Chapter 7
Data Types

What is a data type?

• A set of values

versus

• A set of values + set of operations on those values

Why data types?

• Data abstraction
  – Programming style
  – Modifiability – enhance readability

• Type checking (semantic analysis) can be done at compile time -
  Type checking is the process a translator goes through to determine
  whether the type information is consistent

• Compiler uses type information to allocate space for variables
• Translation efficiency: Type conversion (coercion) can be done at
  compile time
Overview cont...

• Variable Declarations
  – Explicit type information
• Type declarations
  – Give new names to types in type declaration
• Type checking
  – *Type Inference Rules* for determining the types of constructs from available type information
  – *Type equivalence* determines if two types are the same
• Type system
  – Type construction methods + type inference rules + type equivalence algorithm

Simple Types

• Predefined types (float, boolean, int, char)
• Enumerated types

**Haskell**

```haskell
data Color = Red | Green | Blue | Indigo | Violet
  deriving (Show,Eq,Ord)
```

**Pascal**

```pascal
type fruit = (apple, orange, banana);
```

**C/C++**

```c
enum fruit { apple, orange, banana };
```
Simple Types (cont)

• Subrange types

**Pascal**

```pascal
  type byte = 0..255;
  minors = 0..19;
  teens = 13..19;
```

**ADA**

```ada
  subtype teens is INTEGER range 13..19;
```

*Staying within range may or may not be enforced.*

---

Data Aggregates and Type Constructors

• Aggregate (compound) objects and types are constructed from simple types
• Recursive – can also construct aggregate objects and types from aggregate types
  ```
  Tree = Nil | Node int Tree Tree
  ```
• Predefined – records, arrays, strings……
Data Types

- **Strong type checking**
  - language prevents you from applying an operation to data on which it is not appropriate
- **Weak type checking** – if you have type mismatch, a coercion is applied
- **No type checking** – A type mismatch is the user’s problem
  - it just doesn’t work properly.

Error reporting is our friend.

- **Static Typing** means that the compiler can do all the checking at compile time

Type Systems

- **Examples**
  - Ruby is strongly typed, but not statically typed
  - C++ is weakly typed – due to more kinds of implicit conversion
  - Haskell is purely statically typed (stronger than C++) – no implicit conversions
  - Pascal is almost statically typed (variant records)
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically (explicit casts can be illegal)
  - Assembly language – no type checking. Up to programmer to ensure appropriate types

Copyright © 2009 Elsevier
Type Systems

• Common terms:
  – discrete (ordinal) types – countable
    • integer
    • boolean
    • char
    • enumeration
    • subrange
  – Scalar types – single valued (float, int, char, string, boolean)
    • discrete
    • real

Type Systems

• Composite types:
  – records (structs/classes)
  – arrays
    • strings
  – sets
  – lists
  – files
Type Systems

• 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

In typing,
  – arrays/structs/files can contain any type.
  – parameters/return values can be any type
  – operations could be applied to any type (that made sense)

Type Systems

• For example
  – C++ is not completely orthogonal because
    • a function cannot return an array
    • an array cannot be passed by value

• Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about
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 Equivalence

• Two major approaches: structural equivalence and name equivalence
  – Name equivalence is based on declarations. At compilation, know two things are the same type because the base type has a name that is used when you want entities to be same type.
  – Structural equivalence is based on some notion of meaning behind those declarations
  – Name equivalence is more fashionable these days
Structural Equivalence

- Two types are the same if they have the same structure i.e. they are constructed in exactly the same way using the same type constructors from the same simple types.
- May look alike even when we wanted them to be treated as different.

- type Student = record {name: string; age: int;}
- type School = record {name: string; age: int;}
- type Food = record {name: string; calories: int;}
- type Score = record {rank: int; sport: string;}

- Clearly Student and School are the same type (using structural equivalence), but other cases depend on language.

