SIENAcollege
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2019

Topic Notes: Data Types

A *data type* is a collection of values and a set of predefined operations on those values. It is an *abstraction*, as the details of the representation are hidden from the user. Some are hidden by the architecture, some can be hidden by the programming language.

A *descriptor* is the collection of the attributes of a variable. We will elaborate on this idea later. These attributes are sometimes needed only at compile time (if all attributes are static) or may need to be maintained at run time.

An *object*, as you surely know, represents an instance of a user-defined (abstract data) type.

A programming language must address a fundamental design issue for all data types: What operations are defined and how are they specified?

Primitive Data Types

Primitive data types are provided by most languages. These are not defined in terms of other data types.

They are often exactly the kinds of data that the hardware supports, or very similar. This makes operations on these (potentially) very efficient.

Integers

This is certainly true of integer data types supported by programming languages: the types supported frequently correspond directly to the sizes of the chunks of bits that the underlying hardware is designed to operate on.

In some languages, like C, the size of various integer types can vary from implementation to implementation.

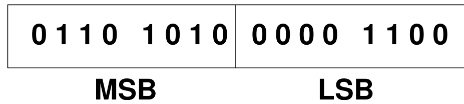
Java defines 4 sizes: `byte`: 8 bits, `short`: 16 bits, `int`: 32 bits, and `long` 64 bits. This is part of the Java language specification.

A 32-bit int has 2^{32} = 4.3 billion possible values, while a 64-bit int has $2^{64} = 1.84 \times 10^{19}$ possible values.

Bit and byte significance is also an important consideration. It is very important from an architecture perspective but can play a role in programming languages as well.

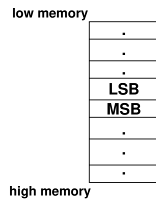
When placing the bits within a byte, they are almost always arranged with the *most significant bit* (msb) on the left, *least significant bit* (lsb) on the right. This nearly never comes into play with a programming language. it is almost entirely the concern of the underlying hardware.

We follow the same idea for stringing together bytes to make up words, longwords, etc.

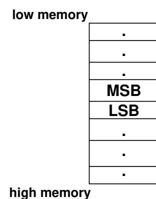


The *endianness* refers to the the order in which we store these bytes in memory.

- *little endian* (x86)



- *big endian* (Sun Sparc, 68K, PPC, IP "network byte order")



An architecture can use either ordering internally, so long as it is consistent. However, endianness becomes important when we think about exchanging data among machines (networks). *Network byte ordering* (big endian) is required of networked data for consistency when exchanging data among machines that may use different internal representations.

The MIPS architecture is bi-endian. It can process data with either big or little endianness.

See Example:
`/home/cs340/examples/show_bytes`

Integer values may be treated as signed or unsigned. For an n-bit integer, the unsigned representation can store values 0 through $2^n - 1$.

Signed representations require must have a way to represent negative numbers.

It turns out there are a number of reasonable or at least seemingly-reasonable options.

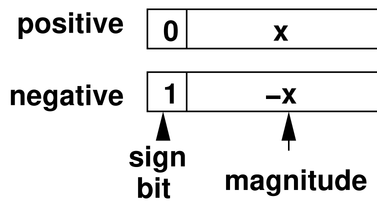
- Signed Magnitude

  The simplest way is to take one of our bits (usually the highest-order bit) and use it to indicate the sign: a 0 means positive, and a 1 means negative.

  With n bits, we can now represent numbers from $-(2^{n-1} - 1)$ to $(2^{n-1} - 1)$

  Positive numbers just use the unsigned representation.

  Negative numbers use a 1 in the *sign bit* then store the magnitude of the value in the rest.

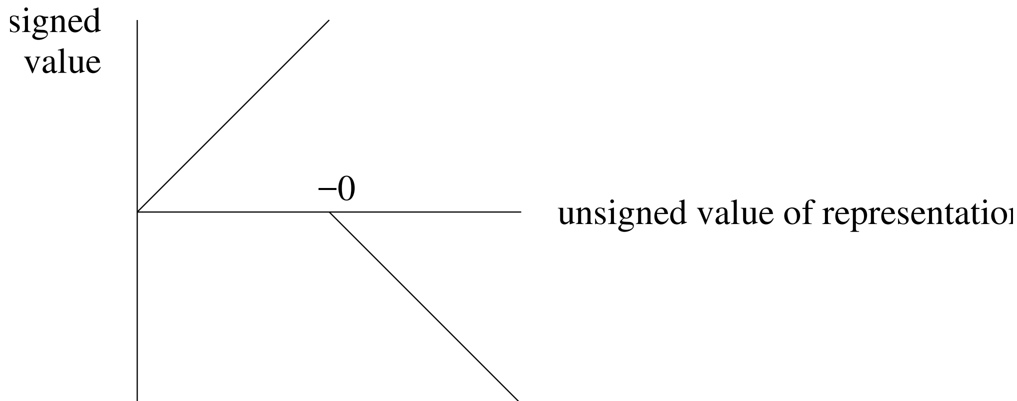


This idea is very straightforward, and makes some sense in practice:

- To negate a value, you just switch its sign bit.
- To check if a value is negative, just look at the one bit.

