

Aggregate Data Types

Jade Abraham

October 7, 2025

Aggregate Data Types in Chapel

- Chapel has two kinds of aggregate data types, records and classes
- Records: represents a block of data that is present in the variable itself (i.e., on the stack)

```
record point {  
    var x: int;  
    var y: int;  
}  
var p = new point(1, 2);
```

- Classes: represents a block of data that is pointed to by the variable (i.e., on the heap)

```
class Point {  
    var x: int;  
    var y: int;  
}  
var p = new Point(1, 2);
```

What is a record?

- Represents a block of data that is present in the variable itself
 - i.e., on the stack

```
record point {  
    var x: int;  
    var y: int;  
}  
var p = new point(1, 2);
```

```
var p = new point();
```

```
var x: int
```

```
var y: int
```

- Can have member variables and methods
- Can be generic
- Cannot use inheritance
- Can use interfaces (will not be discussed today)
- By convention, use camelCase to name them

What is a class?

- Represents represents a block of data that is pointed to by the variable
 - i.e., on the heap

```
class Point {  
    var x: int;  
    var y: int;  
}  
var p = new Point(1, 2);
```

var p = new Point();

var x: int

var y: int



- Can have member variables and methods
- Can be generic
- Can use inheritance
- Can use interfaces (will not be discussed today)
- By convention, use PascalCase to name them

When should I use a class or a record?

- I need to group my data in a named structure
 - Use a record
- I need an is-a- relationship. e.g., I want a parent Animal with multiple children like 'Dog' and Cat
 - Use a class
- I need a has-a- relationship. e.g., I want a Car that has four Wheels
 - Use a record
- I need to represent a tree-like structure. e.g. a linked list or a graph
 - Use a class

Defining methods

```
record point {  
    var x: int;  
    var y: int;  
    proc getX() {  
        return x;  
    }  
    proc ref setY(y: int) {  
        this.y = y;  
    }  
}  
proc point.printMe() {  
    writeln(this);  
}
```

```
var p = new point();  
writeln(p.getX());  
p.setY(10);  
p.printMe();
```

Define the method 'getX()'

'this' is implicit

Define the method 'setY()', making 'this' mutable

'this' is explicitly use, since there are 2 'y' variables

Define the secondary method 'printMe()'

Defining operators

```
record point {  
    var x: int;  
    var y: int;  
}  
  
operator +(lhs: point, rhs: point) {  
    var p: point;  
    p.x = lhs.x + rhs.x;  
    p.y = lhs.y + rhs.y;  
    return p;  
}
```

← A special method that defines what '+' means

```
var p1 = new point(2, 3);  
var p2 = new point(4, 5);  
var p3 = p1 + p2;  
writeln(p3);
```

← Calls our special method

Defining constructors

```
record point {  
  var x: int;  
  var y: int;  
  proc init(c: int) {  
    this.x = c;  
    this.y = c;  
  }  
  proc init() {  
    this.x = 0;  
    this.y = 0;  
  }  
}
```

Define a constructor for 'point'

Define the previously compiler-generated constructor

```
var p1 = new point();  
writeln(p1); // x = 0, y = 0  
var p2 = new point(17);  
writeln(p2); // x = 17, y = 17
```


Using generics

```
record point {  
    var x;  
    var y: x.type;  
}
```

'x' is generic, can be any* type

'y' is generic, must be the same type as 'x'

```
var p1 = new point(1, 2);  
writeln(p1); // x = 1, y = 2  
var p2 = new point(3.1, 9.7);  
writeln(p2); // x = 3.1, y = 9.7
```

Using generics

```
record point {  
  type T;  
  var x: T;  
  var y: T;  
  proc init(type T) {  
    this.T = T;  
  }  
  proc init(x: ?T, y: T) {  
    this.T = T;  
    this.x = x;  
    this.y = y;  
  }  
}
```

'point' is generic over the type 'T'

'x' and 'y' both have type 'T'

I can define various constructors for initializing my type

```
var p1 = new point(int);  
writeln(p1); // x = 0, y = 0  
var p2 = new point(3.1, 9.7);  
writeln(p2); // x = 3.1, y = 9.7
```

Composition

```
record point {  
    var x;  
    var y: x.type;  
}  
record line {  
    var p1: point(?);  
    var p2: p1.type;  
}
```

