
        // nothing in body
      }

Destructors

- A destructor is a function called when an object is destroyed [out of scope]
  - Like constructor, has no return type
  - Unlike constructor, cannot have arguments
  - Name is same as class with tilde in front
    - EngDist::~EngDist() {}
- How long does function x() take to execute?
  - Let's use destructors to create a Timer class
  - Used to measure time inside a function
Designing the Timer Class

- Here's how the Timer class works
  - When you start a function, create a Timer object
    - Pass the name of the function as an argument
    - Timer constructor prints out starting message
    - Use `<ctime>` `clock()` function to get starting time
    - Save starting time in data member
  - When function is finished, destructor is called
    - Use `clock()` to determine current processor time
    - Find difference between current and starting time
    - Print number of seconds using `CLOCKS_PER_SEC`

Improving the Timer class

- Save starting message, print in destructor
  - Could use simple char array
  - Let's use dynamic memory instead
  - 1. Add a char * pointer named message
  - 2. In constructor, allocate memory
    - Use `new`, don't bother checking for failure
  - 3. In destructor, delete memory

const Member Functions

- When do you use `const`?
  - 1) function takes array or structure reference
  - 2) function does not change structure
  - If `Thing` is a structure, then you do this:
    - `void print(const Thing& t) ...`
- What do you do if `Thing` is a class?
  - If function does not change data member:
    - `void Thing::print() const;`
Object Arguments I

- You can pass objects as arguments
  - To free [stand-alone] functions
    ```cpp
    void display(char * s, EngDist d) {
        cout << s;
        d.print();
    }
    EngDist d1(2, 3);
    display("d1 = ", d1);
    ```
  - The display() function is not a member function

Object Arguments II

- What happens when you run the code?
  ```cpp
  EngDist object # 1 constructed
d1 = 2' 3"
EngDist object # 1 destructor called
EngDist object # 1 destructor called
  ```
- Why is the destructor called twice?
  1. Objects [like other args] are passed by value
  2. Uses a constructor called the copy constructor
  3. Temporary copy is destroyed when function ends
- Solution? Pass object arguments by reference

Object Arguments III

- Free functions have no access to object's data
  - Can only call public functions, access public data
    ```cpp
    void display(char * s, EngDist d) {
        cout << s;
        d.print(); // OK, public method
        cout << "feet = " << d.feet; // Illegal, private data
    }
    ```
Object Arguments IV

- Can also pass objects to member functions
  - If object is member of same class, then the method has access to all the private data of the argument

```cpp
void EngDist::add_to(const EngDist& d2)
{
    set_distance( feet + d2.feet,
                    inches + d2.inches);
}
```

Object Arguments V

- Remember, arguments normally passed by value
  - What happens here?

```cpp
void EngDist::copy_to(EngDist d2) const {
    d2.feet = feet;
    d2.inches = inches;
}
EngDist d(5, 4), d1;
    d.copy_to(d1); // Passes a copy of d1
    d1.print();    // d1 is not changed
```

- Changes are not reflected in original argument

Object Arguments VI

- Instead, use pass-by-reference for objects
  - This should work

```cpp
void EngDist::copy_to(EngDist& d2) const {
    d2.feet = feet;
    d2.inches = inches;
}
EngDist d(5, 4), d1;
    d.copy_to(d1); // Pass by reference
    d1.print();    // d1 is changed
```

- If method doesn’t modify argument, use const
Returning Objects I

- What kinds of things can a function return?
  - How about this?

```c
char * get_array() {
    char array[80];
    cin.getline(array, 80);
    return array;
}
```

- Code compiles but array is gone
- Conclusion: Never return a pointer to local data

Returning Objects II

- You can return structures in C and C++
  - Here's an example that does about the same thing:

```c
struct str { char data[80]; }
str get_str() {
    str buf;
    cin.getline(buf.data, 80);
    return buf;
}
```

- What happens when you do this with an object?