One potential concern: we have two zeroes! +0 and -0 are distinct values. This can complicate matters.

Another property of signed binary representations that we will want to consider is how how the signed values fall, as a function of their unsigned representations.



So we do have a disadvantage: a direct comparison of two values differs between signed and unsigned values with the same representation. In fact, all negative numbers look to be "bigger than" all positive numbers. Plus, the ordering of negatives is reverse of the ordering of positives. This might complicate hardware that would need to deal with these values.

- Excess N

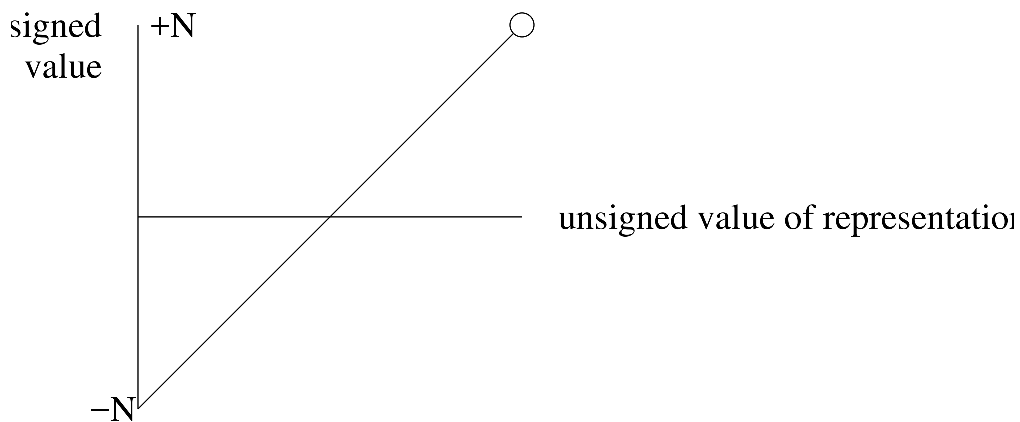
  Here, a value x is represented by the non-negative value $x + N$.

  With 4-bit numbers, it would make sense to use Excess 8, so we have about the same number of negative and positive representable values.

```
1000 = 0 (0 is not the all 0's pattern!)
0111 = -1
0000 = -8
1111 = 7
```

  So we can represent a range of values is -8 to 7.

  This eliminates the -0 problem, plus a direct comparison works the same as it would for unsigned representations.



Excess N representations are used in some circumstances, but are fairly rare.

- 1's complement

  For non-negative x, we just use the unsigned representation of x.

  For negative x, use the *bit-wise complement* (flip each bit) of $-x$.

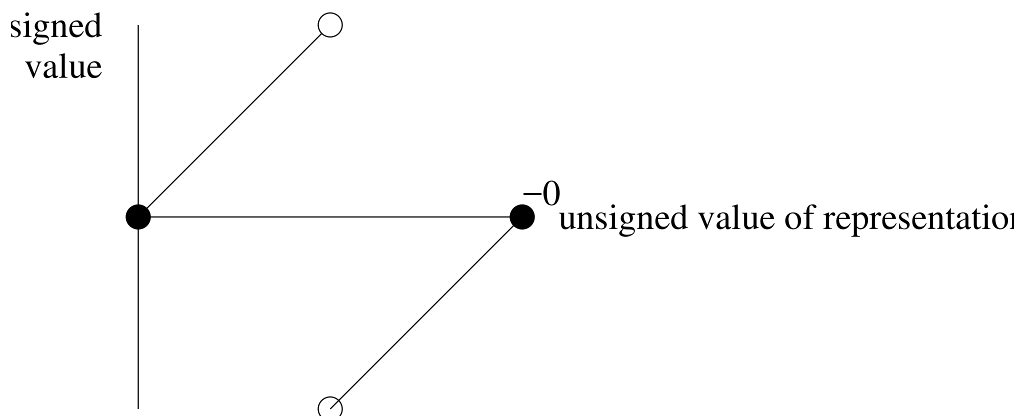
  Programming tip: the $\sim$ operator will do a bitwise complement in C and Java.