'p1' is constrained to 'point'



```
var l1 = new line(new point(1, 2), new point(3, 4));  
writeln(l1);
```

```
var l2 = new line(new point(1.0, 2.2), new point(1.8, 9.9));  
writeln(l2);
```

Inheritance

```
import Math;
class Shape {
  proc area(): real { return 0.0; }
}
class Circle: Shape {
  var radius: real;
  override proc area(): real {
    return Math.pi * radius ** 2;
  }
  proc diameter() {
    return 2.0 * radius;
  }
}
var s: Shape = new Circle(5.0);
writeln(s.area());

var c = s.borrow(): Circle;
writeln(c.area(), " ", c.diameter());
```

All child classes will have an 'area()' method

Circles have a special 'area()' formula

Only Circles can use this method

's' is a Shape and can only call Shape methods

We can recover the Circle and call Circle methods

What's *borrow*?

Memory management and ownership

Classes and memory

- Recall that a class refers to data/memory somewhere else
- That memory must be managed
 - When the memory is no longer needed, it needs to be freed
 - How do we know when memory is no longer needed?
- What about unallocated memory?
 - We need space for some memory, but aren't quite ready to allocate it yet

'owned' and 'borrowed'

- By default, all classes in Chapel are 'owned'
- The compiler handles the memory
- Only one variable "owns" the memory at a time, when it goes out of scope the memory is deleted
- Ownership
 - Who owns the memory?
 - It can be transferred
 - It can be "borrowed" so the data can be accessed in multiple places without transferring ownership
 - Note: this behavior differs for module-scope vs local-scope variables
- The compiler protects you from making a mistake

'owned' and 'borrowed'

```
class MyClass { var x: int; }  
proc main() {  
  var c: owned MyClass = new /*owned*/ MyClass(1);  
  var c2 = c;  
  
  writeln(c2);  
  // writeln(c);  
}
```

'owned' is the default, but we could be explicit

Now 'c2' owns the memory and trying to use 'c' is a compiler error

'owned' and 'borrowed'

```
class MyClass { var x: int; }  
proc main() {  
  var c: owned MyClass = new MyClass(1);  
  var b: borrowed MyClass = c.borrow();  
  
  writeln(c);  
  writeln(b);  
}
```

Now 'b' is an alias to the memory

Both 'c' and 'b' may be used

'owned' and 'borrowed'

```
class MyClass { var x: int; }  
proc main() {  
  var c: owned MyClass = new MyClass(1);  
  {  
    var b: borrowed MyClass = c.borrow();  
    b.x = 2;  
  }  
  var c2 = c;  
  writeln(c2);  
}
```

'b' is a temporary alias to 'c',
but its changes are permanent

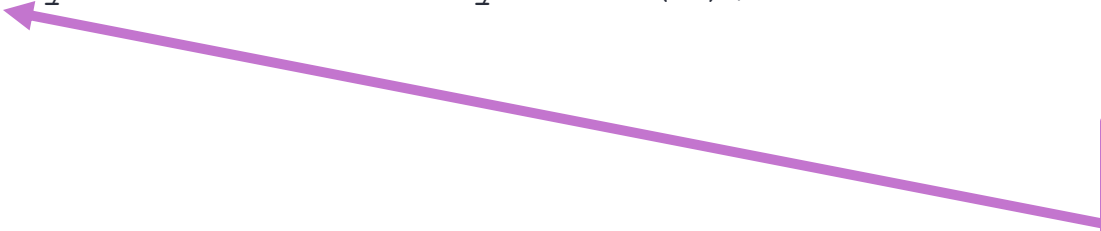
If this block was not here, the
code would not compile.
'b' cannot outlive 'c'

The changes made to 'b' are still
reflected in 'c2'

nilable vs non-nilable

- A variable of a class type must have a value
 - No invalid pointers like **NULL** in other languages
- Sometimes we need to represent the lack of a value
 - We can make a class nilable to allow it to be **nil**
- To get at the value, we need to unwrap it
 - With `!`, this will halt if the value is **nil** (with --checks)
 - With `: nonNilableType`, this will throw if the value is **nil**

```
class MyClass { var x: int; }  
proc main() {  
  var c: owned MyClass? = new MyClass(1);  
  var c2 = c;  
  
  writeln(c2);  
  writeln(c);  
}
```



