COMP 356
Programming Language Structures
Notes for class 5: Data types

Acknowledgment: these notes are based closely on an earlier version by Prof. Tim Wahls.

A **data type** consists of:

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

Approaches to data typing in language design:

- include only a minimal set of data types (Fortran IV)
- include a rich set of built-in types and operators (PL/1)
- include a few built-in types along with type constructors that allow programmers to define their own types (Pascal, C, Java, ...)
- include facilities for defining abstract data types (Java, C++, Ada 95)

Programming language data types are divided into 2 categories:

- **primitive** (**basic**) types
  - size (in memory) is fixed by the language (or implementation)
  - are not defined in terms of other types
  - examples: numeric types, characters, booleans, ...
- **structured** (**constructed**) types
  - defined in terms of other types
  - size can often vary
  - **layout** (organization in memory) is interesting
  - examples: arrays, records, classes, ...

Some types (pointers, references, enumerations, subranges, ...) don't fit nicely into either category.

1 Strings

- usually a constructed type
- common string operations
  - concatenation
  - comparisons
  - pattern matching
    - usually uses some form of regular expressions
    - used for both recognizing and decomposing strings
  - assignment
  - copying (== assignment?)

String length options:

- **static length strings**, in which the length of strings is fixed
  - only works well for immutable strings (that can't be modified)
  - examples: strings in Java and C#
  - are easy to implement, but inflexible
  - each string operation must create and return a new string (inefficient)
- **limited dynamic length strings**, in which the maximum length of strings is fixed (but shorter strings can be stored)
  - examples: strings in C (as arrays of characters)
  - require the programmer to be careful about string lengths
  - often use an EOS character such as `'\0'` to mark the end of meaningful content in the string
  - are more efficient (time) and easy to implement
- **dynamic length strings**
  - examples: strings in JavaScript and Perl
  - are very flexible and easy to use
  - require resizing at runtime (expensive in time and memory)

2 User-Defined Ordinal Types

An **ordinal type** is any type whose values can easily be mapped to the set of positive integers. Typical examples include: int, char, bool. User-defined ordinal types include enumerations and subranges.

2.1 Enumeration Types

An **enumeration type** is a type in which all possible values are listed in the declaration. Example (C++):

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

The values are called **enumeration constants**.

A program can then:

- declare variables of type `stoplight`
- assign any of the constants `red`, `green` or `yellow` to such a variable
- compare variables of type `stoplight` to these constants in conditions
- ...

Using such a type makes programs easier to understand than if integers are used to encode possible values.

Design issues with enumeration types:

- can enumeration constants and variables of enumeration types be treated as integers? (yes in C++, no in Ada, C#)
- can the same enumeration constant be a member of multiple enumeration types (yes in Ada, no in C++)

Enumeration types are included in C, C++, C# and Ada (but not in Java 1.4 and earlier).

In languages without enumerated types, they can be simulated using named constants. E.g. (Java 1.4):

```
public interface Coin {
  public static final int PENNY = 1;
  public static final int NICKEL = 2;
  public static final int DIME = 3;
  public static final int QUARTER = 4;
}
```

This allows programmers to write (for example):

```
  int coin = 0; // legal, but conceptually a type error
  // some computation with coin
  if (coin == Coin.PENNY) {
...
```

which is more readable and less error prone than

```
if (coin == 1) {
```

However, the constants are still integers and can be treated as such in programs.

In Java 5.0, enumerated types:

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

Example (adapted from java.sun.com sample code):

```
// file: Coin.java
public enum Coin {
    penny(1), nickel(5), dime(10), quarter(25); // the possible values as constructor calls
    Coin(int value) { this.value = value; }
    private final int value;
    public int value() { return value; }
}

// file: CoinTest.java
public class CoinTest {
    public static void main(String[] args) {
        for (Coin c : Coin.values()) { // new Java 5.0 for loop syntax
            System.out.println(c + ":   \t"
                  + c.value() +" \t" + color(c));
        }
    }

    private enum CoinColor { copper, nickel, silver } // simple (C++ style) enum declaration

    private static CoinColor color(Coin c) {
        switch(c) {
          case penny:   return CoinColor.copper;
          case nickel:  return CoinColor.nickel;
          case dime:
          case quarter: return CoinColor.silver;
```

```
            default: throw new AssertionError("Unknown coin: " + c);
          }
      }
  }
```

2.2 Subrange Types

A **subrange type** is a contiguous subsequence of an ordinal type. Example (Ada):

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

`SmallInts` is then a type name, and so can be used in variable declarations . . .

Subranges:

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

Advantages of subrange types:

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

3 Arrays

Arrays:

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

Allocating memory for an array consists of choosing a **base address** (location for the first element) and reserving sufficient following locations for the rest of the array elements.