  Examples:

  $$
  \begin{aligned}
  0 &= \texttt{0000} \\
  -1 &= \overline{\texttt{0001}} = \texttt{1110} \\
  -0 &= \overline{\texttt{0000}} = \texttt{1111} \\
  -7 &= \overline{\texttt{0111}} = \texttt{1000}
  \end{aligned}
  $$

  Problems:

  - we have a -0.
  - we can compare within a sign, but otherwise need to check sign.



Range: -7 to +7.

Like Excess N, 1's complement is used in practice, but only in specific situations.

- 2's complement

  For non-negative x, use the unsigned representation of x.

  For negative x, use the complement of $-x$, then add 1 (that seems weird..).

$$0 = \texttt{0000}$$

$$-0 = \overline{\texttt{0000}}\texttt{+1} = \texttt{1111+1} = \texttt{0000}$$

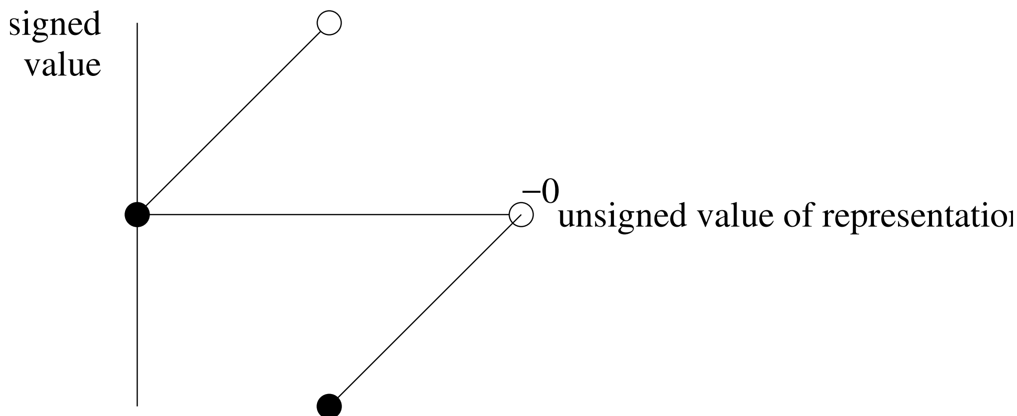
Now, that's useful. 0 and -0 have the same representation, so there's really no -0.

$$1 = \texttt{0001}$$

$$-1 = \overline{\texttt{0001}}\texttt{+1} = \texttt{1110+1} = \texttt{1111}$$

Also, very useful. We can quickly recognize -1 as it's the value with all 1 bits no matter how many bits are in our representation.

Another useful feature: 1's bit still determines odd/even (not true with 1's complement)



Like 1's complement, we can compare numbers with the same sign directly, otherwise we have to check the sign.

Given these convenient properties, 2's complement representations are the standard and default unless there's some specific situation that calls for another representation.

Historical note: Fortran had an `IF` statement:

```
IF (I) GOTO 10,20,30
```

which performed the equivalent of the following (in a more familiar C/Java syntax):

```
if (i < 0) goto 10;
if (i == 0) goto 20;
goto 30;
```

It is easy to check these cases and peform the jump quickly with a 2's complement representation of `I`.

The 4-bit 2's Complement numbers will become very familiar:

```
0000 = 0   1000 = -8
0001 = 1   1001 = -7
0010 = 2   1010 = -6
0011 = 3   1011 = -5
0100 = 4   1100 = -4
0101 = 5   1101 = -3
0110 = 6   1110 = -2
0111 = 7   1111 = -1
```

Notice that the negation operation works both ways: if you take the 2's complement of a number then take the 2's complement again, you get the original number back.

2's complement also allows us to use the same circuits that would add/subtract unsigned values and will produce a correct 2's complement result (subject to overflow restrictions).

Floating-point numbers

Most languages provide floating-point data types which correspond to the floating point representations in hardware. `float` and `double` or something equivalent are usually provided, at a minimum.

Let's think about the way we represent these things in our "normal" base-10 world.

$$3.5, \frac{2}{3}, 1.7 \times 10^{14}$$

We can use decimal notation, fractions, scientific notation.

Fractions seem unlikely as our binary representation, but we can use the decimal notation. More precisely, instead of a decimal point, we have a *radix* point.

