Objectives

1. To explore what is meant by a "data type"
2. To introduce the notions of static and dynamic typing, and strong and weak type checking.
3. To define type equivalence, type compatibility, type inferrence, type conversion and type cast
4. To introduce key issues concerning array, record and pointer types

Introduction

A. Thus far, we have dealt in rather general terms with notions of syntax and semantics. We now want to spend a rather large amount of time looking at different facets of programming languages in general terms - considering what options the language designer has before him/her when designing a new language. We begin our study by looking at features for declaring data types and objects; then we move to different kinds of executable statements.

B. One of the most important discoveries that has been made through work in software engineering is that, in solving a given problem, the most important step is NOT the development of an algorithm. Rather, the most important step is developing a representation for the problem's DATA. Frederick Brooks puts it this way in his book, The Mythical Man-Month, under the heading "Representation is the Essence of Programming":

"Much more often, strategic breakthroughs will come from redoing the representation of the data or tables. This is where the heart of the program lies. Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious."

C. We will consider a number of topics under this heading:

1. What do we mean by a "data type"?

2. What do we mean by "a type system"?, including key issues:

   a. Strong versus weak typing.

   b. Static versus dynamic typing.

   c. Type equivalence.

   d. Type compability.

   e. Type inference.

3. The array structure.

4. The record structure

5. Pointers and references

I. What Do We Mean by a Data Type?

A. At the hardware level, there is support for a limited number of data types - all built around the most basic type of all, the bit. This support takes the form of hardware instructions that support basic operations on those types.

1. Integers - typically of several different sizes (e.g. 8, 16, 32, maybe 64 bit), and with the standard arithmetic operations plus comparisons typically supported.

   Many machines also support logical operations on integers regarded as vectors of bits - e.g. bitwise logical and, or, not, maybe xor.

2. Reals - typically of two different sizes (e.g. IEEE Float, IEEE Double) and with the standard arithmetic operations plus comparisons typically supported.

3. Characters - typically manipulated in the CPU/memory as 8 or 16 bit integers, but handled as characters by various IO devices.

4. Memory addresses (pointers/references) - typically using the same storage representation as an integer (32 or perhaps 64 bits).

5. Most programming languages include primitive data types that map directly to the first four of these (e.g. Java byte, short, int, long, real, double, and char.)

6. Many programming languages also support a boolean type that is actually implemented by using a small integer with false being 0 and true being 1; the hardware bit vector operations can be used on these, with all but the rightmost bit being ignored.

B. What do we mean by a data type in a higher-level language?

   ASK

1. The book suggested that there are three different - but related - ways of looking at this question:

   a. Denotationally - a type denotes a set of values

      - Example: the set of values of the type boolean is false, true

      - Example: the set of values of the programming language type char in languages like C++ is a set that includes 52 letters, 10 digits, 32 punctuation marks/special characters, the space, and 33 control characters.

      - Example: the set of values of the type "int" is actually a finite subset of the mathematical integers (e.g. in most languages the subrange -2,147,483,648 .. 2,147,483,647

   b. Constructively - a type is either one of a set of primitive types, or it is constructed from other types using type constructors like array, record/struct/class etc.

c. Abstractly - a type is an interface consisting of a set of operations.

2. Another way of looking at it sees a type as being characterized by two sets:

   a. A set of values that instances of that type may assume.

   b. A set of operations on instances of that type.

      Example: The Java type int

      Values = { X | X is in the subrange of the mathematical integers:
                  -2,147,483,648 .. 2,147,483,647 }

      Operations = { unary +, binary +, unary -, binary -, *, /, %,
                     <<, >>, >>>, ++, --, ~, &, |, ^,
                     ==, !=, <, <=, >, >= }

      Example: Java array types

      Values = { sequences of instances of some base type, numbered 0 ..
                 some specified maximum }

      Operations = { [] |

3. Data types may be classified as:

   a. Primitive types: types that are built in to the language and correspond directly to hardware data types.

      Examples: boolean, char, int ... (in Java, 8 in all)

   b. Ordinal types: scalar types whose instances are drawn from a fixed set of possible values, such that a one-to-one correspondence can be established with some integer type

      Examples: most primitive types - but _not_ reals, enumerations, subranges of an ordinal type

   c. Scalar types: types whose instances are atomic and cannot be taken apart into smaller pieces.

      Examples: ordinal types plus reals

d. Structured types: the value of an instance of a structured type is a composite of components of other, simpler types. Different languages support different ways of constucting structured types

- arrays (almost all languages)

- records/structs/classes (most languages)

- lists (a few languages)

- maps (very few languages)

- sets (very few languages)

- files (treated as a kind of data type in a few languages)

e. Pointer/reference types: the value of an instance of a pointer or reference type is the memory address of an instance of some other type.

4. An important issue is the notion of a named type versus an anonymous type.

   1. A number of languages have a construct that allows the declaration of a named data type (e.g C/C++: typedef; Pacal, Ada and others: type).

