CS24: Introduction to Computing Systems

Spring 2015
Lecture 10
**Previously: Array Access**

- C doesn’t provide a very safe programming environment
- Previous example: array bounds checking

```c
int a;
int r[4];
int b;
...

r[4] = 12345; /* Compiles! */

r[-1] = 67890; /* Also compiles! */
```

- Depending on variable placement, could affect:
  - `a` and/or `b`
  - Caller’s `ebp`, return address on stack, etc.
- Or, perhaps nothing at all!
**CHECKED ARRAY INDEXING**

- Could add metadata to arrays

  ```c
  struct array_t {
    int length;  /* Number of elements */
    struct value_t values[];
  };
  ```

  - Arrays include length information in their run-time representation
  - Last member of a struct can be an array with no size

- To initialize a new array:

  ```c
  array_t *a = (array_t *)
     malloc(sizeof(array_t) + n*sizeof(value_t));
  a->length = n;
  ```

  - `values` is a pointer to start of variable-size array

| a | length | values[0] | values[1] | ... | values[n-1] |
ARRAY BOUNDS-CHECKING (2)

- Arrays are now a more intelligent data type:
  ```java
  for (int i = 0; i < a.length; i++) {
    compute(a[i]);
  }
  ```
  - A composite type containing multiple related values
  - Ideally, \texttt{length} would be read-only, and every indexing operation would be verified against \texttt{length}

- If only our type could also expose specific behaviors...
  - Operations that can be performed on these values
  - e.g. expose \texttt{length} via a function, or check indexes in an access function
OBJECT ORIENTED PROGRAMMING

- **Idea:**
  - Group together related data values into a single unit
  - Provide functions that operate on these data values
  - This state and its associated behavior is an **object**

- **A class is a definition or blueprint of what appears within objects of that type**

- **Encapsulation:**
  - Disallow direct access to the state values of an object
  - Provide accessors and mutators that control *when* and *how* state is modified

- **Abstraction:**
  - Class provides simplified representation of what it models
  - Compose simpler objects together to represent assemblies
    - e.g. a Car has an Engine, a Transmission, Pedals, a SteeringWheel, Instruments, etc.
OBJECT ORIENTED PROGRAMMING (2)

Idea:
- Object oriented programming paradigm makes it easier to create large software systems
- Promotes modularity and encapsulation of state
- Provides sophisticated modeling and abstraction capabilities for programs to use

(Not everyone believes that OOP is best way to provide these features...)

Many different object-oriented languages now!
- C++, Java, C#, Scala, Python, Ruby, JavaScript, Perl, PHP, ...

Today: focus on some OO features found in Java
Java presents a specific object-oriented programming model

- Includes some kinds of variables we recognize:
  - Global variables
  - Function arguments
  - Local variables

Object-oriented model also introduces:
- Class variables
- Instance variables

```java
public class RGBColor {
    public static RGBColor RED =
        new RGBColor(1.0, 0.0, 0.0);

    private float red, green, blue;

    public RGBColor(...) { ... }

    public void setRed(float v) {
        red = v;
    }

    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
        ...
    }
}
```
Java Types and Abstractions (2)

- How do environments fit together to provide these kinds of state?
  - Global variables
  - Function arguments
  - Local variables
  - Class variables
  - Instance variables

```java
public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);

    private float red, green, blue;

    public RGBColor(...) {
    }

    public void setRed(float v) {
        red = v;
    }

    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
    }
}
```
**Java Types and Abstractions (3)**

- Example environment structure:

```
public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);

    private float red, green, blue;
    public RGBColor(...) { ... }
    public void setRed(float v) {
        red = v;
    }
    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
    }
}
```
Example environment structure:

- Need to introduce new frames to represent classes
- Also need frames to represent specific instances of a class
Java Types and Abstractions (5)

- Example environment structure:
  - Can use memory heap to implement these frames
  - By controlling what programs can do with references, can also provide precise garbage collection for frames
    - Clean up objects when no longer referenced by any frame
    - Can even remove class definitions when not referenced by any object (used by Java application servers for code-reloading)
IMPLEMENTING OOP IN C

- How might we implement this object-oriented programming model in C?
- A very rich topic... Definitely won’t cover it all

- Start with basic object-oriented concepts
  - See how these translate into C-style concepts
- Build up the model until it includes most Java OOP capabilities

- Discussion will necessarily be at a high level
  - Many implementation details left out!!
Implementing OOP in C (2)

