Data types

Data types

A data type consists of

- A set of values
- A set of predefined operations on those values

Approaches to data typing

Include only a minimal set of data types (early Basic, Fortran IV)

Include a rich set of built-in types and operators (PL/1)

Include a few built-in types along with type constructors that allow programmers to define their own types (Pascal, C, Java, ...)

Include facilities for defining abstract data types (Java, C++, Ada 95)

Abstract data types

All types provided by high-level languages are abstract data types

Abstract data types involve:

- Separation of the Interface of a type …
  - What is visible to the user

- … and the representation and set of operations on values of that type
  - Hidden from the user

Why types, and so many?

Types define what you can and cannot do with each data type

- Allows the compiler to look for semantic errors
- Ensures consistency of the interfaces when working with multiple modules

Set rules for equivalence and compatibility

Allows for efficiency within a category of a type

- `byte` instead of `int`, `float` instead of `double`

Two categories of data types

Programming language data type are divided into two (or three) basic categories

Primitive type

Structured type

Types that don’t fit nicely into either category

Primitive types

Size in memory is fixed by the language (or implementation)

- Early primitive types closely followed hardware configuration
- Still true today

Primitive types are not defined in terms of other types

Primitive types include numeric types, characters, and booleans

Structured types

Defined in terms of other types

Size can often vary

Layout (organization in memory) can be interesting

Structured types include arrays, records, and classes

Integers

Most programming languages support several sizes of integers

Some languages include support of signed and unsigned type

Integer types are generally directly supported by the hardware.

Two's compliments is the standard for implementing integers

- One's compliment is still in use

Floating-point

Floating-point types model real numbers

- Values after the decimal point may be approximations

Most languages include two floating point types

- Generally 4 byte and 8 byte implementations
- `float` and `double` in Java

Most modern machines use IEEE Floating point standard 754

- `float` has a precision of 23 bits and a range of 8 bits
  - Range is ± 1.4012984632481707e-45 to 3.40282346638528860e+38
- `double` has a precision of 52 bits and a range of 11 bits
  - Range is ± 4.9406565841246544e-324d to 1.79769313486231570e+308d

Decimal

Many computers are designed to support decimal data types

Primary data type for business data processing

Decimal data types store a fixed number of decimal digits, with the decimal point at a fixed position in the value

Stores values after the decimal point with complete accuracy

Binary Coded Decimal (BCD) data types use one byte per digit

- It is possible to store two base-10 digits in a single byte
- Requires work in software on machines that do not support BCD

Booleans

Early languages didn't include a Boolean data type

Early C used integers for Booleans

- Anything nonzero was considered `true`
- Zero was considered `false`

Generally stored in a byte

Only need a single bit, but it is usually not efficient to work with individual bits

Characters

A means of representing a character in a computer

- Braille and Morse Code are examples of character encoding

Early computers generally used **ASCII**

- Not all ASCII representations were the same
- Lots of Extended ASCII's

EBCDIC was an early IBM's character encoding scheme

- Character set was similar to ASCII
- Evolved from BCD, a six-bit scheme

Unicode is prevalent today

- 1 to 4 byte options – generally uses 2 bytes per character
- First 128 characters are the same as ASCII

Strings

Strings are usually a constructed type

Common string operations include:

- Concatenation
- Comparisons
- Pattern matching
- Assignment
- Copying (is this the same as assignment?)
  - Shallow vs. deep copies

String length

Do strings have a dynamic or fixed length?

Fixed length strings (Java and C#)

- Works well for immutable strings
- Easy to implement, but inflexible
- Each string operation must create and return a new string

Dynamic length strings

- C strings (`char` arrays) will set a maximum length
- Requires more care regarding string length
  - C++ introduced the `string` class to avoid using `char` arrays from C
- How does the program know the length of the string?

Ordinal types

An **ordinal** type is any type whose values can easily be mapped to the set of positive integers.

Typical examples include: `int`, `char`, and `bool`

User-defined ordinal types include enumerations and subranges

Enumerated types

An enumerated type is a type in which ALL possible values are listed in the declaration

```
enum stoplight {red, yellow, green};
```

The vales are called enumeration constants.

