Mahavir Education Trust's

Shah & Anchor Kutchhi Engineering College

Chembur, Mumbai 400 088

UG Program in Information Technology

Data Types

Manya Gidwani
Assistant Professor
IT-SAKEC

Type Systems

- What has a type?
  - things that have values:
    - constants
    - variables
    - fields
    - parameters
    - subroutines
    - Objects
  - A name (identifier) might have a type, but refer to an object of a different (compatible type):

```
double  a = 1;
Person  p = new Student("John");
```

Data Types

- What are types good for?
  - Types can give implicit meaning to operations
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - but it catches enough of them to be useful.

Type Checking

- ***Type checking*** is the process of ensuring that a program obeys the type system's type compatibility rules.

Type Systems

- STATIC TYPING means that the compiler can do all the checking at compile time:
  - types are computed and checked at compile time
  - Common examples of statically-typed languages include Java, C, C++, FORTRAN, Pascal and Scala.

  Example: int data;
  data = 50;
  data = “Hello World!”; // causes an compilation error
- DYNAMIC TYPING: A language is **dynamically-typed** if the type of a variable is checked during **run-time**. Common examples of dynamically-typed languages includes JavaScript, Objective-C, PHP, Python, Ruby, Lisp, and Tcl.

Type Systems

- **Python Example**

```
data = 10;
 data = “Hello World!”; // no error caused
```

Type Systems

- ***Strongly typed*** **:-**A language is called *strongly typed* if it prevents operations on inappropriate types. The key attribute of strongly typed languages is that variables, constants, etc can be used only in manners consistent with their types.
- Example Python

```
temp = “Hello World!”
temp = temp + 10; // program terminates with below stated error
```

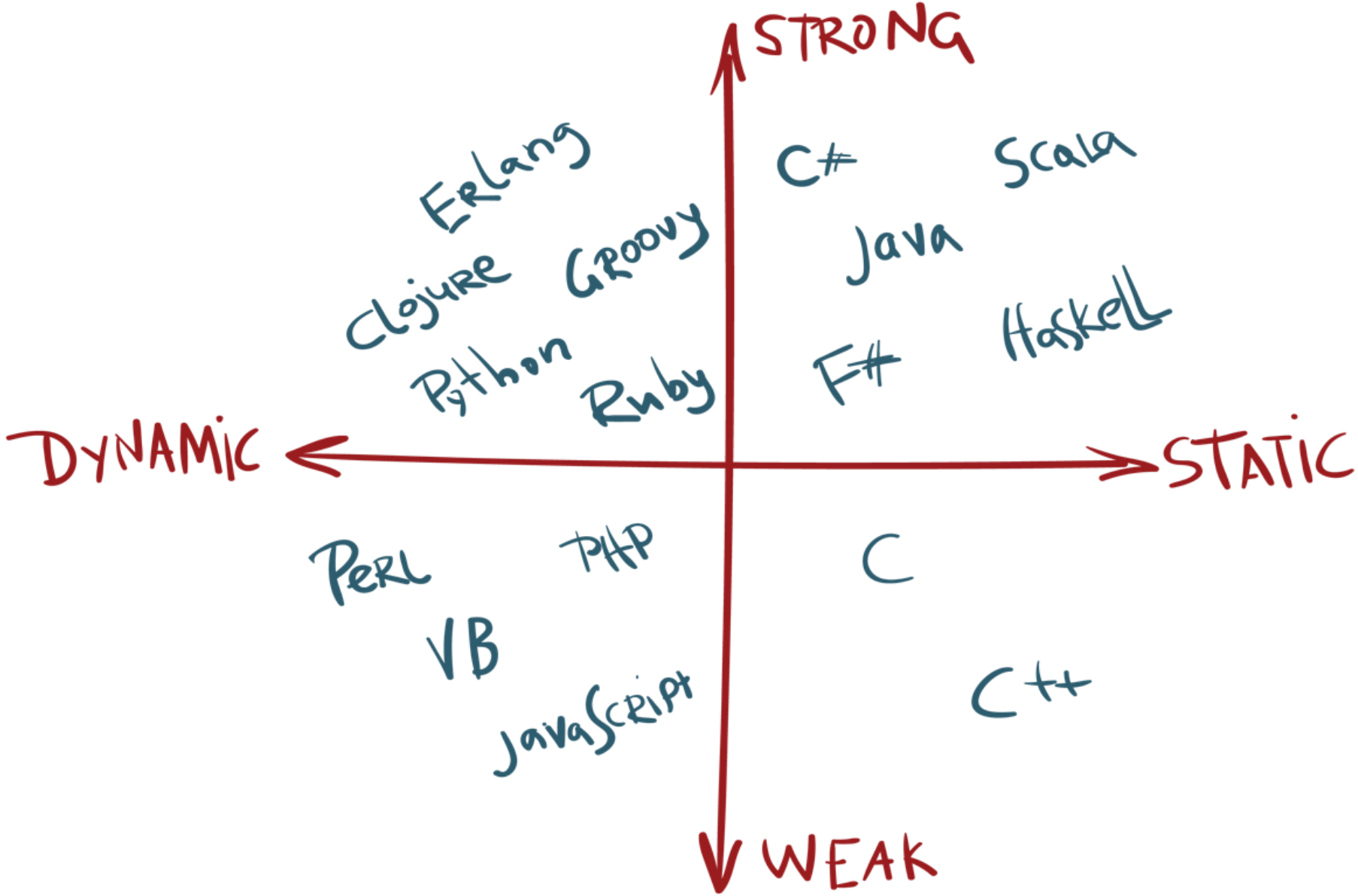
TypeError: cannot concatenate ‘str’ and ‘int’ objects