      Examples (I will use Pascal throughout, but many have analogues in other languages):

```
type
    intarray = array[1..10] of integer - is an array of 10 integers
    person = record
        string name; - assume string already defined
        string address;
    end;                        - person is a structure with 2 fields
    temperature = -20 .. 120; - temperature is a subrange of the
                                  integers
    gender = (male, female);  - gender is an enumeration
    letter_set = set of char; - letter_set is a set
```

   2. Many languages also allow one to use an anonymous type - one that does not have an explicit name

      Examples: (Again, I will use Pascal)

      - the declaration of intarray above used the anonymous subrange 1..10 - that is, the element type of intarray was integer, but the index type was anonymous

```
var
    temperatures_during_month: array[1..31] of integer;
            - temperatures_during_month is a variable whose type
              is an anonymous array type
```

   3. In Java, class, interface, and enum create named data types; but array types are always anonymous.

C. A related notion is the notion of a subtype: a type is considered a subtype of some other (base) type if any instance of the subtype can be used as an instance of the base type.

1. This yields, of course, the familiar law of substitution (isa rule) for inheritance in OO languages.

2. But other kinds of subtypes are also possible - e.g. a subrange is a subtype of its base type.

3. Note that subtypes give rise to the following issue: there may be a distinction between the _declared_ type of a variable and the _actual_ type of its value.

   Example: class Person ...
            class Student extends Person ...
            class Faculty extends Person ...

            Person p ...

   The declared type of p is Person, but at a given moment of time, the actual type of its value may be any subtype of Person - e.g. Student or Faculty.

II. What do we Mean by a Type System?

A. The term "type system of a language" is used to describe the overall approach the language takes to data-type-related issues.

B. Type systems can be extensible or non-extensible.

1. Many early languages had type systems that were not extensible - i.e. the set of possible data types was largely fixed when the language was defined.

2. Most newer languages have extensible type systems - i.e. the user is able to define new data types - typically at program write time.

C. Type systems can be strong or weak.

1. A language is said to be strongly typed if the language system enforces the language's rules of type compatibility - i.e. it does not allow an operation to be performed on an object that is inconsistent with that object's type.

2. A language is said to be weakly typed if such enforcement is not done - i.e. it is possible to perform an operation on an object that is not consistent with that object's type (in which case the object's bit pattern is interpreted in accordance with the expected data type, usually resulting in wrong if not bizarre results).

   Example: FORTRAN is weakly typed. For example, we showed earlier that it is possible to call a subroutine with a very different type of parameter than what it expects.

Example: Traditionally C was considered weakly typed because it did not enforce type equivalence between the formal and actual procedures of a function - e.g. an old style C function declaration would look like this:

```
void foo(a, b c)
  int a, b, c;
{
  ...
```

This specifies that foo expects to be called with three int parameters, but the compiler will not check this, so that the following is perfectly legal

```
foo(1.0, 2.0, 3.0)
```

or even

```
float f[10], g[10], h[10];
foo(f, g, h);
```

ANSI C moved in the direction of much stronger typing, by introducing the notion of a function prototype, which the compiler would enforce - e.g.

```
void foo(int a, int b, int c)
```

In C, the use of prototypes is optional (so one gets strong typing of function calls iff one chooses to do so). In C++, the use of prototypes was made mandatory.

3. Weak versus strong typing is not an absolute either/or thing. There is a spectrum, in the sense that some languages are more strongly typed than others. Sometimes languages that are basically strongly typed have "holes" in their type system, where type misuses can go undetected. [Purists would say such languages are not truly strongly-typed.]

   a. Example: the C/C++ union:

```
union foo
{
    int i;
    float f;
};
```

   This declares a type foo that _either_ holds an int or a float; both fields share the same location in memory because it is assumed that only one is in use at a time. The programmer is responsible for ensuring that if a given foo holds an integer, then foo.f is never accessed, and conversely if it holds a float then foo.i must never be accessed.

   Of course, through either accident or design it is possible to misuse a union - e.g.

```
union foo x;
x.i = 42;
...
cout << x.f << endl;
```

will write 5.88545e-44 on an Intel processor.

b. Note that it is not considered weak typing if the language system detects a type violation and signals an exception at run-time - e.g. in Java, if one casts an Object to a type it actually is not, one gets a ClassCastException, which prevents a misuse of the object.

The above union example was considered a "loophole" in C's typing because no error would be signalled at run-time.

4. Sometimes, the term "Type Safety" is used to speak of the opposite of loopholes like the C union. A construct is not type safe if it can poke a hole in strong typing. The notion of type safety is important in light of two kinds of concerns.

a. Software engineering concerns - a place where a system is not type safe is a place where an undetected error can occur, perhaps causing a symptom somewhere else in the system.

b. Security concerns - a place where a system is not type safe creates a loophole that could be exploited maliciously.

