Data Types

Learning Objective

The objective of this session is to gain an understanding of how data is represented and stored in computers.

Overview

Last session we looked at several language concepts associated with six attributes of a variable (***name, address, type, value, lifetime, and scop***e). This session we'll again be considering variables, focusing on the ***type*** attribute. We will examine various data types: primitive, character strings, ordinal, arrays, records, unions, sets, and pointers. One of the assignments this week will involve running a simple program written in C that allows you to examine how data is stored on your PC.

The purpose of computer programs is to manipulate data. Data comes in many types: integers, characters, floats, Booleans, etc. To further complicate matters, these primitive types can be combined (aggregated) into arbitrarily complicated structures to form derived types. Programming languages vary considerably in their ability to handle the various data types.

The utility of a programming language is dependent on its ability to facilitate the manipulation of the types of data required by the problem domain, e.g., scientific, business, etc. Scientific applications, for example, often require large arrays of floating-point numbers, whereas business applications focus more on data structures designed to hold monetary values, dates, and character strings. In addition, there are data-storage considerations, such as the computer architecture (e.g., the memory word size), and the efficiency of data-storage structures. The manner in which programs allocate data storage space is also an important consideration. Some languages allow (or require) the use of either static and dynamic memory allocation, or both.

Representing Data in a Computer

You may recall from an earlier Session the discussion concerning binary digital computers. We pointed out that for reasons of design simplicity, digital computers use a binary (1s and 0s) scheme to represent instructions and data. Hence, all data types will be represented as strings of 1s and 0s, i.e., bits; the length of the string of bits required to store a given piece of data will vary depending on the data type.

Most modern computers organize their memory into ***bytes.*** A byte is simply 8 ***bits*** of data, with a bit being either a binary 1 or 0. A given data type may require one or more bytes of memory storage space. As you will be able to confirm later for yourself, the lowercase letter "h" on your computer is represented as the single byte containing the binary bits 01101000.

Primitive Data Types

Primitive types are the "atoms" of data types, i.e., they are data types that cannot be defined in terms of other, simpler types. Programming languages provide software engineers with specific basic, or primitive, types and provide operators (e.g., addition, subtraction, etc.) specific to those types. It is important to remember that operators are truly specific to types: the meaning of the addition operator when applied to integers is much different than the meaning when applied to characters. In some cases, a given operator will only apply to one specific data type.

The number of data types varies significantly from language to language. LISP, for example, only has one type: symbolic or S-expression, while C has five primitive types: *character (char), floating point (float), double floating point (double), integer (int),* and *valueless (void).* In some languages (including C), types may be modified (qualified), e.g., as *signed, unsigned, short, long,* etc.

Now, let's look at some individual primitive data types more closely.

Integer

Integers, of course, are simply whole numbers (no fractional part), such as 7, -14, 3675, and so on. The length of the data representation for an integer in a computer's memory is usually determined by the machine's architecture. For a current PC running a C program, a **short int** is usually 2 bytes (16 bits), and an **int** and **long int** are 4 bytes (32 bits).

The qualifiers **signed** or **unsigned** determine whether or not positive and negative integer values are allowed. The **range** of allowed values depends on the machine and the programming language implementation. For C running on a PC, the allowed range of a **short int** integer is: -32,768 to +32,767. For an **unsigned short int**, the range is 0 to 65,536.

The specific representation of integers in a computer's memory is dependent on the machine's design. There are three representations commonly used:

- ***signed integers:*** A **short int** representation of the decimal number +5 in signed-magnitude form is

  0 00000000 00000101, and for -5 it is 1 0000000 00000101. Note that the first underlined bit is the sign bit; a 0 means that the integer is positive, and a 1 indicates that it is negative. Also, note the total of 16 bits (2 bytes) used to represent the number.

- ***2's complement:*** In this representation, +5 = 00000000 00000101, and -5 = 11111111 11111011. The negative representation is obtained by *complementing* each bit (1s become 0s, 0s become 1s), and then adding 1 to the complement.
- ***1's complement:*** In this representation, +5 = 00000000 00000101, and -5 = 11111111 11111010. The negative representation is obtained by just complementing each bit; the step of adding 1 is skipped. The disadvantage of 1's complement is that the integer zero (0) has two different representations: +0=0000000000000000, and -0=1111111111111111.

Most computers use the 2's complement representation, although some older mainframes use 1's complement.

Floating Point

Floats have two advantages over ints: they can hold a much larger *range* of numbers, and they can hold numbers with a fractional part, e.g., 3.14159265. By using fractional and exponent parts in the representation, a much larger range of values can be accommodated (compared with the integer representation). The "specs" on floating point are:

Parameter	Single	Double
Word width in bits	32	64
Exponent width in bits	8	11
Range (decimal)	10^{-38}-10^{+38}	10^{-308}-10^{-+308}
Number of values	$1.98x2^{31}$	$1.99x2^{63}$

Note the tremendous range of numbers (from exceedingly small to huge) that double floats can accommodate.