- Can already implement some aspects of this model
  - Function calls, local variables
  - Implement these with a stack

```
public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);
    private float red, green, blue;
    public RGBColor(...) { ... }
    public void setRed(float v) {
        red = v;
    }
    ...
    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
        ...
    }
}
```
Implementing OOP in C (3)

- How to store object data in C?
  ```c
  public class RGBColor {
    private float red, green, blue;
    ...
  }
  ```

- C provides composite data types using `struct`

- Can use this to represent the data in our objects
  ```c
  struct RGBColor_Data {
    float red;
    float green;
    float blue;
  };
  ```

  - This `struct` loosely corresponds to a class declaration
  - Variables of this type will represent individual objects

- Each `RGBColor` object will have its own frame for its state variables
IMPLEMENTING OOP IN C (4)

- C representation of our object:
  ```c
  struct RGBColor_Data {
    float red;
    float green;
    float blue;
  };
  ```

- Need a way to provide methods as well:
  ```java
  public class RGBColor {
    ...
    public float getRed() {
      return red;
    }
    public void setRed(float v) {
      red = v;
    }
  }
  ```

- No explicit argument representing the object
  ```java
  c.setRed(0.5);
  ```
METHODS AND this

- Need to introduce an implicit parameter into methods
  - **this**: a reference to the object the method is called on

- Object-oriented code:

  ```java
  public class RGBColor {
      ...
      public void setRed(float v) {
          red = v;
      }
      ...
  }
  ```

- Translate into this equivalent C code:

  ```c
  RGBColor_setRed(RGBColor_Data *this, float v) {
      this->red = v;
  }
  ```
Methods and this (2)

- Instance methods include an implicit parameter **this**
  - Allows the object’s code to refer to its own fields
- When a program calls a method on an object:
  - The underlying implementation transparently passes a reference to the called object, to the method code
- A common feature across all OO languages
  - Some languages explicitly specify this parameter
  - e.g. Python:
    ```python
    class RGBColor:
        def __init__(self, red, green, blue):
            self.red = red
            self.green = green
            self.blue = blue

        def get_red(self):
            return self.red
    ```
METHODS CALLING METHODS

- Methods frequently call other methods
  ```java
  public float getGrayScale() {
      return 0.30 * getRed() +
          0.59 * getGreen() +
          0.11 * getBlue();
  }
  ```
  - Calls other methods on the same object

- Must pass the **this** reference to called methods
  ```java
  float RGBColor_getGrayScale(RGBColor_Data *this) {
      return 0.30 * RGBColor_getRed(this) +
          0.59 * RGBColor_getGreen(this) +
          0.11 * RGBColor_getBlue(this);
  }
  ```
  - Again, straightforward translation to support this
**Object Frames**

This approach allows us to implement our object frames:
- Programs can manipulate independent objects of a class.

```java
public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);

    private float red, green, blue;

    public RGBColor(...) {
    }

    public void setRed(float v) {
        red = v;
    }

    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
        ...
    }
}
```
Frames for Classes?

- Our example also has a class-level constant:
  
  ```java
  public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);
    ...
  }
  ```

  - Many OO languages call these static members
  - Member isn’t associated with a specific object
  - Refer to member using the class name:
    
    ```java
    g.setColor(RGBColor.RED);
    ```

- Clearly requires a frame at the class-level for such constants
- Object frames should also reference their class’ frame
  - Specifies object’s type, allow easy use of static members
**RGBColor Class Frame**

- Simple frame for our `RGBColor` class:
  ```
  struct RGBColor_Class {
    RGBColor_Data *RED;    /* static member */
  }
  ```

- Update definition of `RGBColor_Data`:
  ```
  struct RGBColor_Data {
    RGBColor_Class *class;    /* type info */
    float red;
    float green;
    float blue;
  }
  ```

- A new problem: how to initialize the static members?
- Classes define constructors to set up new objects...
- Similarly, init class info first time type is referenced
RGBColor Class Frame

- Mechanism to initialize RGBColor class frame:

  ```c
  RGBColor_class_init(RGBColor_Class *class) {
     /* Initialize static member RED. */
     class->RED = malloc(sizeof(RGBColor_Data));
     RGBColor_init(class, class->RED, 1.0, 0.0, 0.0);
  }
  ```

- Global environment manages class frames, somehow...