$11.1 = 2+1+\frac{1}{2} = 3.5$, $0.11 = \frac{1}{2}+\frac{1}{4} = \frac{3}{4}$

Just like we can't represent some fractions in decimal notation, we can't represent some fractions in binary notation either.

Remember $\frac{1}{3} = .\overline{3}$

Consider: $.\overline{10}$

What value is this? $\frac{1}{2} + \frac{1}{8} + \frac{1}{32} + ...$

$$\begin{aligned} x &= .10\overline{10} \\ \frac{1}{2}x &= .01\overline{01} \\ x + \frac{1}{2}x &= .\overline{1} = 1 \\ x &= \frac{2}{3} \end{aligned}$$

How about $.\overline{1100}$?

$$\begin{aligned} x &= .\overline{1100} \\ \frac{1}{4}x &= .\overline{0011} \\ x + \frac{1}{4}x &= 1 \\ x &= \frac{4}{5} \end{aligned}$$

How can we denote an arbitrary fractional value, say, $\frac{1}{5}$?

We can follow this procedure:

1. Multiply by 2, write integer part.
2. Keep fractional part, repeat until 0, or a repeating pattern emerges.

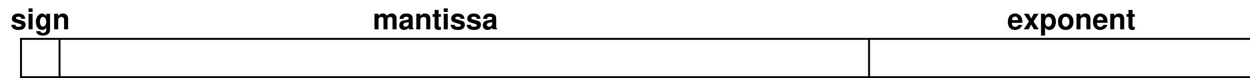
So $\frac{1}{5} = .001100110011...$

When representing these in the computer, we have lots of decisions to make, such as how we place the radix point, etc. We want to store a wide range of values, but we're limited to 2^n unique values in any n-bit representation.

Scientific notation helps us here. Consider some examples:

$$\begin{aligned} .0001011 &= 1.011 \times 2^{-4} \\ .1 &= 1. \times 2^{-1} \\ 1.1 &= 1.1 \times 2^{0} \\ -101 &= -1.01 \times 2^{2} \\ 1111 &= 1.111 \times 2^{3} \end{aligned}$$

Floating point = integer part + mantissa $\times$ 2^{exponent}



If we use binary version of scientific notation, we note that all numbers (other than 0) have a leading 1. So we need not store it! This is known as the *phantom 1* bit.

The *mantissa* is fractional, with the most significant bit representing the $\frac{1}{2}$'s bit, the next the $\frac{1}{4}$'s bit, etc.

The *exponent* is stored in excess notation (which is helpful for hardware that must align fractional values before addition or subtraction).

What about 0? It would be nice if that was the all-0's value. However, 00000000000000000000000000000000 really would represent something like 1.0×2^{-127}.

Trying to store something smaller than that value would result in a *floating point underflow*.

There are many standards, which can be hard to implement. These will include several useful and unusual values, such as $+\infty$, $-\infty$, `NaN` (not a number), etc.

Most modern computers use the IEEE Floating-Point Standard 754 format.

- Single-precision (32 bits)
  - 1 sign bit, 8 bit exponent with bias, 23 bit mantissa
- Double-precision (64 bits)
  - 1 sign bit, 11 bit exponent with bias, 52 bit mantissa

The exponent *bias* indciates than an excess format is used. The smallest possible exponent (-126 for single precision) is represented as a binary 1. Thus, we add 127 to the number we wish to represent. The special exponent 0 is reserved for *subnormal numbers*.

For example, consider the single-precision IEEE 754 representation of the number 51.625. The straight binary representation:

$$110011.101$$

We normalize this to

$$1.10011101 \times 2^5$$

So we need a sign bit of 0, an exponent of 5, represented as 132 in 8 bits: 10000100, and the mantissa of 10011101 (remember the phantom 1!).

Complex Data

Some languages support a complex type, including C99, Fortran, and Python.

Each value consists of two floats, the real part and the imaginary part.

We saw that Scheme could use complex values anywhere. In C, we need to do a bit of extra work, but as of C99, it is supported:

See Example:
`/home/cs340/examples/ccomplex`

Decimal

A built-in decimal type is not as widely supported by common programming languages. But it is included in some languages, usually those that support business applications (*e.g.*, COBOL), but also in C#.

The idea is to store a fixed number of decimal digits, in *binary coded decimal* (BCD) form. Here, each digit is represented by 4 bits.

We gain precision (we can have exactly as many digits as is needed) for some waste (we will need more bits). Also, many architectures do not support BCD directly, so operations are likely handled in software, which will be much slower leading to an efficiency hit.