Example: The C union loophole could be used by a hacker as follows:

```
union hack
{
   int i;
   [ something ] * p;
}
```

Now, the hacker can place any value desired into p by storing a value into i, and thus can access any location in memory under any desired type. While modern computer systems typically have memory protection that prevents access to memory belonging to the system or other users, a hacker could embed code using this loophole into an application program to gain illicit access to information belonging to the user who runs the application - naievely thinking it will do no harm.

6. Arguments for/against strong typing

a. Because strong typing is a major aid to reliability, it has become an increasingly important characteristic of newer languages.

b. One might ask, then, why it is that some languages are not strongly typed. There are two major reasons.

i. In languages which rely on separate compilation for modularity (like C and FORTRAN), it is hard to implement type checking of procedure parameters across separately compiled modules.

- This problem can be solved by including declarations of the interfaces to one's procedures in one file, and the actual body of the procedures in a separate file, as C/C++ do with their header (.h) files. If the programmer uses full function prototypes, then the compiler will type-check all calls to those functions appearing in any file which includes the header file. (However, for compatibility with older versions of C, prototypes are not required in ANSI C - indeed, even function declarations are not mandatory if their result type is int - the compiler infers that the name refers to a function and the number of parameters from use).

- Languages like Modula-2 and Ada formalize this by splitting a packages of procedures, functions, etc. into a package specification and a package body (called definition and implementation modules by Modula-2.) The former must be referenced by all modules using the procedures or functions in the package.

ii. A second reason why languages are not always strongly typed is that strong typing can make it harder to do certain kinds of low-level systems programming. For example, a memory management routine might have to be able to treat pointers as integers.

c. A good example of a thoroughly strong-typed language that still supports systems programming is Ada, which is consistently strongly typed except for one construct that allows one to deliberately poke a hole in type safety when necessary for systems programming. But then the fact that one has done so is documented by the presence in the program text of the use of Unchecked_Conversion.

D. Type systems can be static or dynamic.

1. If the language uses static typing, every variable has a data type that is known at compile-time either because it is explicitly declared or because it is implied.

   Example: Most languages you have seen are statically typed. Even FORTRAN is in this category because, when a variable is not declared, its type is determined by its name.

2. If the language uses dynamic typing, the type of a variable is determined by a value that is assigned to it while the program is running - which generally means that the type of a variable can change as the program runs.

a. Example: Scripting languages such as Python are dynamically typed, as is LISP. The following LISP assignments are valid (all referring to the same variable)

```
(setq x 3.0)
...
(setq x "aardvark")
...
(setq x '(1 2 3))
```

After the first assignment, x would have type real; after the second type string and after the third type list.

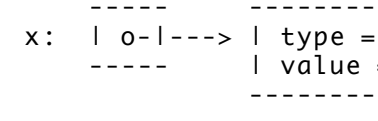
b. Dynamically typed languages are much more flexible, but the flexibility has a price.

- Since the type of a variable can change during execution, the question arises of how to allocate memory for a variable, since different data types require widely differing amounts of memory. This is usually handled by allocating to each variable enough storage for a pointer, and then allocating space from the heap for values as needed.

- Since the type of a variable can change, its representation must encode both its value and its type. There are several ways to handle this:

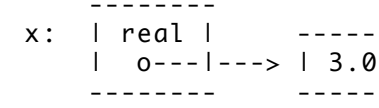
  - reserving the first byte of the node the variable points to for a tag which encodes the type.

    Example: after (setq x 3.0), we might have



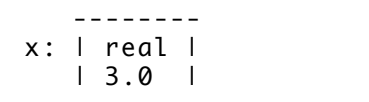
    This approach is called the TAGGED-DATA approach.

- Alternately, the representation for the variable itself might include space for the tag and a pointer, yielding:



  This approach is called the TAGGED-POINTER approach.

- In the latter case, if the value for certain types is small enough to fit along with the tag in the space needed by a pointer, then the value might be stored in the pointer field of the variable, avoiding the need for a separate node - e.g.



- The main limitation to the tagged-pointer approach is that one generally wants a variable to require a multiple of single machine words for its representation. (This makes structured data neat and clean toimplement). However, since a pointer typically requires a machine word by itself, this leaves no room for the tag.

  - Sometimes a few tag bits can be squeezed in by taking advantage of the fact that certain bits of a pointer must be zero. Then a hybrid tagged-pointer/tagged-data approach can be used.

c. Interestingly, though OO languages are typically statically typed, a variant of the tagged data approach is typically used for objects, where the representation for an object encodes information about its class to facilitate polymorphism and safe type casts.

3. Either way, when a variable is used its type must be consistent with the way it is used - for example, an operand of addition must have a numeric type.

   a. In the case of a statically-typed language, most or all of this checking can be done by the compiler.

      Certain runtime checks may still be needed - e.g. in an object-oriented language, a variable can have a declared type, but its value

   b. In the case of dynamic typing, this requires a run-time check of the tag whenever the variable is used in a way that depends on its typed.