Type Systems

- **Weak Typed:-**In contrast weakly typed languages offer many ways to bypass the type system.
- A **weakly-typed** language on the other hand is a language in which variables are not bound to a specific data type; they still have a type, but type safety constraints are lower compared to strongly-typed languages.

Php example

```
$temp = “Hello World!”;
$temp = $temp + 10; // no error caused
echo $temp;
```


STRONG
WEAK
DYNAMIC
STATIC
Erlang
Clojure
Groovy
Python
Ruby
C#
Scala
Java
F#
Haskell
Perl
PHP
VB
JavaScript
C
C++

Type Systems

- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically (for instance, for dynamic binding):

```
String a = 1;          //compile-time error
int i = 10.0;          //compile-time error
Student s = (Student)(new Object());// runtime
```

- Python is strong dynamic typed:

```
a = 1;
b = "2";
a + b          run-time error
```

- Perl is weak dynamic typed:

```
$a = 1
$b = "2"
$a + $b                    no error.
```

Advantages

Static type systems have the following advantages.

- Faster execution: The run-time checks are avoided.
- Typically, better (more extensive) error checking, with errors reported sooner.
- The resulting program is easier to read and maintain.

Dynamic type systems have the following advantages.

- They are more flexible.
- Programs are easier to write.

Type Systems

- A type system consists of:
- A mechanism of defining types and associating them with language constructs
- A set of rules for
  - **Type equivalence:** When do two objects have the same type?
  - **Type compatibility:** Where can objects of a given type be used?
  - **Type synthesis/inference:** How do you determine the type of an expression from its parts or, in some cases, from its context (i.e., the type of the whole is used to determine the type of the parts).

Type Checking

- Type compatibility / type equivalence
  - Compatibility is the more useful concept, because it tells you what you can DO
  - The terms are often (incorrectly, but we do it too) used interchangeably.
  - If 2 types are equivalent, then they are pretty much automatically compatible, but compatible types do NOT need to be equivalent.

Type Checking

- Certainly format does not matter:
  struct { int a, b; }
  is the same as
  struct {
  int a, b;
  }
  We also want them to be the same as

```
struct {
    int a;
    int b;
}
```

Structural Equivalence

- But should:

```
struct {
    int a;
    int b;
}
```

- Be the same as:

```
struct {
    int b;
    int a;
}
```

- Most languages say no, but some (like ML) say yes.

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name equivalence is based on declarations
  - Structural equivalence is based on some notion of meaning behind those declarations
  - Name equivalence is more fashionable these days

- type T1 is new integer;
- type T2 is new integer;
- type T3 is record x:integer; y:integer; end record;
- type T4 is record x:integer; y:integer; end record;
- subtype S1 is integer;

Thus the four types on the right T1,...,T4 are all distinct. In structural equivalence, types are equivalent if they have the same structure so types T3 and T4 are equivalent and aggregates of those two types could be assigned to each other.

Type Casts

- Two casts: converting and non-converting
  - ***Converting cast***: changes the meaning of the object type in question
    - cast of **`double`** to **`int`** in Java
  - ***Non-converting casts***: means to **interpret** the bits as the same type

```
Person p = new Student(); // implicit non-converting
Student s = (Student)p; // explicit non-converting cast
```

- *Type coercion*: May need to perform a runtime semantic check
  - Example: Java references:

```
Object o = "...";
String s = (String) o;
        // maybe after if(o instanceOf String)...
```

Type Checking

- Make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - sometimes the word 'cast' is used for conversions (C is guilty here)

Classification of Types

- Types can be discrete (countable/finite in implementation):
  - **boolean**:
    - in C, 0 or not 0
  - **integer types:**
    - different precisions (or even multiple precision)
    - different signedness
    - Why do we define required precision? Leave it up to implementer
  - **floating point numbers:**
    - only numbers with denominators that are a power of 10 can be represented precisely
  - **decimal types:**
    - allow precise representation of decimals
    - useful for money:Visual Studio .NET:

      decimal myMoney = 300.5m;