- Simple RGBColor constructor translated to C:

  ```c
  RGBColor_init(RGBColor_Class *class, RGBColor_Data *this, float red, float green, float blue) {
     this->class = class;
     this->red   = red;
     this->green = green;
     this->blue  = blue;
  }
  ```
Now we can do everything in our simplified object-oriented programming model!

```java
public class RGBColor {
    public static RGBColor RED = new RGBColor(1.0, 0.0, 0.0);
    private float red, green, blue;
    public RGBColor(...) { ... }
    public void setRed(float v) {
        red = v;
    }
    ...
    public void fromHSV(float h, float s, float v) {
        float p = v * (1.0 - s);
        ...
    }
}
```

### RGBColor object
- red: 0.0
- green: 0.0
- blue: 0.0

### global environment
- RGBColor: RGBColor
- RGBColor.RED: RGBColor
  - red: 1.0
  - green: 0.0
  - blue: 0.0
- h: 0.5
- s: 1.0
- v: 1.0
- p: 0
Object-oriented programming languages also support class inheritance and polymorphism.

**Class inheritance:**
- Can construct hierarchies of classes
- Parent classes represent more general-purpose types
- Child classes are specializations of parent classes
  - Can extend functionality of parent classes with new fields and methods
  - Can override parent-class methods with specialized features

**Polymorphism:**
- Parent class specifies a common interface
- Subclasses provide specialized implementations
- Code written against the parent class behaves differently, depending on which subclass it is given
CLASS INHERITANCE AND POLYMORPHISM

Instead of a simple RGBColor class, provide a color class hierarchy

- Parent class specifies methods that all subclasses will provide
  - `toRGB()` produces an integer value usable by the graphics hardware
    - Bits 16-23 are red value
    - Bits 8-15 are green value
    - Bits 0-7 are blue value

Implement two subclasses

- **RGBColor** subclass, using RGB color space
  - Red, green, blue color components
- **HSVColor** subclass, using HSV color space
  - Hue, saturation, value; effectively implements a color wheel
COLOR SPACES

- RGB and HSV are different *color spaces*
  - Different ways of representing colors
- RGB mixes red, green, and blue components
  - Used by virtually all graphics hardware
- HSV combines hue, saturation, and value
  - Frequently used for color-choosers in UIs
  - Also used frequently in computer vision
Now, can implement functions that use the abstract base-class—
- `Shape.setColor(Color)`
- `Graphics.setColor(Color)`
- etc.

Programs can use the type of color that makes sense for them.

Graphics code can use `toRGB()` method to set up for drawing:
```java
public class Graphics {
    ...
    public void setColor(Color c) {
        device.setRGB(c.toRGB());
    }
}
```
Color Class Hierarchy (2)

- Graphics code:
  ```java
  public class Graphics {
      ...
      public void setColor(Color c) {
          device.setRGB(c.toRGB());
      }
  }
  ```

- **RGBColor** and **HSVColor** provide different versions of this function!
  - **RGBColor** can simply pack up the RGB components into an `int`
  - **HSVColor** must convert to RGB before returning the result

- **Graphics.setColor()** needs to call proper version of `toRGB()`, depending on type of argument!
**Virtual Functions**

- `toRGB()` and `getGrayScale()` are called virtual functions.
- Subclasses have provided their own implementations...
- Code written against base-class must call appropriate version when passed a subclass object.

Public class graphics {
  ...
  public void setColor(Color c) {
    device.setRGB(c.toRGB());
  }
}

- Somehow, objects must indicate which version of `toRGB()` to use.
CLASSES AND FUNCTION POINTERS

- Each object already has a reference to its class info...
- Simple solution:
  - Add details of which methods go with each type, into the class information
  - Can look up which method to call, using object’s class-info
- The C language supports function pointers
  - Instead of a pointer to data, points to a function
  - A function-pointer \( \text{fp} \), which points to a function that takes a \text{double} and returns a \text{double}:
    
    \[
    \text{double (*fp)(double);} \\
    \]
  - Set \( \text{fp} \) to point to the \text{sin()} function:
    
    \[
    \text{fp} = \text{sin}; \quad /* \text{Note: NO parentheses!!} */ \\
    \]
  - Call the \text{sin()} function through \( \text{fp} \):
    
    \[
    \text{result} = \text{fp}(x); \\
    \]
**COLOR CLASS DETAILS**

- **Base class representation:**
  ```c
  struct Color_Class {
    int (*toRGB)(Color_Data *this);
    float (*getGrayScale)(Color_Data *this);
  };
  ```
  - Two function pointers, one for each virtual function

