CSCI: 4500/6500 Programming Languages

Types

What are Types?

- **Denotational:** Collection of values from domain
  - » integers, doubles, characters
- **Abstraction:** Collection of operations that can be applied to entities of that type
- **Structural:** Internal structure of a bunch of data, described down to the level of a small set of fundamental types
- **Implementers**: Equivalence classes

Different Definitions: Types versus Typeless

1. "Typed" or not depends on how the interpretation of data representations is determined
   - » When an operator is applied to data objects, the interpretation of their values could be determined either by the operator or by the data objects themselves
   - » Languages in which the interpretation of a data object is determined by the operator applied to it are called typeless languages;
   - » those in which the interpretation is determined by the data object itself are called typed languages.
2. Sometimes it just means that the object does not need to be explicitly declared (Visual Basic, Jscript …).

What are types good for?

- **Documentation:** Type for specific purposes documents the program
  - » Example: Person, BankAccount, Window, Dictionary
- **Safety:** Values are used consistent with their types.
  - » Limit valid set of operation
  - » Type Checking: Prevents meaningless operations (or catch enough information to be useful)
- **Efficiency:** Exploit additional information
  - » Example: a type may dictate alignment at at a multiple of 4, the compiler may be able to use more efficient machine code instructions
- **Abstraction:** Types allow programmer to think about programs at a higher level
- **Polymorphism:** when the compiler finds that it does not need to know certain things.

What is type again?

Looking at an object:

- **Syntactically compatible:** The object provides all the expected operations (type names, function signatures, interfaces) — `Type`
- **Semantically compatible:** The object's operations all behave in the expected way (state semantics, logical axioms, proofs) — `Class`

- **Type: it provides a particular interface**
- **Class: describes implementation constraints**

Weakly typed and Weak typing

- **Weakly typed:** Can successfully perform an improper operation to an object
- **Weak typing (latent typing):** Type constraints are relaxed to make programming more flexible and powerful ("no type at all typing")

  la·tent

  1. *Present or potential but not evident or active. In existence but not manifest. Latent talent.*
  2. *Psychology. Present and accessible in the unconscious mind but not consciously expressed.*

  - » Example: Duck typing: if it walks like a duck …
  - » Does not imply type safety or unsafety (later)

Weak Type & Type Safety

- Weak typing implicitly/automatically convert (or allow cast) when used.

```
var x = 5;      // (1)
var y = "hi"    // (2)
x + y ;         // (3)
```

```
var x = 5;      // (1)
var y = "4"     // (2)
x + y ;         // (3)
```

- Visual Basic (pre.net)
  - » Left yields "5hi"
  - » Right yields? 9 or "54"?
- Also program does not crash so in this respect Visual Basic appears "type-safe"

Weak Type & Type Safety

- Weak typing implicitly/automatically convert (or allow cast) when used.

```
int x   = 5;
char y[] = "hi";
char *z = x + y;
```

- z points to a memory address 5 character beyond y (pointer arithmetic).
  - » Content of that location may be outside addressable memory (dereferencing z may terminate program)
  - » Unsafe

Strongly and Weakly Typed

- Strongly typed typed languages prevents you from applying an operation on data that is inappropriate
  - » Ada, ML, Haskell
  - » Example of not strongly typed: C's cast operation: forces compiler to accept and allow the conversion at run-time even if it is in appropriate
- Weakly typed languages: can successfully perform and improper operation to an object (un-safe)
- Weak (latently) typed languages (different polymorphism) automatically convert (or casts) types when used
  - » Visual Basic (pre-.Net)

Static versus Dynamic Type Checking

- Static type checking: check constraints at compile time.
- Dynamic type checking: check constraints at run time
  - » Examples: Java, constrains the types and usage of all the symbols at compile time, but also has enough runtime support to perform limited dynamic checks. If it can't check something statically (array bounds or null pointers, for example), it checks them dynamically.