The program can:

- Declare variables of type `stoplight`
- Assign any of the constants `red`, `green`, or `yellow` to the variable
- Compare variables of type `stoplight` to these constants

Enumerated types

Enumerated type can make programs easier to understand than if integers are used to encode possible types

Issues:

- Can enumeration constants and variables be treated as integers?
  - Yes in C++, no in Ada and C#
- Can the same enumerations constant be a member of multiple enumeration types
  - Yes in Ada, no in C++

Java enumerated types

In Java enumerated types:

- are type safe (i.e. can not be compared with or coerced to integers)
- are subclasses of the system class `Enum`
- can have fields, constructors and methods
- the possible values of the enumeration are the only instances of the class
- all possible values can be fetched (as an array) using the static `values` method

Subranges

A subrange is a contiguous subsequence of an ordinal type

Ada:
```
subtype SmallInts is Integer range 0..255;
```

Pascal:
```
type DayInMonth = 1..31;
var currentDay : DayInMonth;
```

Subranges:

- typically inherit the operations of the parent type
- require range checking code to be executed at ***runtime*** to ensure that values assigned to subrange variables are in the legal range
- are useful for array indices and control variables in for loops

Advantages of subrange types:

- help readability by making restrictions on possible values for a variable explicit
- help reliability by detecting violations of such restrictions

Arrays

Arrays:

- contain homogeneous collections of elements - all elements have the same type
- are indexed by integers, enumerations or subranges
- are laid out with elements in consecutive memory locations of the same size

Allocating memory for an array consists of:

- choosing a base address (location for the first element)
- reserving sufficient following locations for the rest of the array elements.

Array issues

Design issues for arrays:

- What types can be used for subscripts?
- Are subscripts range checked?
- When are subscript ranges bound?
- When does the array allocation occur?
- Does the language support true multidimensional arrays, ragged arrays, or both ?
- Can arrays be initialized when created?

Arrays

An array is associated with two data types

- Type of the index
- Type of it's elements

Index issues

- Should the low index always start at 0 or can a lower bound be specified?
- What data types can be used as indices?
  - integer types, subranges, characters, enumerated types, any ordinal type?
- Specifying multiply dimensions
  - Separately or together
- Brackets or parenthesis around the indices?
  - Ada wants array access to look syntactically similar to a function call

Different types of arrays

Static

- Allocation and layout are static
  - The array is in the same location throughout the program – no allocation or deallocation
- Global and static array variables in C++ declared with a fixed size

Stack-dynamic

- Allocation is dynamic (at declaration elaboration time)
- The size cannot be altered during the variables lifetime
- Local array variables in C++
- Fixed static-dynamic array will have a fixed size

Array types

Fixed head-dynamic

- Like stack-dynamic arrays, except that storage is allocated on the heap
- Array variables in C++ allocated using new, or all Java arrays

Heap Dynamic

- Allocation and layout is dynamic
- The size can change dynamically
- Perl arrays and hashes; C and C++ array that are resized using `realloc`

Rectangular Arrays

A rectangular array is a multidimensional array in which all rows have the same length, all columns have the same number of elements and so on.

For example, a C++ array declared as:

```
int matrix[5][10];
```

is a rectangular array.

Jagged or ragged arrays

A jagged array (or ragged array) is a multidimensional array in which the length of rows (columns, etc.) need not all be the same.

For example, a Java array constructed as follows:

```
int [][] intArray = new int[3][];
intArray[0] = new int[3];
intArray[1] = new int[7];
intArray[2] = new int[5];
```

is a jagged array. Jagged arrays are possible when multidimensional arrays are implemented as ***arrays of arrays***.

Array Layout

The compiler must generate code for each array element reference to compute the address of the referenced element

Computation of the array elements addresses:

- Can be done relative to the base address after layout (determining size)
- Can be done absolutely after allocation (determining location)

Determining element location

To determine the location of an array element the compiler must know:

- The base, or starting address, of the array
- The size of each element in the array
- The "low" index of the array

The location of an element `x` in an array is

```
(base – low * size) + x * size
```

computed once at allocation time

computed for each reference at runtime

For 0-based arrays, as in Java, C, or C++, `low=0`, so the formula becomes `base + x * size`

Multi-dimensional arrays

Multidimensional array elements must be “mapped” into 1-dimensional memory.

Logically, a 2-dimensional array such as:

```
var A: array [low1 .. high1][ low2 .. high2] of basetype
```

is laid out as:

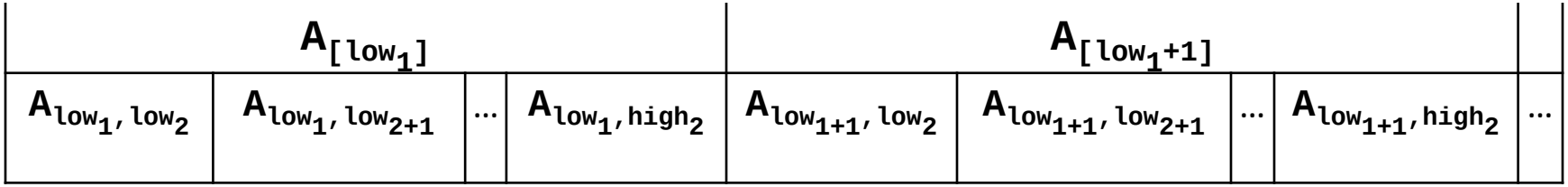
A_{low_1, low_2}	$A_{low_1, low_{2+1}}$	…	$A_{low_1, high_2}$
A_{low_{1+1}, low_2}	$A_{low_{1+1}, low_{2+1}}$	…	$A_{low_{1+1}, high_2}$
…	…	…	…
A_{high_1, low_2}	$A_{high_1, low_{2+1}}$	…	$A_{high_1, high_2}$

Major Layout

- There are two common layouts for multi-dimension arrays
  - Row major layout
  - Column major layout

Row Major Layout

Under row major layout, the rows are laid out “side by side” in memory. For the previous example, this becomes:



Note that the second index varies faster

A[low1] can be viewed as a single dimensional array and can be treated as such in C, C++ and Java

Computing row major addresses

Given:

- w_2 = the size of the base type
- w_1 = the "width" of a row
  = `((high2 – low2) + 1)` * w_2
  (note that this can be computed at layout time)
- `base`= the base address

The address of `A[i][j]` (for rectangular arrays) is:

`base + (i -` low_1`) *` w_1 + `(j -` low_2`) *` w_2

skipping rows — skipping columns

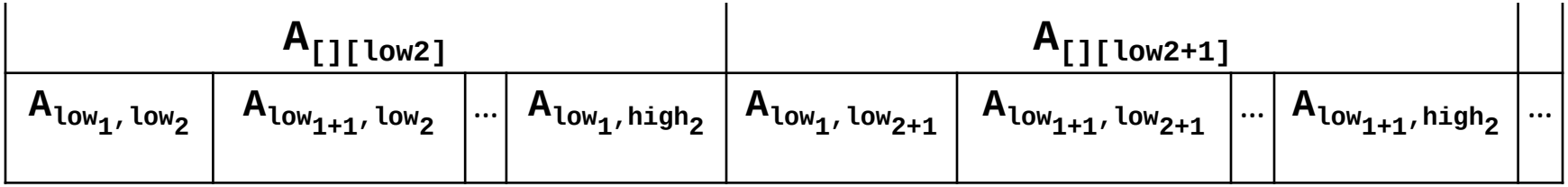
which is usually expressed as:

`(base -` low_1 * w_1 - low_2 * w_2`) + i * w1 + j *` w_2

computed at layout time — computed for each reference

Column Major Layout

Under column major layout, the columns are laid out “side by side” in memory. For the previous example, this becomes:



Note that the first index varies faster

A[][low2] cannot be viewed as a single dimensional array

Implications of Array Layout

Layout is usually transparent to the programmer, but it can matter:

- when doing pointer arithmetic
- to minimize paging when accessing all elements of an array
- when passing subarrays as parameters (slices)
- when passing multidimensional arrays as parameters. For example, in C and C++ a formal parameter of a multidimensional array type must include all dimensions ***except*** the first because all dimensions except the first are used in address calculations for row-major layout

```
void someArrayOperation(double array[][5][10]) {
   // some references to elements of array here
}
```

Implications of Jagged/Ragged Arrays

A jagged array is stored as an array of pointers or references to other arrays.

An array reference such as `intArray[2][3]` (for a 2 dimensional jagged array `intArray`) is resolved as follows:

- treating `intArray` as a single dimensional array and finding the pointer or reference stored at index 2. (Note that each element of `rArray` is of a fixed size since it is a reference or pointer.)
- following that pointer or reference to find the “referenced” array
- finding index 3 within that array

Associative Arrays

An associative array is an unordered collection of data, where each element is indexed by some key

The keys can be nonordinal types such as strings, reals and instances of classes

Associative arrays:

- are like “lookup” tables
- are often implemented as hash tables
- are efficient for accessing individual elements, but not for processing all elements systematically
- Found in languages such as:
  - Perl (hashes)
  - Java (maps)

Structured, or Composite Types

Types constructed from simpler types

- Can be defined by the user
- Basis for abstract data types

Many name for composite types

- C/C++: struct
- Pascal: record
- Python: tuples
- OOP: classes (from the data point of view)

Recursive types

- Composite types that are (partially) defined in terms of themselves
    - Lists, trees, …