Continue…

- **character**
  - often another way of designating an 8 or 16 or 32 bit integer
  - Ascii, Unicode (UTF-16, UTF-8), BIG-5, Shift-JIS, latin-1
- **subrange numbers**
  - Subset of a type (**for i in range(1:10)**)
  - Constraint logic programming: **X in 1..100**
- **rational types:**
  - represent ratios precisely
- **complex numbers**

Continue…

- Types can be composite :
  - **records (unions)**
  - **arrays**
    - **Strings** (most languages represent Strings like arrays)
      - list of characters: null-terminated
      - With length + get characters
  - **sets**
  - **pointers**
  - **lists**
  - **files**
  - **functions, classes**, etc.

Record Types

- A record consists of a number of fields:
  - Each has its own type:

```
struct MyStruct {
boolean ok;
int bar;
};
MyStruct foo;
```

- There is a way to access the field:

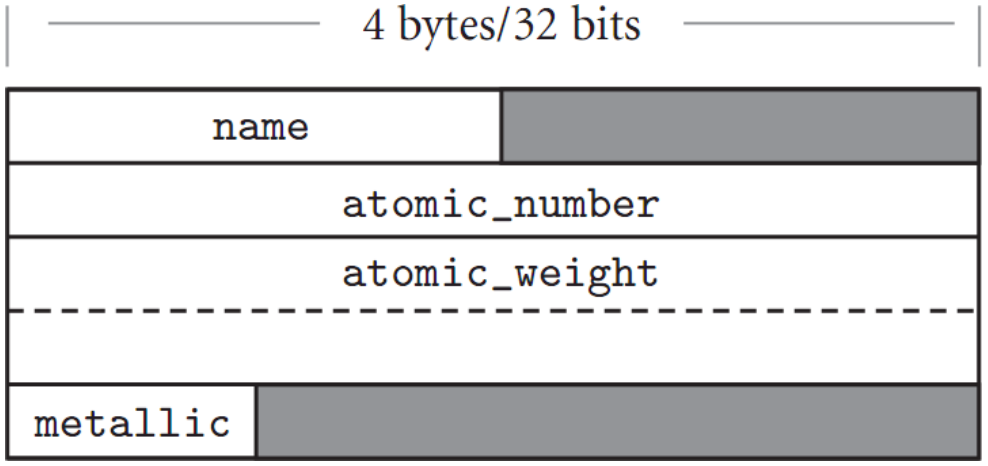
`foo.bar;` <- C, C++, Java style.
`bar of foo` <- Cobol/Algol style
`person.name` <- F-logic *path expressions*

Record Types

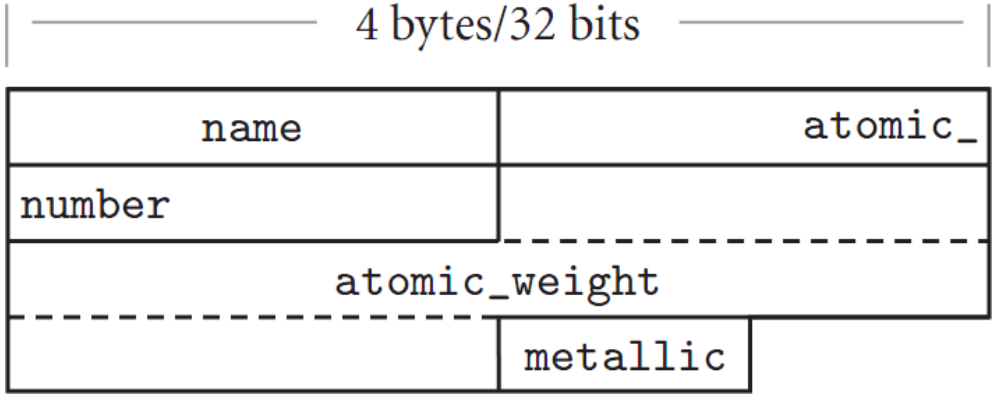
- ***Records***: multiple properties in one complex type
  - **usually laid out contiguously**
  - possible holes for alignment reasons
  - smart compilers may rearrange fields to minimize holes (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays

Record Types

- Memory layout

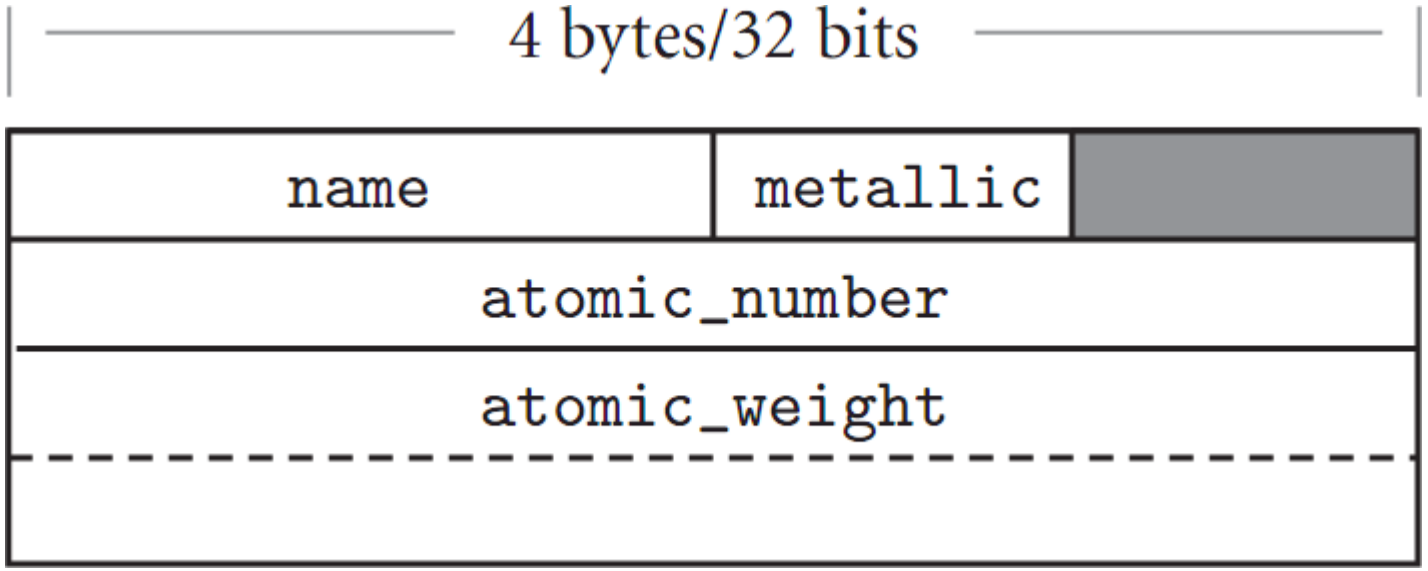


- memory layout for **packed** element records



Record Types

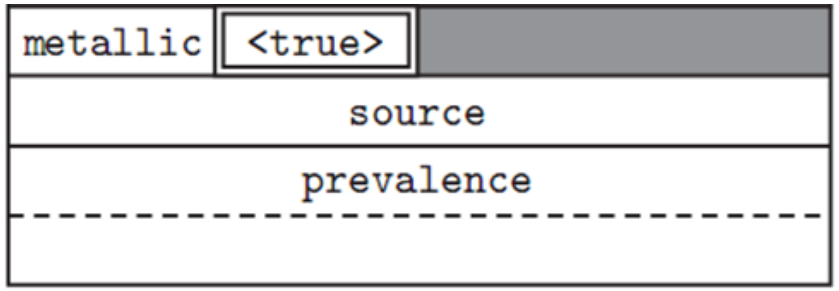
- Rearranging record fields to minimize holes and keep fields optimally addressable:

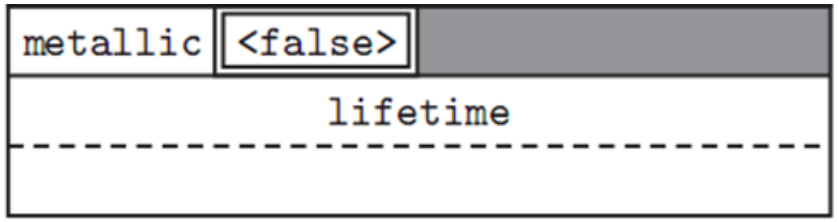


Record Types

- ***Unions*** (a.k.a., ***variant records***):
  - **overlay space**
  - cause problems for type checking
  - Lack of tags means you don't know what is there
    - Ability to change tag and then access fields hardly better: can make fields "uninitialized" when tag is changed (requires extensive run-time support)

      –
  - Memory layout for unions example:





Assignment vs Equality

- **Assignment** sets and/or re-sets the value stored in the storage location denoted by a variable name.
- **Equality** is a relational operator that tests or defines the relationship between two entities.

```
If (pig = 'y’)
Output "Pigs are good"
Else
 Output "Pigs are bad."
```

```
If (pig == 'y’)
Output "Pigs are good“
 Else
 Output "Pigs are bad."
```

Link for detailed Handouts

https://cs.nyu.edu/~gottlieb/courses/2000s/2009-10-fall/pl/lectures/lecture-06.html