3.1 Kinds of Arrays

Arrays can be categorized by their allocation and layout times. The layout of an array is known when the number of dimensions, the size of each dimension and the size of the base type is known.

1. **static arrays**
    - allocation and layout are static
    - examples: global and static array variables in C++ declared with a fixed size
2. **fixed stack-dynamic arrays**
    - allocation is dynamic (at declaration elaboration time)
    - layout is static
    - example: local array variables in C++ declared with a fixed size
    - advantage over static arrays: local array variables in different functions can use the same memory (share storage)
3. **stack-dynamic arrays**

- allocation and layout is dynamic
- the size (layout) can not be altered during the variable's lifetime
- storage is allocated on the stack

4. **fixed heap-dynamic arrays**
   - like stack-dynamic arrays, except that storage is allocated on the heap
   - example: array variables in C++ allocated using `new`, all Java arrays

5. **heap-dynamic arrays**
   - allocation and layout is dynamic
   - the size (layout) can change dynamically
   - examples:
     - Perl arrays (hashes)
     - C and C++ arrays that are resized using `realloc`

3.2 Rectangular and Jagged Arrays

A **rectangular array** is a multidimensional array in which all rows have the same length, all columns have the same number of elements and so on. For example, a C++ array declared as:

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

is a rectangular array.

A **jagged array** (or **ragged array**) is a multidimensional array in which the length of rows (columns, etc.) need not all be the same. For example, a Java array constructed as follows:

```
int [][] rArray = new int[3][];

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

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

3.3 Array Layout and Element Address Calculation

The compiler must generate code for each array element reference (index expression) to compute the address of the referenced element. Computation of array element addresses:

- can be done relative to the base address after layout
- can be done absolutely after allocation

Consider a general array variable declaration:

`var` A`:` `array` [low `..` high] `of` basetype

Let w = the size of the basetype
$\quad base$ = the base address

The address of A[k] is $base$ + (k - low) $*$ w
This is usually expressed as:

($base$ - low $*$ w)	+	k $*$ w
computed once at allocation time		computed for each reference at runtime

In C, C++ and Java, low $\equiv$ 0, so this formula becomes $base$ + k $*$ w

For multidimensional arrays, array elements must be "mapped" into 1-dimensional memory. Logically, a 2-dimensional array such as:

`var` A: `array` [low$_1$ `..` high$_1$] [low$_2$ `..` high$_2$] `of` basetype

is laid out as:

A_{low_1,low_2}	A_{low_1,low_2+1}	...	$A_{low_1,high_2}$
A_{low_1+1,low_2}	A_{low_1+1,low_2+1}	...	$A_{low_1+1,high_2}$
...	...	...	...
A_{high_1,low_2}	A_{high_1,low_2+1}	...	$A_{high_1,high_2}$

There are two common layouts for this:

- **row major layout**
- **column major layout**

3.3.1 Row Major Layout

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

A[low$_1$]				A[low$_1$ + 1]				
A_{low_1,low_2}	A_{low_1,low_2+1}	...	$A_{low_1,high_2}$	A_{low_1+1,low_2}	A_{low_1+1,low_2+1}	...	$A_{low_1+1,high_2}$	...

- the second index varies faster
- A[low$_1$] (for example) is a single dimensional array and can be treated as such in C, C++ and Java

The address of A[i][j] (for rectangular arrays) is computed as follows:

Let w_2 = the size of the basetype
$\quad w_1$ = the "width" of a row
$\quad\quad$ = ((high$_2$ - low$_2$) + 1) $*$ w_2
$\quad\quad\quad$ (note that this can be computed at layout time)
$\quad base$ = the base address

Then the address of A[i][j] is:

$$base \quad + \quad \underset{\text{skipping rows}}{(\mathrm{i} - \mathrm{low}_1) * w_1} \quad + \quad \underset{\substack{\text{skipping}\\ \text{columns}}}{(\mathrm{j} - \mathrm{low}_2) * w_2}$$

which is usually expressed as:

$$\underset{\text{computed at layout time}}{(base - \mathrm{low}_1 * w_1 - \mathrm{low}_2 * w_2)} \quad + \quad \underset{\substack{\text{computed for}\\ \text{each reference}}}{\mathrm{i} * w_1 + \mathrm{j} * w_2}$$

3.3.2 Column Major Layout

Under column major layout, the columns are laid out "side by side" in memory. For the previous example (the 2-dimensional array A), this becomes:

A[ ][low_2]				A[ ][low_2 + 1]				
$A_{\mathrm{low}_1,\mathrm{low}_2}$	$A_{\mathrm{low}_1+1,\mathrm{low}_2}$	$\cdots$	$A_{\mathrm{high}_1,\mathrm{low}_2}$	$A_{\mathrm{low}_1,\mathrm{low}_2+1}$	$A_{\mathrm{low}_1+1,\mathrm{low}_2+1}$	$\cdots$	$A_{\mathrm{high}_1,\mathrm{low}_2+1}$	$\cdots$

i.e. the first index varies faster.

3.3.3 Implications of Array Layout

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

- when doing pointer arithmetic
- to minimize paging when accessing all elements of an array
- when passing subarrays as parameters (slices)
- when passing multidimensional arrays as parameters. For example, in C and C++ a formal parameter of a multidimensional array type must include all dimensions except the first. E.g:

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

  because all dimensions except the first are used in address calculations for row-major layout.

A jagged array is stored as an array of pointers or references to other arrays. Hence, an array reference such as `rArray[2][3]` (for a 2 dimensional jagged array `rArray`) is resolved as follows:

- treating `rArray` as a single dimensional array and so finding the pointer or reference stored at index 2 using techniques already discussed. (Note that each element of `rArray` is of a fixed size since it is a reference or pointer.)
- following that pointer or reference to find the "contained" array
- finding index 3 within that array using techniques already discussed

3.4 Associative Arrays

An **associative array** is an unordered collection of data, where each element is indexed by some **key**. The keys can be nonordinal types such as strings, reals and instances of classes.

Associative arrays:

- are like "lookup" tables
- are often implemented as hash tables since the keys need not be ordered or discrete
- are efficient for accessing individual elements, but not for processing all elements systematically
- are found in languages such as:
  - Perl (hashes)
  - Java (maps)

4 Records

Records:

- are nonhomogeneous collections of data
- have named data elements called **fields**
- are similar to (but simpler than) class instances
  - all instance variables are public
  - no methods
- are heavily used in languages such as Cobol and C (`structs`)

Record layout:

- is static
- is usually done by putting all fields in contiguous memory to make allocation and addressing easier
  - each field is a fixed offset from the base address
  - these offsets are computed at compile time

The size of a record is the sum of the sizes of all fields plus any required padding (rare).

5 Unions and Variant Records

Union types can store any of some fixed set of other types. For example, in C++:

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

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

Unions are primarily used when data of different types must be stored in the same collection. For example, consider defining an array of shapes, where each kind of shape (circle, square, parallelogram, ...) has different fields.

In many languages (Pascal, Ada), unions must always be embedded in records. This creates a **variant record** whose values have:

- a common part (with the same type structure for all values of the variant record type)
- a variant part (with a different type structure for different values of the type)

For example, all shapes might have a location and a color, but other fields would differ. In C and C++, a variant record can be implemented using a `struct` or `class` with a `union` field or data member.

The size of a union is the size of the largest alternative to ensure that a variable of the union type can hold any value that can legally be assigned to it. This uses less space than allocating separate storage for each alternative. The size of a variant record is the sum of the size of the common part and the size of the largest alternative in the variant part.

In many languages (C, C++, Pascal), some type errors involving unions can not be detected, and so unions prevent the language from being strongly typed. For example:

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

...

  u.i = 3;

  ...

  std::cout << u.d;
```

