Types

COMP360

Some slides are from

“Programming Language Paradigms” by Michael Scott

“It’s hardware that makes a machine fast. It's software that makes a fast machine slow.”

Craig Bruce

Bring your laptop

- On Wednesday we will be discussing Drag & Drop programming
- It would be useful to have you try some examples during the lecture
- read http://appinventor.mit.edu/explore/ai2/tutorials.html

Remaining Schedule

<table>
<tr><td>Monday, April 17
Types
read sections 7.1 & 7.2</td><td>Wednesday, April 19
Drag & drop programming</td><td>Friday, April 21
Drag & drop programming</td></tr>
<tr><td>Monday, April 24
Concurrent Programming
read chapter 13</td><td>Wednesday, April 26
Concurrent Programming</td><td>Friday, April 28
Concurrent Programming</td></tr>
<tr><td>Monday, May 1
Exam 3</td><td>Wednesday, May 3
final review</td><td></td></tr>
<tr><td colspan="2">Tuesday, May 9
Final Exam
10:30am – 12:30pm</td><td></td></tr>
</table>

Data Format

- Different types of data are stored differently
- Some operations are not allowed on some data types

What is 5 * “aardvark” ?

- Some operations are implemented differently for different data types

Data Types

- What are types good for?
  - implicit context
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Old Languages Without Types

- In the pre-Cambrian days of programming, C didn’t really care about types
- Functions returning an integer or pointer with the same size as an **int** are not required to have a declaration
- Parameters did not have to be declared

Modern Example

- Consider this C function written in a modern style

```
double new_style( int cat, double *dog);  /* prototype */

double new_style( int cat, double *dog) {
        return *dog + (double)cat;
}
```

Old C Example

- Consider this C function written in an ancient style

```
double old_style( );          /* prototype */

double new_style( cat, dog )
            int cat, double *dog
{
      return *dog + (double)cat;
}
```

Strong Typing

- **Strong Typing** has become a popular buzz-word
  - like *structured programming*
  - informally, it means that the language prevents you from applying an operation to data on which it is not appropriate
- **Static Typing** means strongly typed and that the compiler can do all the checking at compile time

Example Type Systems

- Common Lisp is strongly typed, but not statically typed
- Ada is statically typed
- Pascal is almost statically typed
- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
- C has become more strongly typed with each new version, though loopholes still remain

Common Hardware Types

- Most CPUs provide instructions to handle certain data types
    - integer
    - long
    - short
    - char
    - float
    - double
    - pointers

Software Created Types

- Arrays
- Strings
- Classes
- Records
- Sets
- Files
- Enumeration

Coercion

- When an expression of one type is used in a context where a different type is expected, one normally gets a type error
- But what about

```
int cat;  double dog, goat;
     ...
dog = cat + goat;
```

Which Java statement will work?

```
double dog, goat;      int cat;
```

A. cat = dog + goat;
B. dog = cat + goat;
C. cat = (int)dog + goat;
D. All the above
E. None of the above

Coercing Expressions

- Many languages allow things like this, and COERCE an expression to be of the proper type
- Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
- Fortran has lots of coercion, all based on operand type

ELSEVIER

Simple Coercion

- C has lots of coercion, too, but with simpler rules:
  - all **`float`**s in expressions become **`double`**s
  - **`short`**, **`int`**, and **`char`** become **`int`** in expressions
  - if necessary, precision is removed when assigning into LHS

ELSEVIER

No Static Types

- Some languages, such as JavaScript, do not statically type the variables
- A variable can hold any data given to it

```
var  rabbit;
rabbit = 7;
rabbit = "hare";
```

- Data types are checked at run time

Generics

- Generic classes are particularly valuable for creating *containers*: data abstractions that hold a collection of objects
- Generics allow type determination without casting
- Generic methods are needed in generic classes

Generics Example

- Casting is required without generics

```
ArrayList cow = new ArrayList();
String bull = (String)cow.get();

ArrayList<String> goat = new ArrayList<>();
String bull = goat.get();
```

Strings

Usual string operations include:

- Concatenation
- Substring
- Comparison
- Length
- Assignment
- *Pattern matching*

String Design Consideration

- Character arrays
- Length
  - Static
  - Dynamic
  - Limited length dynamic
- Immutable

String Implementation

- Length determination
  - Special termination character
  - Property
    - Stored separately from the string data
    - Stored with the string data
- How do you handle changes in string length?