Returning Objects III

- Let's look at EngDist and see
  - Let's add following free function

```c
EngDist get_distance() {
    EngDist local(3, .5);
    return local;
}
```

- Constructor called once, destructor twice
- Temporary copy made for return value
Returning Objects IV

- What if constructor allocates resource?
  - If constructor allocates memory, it will be deleted twice
  - Conclusion?
    - Don't return objects from functions, if the object allocates resources in its constructor.
  - Long-term answer to all of these problems?
    - Add a new kind of constructor: copy constructor
    - Add an overloaded assignment operator
  - We'll tackle these problems in Chapter 12

Object Arrays I

- You can easily create arrays of objects
  - Here's an example:
    ```
    EngDist a1[2];
    display("a1[0] = ", a1[0]);
    display("a1[1] = ", a1[1]);
    ```
  - What happens when this is run?
    - Default constructor called for every object
    - Your class must have a default constructor if you wish to create uninitialized arrays of objects.

Object Arrays II

- You can initialize an array by explicitly calling a constructor for every element
  - Here's an example of explicit object array initialization
    ```
    EngDist a2[3] = {
        EngDist(),       // Use default constructor
        EngDist(2),     // Use 1-arg constructor
        EngDist(3, 4)  // Use working constructor
    };
    ```
Object Arrays III

- You can also implicitly initialize an array if the array has a single-argument constructor
  - Here's an example:
    ```cpp
    EngDist a3[3] = { 5, 7, 9 };  
    ``
  - This is interpreted like this:
    ```cpp
    EngDist a3[3] = {
    EngDist(5),  
    EngDist(7),  
    EngDist(9)   
    }; 
    ``

The Pointer Named this I

- In modular-style C++ programming, you first design a data structure, then provide functions that operate on that data structure
  - Here's an example (modified from Stroustup 10.2.1):
    ```cpp
    struct Date {
    int month, day, year;
    };  
    void init(Date* dp, int m, int d, int y);  
    void addDay(Date* dp, int n);  
    string toString(const Date* dp); 
    ```

The Pointer Named this II

- When you call a function like init(), you pass the address of a Date variable.
  - Here's how you'd use this code:
    ```cpp
    Date today, birthday;  
    init(& today, 2, 9, 2004); // Initialize today  
    init(& birthday, 2, 2, 2004); // Init birthday  
    addDay(& birthday, 7); // Add to birthday  
    cout << toString(& today) << endl;  
    cout << toString(& birthday) << endl;  
    ```

- Each function gets a pointer to the "current" Date
The Pointer Named this III

Object-based programming is similar, but you don’t pass an explicit Date pointer:
– Design a class and methods, removing Date *:

```cpp
class Date {
  int month, day, year;
public:
  Date(init(Date* dp, int m, int d, int y);
  addDay(Date* dp, int n);
  string toString(const Date* dp) const;
};
```

The Pointer Named this IV

Every member function is passed an invisible pointer to the “current” object, named this:
– Here’s how the previous example looks:

```cpp
Date today, birthday;
today.init(2, 9, 2004); // Initialize today
birthday.init(2, 2, 2004); // Init birthday
birthday.addDay(7); // Add to birthday
cout << today.toString() << endl;
cout << today.toString() << endl;
```

Application: Arrays and this I

Let’s find the largest element in an object array, using this algorithm:
– 1. Assume that first element is the largest
– 2. Look through array comparing each element
– 3. When larger element found, assign to largest

```cpp
EngDist largest = a3[0]; // first element
for (int i = 1; i < 3; ++i)
  largest = max(a3[i];
```
Application: Arrays and this II

- Let's add the `max()` function to `EngDist` class
  - 1) Will compare two objects, return the largest
  - 2) Argument is not modified so it should be `const`
  - 3) Return not modified, also should be a `const`

  ```cpp
  const EngDist& max( const EngDist& rhs );
  ```

- To calculate the max, use this algorithm
  - If current-object.distance >= rhs.distance then return reference to current-object, otherwise return reference to rhs object

Application: Arrays and this III

- Inside the `max()` method, we can use `*this` to return a reference to the current object
  - Here's the finished `max()` member function:

  ```cpp
  const EngDist& EngDist::max(const EngDist& rhs) const
  {
    if (feet * 12 + inches) >=
        (rhs.feet * 12 + rhs.inches)
    return *this;
    return rhs;
  }
  ```

Enumerated Types I

- What are enumerated types?
  - User-defined, single-value, integer-like types
  - Defined using the `enum` keyword
    ```cpp
    enum gender { female, male }; 
    ```
  - female, male are `values` of the gender type