E. A type system includes rules of type equivalence - when are two data types considered to be the same?

   1. Some languages use some form of NAME EQUIVALENCE - two types are the same only when they have the same name. There are a number of issues that arise in this case.

      a. There is a subtle problem with operators. Consider the following example (using Pascal, which allows named subranges):

```
type
    gradetype = 0..100;

var
    grade, oldgrade, difference: gradetype;

begin
    ...
    difference := grade - oldgrade;
```

      Is this computation legal? The operator - is defined as addition for integers and reals, but not for gradetype.

b. Constants pose a problem, too. Consider the following:

```
grade := 0;
```

Should this assignment be permitted? The type of 0 is integer, while grade is of type gradetype, and these names are not the same. .

c. Then, we get perfectly reasonable operations like the following, nvolving a clear mixing of different named types:

```
var
    classgrades: array[1..size] of gradetype;
    sum, i: integer;
begin
    sum := 0;
    for i := 1 to size do
        sum := sum + classgrades[i];
    ...
```

d. To handle problems like these three, strict name equivalence is typically modified to allow subrange types to "inherit" operators and constants from the parent type, and to participate in arithmetic with the parent type. In effect, objects of type gradetype can be treated as integers in some cases, but not others.

e. However, the line must be drawn somewhere. We probably DON'T want to allow:

```
type
    agetype = 0..100;
    gradetype = 0..100;
var
    age: agetype;
    grade: gradetype;

begin
    ... age + grade ...
```

(This is precisely the kind of nonsensical mixing of types that name equivalence tries to prevent.)

Figuring out how to draw the line - and how to document the decisions in a readable way - is non-trivial.

f. Note that Java uses name equivalence, but the only kind of named types the user can create is classes/interfaces/enums, where strict name equivalence makes sense and does not create problems like the above.

2. Some languages use some form of STRUCTURAL EQUIVALENCE - two types are the same if they have the same structure but different names (or one or both is anonymous).

Example: Pascal-like language

```
type foo = array[1..10] of integer;
     bar = array[1..10] of integer;
```

a. From the standpoint of name equivalence, foo and bar are two different types.

b. From the standpoint of structural equivalence, they are the same type.

c. Even the exact rules of structural equivalence vary quite a bit between languages, but the following is a typical approach:

   i. Two objects are always structurally equivalent if they are name equivalent.

   ii. A subrange of a given type is structurally equivalent with its parent type, and two subranges of the same type (or which ultimately trace their ancestry back to the same type) are structurally equivalent. Of course, if a variable of one subrange type is assigned a value from a variable of another subrange type, the value assigned must lie within the declared subrange of the receiving variable, too, which may require a runtime check.

   iii. Two arrays are structurally equivalent iff:

        - Their subscripts have the same lower and upper bounds.
        - Their basetypes are stucturally equivalent

   iv. The rule for records may vary - one requirement may or may not be present. Two records are structurally equivalent iff:

        - They have the same number of fields
        - Their corresponding fields are structurally equivalent
        ? Their corresponding fields have the same names ?

        In the case of variant records, the two records must have the same number of variants, with the same tags, and each pair of corresponding variants must be structurally equivalent.

   v. Two pointers are structurally equivalent if what they point to is structurally equivalent.

   vi. No other objects are structurally equivalent.

3. Because strict name equivalence may be too restrictive, and structural equivalence is sometimes too loose, it is also possible to follow a mixed approach. For example, ISO Pascal uses the following rules:

   a. Scalar types and character strings follow the rules of structural equivalence.

   b. All other structured types follow a modified rule of name equivalence sometimes called DECLARATION EQUIVALENCE. Two structured types are equivalent if they are name equivalent, or can be traced back to name equivalent types.

Example:

```
type
    T1 = array[1..10] of integer;
    T2 = T1;
var
    A1: T1;
    A2: T2;
```

A1 and A2 are NOT name equivalent, but they are declaration equivalent, because both their declarations trace back to T1.

F. A type system includes rules of type compatibility - when can an instance of one type be used where an instance of some other type is expected?

1. Of course, equivalent types - by whatever rule the language uses - are always compatible - the issue arises only when we use types that are not equivalent.

2. In many languages, this varies with the way in which the instance is being used - e.g. the rules of compatibility may be looser for assignments or value parameters than for reference parameters.

   Example: the following is valid in C++

```
void foo(int i);
int main(int argc, int * argv) {
   short s;
   foo(s);
}
```

   But if we change the parameter type int i to int & i, it becomes invalid

3. Sometimes, in order to allow an operation, a compiler must generate some sort of type conversion. We will now consider three terms that are ofen confused, though each has a precise technical meaning.

   a. A CONVERSION involves transforming an instance of one type into an instance of another type that has the same meaning.

   Example: Many languages allow mixed mode arithmetic - e.g.

```
int a;
double b;
...
... a + b ...
```

Typically, this is handled by converting the value of a from integer to double - e.g. if a is 1, then it is converted to 1.0.