'c' is nilable, so when ownership changes it is still usable, but its value is 'nil'

nilable vs non-nilable

```
proc getXUnsafe(c: borrowed MyClass?) {  
  return c!.x;  
}  
proc getX(c: borrowed MyClass?) throws {  
  return (c: borrowed MyClass).x  
}
```

'c' is nilable, so to get 'x' we have to unwrap it

If 'c' is 'nil', this will halt or crash

By using a cast, if 'c' is 'nil' we get a thrown 'Error' instead

```
proc getXUnsafe(c: borrowed?) {  
  return c!.x;  
}  
proc getX(c: borrowed?) throws {  
  return (c: borrowed).x  
}
```

We can also write the above generically

'shared'

- Like owned, but can have multiple owners of the data
 - The memory is only deleted when all instances go out of scope
 - The memory can still be borrowed without affecting ownership
- shared can have higher runtime overheads

```
class MyClass { var x: int; }  
proc main() {  
  var c1: shared MyClass = new shared MyClass(1);  
  {  
    var c2: shared MyClass = c1;  
    writeln(c2);  
  }  
  writeln(c1);  
}
```

At this point, there is 1 owner of the data

'c2' also owns the data and can use it freely

There are 2 owners

'c1' can still be used
There is only 1 owner

'unmanaged'

- Unlike 'owned'/'shared'/'borrowed', the compiler will do nothing special with this memory
- The memory will never be deleted unless it is deleted explicitly
 - Similar to raw pointers in C/C++

```
class MyClass { var x: int; }  
proc main() {  
  var c1: unmanaged MyClass = new unmanaged MyClass(1);  
  var c2 = c1;  
  writeln(c1);  
  writeln(c2);  
  
  delete c1;  
}
```

The compiler will do nothing special

I can do whatever I want with this variable

The programmer is responsible for cleaning up

Calling functions

- Calling function can make dealing with ownership difficult

```
proc takesOwnership(in x: owned) { }
```

```
proc kindaBorrows(/*ref*/ x: owned) { }
```

```
proc actuallyBorrows(x: borrowed) { }
```

Example: Linked List

The 'Node' class

```
class Node {  
  type T;  
  var data: T;  
  var next: owned Node(T)?;  
  proc init(data: ?T) {  
    this.T = T;  
    this.data = data;  
    next = nil;  
  }  
}
```

'Node' is a generic
container for data

'next' is a nilable owned 'Node(?)'

The 'LinkedList' record and 'printList' method

```
record LinkedList {  
  type T;  
  var head: owned Node(T)?;  
  proc init(type T) {  
    this.T = T;  
    head = nil;  
  }  
}  
  
proc LinkedList.printList() {  
  var current = head.borrow();  
  var sep = "";  
  while current != nil {  
    write(sep, current!.data);  
    current = current!.next.borrow();  
    sep = " -> ";  
  }  
  writeln();  
}
```

We don't want ownership of 'head',
we 'borrow()' it instead

Since current is nilable, we need to
unwrap it

The 'append' method

```
proc ref linkedList.append(data: T) {  
  if head == nil {  
    head = new Node(data);  
  } else {  
    appendHelper(head!, data);  
  }  
}  
proc appendHelper(node: borrowed, data: node.T) {  
  if node.next == nil {  
    node.next = new Node(data);  
  } else {  
    appendHelper(node.next!, data);  
  }  
}
```

The base case, just set 'head'

Use the 'appendHelper' function

Since 'node' is 'borrowed',
we get an implicit borrow

Recursively call 'appendHelper'

Exercise: Expression Evaluator

Problem

- Make this program work!

```
proc main() {  
    var syms = new symbols();  
    syms.add("x", 10);  
    syms.add("y", 5);  
  
    var expr = new BinExpr("+",  
        new BinExpr("*",  
            new Variable("x"),  
            new Number(2)),  
        new BinExpr("-",  
            new Variable("y"),  
            new Number(3)));  
  
    printExpr(expr);  
    printVal(expr, syms);  
}
```

Example output

Expression: ((x * 2) + (y - 3))
Value: 22

Starter code

```
use Map;
record symbols {
  var table: map(string, int);
  proc ref add(name: string, value: int) {
    table[name] = value;
  }
  proc get(name: string): int {
    return table[name];
  }
}
class Expr {
  proc eval(syms: symbols): int {
    halt("Not implemented");
  }
  proc stringify(): string {
    halt("Not implemented");
  }
}
```



Thank you