Example: Static and Dynamic Type Checking

```
var x;          // (1)
x := 5;         // (2)
x := "hi"       // (3)
```

- (1) declares the name x (2) associates integer value to x and (3) associates string value "hi" to x
  - » Most statically typed languages illegal because (2) and (3) bind x to values of inconsistent types
  - » Pure dynamically typed languages would permit it because the name x would not have to have consistent type.
    - – Dynamic languages catch errors related to misuse of values "type errors" at the time of the computation of the statement/expression.

Example Implementation

```
var x   = 5;            // (1)
var y := "hi";          // (2)
var z = x + y;          // (3)
```

- (1) binds 5 to x (2) binds "hi" to y; and (3) attempts to add x to y.
- Binding may be represented by pairs
  - » (integer, 5) and (string, "hi")
- Executing line 3 … error…

Advantages of Static Typing

- More reliable [?]
- more efficient at run time [yes!]

Implicit and Explicit Types

- Explicit (Manifest) Typing: Types appears as syntax in the program
  - » int a, char b; C++/C
- Implicit Typing: Can infer type, not part of the text of the program
  - » ML and Scheme

Type Equivalence

- Determining when the types of two values are the same

```
struct student { string name; string address;}
struct school  { string name; string address;}
```

- Are these the same?

Structural Equivalence

```
struct student { string name; string address;}
struct school  { string name; string address;}
```

- The same component, put together the same
  - » Algol, early Pascal, C (with exception) and ML
- Straightforward and easy to implement
- Definition varies from language to language
  - » Does ORDER of the fields matter
- Yes! Example is structurally equivalent

Name (Nominative) Equivalence

```
struct student { string name; string address;}
struct school  { string name; string address;}
```

- More popular recently
- A new name or definition = a new type
  - » Java, current Pascal, Ada
- Assume that: if programmer wrote two different definitions, then wanted two types
  - » Is this a good or bad assumption to make?
- Example: No! not name equivalent

Strict & Loose Name Equivalence

- Aliasing: Multiple names for the same type
  - » `typdef` in C/C++
  - » `TYPE new_type = old_type` in Modula 2
- Subtyping: explicit naming of parent

- Strict: aliases are distinct types
- Loose: aliases are equivalent types

Duck Typing

- Variable itself determines what it can do, if it implements a certain interface.
- Object is interchangeable with any object that implements the same interface, no matter whether the two have related inheritance hierarchy
  - » Templates in C++ is similar but cannot do it at run-time

if it walks like a duck, and talks like a duck, then it might as well be a duck. One can also say that the language ducks the issue of typing.

"duck the issue of typing"

Duck typing in Python

- Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object
- By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution.
- Duck-typing avoids tests using type() or isinstance(). Instead, it typically employs hasattr() tests or EAFP [Easier to Ask Forgiveness than Permission] programming.

Duck Typing: Python's File Like classes

- Classes can implement some or all of the methods of file and be used where files would normally be used.
  - » For example, GzipFile implements a file-like object for accessing gzip-compressed data.
  - » cStringIO allows treating a Python string as a file.
  - » Sockets and files share many of the same methods as well, however,
    - – sockets lack the tell() method and can't be used everywhere a GzipFile can be used.
  - » a file-like object can implement only methods it is able to, and consequently it can be only used in situations where it makes sense.
    - – Flexibility of Duck typing!

Duck typing in Java

- 2003, Dave Orme, leader of Eclipse's Visual Editor Project
  - » Needed a generic way to bind any SWT control to any JavaBeans-style object,
  - » Observed SWT reuses names across its class hierarchy.
    - – For example, to set captions set the "Text property"
      - true for an SWT Shell (window), a Text control, a Label, and many more SWT controls
  - » Realized that if he could implement data binding in terms of the methods that are implemented on a control,
    - – could improve re-use instead of implementing separate data binding for an SWT Shell, Text, Label, and so on.
  - » Created a class that makes duck typing simple and natural for Java programmers.