Structural Equivalence

- Consider…

```plaintext
Type array1 = array[-1..9] of integer;
array2 = array[0..10] of integer;
equivalence depends on language
```

- Dynamic arrays

```plaintext
Array (INTEGER range <>) of INTEGER
```
Name Equivalence

- Two name types are equivalent only if they have the exact same type name

```plaintext
typedef int Atype[10];
typedef Atype Btype;
typedef int age;
```

- Name equivalence in Ada and C (strict)
- ar1 and ar2 are not considered name equivalent

```plaintext
type ar1 is array (INTEGER range1..10) of INTEGER;
type ar2 is new ar1;
type age is new INTEGER;
```

Name equivalence...

```plaintext
v1: ar1;
v2: ar1;
v3: ar2;
```

```plaintext
v4: array (INTEGER range 1..100) of INTEGER;
v5: array (INTEGER range 1..100) of INTEGER;
v4 and v4 are not name equivalent.
Name equivalence is usually an easy test by the compiler.
```

```plaintext
v6,v7: array (INTEGER range 1..100) of INTEGER;
v6 and v7 ARE name equivalent (even though the type is unnamed)
```
Loose Name Equivalence

- Lead back to the same original structure declaration via a series of redeclarations

```plaintext
type CREDITS = int;
type REQUIREMENTS = CREDITS;
type UPPERDIV = REQUIREMENTS;

hours: CREDITS;
missing: UPPERDIV;
scholarship: REQUIREMENTS;
age: integer;
All variables here are the same “loose” type.

type SCHED = array [1..10] of integer;
AMTS= array [1..10] of integer;
These are different types.
```

Type Checking

- Coercion
  - When an expression of one type is used in a context where a different type is expected, one gets a type error unless implicit casts are used.

```plaintext
var a : integer; b, c : real;
...
c := a + b;
```
Type Checking

• Coercion
  – 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

Type Checking

• C has lots of coercion - simpler rules:
  – all floats in expressions become doubles
  – short int and char become int in expressions
  – if necessary, precision is removed when assigning into left hand side (can assign a float to an int)
Type Checking

- Coercion rules are a relaxation of type checking
  - probably a bad idea
  - Languages such as Modula-2 and Ada do not permit coercions
  - C++, generous with them
  - They're one of the hardest parts of the language to understand

Type Checking

- Make sure you understand the difference between
  - type conversions (or casts) (explicit)
  - type coercions (implicit)
  - nonconverting type casts (reinterpret bits a different way) Ex. allocate large block as bytes, but want to look at portions as integer.
Records (Structures, Classes)

- Records
  - usually laid out contiguously
  - possible gaps for alignment reasons
  - smart compilers may re-arrange fields to minimize gaps (C compilers promise not to)
  - implementation problems are caused by records containing dynamic arrays

Records (Structures)

- Memory layout and its impact (structures)

![Diagram](image)

*Figure 7.1 Likely layout in memory for objects of type element on a 32-bit machine. Alignment restrictions lead to the shaded "holes."*
Figure 7.2 Likely memory layout for packed element records. The atomic_number and atomic_weight fields are nonaligned, and can only be read or written (on most machines) via multi-instruction sequences.

Figure 7.3 Rearranging record fields to minimize holes. By sorting fields according to the size of their alignment constraint, a compiler can minimize the space devoted to holes, while keeping the fields aligned.
Records (Structures) and Variants (Unions)

- Unions (variant records)
  - overlay space
  - cause problems for type checking
- Lack of tag 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)
  - can require assignment of entire variant, as in Ada

Modula 2 variant record

Person = RECORD
  lastname, firstname : Name;
  birthdate : Date;
  isMale : BOOLEAN;
CASE status : Classification   -- called the tag
  student: idnumber :Int;   year : Int;
  | faculty: position : Rank; pay : REAL
  | staff: occupation : Job;
END; (* case *)
  isMarried : BOOLEAN;
END; (* of the record Person *)
C++ Union

```c
union Mix {
  char c;
  int i;
  float f;
} any;
```

defines three elements, each with a different type (all sharing the same space):

```c
any.c  any.i  any.f
```

---

**Longest is used.**

- Memory layout and its impact (unions)

<table>
<thead>
<tr>
<th>4 bytes/32 bits</th>
<th>4 bytes/32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>atomic_number</td>
<td>atomic_number</td>
</tr>
<tr>
<td>atomic_weight</td>
<td>atomic_weight</td>
</tr>
<tr>
<td>metallic&lt;true&gt;</td>
<td>metallic&lt;false&gt;</td>
</tr>
<tr>
<td>source</td>
<td></td>
</tr>
<tr>
<td>prevalence</td>
<td>lifetime</td>
</tr>
</tbody>
</table>

*Figure 7.15 (CD)* Likely memory layouts for element variants. The value of the naturally occurring field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type string_ptr is assumed to be represented by a (four-byte) pointer to dynamically allocated storage.
Arrays

- Arrays are the most common and important composite data types.
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous.
- Semantically, they can be thought of as a mapping from an index type to a component or element type.
- A slice or section is a rectangular portion of an array (See figure 7.4).