Boolean

And the simplest data type: the `boolean`. All we have is `true` or `false`.

Most languages have some support for a genuine Boolean data type, but C traditionally did not, and used integer values to represent Booleans. Zero represents "false", any non-zero value represents "true".

Most of the time, these values are not represented by a single bit, as bit-level access is expensive and inconvenient on most architectures.

Characters

Computers only deal with numbers. Humans sometimes deal with letters and text. So we just need agree on how to encode letters as numbers if we want to have a computer process them.

- ASCII (1963) – American Standard Code for Information Interchange
  - fits in a byte
  - some you'll want to know:
    space (32 = 0x20)
    numbers ('0'-'9' = 48-57 = 0x30-0x39)
    lowercase letters ('a'-'z' = 97-122 = 0x61-0x7a), 96+letter pos
    uppercase letters ('A'-'Z' = 65-90 = 0x41-0x5a), 64+letter pos

  - See: `man ascii`

- Of historical interest: EBCDIC (Extended Binary Coded Decimal Interchange Code) developed by IBM for punched cards in the early 1960s and IBM still uses it on mainframes today, as do some banking systems in some circumstances.

- Unicode (1991), ASCII superset (for our purposes) – 2-byte characters to support international character sets

Most modern languages provide primitive data types to represent characters.

Strings

Most programming languages allow programmers to use *string* data types as a representation of sequences of characters.

There are many design decisions that need to be made.

- Are they arrays of characters, an abstract data type, or a primitive type in their own right?
- Are lengths static or dynamic?
- Are they counted or null-terminated?
- What operations are supported directly?

For static length strings, a descriptor is needed only at compile time to store the static string value, its length, and the starting address. For dynamic strings, we need a run-time descriptor which two lengths: the current length and the maximum length (as currently) allocated.

Language String Examples

We will look at the support for strings in a set of representative languages.

- C and C++
  - strings are not primitives
  - they are `char` arrays and a library of functions that provide operations in C or C++
  - end of meaningful portion of allocated array indicated by a null (`'\0'`) character – means length is an $O(n)$ operation
  - C++ provides a `String` class that encapsulates a C `char` array
  - no run-time descriptor needed
- SNOBOL4 (a string manipulation language)

- strings are primitive types with many operations, including elaborate pattern matching

- Fortran and Python
  - primitive type with assignment and several operations
  - Python strings are static length

- Java
  - implemented in the `String` class, but some "primitive type-like behavior" with + used for concatenation
  - `String`s are *immutable*
  - mutable strings are provided by the `StringBuffer` class

- Perl, JavaScript, Ruby, PHP
  - primitives with built-in pattern matching, using regular expressions
  - dynamic length

Enumeration Types

An *ordinal type* has a range of possible values can be easily associated with the set of positive integers. (*e.g.*, Java `int`, `char`, `boolean`).

With *enumeration types*, all possible values are provided in the definition.

C and Pascal were the first to use `enum`s. A C example:

See Example:
`/home/cs340/examples/enum`

As we can see with C, these are often implemented by integers.

Use of enumeration types improves readability, and can improve reliability in certain cases. We have essentially created a group of related named constants, and defined a datatype, instances of which are restricted to the available types.

There are a few design decisions that need to be made for a language that supports this:

- ss an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?

- are enumeration values *coerced* to integer?

- can other types be coerced to an enumeration type?

If the enumeration types are not coerced, this can allow better error checking (as in Java, C#) and restriction of operations to those apprpriate for an enumeration rather than integers.

We can see the power of Java `enum` types by looking at the examples at Oracle:

On the web: Enum Types at Oracle's Java Tutorials at
`http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`

Subranges

Closely related to enumeration types are *subrange types*. These consist of an ordered, continuous subsequence of an ordinal type.

For example in Pascal, we can define a variable such as:

```
var HoursWorked: 0..24;
```

This would define a variable that behaves much like an `integer`, except that the compiler and run-time system will not allow any value outside of the defined range.

See Example:
`/home/cs340/examples/hello_pascal`

See Example:
`/home/cs340/examples/pascal_subrange`

Ada provides a similar capability, as described in the text.

This has advantages in readability, efficiency (an appropriate internal storage can be chosen), and program safety (we can avoid using a clearly incorrect value).

Arrays

Our next data type is the very familiar and widely used *array* construct. For a data type that has such a wide usage, there are many variations and design decisions that lead to different implementations.

Our text describes an array as "a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element."