This prints `u` just as if it contained a double.

To enable typechecking of unions:

- each union type must include a **tag** (or **discriminant**) to indicate the type it currently contains
- tags must be checked dynamically to ensure that usage is consistent with the type

So some dynamic type checking is required, even in a statically typed language.

A union with a tag is called a **discriminated union**.

Pascal variant records are discriminated, but:

- the tag is optional
- the tag can be changed without changing other fields
- most compilers don't generate code to check tags anyway

So Pascal is not strongly typed.

In Ada:

- tags are required in variant records
- compilers are required to generate code to check tags
- tags must be set before any other field can be assigned to
- tags can only be changed (after initialization) by assigning one entire variant record to another

So variant records do not keep Ada from being strongly typed. (In fact, Ada is strongly typed.)

Java and C# do not have unions.

Unions are dangerous because they can lead to:

- undetected type errors
- lack of portability

6 Pointers and References

6.1 Pointers

Pointers:

- can contain addresses or NULL
- are typically implemented as address values stored in 2, 4 or 8 byte memory cells
- uses:
  - data structures that change size dynamically
  - reference parameters
  - efficiency in parameter passing
  - indirect addressing/access

Recall that **heap-dynamic variables** are variables that are dynamically allocated from the heap.

- in Pascal, all "variables" referred to by pointers are heap-dynamic
- in C++, variables referenced by pointers may or may not be heap-dynamic (because of `&`)

operation name	C++ notation	C notation (if different)
dereferencing	`*`	
assignment	`=`	
allocation	`new`	`malloc()`
deallocation	`delete`	`free()`
changing size of allocated memory		`realloc()`
arithmetic	`+`, `-`, `++`, ...	

Table 1: Operations on Pointers.