In general, conversions alter the FORM (the bit pattern), but not the MEANING of the instance. For example, if 1 is converted to 1.0, the meaning is preserved at the expense of going to a very different bit pattern (typically an IEEE Double rather than a two's complement integer).

b. A COERCION is a conversion that is done automatically when it is needed, rather than being explicitly specified by the programmer. For example, the conversion of a from int to double above is done as a coercion in most languages, though some languages require the programmer to explicitly specify a conversion.

- Example: Modula-2 does not do coercion. So the following is not legal in Modula:

```
VAA a: INTEGER;
    b: REAL;
...
... a + b ... -- Not legal
```

Instead, one must write:

```
... REAL(a) + b ... - explicitly specifying the conversion
```

i. Languages that do type coercion automatically must include a set of rules that specifies what combinations of operands can be mixed and what the resulting type will be.

For example, the following are the CONTAGION rules of COMMON LISP, which has a particularly rich numeric type structure:

	fixnum	bignum	ratio	short-float	long-float	complex
fixnum	fixnum	bignum	ratio	short-float	long-float	complex
bignum	bignum	bignum	ratio	short-float	long-float	complex
ratio	ratio	ratio	ratio	short-float	long-float	complex
short-float	short-f	short-f	short-f	short-float	long-float	complex
long-float	long-f	long-f	long-f	long-float	long-float	complex
complex	complex	complex	complex	complex	complex	complex

ii. Coercions are generally performed in such a way that the set of values of the result type is a superset of the set of values of the source type.

- Thus, for example, many languages allow integer to be coerced to real because the set of reals includes all integers as a subset.

- However, most languages require explicit conversion when the source type is not a subset of the target type - i.e. with automatic coercion, loss of information is possible - e.g.

```
int a;
double b;
...
b = a; -- Many languages will allow
a = b;  -- Many languages will _not_ allow, because some
           information will almost certainly be lost - the
           fractional part of b, and possibly the magnitude if
           the magnitude is 2^31 or greater.
```

- But note that some languages _do_ perform coercions that might lose information - e.g. the following is legal FORTRAN

```
REAL R
INTEGER I
I = R
R = I
END
```

iii. One question that confronts the language designer is how far to go with coercion. t is instructive to look, for comparison languages like C or Java, at some of the extremes.

- Modula-2 is an example of a very conservative language. There are NO automatic conversions done within expressions - not even between different integer types. (But subranges may be freely mixed with their parent type.) Some automatic conversion is done on assignment, but only between integer types.

- PL/1 is an example of a very liberal language. The basic rule is that if a meaningful conversion is at all possible, it should be done. Thus, for example, character strings and numbers can be mixed in an expression; an attempt will be made at run time to convert the character string to a number or vice-versa, depending on the operator being used. Likewise, there is free conversion between bit-string (boolean) items and numbers.

- Java, though generally conservative, will coerce any data type to a string if concatenated to a string. (The compiler does this automatically for a primitive, and invokes the toString() method which is defined in Object though it may be overridden by any class to do this in a more appropriate way.)

c. The term CAST is used in two quite different ways.

- Sometimes, cast is used as a synonym for explicit conversion (a converting cast).

  For example, the above illegal assignment could be made legal in Java by using a cast:

```
a = (int) b;  -- In this case, the compiler understands this
                 as a statement by the programmer that he is
                 willing to accept whatever loss of information
                 occurs in this conversion
```

- Sometimes, cast is used to refer to a directive to the compiler to preserve a bit pattern but look at in a different way. In such a case, the FORM of the information is preserved, but its MEANING is changed. In fact, there is a "trick" a programmer can use to force this sort of cast even when the normal cast would do a conversion

  Example (based on one in the book) Treat the bit pattern for an int as if it were a float (preserving the form but changing the meaning)

```
int n;
float f;
...
f = *((float *) &n);
```

- C++ introduced various cast specifiers, including

  static_cast - always does a conversion (preserves meaning while possibly changing the bit pattern)
  reinterpret_cast - never does a conversion (preserves bit pattern while possibly changing the meaning)

G. A type system includes rules of type inference - what is the type of an expression?

1. This may be important in deciding questions of compatibility.

   Example: given the assignment a = b / c, whether b / c is assignment compatible with a depends on the type of b / c.

   It turns out that this particular example is handled differently by different languages.

   - Suppose a, b, and c are all declared to be integers. Then in most languages, this assignment is legitimate.

   - However, some languages (e.g. Pascal) specify that the result of / is _always_ a real number. (A separate operator - e.g. div in Pascal - is used for integer division). In such a language, the assignment may not be legitimate, since the right hand side is a real number, but the destination variable is an integer.

2. The issue becomes complex when subranges are involved.

   Example: given type smallint = 1..100;
   var a, b: smallint;

   the type of a + b is not necessarily smallint!

3. One language (ML) allows one to user variables without declaring a type, and infers the type from the way the variable is used (i.e. the types of constants or other variables it is used with.)

H. In learning about the type system of a given language, here are some questions you will want to ask:

1. What data types are supported?

   a. What data types are built in to the language?

   b. What type constructors are available for creating user-defined types?

   c. Can user-defined types be given names, or are they anonymous?