There are several major categories of arrays supported by various languages:

- Static
  - ranges/allocation done at compile time
  - *e.g.*, C/C++ arrays with `static` keyword
  - efficient: no dynamic allocation

- Fixed stack-dynamic
  - static binding of subscript range (declaration time)
  - *e.g.*, normal C/C++ arrays declared as locals
  - space efficient
- Stack-dynamic
  - run-time bindings for subranges and run-time allocation
  - fixed for the entire life of the array
  - *e.g.*, Ada arrays
  - flexibility advantage: need not know the array size until it is used
  - but still allocated on the stack
- Fixed heap-dynamic
  - allocation occurs by user specification, not at elaboration time
  - memory comes from heap, not stack
  - *e.g.*C++ arrays created by `new`, Java arrays
  - similar to stack dynamic except allocated on the heap
- Heap-dynamic
  - subscript ranges and memory allocation dynamic
  - arrays can grow or shrink during execution
  - very flexible, but more complex run-time
  - *e.g.*, Perl, JavaScript, Python, Ruby

The array notation needs to be resolved to addresses and values to be used.

Two options:

```
Address(X[n]) = Address(X[0]) + (n - 1) * element_size
```

or

```
Address(X[n]) = Address(X[0]) - element_size + (n * element_size)
```

In the latter, the first part can be done at compile time in certain situations (faster array access).

A compile-time descriptor for a single-dimensional array needs (most of):

- element type
- index type
- index lower bound (forced to 0 in C/C++/Java)
- index upper bound
- address

We also need to distinguish between rectangular and jagged multidimensional arrays.

- Rectangular arrays have congruent rows and columns
  - *e.g.*, Fortran, Ada, C#
  - must distinguish row major (most languages) *vs.*column major linearization (Fortran)
- Jagged arrays can have different length rows
  - essentially an array of arrays
  - *e.g.*, C++, Java

Think about the memory layout and run-time mechanism to access values in each option.

Additional array design issues:

- What types are legal for subscripts?
  - Fortran, C: integer only
  - Ada: integer or enumeration (includes `Boolean` and `char`)
  - Java: integer types only
- Are subscripting expressions in element references range checked?
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - In Ada, by default it requires range checking, but can be turned off
- When are subscript ranges bound?
- What is the maximum number of subscripts?
- Can array objects be initialized?
  e.g., in C, C++, Java, C#:

```
int a[] = {3, 9, 18, 21};
```

e.g., in Ada:

```
List : array (1..5) of Integer :=
    (1 => 17, 3 => 34, others => 0);
```

- Is there support for heterogeneous arrays?
  - elements need not be all of the same type
  - *e.g.*, Perl, Python, JavaScript, Ruby
- Additional operations (beyond standard indexing)?
  - array assignment (deep *vs.*shallow)
  - array catenation
- Are any kind of slices supported?

Hashes/Associative Arrays

An *associative array* also known as a *hash* is an unordered collection of data elements that are indexed by an equal number of values called *keys*.

For example, a Perl program of mine defines this associative array:

```
%team_array = ('ATLANTA','Atlanta Falcons',
'ARIZONA','Arizona Cardinals',
'BALTIMORE','Baltimore Ravens',
'BUFFALO','Buffalo Bills',
'CAROLINA','Carolina Panthers',
'CHICAGO','Chicago Bears',
'CINCINNATI','Cincinnati Bengals',
'CLEVELAND','Cleveland Browns',
'DALLAS','Dallas Cowboys',
'DENVER','Denver Broncos',
'DETROIT','Detroit Lions',
'GREENBAY','Green Bay Packers',
'HOUSTON','Houston Texans',
'INDIANAPOLIS','Indianapolis Colts',
'JACKSONVILLE','Jacksonville Jaguars',
'KANSASCITY','Kansas City Chiefs',
'MIAMI','Miami Dolphins',
```

```
'MINNESOTA','Minnesota Vikings',
'NEWENGLAND','New England Patriots',
'NEWORLEANS','New Orleans Saints',
'NYGIANTS','New York Giants',
'NYJETS','New York Jets',
'OAKLAND','Oakland Raiders',
'PHILADELPHIA','Philadelphia Eagles',
'PITTSBURGH','Pittsburgh Steelers',
'SANDIEGO','San Diego Chargers',
'SANFRAN','San Francisco 49ers',
'SEATTLE','Seattle Seahawks',
'STLOUIS','St. Louis Rams',
'TAMPABAY','Tampa Bay Buccaneers',
'TENNESSEE','Tennessee Titans',
'WASHINGTON','Washington Redskins');
```

This defines mappings between a token form of a football team's name with the team's full name. Later in the program, it is used to translate the tokenized version (which is found in the program's data files) to the full name:

```
  printf("<tr><td>$boldroad $team_array{$roadteam} $score1
           $endboldroad</td><td>at</td><td>$boldhome
           $team_array{$hometeam} $score2 $endboldhome</td>
           <td>${gametime} PM</td></tr>\n");
```

Here, both `$roadteam` and `$hometeam` have been assigned values when reading a data file.