Most modern programming languages and machines represent floats according to the IEEE 754 Standard**.**

Binary-Coded Decimal (BCD)

Binary-coded decimal (BCD) is a straightforward method of representing decimal numbers. With BCD, every decimal digit is coded using 4 bits (1/2 byte). As an example:

4093 = 0100 0000 1001 0011
(4) (0) (9) (3)

The advantage of BCD is that it allows "infinite" precision, i.e., decimal numbers of any size can be represented. The drawbacks are that BCD wastes memory, and arithmetic operations can be very time consuming.

Boolean Types

It is often important in programs to be able to label something as "true" or "false." For example, reading a file from a hard drive will be successful ("true"), or it will fail ("false"). Hence, many programming languages accommodate Boolean data types that can store these two values.

Computers could use just one bit to represent Boolean values, e.g., 1 = true, and 0 = false. However, as we've said earlier, most machines are byte-oriented, so typically an entire byte is used to represent a Boolean data type. Using an entire byte, true = 00000001, and false = 00000000. The C language actually assumes that any non-zero value is true.

Characters

Like all other data, characters are represented in computers by binary strings. A common binary representation of characters is ASCII (American Code for Information Interchange). 7-bit ASCII codes (0-127 decimal) are used to store printable characters (alphanumerics) as well as control (non-printable) characters (such as line feeds and carriage returns). When more characters are needed, 8-bit codes are used.

An alternative to ASCII, Unicode , provides a more modern and much more powerful representation of characters, and is sufficiently versatile to accommodate international character sets.

Character Strings

Character strings have the interesting property of being primitive data types in some languages (e.g., Java, FORTRAN 77 & 90, and BASIC), but arrays of primitives in other languages (C, C++, Pascal, Ada, etc.). As the name implies, a character string is nothing more than a sequence (or array) of characters.

Strings need not be of fixed length; hence, some sort of delineator is required to mark the end of a character string. C and C++ use the NULL character to terminate strings.

We mentioned **descriptors** at the beginning of the session as the collection of data-type attributes. Strings can be stored as static (fixed length) or dynamic (variable-length) arrays.

Ordinal Type

Ordinal implies an association of objects or values with the set of positive numbers. There are two major categories of ordinal data types: **enumeration** and **subrange** types.

An *enumeration type* simply lists all possible values of a variable as symbols, e.g.,

```
type car is (Honda, Camry, Chevrolet, Dodge)
```

Here, the variable is car, and car may assume any of the values (Honda, Camry, etc.) shown. Internally (in memory), enumeration types are usually stored as integers starting with 0. In the example above, car would have a value of 0 for Honda, 1 for Camry, etc., but would be referenced by the program as Honda, Camry, etc. The primary purpose for using enumerated types is the enhanced readability that results.

Subrange types constrain values of a parent type to a specified range. Pascal only allows subranges for ordinal types, while Ada allows subranges for fixed and float types. An example of an Ada subtype is:

```
subtype INDEX is INTEGER range 1..10;
```

This statement simply means that the variable INDEX is an integer that is allowed to have a value from 1 to 10.

Arrays

An **array** is a collection of elements of homogeneous (identical) type. The entries in an array are selected by an index or subscript. Depending on the language, the index is usually enclosed in brackets or parentheses:

```
AGE(INDEX) = 49       {FORTRAN}
age[index] = 49;      {C}
```

Parentheses are also used by many programming languages to enclose subroutine and function arguments, e.g., X = sqrt (3.1415), where sqrt is the math square-root function. This can be confusing to a compiler if the language also uses parentheses for arrays. The programming language compiler resolves this problem by using a table of declared arrays to distinguish arrays from functions/subroutines. If parentheses are encountered, the array table is searched for the name preceding the parentheses; if the name is not found, the compiler assumes that it is a function or subroutine.

Memory (RAM) must be made available to store arrays; when memory is allocated (made available) to store an array, we say that memory is ***bound*** to the array. Memory can be bound to arrays either at compile time (static allocation) or at run time (dynamic allocation). Static allocation can be used if the size of an array is known prior to running a program; if the size is unknown, dynamic allocation (while the program is running) is required. For the case of dynamic allocation, there are two sources of memory: the **heap** and the **stack.** The heap is simply an area of unused memory that can be allocated in hunks, and later recovered when it's no longer needed. The stack, as the name implies, is a push-down list of memory locations. An item removed from the stack is the last item placed on the stack (i.e., last-in, first out).