How do we do address calculation?

- What if we pass an array as an argument?
Arrays Slices

Figure 7.4 Array slices (sections) in Fortran90. Much like the values in the header of an enumeration-controlled loop (Section 6.5.1), a: b: c in a subscript indicates positions a, a+c, a+2c, ... through b. If a or b is omitted, the corresponding bound of the array is assumed. If c is omitted, 1 is assumed. It is even possible to use negative values of c in order to select positions in reverse order. The slashes in the second subscript of the lower right example delimit an explicit list of positions.

Copyright © 2009 Elsevier

Arrays

• Dimensions, Bounds, and Allocation
  – *global lifetime, static shape* — If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory
  – *local lifetime, static shape* — If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine’s stack frame at run time.
  – *local lifetime, shape bound at elaboration time (point at which declaration is first seen at entrance to scope)*
Arrays

• Contiguous elements (see Figure 7.7)
  – column major - in Fortran, Matlab, R
  – row major - used by everybody else
    • makes array [a..b, c..d] the same as array [a..b] of array [c..d]

Dope vector stores info about array

Figure 7.7 Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from $A[0,0]$ to $A[9,9]$, then in the row-major case elements $A[0,4]$ through $A[0,7]$ share a cache line; in the column-major case elements $A[4,0]$ through $A[7,0]$ share a cache line.

Arrays

- Two layout strategies for arrays (Figure 7.8):
  - Contiguous elements (row major or column major)
  - Row pointers

- Row pointers
  - an option in C
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication in accessing formulas
  - nice for matrices whose rows are of different lengths (ragged arrays)
    - e.g. an array of strings
      - requires extra space for the pointers and extra time to locate
Arrays

```c
char days[10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};

... days[2][3] = 's'; /* in Tuesday */
```

```c
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};

... days[2][3] = 's'; /* in Tuesday */
```

Figure 7.8 Contiguous array allocation v. row pointers in C. The declaration on the left is a true two-dimensional array. The slashed boxes are NULL bytes; the shaded areas are holes. The declaration on the right is a ragged array of pointers to arrays of characters. In both cases, we have omitted bounds in the declaration that can be deduced from the size of the initializer (aggregate). Both data structures permit individual characters to be accessed using double subscripts, but the memory layout (and corresponding address arithmetic) is quite different.

Copyright © 2009 Elsevier

---

Compile-Time Descriptors (aka Dope Vectors) - Why do we need?

<table>
<thead>
<tr>
<th>Array</th>
<th>Element type</th>
<th>Index type</th>
<th>Number of dimensions</th>
<th>Index range 1</th>
<th>...</th>
<th>Index range n</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-dimensioned array</td>
<td>Element type</td>
<td>Index type</td>
<td>Number of dimensions</td>
<td>Index range 1</td>
<td>...</td>
<td>Index range n</td>
<td>Address</td>
</tr>
<tr>
<td>Multi-dimensional array</td>
<td>Element type</td>
<td>Index type</td>
<td>Number of dimensions</td>
<td>Index range 1</td>
<td>...</td>
<td>Index range n</td>
<td>Address</td>
</tr>
</tbody>
</table>
Locating an Element

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td>1</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>a1</td>
<td>b1</td>
</tr>
<tr>
<td>4</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
<td>f1</td>
<td>g1</td>
<td>h1</td>
<td>i1</td>
</tr>
</tbody>
</table>

Logical view

Physical View

Accessing Formulas – 1D

Address(A[i]) = StartAddress + (i-lb)*size
= StartAddress - lb*size + i*size
= VirtualOrigin +i*size

lb: lower bound
size: number of bytes for one element
Virtual origin allows us to do some of the math once, so don’t have to repeat each time.
You must check for valid subscript before you use this formula, as obviously, it doesn’t care what subscript you use.
Accessing Formulas Multiple Dimensions

- $ub_i$: upper bound in $i^{th}$ dimension
- $lb_i$: lower bound in $i^{th}$ dimension
- $\text{length}_i = ub_i - lb_i + 1$