Language	Static Dynamic	Strong Weak	Safety	Nominative Structural
Assembly	none	weak	unsafe	structural
BASIC	static	weak	safe	nominative
C/C++	static	weak	unsafe	structural
Fortran	static	strong	safe	nominative
Java	static	strong	safe	nominative
C#	static	weak	unsafe	nominative
ML	static	strong	safe	structural
Scheme	dynamic	weak	safe	nominative
Python/Ruby	dynamic	strong	safe	duck
Perl 6	hybrid	weak	safe	nominative

Data Types

Records

- Also known as 'structs' and 'types'.
  - » C

```
struct resident {
  char initials[2];
  int ss_number;
  bool married;
};
```

- fields – the components of a record, usually referred to using dot notation.

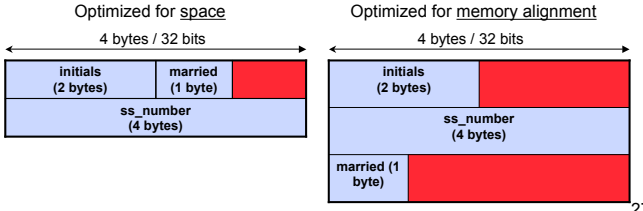
Nesting Records

- Most languages allow records to be nested within each other.
  - » Pascal

```
type two_chars = array [1..2] of char;
type married_resident = record
  initials: two_chars;
  ss_number: integer;
  incomes:
    husband_income: integer;
    wife_income: integer;
  end;
end;
```


Memory Layout of Records

- Fields are stored adjacently in memory.
- Memory is allocated for records based on the order the fields are created.
- Variables are aligned for easy reference.



Simplifying Deep Nesting

- Modifying records with deep nesting can become bothersome.

```
book[3].volume[7].issue[11].name := 'Title';
book[3].volume[7].issue[11].cost := 199;
book[3].volume[7].issue[11].in_print := TRUE;
```

- Fortunately, this problem can be simplified.
- In Pascal, keyword `with` "opens" a record.

```
with book[3].volume[7].issue[11] do
begin
      name := 'Title';
      cost := 199;
      in_print := TRUE;
end;
```


Simplifying Deep Nesting

- Modula-3 and C provide better methods for manipulation of deeply nested records.
  - » Modula-3 assigns aliases to allow multiple openings

```
with var1 = book[1].volume[6].issue[12],
     var2 = book[5].volume[2].issue[8]
DO
  var1.name = var2.name;
  var2.cost = var1.cost;
END;
```

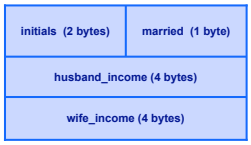
  - » C allows pointers to types
    - – What could you write in C to mimic the code above?

Variant Records

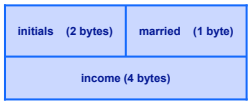
- variant records – provide two or more alternative fields.
- discriminant – the field that determines which alternative fields to use.
- Useful for when only one type of record can be valid at a given time.

Variant Records – Pascal Example

```
type resident = record

  initials: array [1..2] of char;

  case married: boolean of
    true: (
      husband_income: integer;
      wife_income: integer;
    );
    false: (
      income: real;
    );

  id_number: integer;

end;
```



Case is TRUE



Case is FALSE

Unions

- A union is like a record
  - But the different fields take up the same space within memory

```
union foo {
    int i;
    float f;
    char c[4];
}
```

- Union size is 4 bytes!

Union example (from an assembler)

```
union DisasmInst {
#ifdef BIG_ENDIAN
  struct { unsigned char a, b, c, d; } chars;
#else
  struct { unsigned char d, c, b, a; } chars;
#endif
  int intv;
  unsigned unsv;
  struct { unsigned offset:16, rt:5, rs:5, op:6; } itype;
  struct { unsigned offset:26, op:6; } jtype;
  struct { unsigned function:6, sa:5, rd:5, rt:5, rs:5, op:6; } rtype;
};
```


Arrays

- Group a homogenous type into indexed memory.
- Language differences: A(3) vs. A[3].
  - Brackets are preferred since parenthesis are typically used for functions/subroutines.
- Subscripts are usually integers, though most languages support any discrete type.