Immutable String

- Java uses immutable strings
- You cannot change a string although you can replace one

```
String cat = “meow”;
String dog = cat;
cat = cat + “ purr”;
System.out.println( dog );      // displays meow
```

Array Considerations

- Specification of index range
- Dynamic or Static size
- Number of dimensions
- Range checking
- Initialization
- Slices

Array Organization

- Vectors are almost always stored as consecutive addresses.
- 2 dimensional arrays can be:
  - Consecutive addresses
  - Array of arrays
- 3+ dimensional arrays

2D Array Allocation

```
int a[3][2];
```

0,0	0,1
1,0	1,1
2,0	2,1

Allocation in memory

0,0	0,1	1,0	1,1	2,0	2,1

2D Array Index Calculation

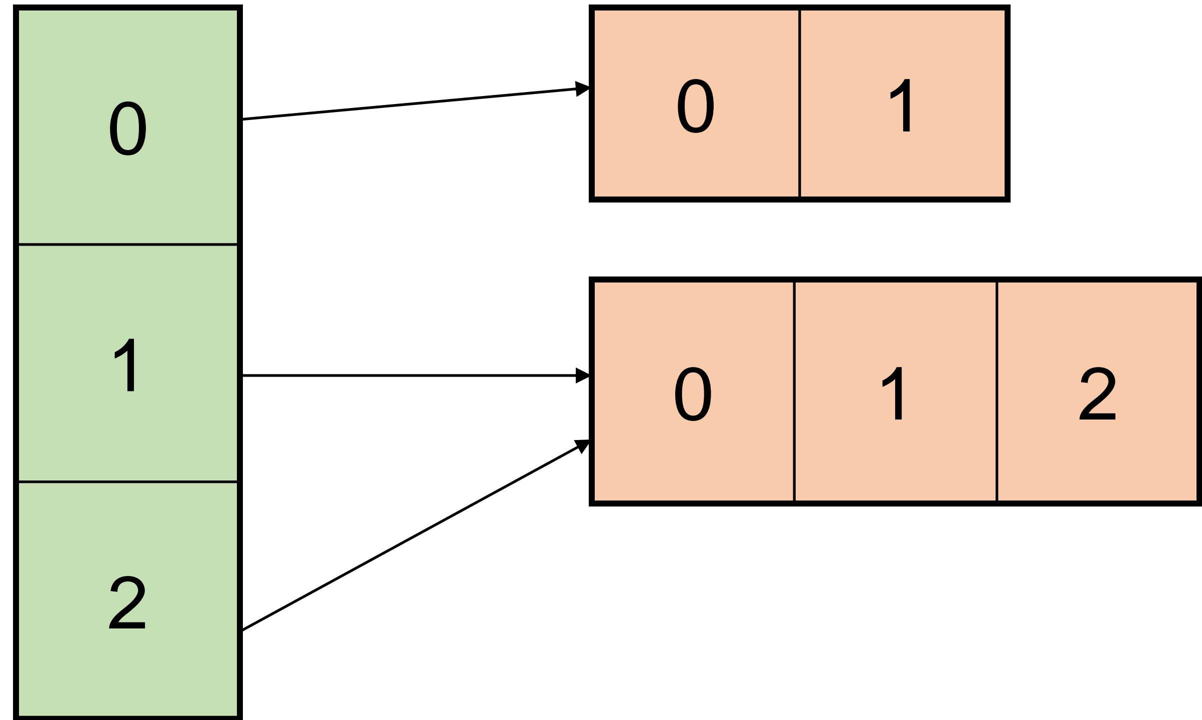
Location of element i,j =

Base address +

object size * num col * i +

object size * j;

Java Arrays of Arrays



Array Slices

- Some languages allow you to operate with an arbitrary portion of an array
- C allows you to pass a row of a 2D array to a function as a vector, but you can’t pass a column as a vector. Fortran 90 allows this