2. How strongly typed is the language?

3. Does the language use static or dynamic typing?

   - If the language uses static typing, under what circumstances is a dynamic (runtime) check necessary to verify something that cannot be staticly checked?

4. What are the rules of type equivalence, compatibility, and inference?

Example: Java

ASK CLASS

III. Array Types

A. Historically, the oldest form of structured type supported by programming languages (and most do) is the array.

1. Typically, an array is represented with a subscript used to select a particular element.

   Example: the C construct int n [10]; would cause n to be associated with a block of storage 10 times the size of an int, with n[0] referring to the first, n[1] the second ...1.

2. An array is actually associated with two data types: the type of its elements, and the type of its index.

   For example in the above C declaration, the element type is int and the index type is the integer subrange 0..9.

   a. The oldest languages restricted the element type to primitives, but most languages allow the element type to be any data type.

   b. There is more variation in terms of the index type.

      - FORTRAN: A subrange of integers, starting at 1

        e.g. DIMENSION N(10) says the elements of N are numbered 1 .. 10

      - The C family of languages: a subrange of integers, starting at 0

        e.g. The above C example says the elements of n are numbered 0..9

- The Pascal family of languages allows any subrange of any ordinal type

  e.g. n: array [-4 .. 5] of integer says the elements of n are numbered -4, -3, -2 ... 5

  e.g. n: array ['A' .. 'Z'] of integer says the elements of n are "numbered" 'A', 'B', 'C .. 'Z'

- Some languages (e.g. Python) allow "associative arrays" in which the index type can be non-ordinal, and C++ allows the subscript operator ([]) to be declared for any class. These aren't really arrays, but rather maps implemented by a hashtable or a tree. The language simply allows "array syntax" to be used with a structure. (We will not discuss this possibility further here).

c. In any case, the compiler applies a "subscripting formula" to an index to calculate the actual location of an element.

   - Assume the storage allocated for the array begins at an address b in memory.

   - If the index is not an integer, it is treated as an integer (something always possible with an ordinal type, since ordinals are actually represented as small integers).

   - Assume the index has range lo .. hi.

   - Assume each element has size s.

   - Then the location in memory of element[i] is calculated as b + (i - lo) * s

   - This would seem to argue for 0 origin indexing, since the calculation is simpler. But since the above is equivalent to (b - lo * s) + i * s, if is possible to calculate the first term once (possibly at compile time) and use a "virtual base" when actually interpreting a subscript

B. One issue that arises with arrays is how/when is the needed storage allocated.

   1. In original FORTRAN, arrays (and indeed all variables) were allocated statically. This is possible because the size of the array is known at compile time.

   2. Many block-structured languages (and some FORTRAN implementations) allocate arrays local to a block on the stack frame, just like other variables. This is possible provided the size of the array is known when the name comes into scope (elaboration time).

Example: C++

```
void foo(int s)
{
   ...
   int n[s];
```

is valid C++ (and also C if there is no executable code before the declaration). The amount of storage allocated for a will vary between calls to the function.

3. Many languages allow/require arrays to be allocated dynamically, using the new operator. In this case, the size of of the arrya must be known when the new statement is executed.

   Example: Java

   ```
   int s = a * b;
   int [] n = new int[s];
   ```

   is valid as long as the a * b yields a non-negative integer value.

4. Some languages use just one of these three approaches (e.g. historical FORTRAN always static; Java always dynamic), while others allow two or all three.

   Example: C++

   int n[10] uses static allocation if it appears in global scope
   int n[s] uses stack allocation if it appears in local scope
   int n [] = new int[s]; is allowed anywhere in executable code.

C. Another issue that arises with arrays is the matter of multi-dimensional arrays.

1. FORTRAN, for example, allowed arrays to have up to 7 dimensions - e.g.

   DIMENSION N(2, 3, 4, 5, 6, 7, 8) is valid FORTRAN that sets aside storage for 40,320 integers.

   (Many other languages - including the Pascal and most of the C family - allow something similar.)

2. The subscripting formula now becomes a bit more complex. For example, given the C declaration int n[3][4], what we get is an allocation that looks like this:

```
-----------
| n[0][0] |
| n[0][1] |
| n[0][2] |
| n[0][3] |
| n[1][0] |
| n[1][1] |
| n[1][2] |
| n[1][3] |
| n[2][0] |
| n[2][1] |
| n[2][2] |
| n[2][3] |
-----------
```

and the address of n[i][j] is calculated as

```
b + s * (4*i + j)
```

3. A significant issue is whether the array is laid out using row major order or column major order. (FORTRAN uses column major; most other languages use row major.)