Array Dimensions

- C uses 0 -> (n-1) as the array bounds.
  - `float values[10]; // 'values' goes from 0 -> 9`
- Fortran uses 1 -> n as the array bounds.
  - `real(10) values ! 'values' goes from 1 -> 10`
- Some languages let the programmer define the array bounds.
  - `var values: array [3..12] of real;`
    `(* 'values' goes from 3 -> 12 *)`

Multidimensional Arrays

- Two ways to make multidimensional arrays
  - Both examples from Ada
  - Construct specifically as multidimensional.

```
matrix: array (1..10, 1..10) of real;
-- Reference example: matrix(7, 2)
```

    - Looks nice, but has limited functionality.
  - Construct as being an array of arrays.

```
matrix: array (1..10) of array (1..10) of real;
-- Reference example: matrix(7)(2)
```

    - Allows us to take 'slices' of data.

Array Memory Allocation

- An array's "shape" (dimensions and bounds) determines how its memory is allocated.
  - » The time at which the shape is determined also plays a role in determining allocation.
- At least 5 different cases for determining memory allocation:

Array Memory Allocation

- Global lifetime, static shape:
  - » The array's shape is known at compile time, and exists throughout the entire program.
    - – Array can be allocated in static global memory.
    - – int global_var[30]; void main() { };
- Local lifetime, static shape:
  - » The array's shape is known at compile time, but exists only as locally needed.
    - – Array is allocated in subroutine's stack frame.
    - – void main() { int local_var[30]; }

Array Memory Allocation

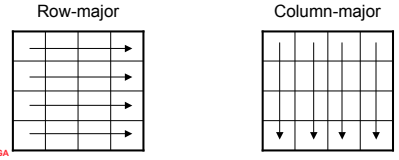
- Local lifetime, bound at elaboration time:
  - » Array's shape is not known at compile time, and exists only as locally needed.
    - – Array is allocated in subroutine's stack frame and divided into fixed-size and variable-sized parts.
    - – main() { var_ptr = new int[size]; }
- Arbitrary lifetime, bound at elaboration time:
  - » Array is just references to objects.
    - – Java does not allocate space; just makes a reference to either new or existing objects.
    - – var_ptr = new int[size];

Array Memory Allocation

- Arbitrary lifetime, dynamic shape
  - » The array may shrink or grow as a result of program execution.
    - – The array must be allocated from the heap.
    - – Increasing size usually requires allocating new memory, copying from old memory, then de-allocating the old memory.

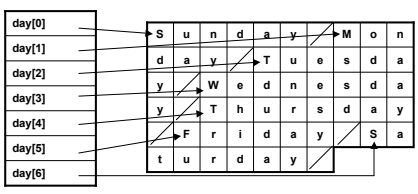
Memory Layout Options

- Ordering of array elements can be accomplished in two ways:
  - » row-major order – Elements travel across rows, then across columns.
  - » column-major order – Elements travel across columns, then across rows.



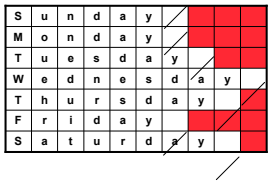
Row Pointers vs. Contiguous Allocation

- Row pointers – an array of pointers to an array. Creates a new dimension out of allocated memory.
- Avoids allocating holes in memory.



Row Pointers vs. Contiguous Allocation

- **Contiguous allocation - array where each element has a row of allocated space.**
- **This is a true multi-dimensional array.**
  - » **It is also a ragged array**



Array = 70 bytes

Array Address Calculation

Contiguous allocation arrays: can calculate sizes

Given: (L - lower limit ; U - upper limit)

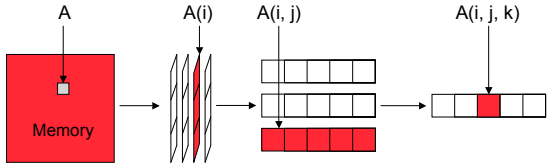
- **Calculate the size of an element (1D)**
  - » **element_size = sizeof element_type**
- **Calculate the size of a row (2D)**
  - » **row_size = element_size * ($U_{element}$ - $L_{element}$ + 1)**
- **Calculate the size of a plane (3D)**
  - » **plane_size = row_size * (U_{rows} - L_{rows} + 1)**
- **Calculate the size of a cube (4D)**