- In C++ enumerations are:
  - Strictly type-checked [ unlike C ]
  - Useful for creating "class constants"
  - See pages 119-122 for more info on basic usage
Enumerated Types II

- Most common use is "class constants"
  - Class definitions can't have initialized data values
    ```cpp
    class Label {
    const int LEFT = 0; // Illegal
    };
    ```
  - Use an in-class enumeration instead
    ```cpp
    class Label {
    enum Align { LEFT, RIGHT};
    }
    ```

- How do you use a class constant?
  - Inside methods, simply refer to values like this:
    ```cpp
    Align cur_align = LEFT;
    ```
  - Use enum type as formal argument to methods
    ```cpp
    Label(char * s, Align a) {
    }
    ```
  - Outside of class, use scope resolution to pass arg
    ```cpp
    Label two("Barney", Label::RIGHT);
    ```

- Note: Enumeration must be public to do this

Enumerated Types III

- How do you use a class constant?
  - Inside methods, simply refer to values like this:
    ```cpp
    Align cur_align = LEFT;
    ```
  - Use enum type as formal argument to methods
    ```cpp
    Label(char * s, Align a) {
    }
    ```
  - Outside of class, use scope resolution to pass arg
    ```cpp
    Label two("Barney", Label::RIGHT);
    ```

- Note: Enumeration must be public to do this

Constant static Data Members

- Standard C++ now allows you to define integral constants as part of your class
  - Only works for integer & enumerated values
  - Could now define Label constants like this:
    ```cpp
    class Label {
    static const int LEFT = 0;
    static const int RIGHT = 1;
    };
    ```
  - Lose the type-checking ability built-in to enums
Abstract Data Types I

- What is a data type?
  - Region of memory that holds a value
  - Value behaves in a particular way
  - An integer is an abstract data type
    » Bits in memory are interpreted as whole numbers
    » Can be added, subtracted, etc.
    » Can't hold a fractional amount
  - Specific [concrete or implementation-dependent]
    data types provide additional limitations
    » Size of variables, range, two’s complement, etc.

Abstract Data Types II

- One common ADT is the container class
  - Described in terms of its behavior
    » Are items ordered, unordered?
    » How are items added and removed?
    » How are items processed?
  - Don't care what kind of thing is stored
  - Don't care about the underlying implementation
    » Can store items in arrays or linked-lists

Abstract Data Types III

- To design an ADT, describe its behavior
  - One of simplest ADTs is the Stack
  - Only two required operations:
    » push() : add an item to the top of the stack
    » pop() : remove an item from the top of the stack
  - Additional operations are not part of abstraction
    » isempty(): used to prevent an error condition
    » isfull(): used if implementation is fixed size
  - Also need constructor & destructor
The <cassert> Header I

- C++ Standard Library includes 18 C headers
  - Want to get some exposure to most facilities
- The assert macro helps discover code errors
- Use to specify invariants in your functions
  - Invariant: something that must always be true

```cpp
#include <cassert>
Duck::Duck(const char * name) {
    assert(name != 0); // name must not be null
}
```

The <cassert> Header II

- What happens when assert is called?
  - The invariant condition is tested
  - If the condition is false, assert writes an error message to standard error and aborts the program
- Assert is only active in "debug builds"
  - You can "turn off" assert by #defining NDEBUG
- Assert is used to fix logic errors, not handle exceptional situations

Homework

- Homework 3: Constructors
  - Due Sunday, Feb 22, [Grade early Monday]
  - Add three constructors to the Fraction class
  - Add private reduce() member function to class
  - Use hmwk03.cpp and Fraction02.h without changes
  - Note that the Fraction class now uses const, so you will have to change your method definitions
  - Note that next Monday is a holiday
Homework 4: Classes as data members
– Due Sunday, Feb 22, [Grade early Monday]
– Create Calculator class
  » 2 Fraction fields, 1 Operator field
  » Operator is in-class enumerated type
– Use hmwk04.cpp as a starter
  » Declare & define Calculator class in hmwk04.cpp
  » Add four "free" functions: add(), subtract(), etc.
  » Each takes two Fraction objects, produces another
– Use Fraction02.h, Fraction02.cpp from Homework 3