The ways of allocating memory space to arrays include:

- ***Static***: the subscript range and memory storage are bound at compile time
- ***Fixed stack - dynamic:*** the subscript range is fixed, but storage is allocated during run time when the array is declared; this technique allows the reuse of the memory.
- ***Stack-dynamic***: the range and memory storage are bound during run time, but remain fixed during the lifetime of the array.
- ***Heap-dynamic:*** the range and memory storage are bound during run time, and can change (increase or decrease) during the lifetime of the array.

FORTRAN 77 and earlier versions used only static arrays. C allows heap-dynamic arrays using library functions ***malloc*** and ***free.***

Most programming languages allow *multidimensional* arrays, e.g., a 2-dimensional (rows and columns) array used to store a table. Such multi-dimensional arrays are either stored in memory by row (***row major order*)** or by column (***column major order***).

Arrays, like other data types, are bound to descriptors.

Records

A ***record*** is an aggregate storage structure in which the entries may be heterogeneous. Consider a record used to store employee information: name, date of birth, social security number, salary, etc. These items are of various different primitive data types: integer for age, string for name, etc. Records are convenient because they permit grouping information concerning a particular item (object). To summarize the differences between records and arrays:

Arrays	**Records**
Homogeneous elements	Heterogeneous elements
Elements referenced by index	Fields referenced by identifiers

Some languages (e.g., Ada) use the "dot" notation to access fields of a record; for an employee record, the employee's name would be accessed as:

```
employee.name
```

The obvious benefit of this syntax is that it is very readable and maintainable.

Unions

Unions allow the storage of more than one type of variable in the same memory location (at different times). As an example, consider the format of the double float that we looked at earlier. Double floats require 64 bits, or 8 bytes, of memory, which is a considerable amount. A union type would allow the 8 bytes allocated to store a double float to be used also to store two 4-byte floats, though not at the same time. Unions must have sufficient storage space for the largest expected data type.

```
type
        intReal = union
  i: integer;
          r: real;
end union;
          .
          .
     var
x: intReal;
        y: real;
```

```
y:= x.r;
x.i:= 3;
```

In this example, we've declared a union type called intReal that is designed to hold a single integer or real (float) number. At the bottom, we declare a variable x as type intReal (i.e., a union). The statement y:= x.r; then treats the union as a real; the last statement x.i:= 3; treats the union as an integer. Keep in mind that that the last statement will overwrite the float, since they are both occupying the same memory cells.

The small C program included in this session's lab includes a union; take a look at the code and see if you can understand its design.

Sets

A set is any unordered collection of distinct elements, unlike arrays which must be ordered. Consider this Pascal example (Appleby and VandeKopple, p. 55):

```
  type
        intSet = set of 1 .. 10; {intset is the basetype}
var
        s: intSet
begin
        s:= [1, 3, 5, 9]
 ...
end
```

In this example, we've defined a type called intSet that consists of any of the integers 1 to 10. We then declared a variable s of type intSet, and then set s equal to a particular subset of the type.

Sets are usually implemented with bit strings in memory to show which base type elements are in the set; e.g., in the above example, s would be represented in memory as 1010100010 (bits are set to 1 at positions 1, 3, 5, and 9).

Pointers

A ***pointer*** type contains the location (address) of a data object, e.g., a character or an array. Pointer values are addresses of other objects, similar to the indirect addressing scheme used in assembly or machine language. Pointers are also known as ***reference*** or ***access*** types in some languages. If a pointer is not in use, it is set to nil (usually 0, and sometimes called null rather than nil). Pointers are usually associated with a single type, e.g., an integer pointer (contains the address of an integer variable), or an array pointer (contains the address of the the first value of an array). To make life really complicated, it is possible to have a pointer to a pointer.

Let's take a look at an example. Suppose "i" is an integer variable containing the value 12 at memory location 1876. Schematically, using square brackets to represent a memory cell,

i [12] {at memory location 1876}

Read the above as: "the integer 'i', of value = 12, is located in memory cell 1876. Now, let's introduce a pointer "p" at memory location 1548 that points to the integer 'i'."

p[1876] {at memory location 1548} ------>i[12] {memory location 1876}

Read the above as: " the pointer 'p', of value = 1876, is located in memory cell 1548 and points to the variable 'i'." Pointers are a very powerful feature of some programming languages (e.g., C and C++). They are useful in accessing fields in complex data structures as well as particular characters in a string. They are also useful for accessing newly allocated blocks of dynamic memory. Unfortunately, the use of pointers is fraught with dangers, such as:

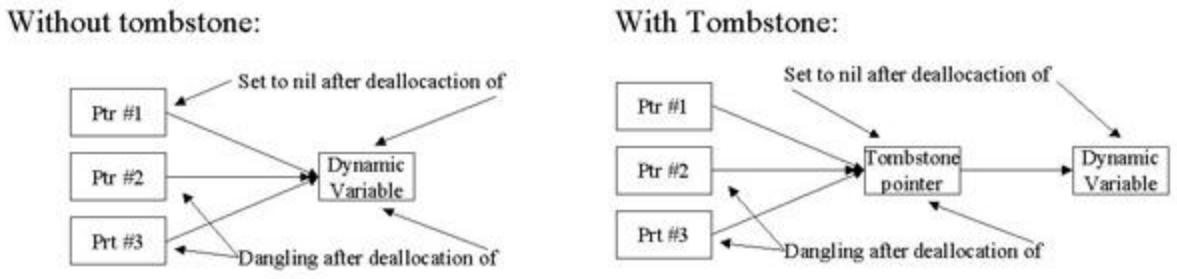
- **Dangling Pointers**: pointers to variables that have been deallocated (i.e., variables no longer needed by the program). The pointer will point to whatever (random) new object that is in the old location. This normally happens when a pointer value is assigned to another pointer, and the object being pointed at is deallocated. The original pointer will be set to nil upon deactivation, but the assigned pointer will not.
- **Lost heap-dynamic variable**: A pointer is set to point at an object in the heap; the pointer is then re-assigned to point elsewhere. The result: there is no longer a way to reference the object in the heap, rendering that part of memory unusable. This effect is known as *memory leak*.
- **Illegal index**: Arrays are often accessed by off-setting pointers, e.g., newptr = ptr + index, where ptr points to the first element of an array. By incrementing the variable index, one can access each element of the array. However. there is no guarantee that areas outside of the array won't be accidentally accessed, and perhaps, overwritten.

The dangers of pointers were aptly summarized by Hoare in 1973: "*Their introduction into high-level languages has been a step backward from which we many never recover."* In the terminology that we've used in earlier sessions, pointers adversely affect the reliability of a programming language.

Pointers have other uses. C allows pointers to functions (as opposed to variables). This can be a useful (albeit dangerous) means of passing functions as parameters to subprograms.

Given the utility of pointers, there have been attempts to render them less dangerous. There are two common approaches to mitigate their dangers:

- **Tombstones**: the idea is, rather than point directly at an object (integer, array, or whatever), point instead to a "tombstone" object which in turn points to the desired object. Tombstones are intended to solve the problem of dangling pointers, i.e., pointers to deallocated memory. When memory is deallocated, the tombstone pointer itself is set to nil, ensuring that any dangling pointers will also be pointing to a nil location.



The primary drawbacks of tombstones are two-fold: memory use is increased (the tombstone itself takes memory space), and performance is impacted due to the double layer of pointers.

- **Lock and Key**: With this approach, a lock is associated with every dynamic variable, and a key with every pointer to that variable. When a dynamic variable is deallocated, the lock is changed so that no pointer can gain access to the object.

As we've mentioned above, an important use of pointers is for memory allocation and management. Dynamic memory allocation happens during the execution of a program, i.e., memory is allocated from the heap during run time. Dynamic-heap allocation leads to the problem of heap management, i.e., reclaiming deallocated memory for reuse. Memory management is generally based on a table that marks memory cells as available or unavailable. Two approaches are:

- **Reference counter method**: Associate each memory location with a counter that shows the number of pointers to that cell. When the number of pointers equals zero, mark the memory location as available. This is an expensive approach -- lots of extra memory is used, and there is significant run-time overhead.
- **Garbage collection**: The approach is to wait until available memory is gone, and then to check the pointers that are in use. Memory locations with no pointers are marked "available" for reallocation. This is a time-consuming process.

If memory is allocated in variable-sized "chunks," a more sophisticated approach is needed. Dynamic memory can become fragmented (like a hard disk); defragmentation involves moving data objects around to maximize large contiguous hunks of memory.

Wrap Up

This session we've look at how data is represented and stored in computers. We've looked at the simplest (primitive) data types such as integers and floats, as well as more complex types such as structures and unions. We've looked at pointers -- their power for accessing data and for implementing memory management -- as well as their considerable dangers.

The lab session this week will let you look at how primitive data types are stored in your PC.

Next session, we'll look at two basic programming language constructs: expressions and assignments.

Recommended Readings:

Programming Languages: Paradigm and Practice
by Doris Appleby and Julius J. VandeKopple; 444 pages,
Published by McGraw-Hill Companies; 1997; ISBN 0-07-005315-4

Sams Teach Yourself Data Structures and Algorithms in 24 Hours
by Robert Lafore; 523 pages,
Published by Sams; 1999; ISBN: 0672316331