⋮

Array Address Calculation

- **Address of a 3-dimenional array A (i, j, k) is:**

address of A (in C &A)
+ ((i - L_{plane}) * size of plane)
+ ((j - L_{row}) * size of row)
+ ((k - $L_{element}$) * size of element)



Sets

- **Introduced by Pascal, found in most recent languages as well.**
- **Common implementation uses a bit vector to denote "is a member of".**
  - » **Example:**
    U = {'a', 'b', …, 'g'}
    A = {'a', 'c', 'e', 'g'} = 1010101
- **Hash tables needed for larger implementations.**
  - » **Set of integers = (2^{32} values) / 8 = 536,870,912 bytes**

Enumerations

Enumerations

- enumeration **– set of named elements**
  - » **Values are usually ordered, can compare**

```
enum weekday {sun,mon,tue,wed,thu,fri,sat}
if( myVarToday > mon ) { . . . }
```

- **Advantages**
  - » **More readable code (originally created for Pascal)**
  - » **Compiler can catch some errors**
- **Is `sun==0` and `mon==1`?**
  - » **C/C++: yes; Pascal: integers are not compatible with enumerated types!**
- **Can also choose ordering in C**

```
enum weekday {mon=28,tue=55,wed=30…}
```

Lists

Lists

- list – the empty list or a pair consisting of an object (list or atom) and another list

```
(a . (b . (c . (d . nil))))
```

- improper list – list whose final pair contains two elements, as opposed to the empty list

```
(a . (b . (c . d)))
```

- basic operations: cons, car, cdr, append
- list comprehensions (e.g. Miranda, Haskell and Python) construct for creating lists.
  - » Expression, enumeration and filter(s).
  - » list of squares of all odd numbers less than 100,
    - – | means "such that"
    - – <- means "is a member of"
  - » `[ i * i | i <- [1..100], i mod 2 = 1 ]`

List Comprehensions in Python

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
>>>
```


Recursive Types

Recursive Types

- recursive type - type whose objects may contain references to other objects of the same type
  - » Most are records (consisting of (1) reference objects and (2) other "data" objects)
  - » Used for linked data structures: lists, trees

```
struct Node
  {
  Node *left, *right;
  int data;
  }
```


Recursive Types

- In *reference model* of variables (e.g. Lisp, Java), recursive type needs no special support. Every variable is a reference anyway.
- In *value model* of variables (e.g. Pascal, C), need pointers (memory capsule model)

Refresher: Value vs. Reference

- Functional languages
  - » almost always reference model
- Imperative languages
  - » value model (e.g. C)
  - » reference model (e.g. Smalltalk)
    - – implementation approach: use actual values for *immutable* objects
  - » combination (e.g. Java)

Pointers

- pointer – a variable whose value is a reference to some object
  - » pointer use may be restricted or not
    - – only allow pointers to point to heap (e.g. Pascal)
    - – allow "address of" operator (e.g. ampersand in C)
  - » pointers not equivalent to addresses!
    - – In C it is implemented using addresses, but in some systems a pointer may contain a record, e.g. machine with segmented memory it contains
      - a segmentation ID and an offset.
  - » how reclaim heap space?
    - – explicit (programmer's duty)
    - – garbage collection (language implementation's duty)

Value Model – More on C

- Pointers and single dimensional arrays interchangeable, though space allocation at declaration different

```
int a[10]; int *b;
```

- For subroutines, pointer to array is passed, not full array
- Pointer arithmetic
  - » <Pointer, Integer> addition

```
int a[10];
int n;
n = *(a+3);
```

  - » <Pointer, Pointer> subtraction and comparison