In row-major order

$$\text{Address}(A[i,j]) = \text{StartAddress} + \text{size}((i-lb_i)\times\text{length}_j + j-lb_j) = \text{StartAddress} + \text{size}\times i \times \text{length}_j - \text{size}\times lb_i + \text{size}\times j - \text{size}\times lb_j$$

= VirtualOrigin + $i\times\text{mult}_i + j\times\text{mult}_j$

**Virtual Origin** – as is where array would begin if $i$ and $j$ were zero (the origin)

For Example: array of floats $A[0..6, 3..7]$ beginning at location 100

StartAddress = 100

- $i$ = 4 (if floats take 4 bytes)
- $lb_i = 0$  $ub_i = 6$  $\text{length}_i = 7$
- $lb_j = 3$  $ub_j = 7$  $\text{length}_j = 5$

VO = 100 + 4*(-3)*5 = 40

$\text{mult}_i = 28$

$\text{mult}_j = 20$

For each dimension

<table>
<thead>
<tr>
<th>$\text{VO}$</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lb_i$</td>
<td>0</td>
</tr>
<tr>
<td>$ub_i$</td>
<td>6</td>
</tr>
<tr>
<td>$\text{mult}_i$</td>
<td>28</td>
</tr>
<tr>
<td>$lb_j$</td>
<td>3</td>
</tr>
<tr>
<td>$ub_j$</td>
<td>7</td>
</tr>
<tr>
<td>$\text{mult}_j$</td>
<td>20</td>
</tr>
</tbody>
</table>

repeated for each dimension
Accessing Formulas
Multiple Dimensions

- In column-major order:
  \[ \text{Address}(A[i,j]) = \text{StartAddress} + \text{size}((i-lbi) + (j-lbj)*\text{length}_i) \]

- In 3D in row major:
  \[ \text{Addr}(A[I,j,k]) = \text{StartAddress} + \text{size}((i-lbi)*\text{length}_j*\text{length}_k) + (j-lbj)\text{length}_k + k-lbk) \]

Strings

- Strings are really just arrays of characters
- They are often a special-case, to give them flexibility (like dynamic sizing or polymorphism to act like a scalar) that is not available for arrays in general
Sets

- Lot of possible implementations
  - Bitsets are what usually get built into programming languages
  - Intersection, union, membership can be implemented efficiently with bitwise logical instructions
  - Some languages place limits on the sizes of sets to make it easier for the implementor
    - There is really no excuse for this

Pointers And Recursive Types

- Pointers serve two purposes:
  - Efficient (and sometimes intuitive) access to objects
  - Dynamic creation of linked data structures, in conjunction with a heap storage manager
- Several languages (e.g. Pascal) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model
Pointers And Recursive Types

• C pointers and arrays
  
  \[ \text{int } *a == \text{int } a[] \]
  \[ \text{int } **a == \text{int } *a[] \]

• BUT equivalences don't always hold
  – Specifically, a declaration allocates an array if it specifies a size for the first dimension
  – otherwise it allocates a pointer
  
  \[ \text{int } **a, \text{int } *a[] \text{pointer to pointer to int} \]
  \[ \text{int } *a[n], \text{n-element array of row pointers} \]
  \[ \text{int } a[n][m], \text{2-d array} \]

Pointers And Recursive Types

• Problems with dangling pointers are due to
  – explicit deallocation of heap objects
    • only in languages that have explicit deallocation
  – implicit deallocation of elaborated objects

• Two implementation mechanisms to catch dangling pointers
  – Tombstones
  – Locks and Keys
Garbage Collection

- What is garbage and how can we deal with it?
- Garbage collection schemes
  - Reference Counting
  - Mark and Sweep
  - Stop and Copy

How Java Reclaims Objects Memory

- Java does not provide the programmer any means to destroy objects explicitly
- The advantages are
  - No *dangling reference* problem in Java
  - Easier programming
  - No *memory leak* problem
What is Garbage?

Garbage: unreferenced objects

Student ali= new Student();
Student khalid= new Student();
ali=khalid;

Now ali Object becomes a garbage,
It is unreferenced Object

What is Garbage Collection?

• What is Garbage Collection?
  – Finding garbage and reclaiming memory allocated to it.