   For example, the FORTRAN declaration DIMENSION N(3, 4) yields an allocation that looks like this:

```
-----------
| N[1][1] |
| N[2][1] |
| N[3][1] |
| N[1][2] |
| N[2][2] |
| N[3][2] |
| N[1][3] |
| N[2][3] |
| N[3][3] |
| N[1][4] |
| N[2][4] |
| N[2][4] |
-----------
```

   and the address of N[I][J] is calculated as

```
b + s * (4*J + I)
```

   - Knowing which order is used can make a difference if how one writes nested loops if one is concerned about performance on a system that uses cache memory (as all modern machines do)

     e.g. for a large array n[rows][cols] the C code

```
for (int i = 0; i < row; i ++)
   for (int j = 0; j < cols; j ++)
     sum += n[i][j];
```

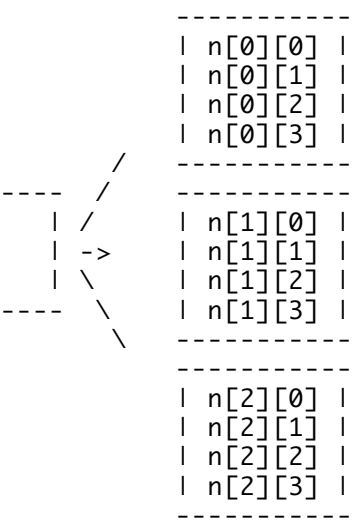
is better written in FORTRAN as

```
DO 1, J = 1, COLS
    DO 1, I = 1, ROWS
        SUM = SUM + N[I][J]
```

4. The above constrasts with a different approach, which is to treat a multi-dimensional array as an array of pointers to arrays of one less dimension - e.g.

   Java int n = new int [3][4]

   produces this:



   So the subscripting operation is a bit more complex: to access n[i][j], first use the formula b + size of pointer * i to locate the base of the array n[i], then add to this size of int * j to locate n[i][j]

   a. This has a bit more space overhead, but allows arrays with differing length rows.

   b. It does not, however, map well to array structures in other languages if one is writing a multi-language program.

D. Character strings constitute an important "array-like" data type.

   1. Conceptually, a string is an array of characters.

   2. Character strings deserve special attention for several reasons:

      a. String processing is important in many application areas - at least for IO.

      b. There are many operations on strings which would be desireable to include in a language: length, concatenation, substring, comparison, etc.

3. Unlike other builtin types, the amount of storage needed for a string can vary greatly at runtime, due to application requirements.

   Three approaches may be taken to building strings into a language:

   a. Treat strings as arrays of characters, with each string having a fixed length, with unused spaces filled with some padding character, or with the end of the significant part of the string marked with some marker.

      Example: C with \0

   b. Represent a string by a record structure consisting of a length indicator and an array of some maximum maximum length. At any time, the current length of the string can be 0 .. maximum specified when it was created.

      Example: FORTRAN 77

      Both of these approaches lead to potentially large amounts of wasted space, as well as the possibility that a value that is needed won't fit.

   c. Allocate storage for a string "on the fly". A string variable may be a record structure consisting of a length indicator and a pointer to heap allocated storage where the characters are actually stored.

      Example: C++ (where strings are not actually part of the language, but rather a standard library that uses operator overloading to make strings behave like part of the language.)

   d. Allocate storage for a string "on the fly", with a string variable simply a pointer to a structure constaining the length indicator and data.

      Example: Java (where strings are implemented by a library class, though the compiler gives them some special status)

IV. Record Types

A. Another data type that entered into programming languages and now exists in some form in many is the record (struct) type.

1. An array is a block of memory holding a sequence of values of the same type, referenced by index.

2. A record is a block of memory holding a sequence of values of potentially different types, referenced by name.

Example (C/C++)

```
struct
{
    int i;
    double d;
    char c;
} x;
```

results in x referring to a block of memory that looks like this

```
x i [ 4 bytes ]
  d [ 8 bytes ]
  c [ 1 byte  ]
```

B. One issue that arises in conjunction with records is the notion variant records. A variant record is one that has two or more different possible layouts.

1. A simple example: the C/C++ union

2. Most languages that support variant records include a tag which indicates which variant is being used.

Example: Pascal (assume separate declarations for name_type, address_type, major_type)

```
type
    status_type = (student, faculty);

    cruelty_type = (nice, tough, rotten);

    person = record
    name: name_type;
    address: address_type;
    case status : status_type of
        student: (gpa: real; major: major_type);
        faculty: (cruelty: cruelty_type);
    end;
```

- When there is a tag, the value of the tag determines which variant is actually present.

  In the above, person.name, person.address are always legal
  In the above, if tag is student, then person.gpa and person.major are legal, but person.cruelty is not
  However, if tag is faculty, then person.cruelty is legal, but person.gpa and person.student are not
  Referencing a field that is not consistent with the tag results in the same sort of violation of type safety as happens with a C union.

3. Type-safety imposes the following requirements on variant records (enforced in Ada, but not in Pascal)

   a. The presence of an explicit tag.

   b. A runtime check to ensure that, when a field in the variant portion is accessed, the access is consistent with the tag.

   c. If the tag is changed, then the variant value must be changed at the same time (or perhaps automatic initialization to a "safe" value.)