```
int a[10];
int * x = a + 4;
int * y = a + 7;
int closer_to_front = x < y;
```


Dangling References

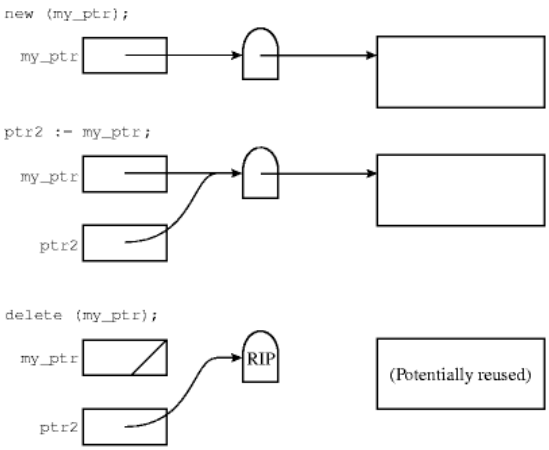
- dangling reference – a live pointer that no longer points to a valid object
  - » to heap object: in explicit reclamation, programmer reclaims an object to which a pointer still refers: UGH!
  - » to stack object: subroutine returns while some pointer in wider scope still refers to local object of subroutine: OUCH!
- How do we prevent them?

Dangling References

- Prevent pointer from pointing to objects with shorter lifetimes than the pointer itself
  - » (e.g. Algol 68, Ada 95). Difficult to enforce (pass pointers into arguments of functions.
- Tombstones (next slide)
- Locks and Keys

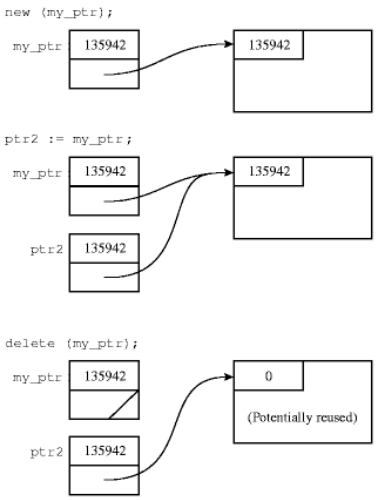
Tombstones

- Idea
  - » Introduce another level of indirection: pointer contain the address of the tombstone; tombstone contains address of object
  - » When object is reclaimed, mark tombstone (zeroed)
- Time overheads
  - » Create tombstone
  - » Check validity of every access (could make it outside program space and therefore create an interrupt).
  - » Double indirection
- Space overheads
  - » when to reclaim??
- Extra benefits
  - » easy to compact heap (can change addresses to objects under the hood).
  - » works for heap and stack



Locks and Keys

- Idea
  - » Every pointer is `<address, key>` tuple
  - » Every object has a lock
  - » A pointer to an object is valid if the key in the pointer matches the lock in the object.
  - » When object is reclaimed, object's lock marked (zeroed)
- Advantages
  - » No need to keep tombstones around
- Disadvantages
  - » Objects need special key field (usually implemented only for heap objects)
  - » Probabilistic protection
- Time overheads
  - » Lock to key comparison costly



Garbage Collection

- Language implementation notices when objects are no longer useful and reclaims them automatically
  - » essential for functional languages
  - » trend for imperative languages
- When is object no longer useful?
  - » Reference counts
  - » Mark and sweep
  - » "Conservative" collection

Reference Counts

- Idea
  - » Counter in each object that tracks number of pointers that refer to object
  - » Recursively decrement counts for objects and reclaim those objects with count of zero
- Must identify every pointer
  - » in every object (instance of type)
  - » in every stack frame (instance of method)
- Type descriptor
  - » Most implemented as a table that lists offsets within the type where pointers can be found

Mark-and-Sweep

- Idea

  … when space low
  1. Mark every block temporarily as "useless"
  2. Beginning with pointers outside the heap, recursively explore all linked data structures and mark each traversed as useful
  3. Return still marked blocks to freelist
- Must identify pointers
  - » must know begin and end of blocks
  - » find pointers within blocks

Garbage Collection Comparison

- **Reference Count**
  - » **Will never reclaim circular data structures**
  - » **Must record counts**
- **Mark-and-Sweep**
  - » **Lower overhead during regular operation**
  - » **Bursts of activity (when collection performed, usually when space is low)**