• Why Garbage Collection?
  – the heap space occupied by an un-referenced object can be recycled and made available for subsequent new objects

• When is the Garbage Collection process invoked?
  – When the total memory allocated to a Java program exceeds some threshold.

• Is a running program affected by garbage collection?
  – Yes, the program suspends during garbage collection.
Disadvantages of Garbage Collection

- Garbage collection adds an overhead that can affect program performance.
- GC requires extra memory.
- Programmers have less control over the scheduling of CPU time.

Reference Counting Garbage Collection

- Main Idea: Add a reference count field for every object. Keep updated all the time.

- This Field is updated when the number of references to an object changes.

Example

Object p = new Integer(57);
Object q = p;
Reference Counting (cont'd)

- The update of reference field when we have a reference assignment (i.e., \( p=q \)) can be implemented (by system not programmer) as follows:

```java
if (p!=q)
{
    if (p!=null)
        --p.refCount;
    p=q;
    if (p!=null)
        ++p.refCount;
}
```

Example:

```java
Object p = new Integer(57);
Object q = new Integer(99);
p=q
```

Reference Counting (cont'd)

- Must be able to identify the location of every pointer.
- Reference counting will fail whenever the data structure contains a cycle of references and the cycle is not reachable from a global or local reference.
Reference Counting (cont'd)

- **Advantages**
  - Conceptually simple: Garbage is easily identified
  - It is easy to implement.
  - Immediate reclamation of storage
  - Objects are not moved in memory during garbage collection.

- **Disadvantages**
  - Reference counting does not detect garbage with cyclic references.
  - The overhead of incrementing and decrementing the reference count each time.
  - Extra space: A count field is needed in each object.
  - It may increase heap fragmentation.

Mark-and-Sweep Garbage Collection

*Happens periodically (not continually)*

- The mark-and-sweep algorithm is divided into three phases:
  - **Clear phase** – mark every block as useless. Each object has an extra bit: the mark bit – initially the mark bit is 0.
  - **Mark phase**: the garbage collector traverses the graph of references from the *root nodes* and marks each heap object it encounters, Mark bit is set to 1 for the *reachable objects* in the mark phase.
  - **Sweep phase**: the GC scans the heap looking for objects with mark bit 0 – these objects have not been visited in the mark phase – they are garbage. Any such object is added to the free list of objects that can be reallocated. The objects with a mark bit 1 have their mark bit reset to 0.
Mark and Sweep (cont'd)

• Advantages
  – It is able to reclaim garbage that contains cyclic references.
  – There is no overhead in storing and manipulating reference count fields.
  – Objects are not moved during GC – no need to update the references to objects.

• Disadvantages
  – It may increase heap fragmentation.
  – It does work proportional to the size of the entire heap.
  – The program must be halted while garbage collection is being performed.

Pointers And Recursive Types

• Mark-and-sweep
  – commonplace in Lisp dialects
  – complicated in languages with rich type structure, but possible if language is strongly typed
  – achieved successfully in Cedar, Ada, Java, Modula-3, ML
  – complete solution impossible in languages that are not strongly typed
  – conservative approximation possible in almost any language
Stop-and-Copy Garbage Collection

- The heap is divided into two regions: Active and Inactive.

- Objects are allocated from the active region only.

- When all the space in the active region has been exhausted, program execution is stopped and the heap is traversed recursively. Live objects are copied to the other region as they are encountered by the traversal. Pointers are updated (by looking into mapping).

- The role of the two regions is reversed, i.e., swap (active, inactive). …

Stop-and-Copy Garbage Collection (cont'd)

- A graphical depiction of a garbage-collected heap that uses a stop and copy algorithm. This figure shows nine snapshots of the heap over time:
Stop-and-Copy Garbage Collection (cont'd)

• Advantages
  – Only one pass through the data is required.
  – It de-fragments the heap – as gaps are squeezed out
  – It does work proportional to the amount of live objects and not to the memory size.
  – It is able to reclaim garbage that contains cyclic references.
  – There is no overhead in storing and manipulating reference count fields.

Stop-and-Copy Garbage Collection (cont'd)

• Disadvantages
  – Twice as much memory is needed for a given amount of heap space.
  – Objects are moved in memory during garbage collection (i.e., references need to be updated)
  – The program must be halted while garbage collection is being performed.