These can be very convenient to use.

Design issues include:

- What is the form of references to elements?
- Is the size static or dynamic?
- How can we access elements efficiently (likely answer: hashing)

For Perl, in the example above, we could add a new team to the association with:

```
$team_array{'LOSANGELES'} = 'Los Angeles Californians';
```

And if we wanted to remove an element:

```
delete($team_array{'DALLAS'});
```

Python, Ruby, and Lua are other languages with direct support for associative arrays.

Records

A *record* is a (potentially) heterogeneous aggregate of data elements.

Examples include Pascal records, C/C++ structs, and C++ classes.

See Example:
`/home/cs340/examples/ratios`

In Ada:

```
type Emp_Rec_Type is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
        Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```

Individual members of a record can be accessed in various ways

- `Of` keyword in COBOL
- `.` operator in C/C++

Record storage is typically in a block of contiguous memory locations.

The names, types, and offsets (from the start of the record) associated with each field need to be stored in a (likely compile-time) record descriptor.

Pascal examples:

On the web: Pascal Programming Lesson 11: Record Data Structure at
`http://pascal-programming.info/lesson11.php`

Note the use of either the `.` notation or a `with` construct to access fields.

What other operations might be included?

- assignment is often supported if the types are identical (fieldwise copy)
- Ada allows record comparison
- COBOL provides `MOVE CORRESPONDING`, which copies a field of the source record to the corresponding field in the target record

Unions

A *union* type is similar to a record, but it can store different type values at different times.

Pascal provides this functionality with case variant records, C/C++ with the `union` types.

An example from C:

```
union nodeTypeStruct {
   DRUM_computingNode *compNode;
   DRUM_networkNode   *netNode;
} type;
```

Here, this defines a datatype that could either contain a `DRUM_computingNode` or a `DRUM_networkNode`, but never both. We treat it just like a `struct` except that the two fields share the same memory. So a modification to either is a modification of both.

This is implemented by assigning largest possible memory unit needed, along with all descriptions of possible types in the descriptor.

In C/C++/Fortran, unions are *free unions* – that means we can access the data using any of the names, regardless of which name was used to place the value into the union.

See Example:
`/home/cs340/examples/union`

A *discriminated union* remembers what type is currently stored in the union (which must be tracked in a run-time descriptor), and restricts access to that type. This is supported in Ada.

Since free unions are unsafe, they are not supported in Java or C#.

Tuple Types

We can think of a *tuple* as a record but where the elements are unnamed. In many ways, they are like lists, and can be used for purposes such as returning multiple values from a function.

The text mentions tuple implementations in Python, ML and F#. In particular, tuples in Python are essentially immutable lists.

List Types

We already looked at *lists* in Scheme, so we will say just a little more here.

Python has brought more fundamental list support into an imperative language.

See Example:
`/home/cs340/examples/python_list`

Pointers and References

A *pointer* is a type whose values consists of memory addresses and a special `nil` (or sometimes, `null`) value. C/C++ pointers and Java references are prime examples.

Pointers allow for *indirect addressing* – the data is in the memory location referred to by the contents of a pointer variable.

This provides the mechanism for dynamic memory allocation (*e.g.*, `malloc`, `new`).

Important operations on pointers:

- *assignment* – set a pointer's value to some useful address
  - result of a dynamic memory allocation
  - take address (`&`) of existing variable
  - copy value from an existing pointer with a useful address
- *dereferencing* – "follow" the pointer. In C/C++, this is the `*` operator preceding a pointer variable's use
- *pointer arithmetic* – `ptr++` – advances the pointer to the next memory location (relative to the data size of what `ptr` points to)

  See Example:
  `/home/cs340/examples/isort`

There are some dangers associated with allowing pointers in a programming language:

- *dangling reference* – a pointer to memory no longer allocated for its previous purpose

```
int *x = (int *)malloc(10 * sizeof(int));
// use x
free(x);