Orthogonality

- **Orthogonality** is a useful goal in the design of a language, particularly its type system
- A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

Non-Orthogonality in C

- Structures (but not arrays) may be returned from a function
- An array can be returned if it is inside a structure
- A member of a structure can be any data type (except void), or the structure of the same type
- An array element can be any data type (except void)
- Everything is passed by value (except arrays)

Type Inference

- Haskell is strongly typed
- While you can, you are not required to specify the type of variables
- Haskell determines the type of the variables based on the operations done on them

Type Checking

A type system has rules for

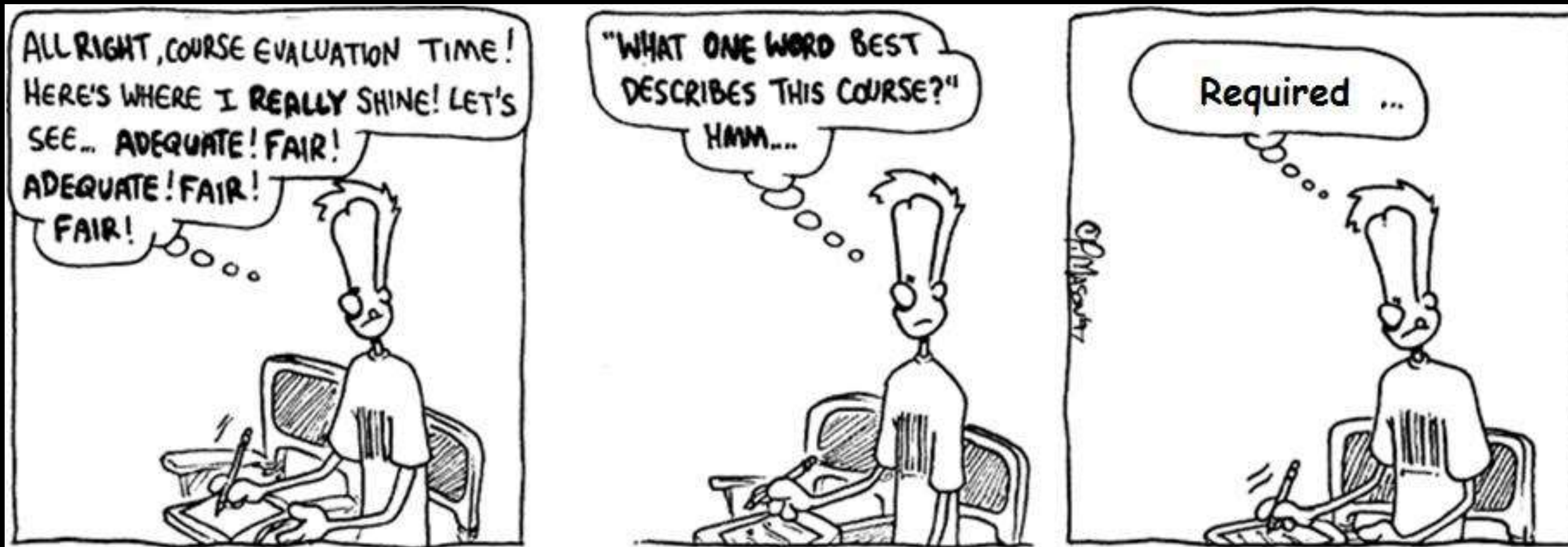
- type **equivalence** – when are the types of two values the same?
- type **compatibility** – when can a value of type A be used in a context that expects type B?
- type **inference** – what is the type of an expression, given the types of the operands?

Type compatibility / type equivalence

- Compatibility is the more useful concept, because it tells you what you can DO
- The terms are often used interchangeably

Course Evaluations

- Course evaluations are available on Blackboard
- Be sure to complete all evaluations for all classes
- Only 18% have completed the COMP360 survey



Bring your laptop

- On Wednesday we will be discussing Drag & Drop programming
- It would be useful to have you try some examples during the lecture
- read http://appinventor.mit.edu/explore/ai2/tutorials.html