Composite types

Records/structs are nonhomogeneous collections of data

- Data elements called fields
- Similar to (but simpler than) class instances
- all instance variables are public
- no methods

The layout is static

- all fields are usually put in in contiguous memory to make allocation and addressing easier
- each field is a fixed offset from the base address
  - these offsets are computed at compile time
- The size of a record is the sum of the sizes of all fields plus any required padding (rare)

Unions

Unions can store a collection of different types in the same memory location

```
union {
   int i;
   double d;
} u;
```

u is a variable that can hold either an `int` or a `double`, but (logically) not both at the same time

- `u.d` will return a `double` value,
- `u.i` will return an `int` value

Unions can be used in `structs` or nested in order `unions`

Unions are primarily used when mutually exclusive data of different types must be stored in the same collection

- The size of a union is the size of the largest field in the union
- This uses less space than allocating separate storage for each alternative

Pointers

Pointers contain memory addresses

Pointers can contain a NULL value

Pointers are generally typed, such that they know where the data is in memory AND what type of data is at that location

Pointers are typically implemented in 2, 4 or 8 byte memory cells

Pointers are commonly used to create reference parameters

- efficiency in parameter passing
- indirect addressing/access

Dangling pointers

Some language only allow pointers to refer to items that are heap-dynamic

Other languages (for example, C/C++) allow for pointers that do not have to refer to heap-dynamic variables.

The address-of operator (&) returns a pointer

A pointer is ***dangling*** if it refers to memory that isn't allocated - or has been deallocation

- Before memory is allocation
- After memory is allocation
- When a pointer into an array is incremented past the end of the array (C, C++)
- When a function returns a point to a nonstatic local variable

Dereferencing a dangling pointer is dangerous and unpredictable

Garbage

Garbage is memory that has been allocated but is now inaccessible

Garbage is created whenever all references to an allocated heap-dynamic variable are lost

```
int *p1, *p2;
p1 = new int;
p2 = new int;
p1 = p2;
```

The reference to the memory original pointed to by p1 is lost

Programs that create garbage have a ***memory leak***

References

Reference types are generally restricted forms of pointer types.

They have the same representation as pointers but do not support the full range of pointer operations

- Usually safer to use than pointers.

In C++, reference types are special pointer types that:

- are equivalent to constant pointers
- are always dereferenced when used
- are often used for passing parameters by reference

Because references are constants:

- they must be initialized with an (implicit) address when they are declared
- the value of the reference itself can not be changed

Aliases

A reference usually create an ***alias***

An alias occurs when a second (or more) variable refers to the location of some other variable

```
int i = 3;
int &r = i;
// makes r a reference to i (a constant pointer to
// i's memory location)
r = 4;
std::cout << r << std::endl; // prints 4
std::cout << i << std::endl; // prints 4
```

References allow functions to modify parameters without using pointer syntax

Java references

In Java, references:

- can contain null
- are allocated using new
- can only refer to instances of classes. In fact, a variable of a class type is always a reference.
- are implicitly dereferenced by the . operator

This design has a number of implications:

- references can (and do) completely replace pointers, because Java references can be used to build dynamic data structures
- references can be used to create aliases (as in C++) and garbage (different from C++)

Java references

Pointer arithmetic is not legal

- No references to non-class types – doesn't make sense

Instances of classes are always passed to method by reference

Primitive types are always passed by value

Java does not have a deallocation operator

- When an object becomes garbage the runtime system needs to find and deallocate the object
- No dangling references
- The programmer doesn't have to worry about when it is safe to deallocate an object

Heap management

The heap is usually maintained as a linked list of memory locations – called the free list

- When the program begins there is only one free cell – the entire heap
- Over time the available memory cells end up being of all different sizes

Allocation

- The free list is search for a memory cell large enough to meet the requested size
- Any remaining memory from that cell is added back to the free list

Deallocation

- The free list can be searched for free memory adjacent to the deallocated cell, merging the two locations into one larger location
- The memory can be added to the free list
  - Eventually the free list contains a large list of small sizes
  - Merging of adjacent locations must happen eventually

Deallocation

Explicit – The programmer must explicitly call and command, such as delete, to deallocate head storage

- C++ and Pascal

Implicit – controlled by the runtime system of the programming language

- Java and Python
- Objects can be identified for garbage collection using ***reference counting (eager approach)*** or ***mark-scan (lazy approach)***
  - Reference counting involves keeping track of the number of references to an object – the object can be deallocated when the count drops to 0
  - Mark-scan involves identifying used objects in the heap by scanning the runtime stack for references and marking objects used directly (or indirectly) – unmarked objects can be deallocated