// use of x here is a dangling reference
```

- *memory leaks* – allocate memory but never free it for reuse
- *lost heap-dynamic variables* – an allocated heap-dynamic variable that is no longer accessible because we no longer are maintaining a pointer to that variable (*i.e.*, it is *garbage*, and a *memory leak* if nothing is done about it)

```
int *x, i;
for (i=0; i<10; i++) {
  x = (int *)malloc(10 * sizeof(int));
  // do other stuff?
}
// only the last allocated x is still accessible here
```

- casts to treat a pointer to one datatype as a pointer to another (as is allowed in C/C++)

References

- C++ includes a special kind of pointer type called a *reference type* that is used primarily for formal parameters (more on this later)
- Java reference variables replace pointers entirely – all are references to objects, rather than being addresses
- C# includes both the references of Java and the pointers of C++

References are more restrictive, and hence safer. No taking arbitrary addresses or pointer arithmetic.

Dealing with Memory Leaks

We have seen that languages sometimes require dynamically allocated memory to be explicity returned to the system (*e.g.*, C's `free`, C++'s `delete`), or have the system determine which memory is still in use and reclaim what is no longer used (*e.g.*, Java).

We also saw that with explicit deallocation, the potential may exist for references to remain to variables that no longer are allocated (or worse, since reallocated for some other purpose): the dangling reference problem.

The text describes two approaches: *tombstones* and *locks-and-keys* that can deal with this problem, but both are quite expensive. The most common approach to solving the problem is done at the programmer level rather than the language level - making sure any references or pointers to chunks of heap memory that have been deallocated are set to some null value as soon as the deallocation occurs. For example, one project I worked on that was developed in C and C++ has this requirement in the programmer's style guide for the project.

The most common approach is used by Java and many other modern languages is to take the responsibility for deallocating memory out of the hands of the programmer. In these cases, memory is reclaimed using a process called *garbage collection*.

Approaches:

- *reference counters* (eager approach)

  - maintain a counter for each allocation unit, if counter goes to 0, unit can be reclaimed
  - approach suffers from space/time considerations
  - circular lists lead to reference counting problems
  - process is incremental, so costs are spread over time

- *mark-sweep* (lazy approach)
  - allocation continues until heap space becomes low
  - when a garbage collection is triggered
    * mark all allocation units for removal
    * follow all pointers, unmarking any reachable unit
    * when no more pointers to trace, deallocate remaining marked allocation units
  - main disadvantage: it is an expensive process that causes the program to pause its execution for a significant time, though this can be reduced by running periodically not just when memory available becomes very low

Garbage collection has been an active area of research in programming language design for many years – we are just scratching the surface of the approaches and ways to do it more efficiently.

Type Checking

A language's *type checking* system ensures that the operands of an operator are of *compatible types*.

Types are compatibile with an operator if they either match the legal types for the operator or can be converted automatically (*i.e.*, *coerced*) to a legal type.

If such a conversion is not possible, we have a *type error*.

Type checking is closely related to type binding:

- static type bindings mean that type checking can (usually) be done statically
- dynamic type bindings necessitates a dynamic type checking
- with dynamic type binding and checking, type errors can only be detected at run time rather than compile time (when we'd like to detect them, if possible)

A language is called *strongly typed* if all type errors will be detected. This is helpful, as it ensures any type errors will be detected.

C and C++ allow unchecked union types, hence are not strongly typed.

Java, C#, and Ada are nearly strongly typed.

Type coercion is convenient for programmers (consider adding an `int` to a `float`, resulting in a `float`), but could cause some programmer errors to go undetected.

Type Equivalence

Even if two variables (or more generally, expressions) are of different type names, they could represent the same data type.

Two expressions that have *name type equivalence* are known to be exactly the same type. A name type equivalence requirement is relatively easy for a language to enforce, but unnecessarily restricts some otherwise legal constructs:

- subranges would not be equivalent to integer types

A more flexible but difficult requirement to enforce is *structure type equivalence*. Here, two data types that have the same "structure" even if not the same name, can be considered equivalent for type checking purposes.

Some issues to consider:

- Would two record types be considered equivalent if they happen to have the same types of fields and in the same order, but have different names?
- In languages that allow array subscript ranges, do the subscript ranges have to be the same or just the same "size"?
- What about programmer type definitions (think C's `typedef`) where the programmer may intend a value to be distinct from a type it has been defined as:

  ```
  typedef int grid_size_t;
  ```

  and the programmer did so to ensure that only integers of type `grid_size_t` would be allowed to be used where a `grid_size_t` is expected.

The text also goes into some information about type theory. You are encouraged to read about it but we will not cover it this semester.