4. How do OO languages provide the equivalent of variant records?

   ASK

   By subclassing a base class for each variant

V. Pointers and References

A. Programming languages tend to follow one or the other of two models with regard to variables.

1. The value model: A variable stands for a location in memory that holds a value.

   Example: Java

```
double d = 1.0;
int i = 42;
char c;
```

   results in the following in memory (assume static starting at address 1000 hex)

```
1000-1007  d (8 bytes) holds IEEE double representation for 1.0
1008-100b  i (4 bytes) holds 0x0000002a
100c       c (1 byte)  holds 0x63
```

2. The reference model: A variable stands for a location in memory that holds the address of another location in memory where the value is actually stored.

   Example: Java again

   ```
   Robot r = new Robot();
   ```

   results in the following in memory (assume r is at location 1000 hex, and the Robot is created at location 2000 hex and is of size 20 hex

   ```
   1000-1003  r (4 bytes) holds 0x00002000
   ...
   2000-201f  [ Robot object - 20 bytes ]
   ```

3. Languages that are statically-typed typically use the value model as their primary model; languages that are dynamically-typed generally use the reference model (since the memory needed for a value can change as the type changes).

   Java, as the illustration above illustrates, actually uses both models - the value model for primitives, and the reference model for objects and arrays. (Due to polymorphism, object types are actually dynamic, since a variable of object type can point to objects of a subclass instead; arrays can vary in size.)

B. Older programming languages did not include user-visible pointer types; but they quickly became part of statically typed languages to facilitate the creation of other types such as lists, trees, graphs, etc. Thus, this feature is usually present in newer statically-typed languages.

   In fact, some sort of pointer or reference facility is mandatory if one is going to allow recursive data types like trees. A declaration like the following using the value model would actually declare an infinitely big structure!

   ```
   struct ListNode
   {
      ... data
      ListNode next;
   }
   ```

   which makes a ListNode look like this

   ```
   ----------------
   | data
   |---------------
   | next
   | --------------
   | | data
   | |-------------
   | | next
   | --------------
   | | | data
   | | |-----------
   | | | next
   ...
   ```

Instead, one must code this as

```
struct ListNode
{
   ... data
   ListNode * next;
}
```

which makes a ListNode look like

```
---------
| data  |
| next  | ---> to another node
---------
```

(Java allows a form analogous to the first example because in Java object variables are _always_ references, but the structure is like the second.)

C. Languages like C, Pascal, and Ada support pointers as well as ordinary value variables. C++ supports references and pointers as two distinct kinds of entity. Java supports only references, and then only for "reference types". The difference between a pointer and a references is basically a matter of syntax; both are implemented similarly.

1. Pointer syntax: a pointer must be explicitly dereferenced - e.g.

   ```
   int * c;
   ...
   cout << *c + 1;
   * c = 42;
   ```

   An assignment to a pointer is to the pointer, not the thing pointed to.

2. Reference syntax: no explicit deference:

   ```
   int & c;
   ...
   cout << c + 1;
   c = 42;
   ```

   An assignment to a reference is to the thing referred to; the reference is immutable.

3. Java uses reference syntax, with a mix of reference and pointer semantics.

   a. When a reference appears in an expression, it is implicitly dereferenced. [Reference semantics]

   b. When a reference is the target of an assigment, the reference is assigned to, not the object referred to. [Pointer semantics]

D. In C/C++, there is an equivalence between arrays and pointers.

a. The following two declarations are equivalent:

```
int * p;
int [] p;
```

b. Arithmetic on pointers is done in units of the size of the type pointed to.

- Example

  int * p, *q;

  Assume p = 1000x. Then p+1 = 1004x (integers are 4 bytes), etc.
  Assume p = 1000x and q = 1004x. Then q - p = 1.

E. An important issue with regard to pointers is how memory allocation and deallocation is handled.

1. Languages that support pointers typically provide a syntax for creating a new object on the heap and setting a pointer to it - typically an operator such as new.

2. We have already seen that languages typically take one of three approaches to handling the dealloction of storage for objects that are no longer needed. Let's Review. What are they?

   ASK

   a. Explicit deallocation - e.g. the C++ delete operator.

   b. Reference counting - heap-allocated objects contain an extra field recording the number of pointers referring to the object, which the language system keeps up to date when assigning pointer values. When the reference count for a node drops to 0, it is recycled.

   c. Garbage collection - the language system automatically "recycles" storage used by objects that have no pointers to them.

3. Each approach has significant problems. What are they?

   ASK

   a. Explicit deallocation can lead to dangling references or storage leaks.

   b. Reference counting will be fooled by circular structures, resulting in storage leaks.

   Example: External pointer -> [2 ref] -> [1 ref]-->
                                  _____________________|

   If the external pointer is made to point elsewhere, the structure becomes garbage, but both reference counts remain at 1!

   c. Garbage collection - considerable overhead; dependent on language design facilitating locating pointers.

4. Some examples of choices various languages have made:

   a. C/C++ : require explicit deallocation.

   b. Ada: implementor can choose, but most implementations use explict deallocation

   c. LISP, Java, PROLOG: Garbage collection