- **Subclass type information is identical**
  - (These subclasses don’t have static members...)
  ```c
  struct RGBColor_Class {
    int (*toRGB)(RGBColor_Data *this);
    float (*getGrayScale)(RGBColor_Data *this);
  };
  ```
  ```c
  struct HSVColor_Class { ... /* same idea */ };
  ```
COLOR CLASS DETAILS (2)

- Can model basic class-inheritance with C structs
- Declare “base-type” struct with certain members
  ```c
  struct Color_Data {
    Color_Class *class;
  };
  ```
- “Sub-type” structs can add other members if needed, but **must** have same types of members at the start!
  ```c
  struct RGBColor_Data {
    RGBColor_Class *class;
    float red;
    float green;
    float blue;
  };
  ```
- Then, can cast a base-type pointer to subtype pointer
  - The common members are at same offsets in both structs
COLOR CLASS INITIALIZATION

Now our class-initialization code becomes:

```
RGBColor_class_init(RGBColor_Class *class) {
    /* Initialize function pointers */
    class->toRGB = RGBColor_toRGB;
    class->getGrayScale = RGBColor_getGrayScale;
}

HSVColor_class_init(HSVColor_Class *class) {
    /* Initialize function pointers */
    class->toRGB = HSVColor_toRGB;
    class->getGrayScale = HSVColor_getGrayScale;
}
```

Objects of each type can easily invoke the proper version of `Color.toRGB()` now.
COLOR-OBJECT DATA TYPES

- **RGBColor_Data** definition is same as before:
  ```c
  struct RGBColor_Data {
    RGBColor_Class *class; /* type info */
    float red;
    float green;
    float blue;
  };
  ```

- **HSVColor_Data** definition:
  ```c
  struct HSVColor_Data {
    HSVColor_Class *class; /* type info */
    float hue;
    float saturation;
    float value;
  };
  ```
Graphics Code Translation

- Our Graphics code from before:
  ```java
  public class Graphics {
      ...
      public void setColor(Color c) {
          device.setRGB(c.toRGB());
      }
  }
  ```

- Translate into C code:
  ```c
  void Graphics_setColor(Graphics_Data *this, 
                          Color_Data *c) {
      Device_setRGB(this->device, c->class->toRGB(c));
  }
  ```

- If RGBColor passed in, RGBColor_toRGB() is used
- If HSVColor passed in, HSVColor_toRGB() is used
Note the two different calling patterns:

```c
void Graphics_setColor(Graphics_Data *this, Color_Data *c) {
    Device_setRGB(this->device, c->class->toRGB(c));
}
```

- Non-virtual methods do not support polymorphism
  - The method is chosen at compile-time, and cannot change
    - Also called static dispatch
  - Doesn’t require an extra lookup, so it’s faster
- Virtual methods do support polymorphism:
  - Method is determined at run-time, from the object itself
    - Also called dynamic dispatch
  - *Essential* when methods are overridden by subclasses!
  - Slightly slower, due to the extra lookup
Object Oriented Programming Model

- Our simple example now supports simple class hierarchies and polymorphism
- Conceptually straightforward to implement in C
  - Structs to represent data for objects and classes
  - Implement virtual functions by storing function-pointers in the class descriptions
  - Look up which virtual function to call at run-time, *directly from the object itself*

- Note 1: almost all Java methods are virtual
  - ...unlike C++, where member functions must explicitly be declared virtual
OOP Model (2)

- Note 2:
  - Many OOP languages represent virtual function pointers with a *virtual-function pointer table* (a.k.a. vtable)

- Our representation:
  ```
  struct color_class {
    int (*toRGB)(Color_Data *this);
    float (*getGrayScale)(Color_Data *this);
  };
  ```
  - Our simple example includes more type information

- Frequently:
  - Class information contains an array of virtual function pointers (or references)
  - Individual functions are often referred to by slot-index
    - e.g. slot 0 = toRGB(), slot1 = getGrayScale()
  - Some languages (like Java) refer to functions by name
Note 3: Our example is distinctly hard-coded...
- Mapped our example classes to C structs and code
- Doesn’t support the same ability to dynamically load and run code that the Java VM provides!

Java virtual machine uses sophisticated data structures to represent class information
- ...including fields, method signatures, method definitions, class hierarchy information...

Allows Java VM to dynamically load class definitions and execute them
- Even allows Java programs to generate new classes on the fly, then load and run them!