Additionally, the address-of operator (`&`) returns a pointer.
Notes on pointers:

- a pointer is **dangling** if it refers to memory that isn't allocated. Pointers are dangling:
  - before memory is allocated
  - after memory is deallocated
  - when a pointer into an array is incremented past the end of the array (C, C++)
  - when a function returns a pointer to a (nonstatic) local variable

  The following program creates a more subtle dangling pointer:

```
int *p1, *p2;

p1 = new int;
p2 = p1;
...
delete(p1);
```

p1 is dangling after the call to delete, but so is p2.

Dereferencing a dangling pointer is dangerous and unpredictable.

Because pointers in Pascal always refer to memory on the heap, deallocation (`dispose()`) is the only operation that can cause a pointer to become dangling.

- **garbage** is memory that has been allocated but is now inaccessible. Garbage is created whenever all references to an allocated heap-dynamic variable are lost, and so such variables are called **lost heap-dynamic variables**.

  The following program creates garbage:

  ```
   int *p1, *p2;

   p1 = new int;
   p2 = new int;

   p1 = p2;
  ```

  because the last assignment statement causes all references to the memory that was originally pointed to by p1 to be lost.

- a program that creates garbage has a **memory leak**
- C and C++ treat arrays as pointers to blocks of memory, so pointers can be used to access array elements in those languages. For example:

  ```
  int *iptr, iarr[10], i;

  // add values to iarr here
  iptr = iarr; // legal and equivalent to: iptr = &iarr[0];
  // printing the contents of iarr
  for (i 0; i < 10; i++) {
    std::cout << *iptr << std::endl;
    iptr = iptr + 1; // pointer arithmetic
  }
  ```

  After the assignment `iptr = iarr;`, the last element of `iarr` can be accessed with either `iarr[9]` or `*(iptr + 9)`. Note that the value added to `iptr` before dereferencing is 9 times the size in memory of an integer, not simply 9.

6.2 References

In general, reference types are restricted forms of pointer types. They have the same representation as pointers, but do not support the full range of pointer operations, and so are usually safer to use than pointers.

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

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

Because references are constants:

- they must be initialized with an (implicit) address when they are declared

- the value of the reference itself can not be changed

This prevents any form of pointer arithmetic or other means of changing the address stored in the reference. However, the value stored in the memory location that the reference "points to" can change:

```
int i = 3;
int &r = i;  // makes r a reference to i (a constant pointer to
             // i's memory location)

r = 4;

std::cout << r << std::endl; // prints 4
std::cout << i << std::endl; // prints 4
```

In particular, a reference (almost) always creates an alias. In this example, the names `i` and `r` always refer to the same memory location.

References allow functions to modify parameters without using pointer syntax. For example:

```
#include <iostream>
void foo(int& r) {
  r = 3;
}

void foo(int *r) {
  *r = 3;
}

int main() {
  int x = 1, y = 1;
  foo(x);
  foo(&y);
  std::cout << x << std::endl; // prints 3
  std::cout << y << std::endl; // prints 3
}
```

In Java, references:

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

This design has a number of implications:

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

  ```
  Integer i1 = new Integer(3);
  Integer i2 = new Integer(4);

  i1 = i2;  // now i1 and i2 are aliases, and the instance previously
            // referred to by i1 is garbage
  ```

  So a variable of a class type denotes a reference, and not an instance of the class directly.

- pointer arithmetic is not sensible
- instances of classes are always passed to functions by reference (unless the keyword `final` is used in the parameter declaration of the function). Values of nonclass types are always passed by value.

Java does not have a deallocation operator. When a memory cell becomes garbage, it is the responsibility of the runtime system to find and deallocate that cell. The advantages of this approach are:

- no reference is ever dangling (unless the runtime system contains an error)
- the programmer need not be concerned with determining when it is safe to deallocate a memory cell (or with dangling references)

C# includes both pointers and Java-style references, but the use of pointers is discouraged.

6.3 Heap Management

The heap is usually maintained as a linked list of available memory cells, called the **free list**.

- in general, the memory cells are of different sizes
- when program execution begins, the free list contains only one cell (the entire heap)
- when an allocation request occurs:
  - the free list is searched for a memory cell large enough to satisfy the request
  - any portion of the cell not used is linked back in to the free list
- when deallocation occurs, there are two possibilities:
  - the free list could be searched for a cell adjacent in memory to the deallocated cell, and if such a cell is found, the two cells are merged on the free list.
  - or, the deallocated cell could simply be linked in to the free list. This approach eventually causes the free list to contain a large number of small cells, and so at some point no cell will be large enough to satisfy a request. When this occurs, the free list must be scanned to merge cells that are adjacent in memory.

Deallocation of heap-dynamic variables may be:

- explicit (controlled by the programmer). This is the case in C++ and Pascal, where the programmer must explicitly call an operator (such as `delete`) to deallocate heap storage.
- implicit (controlled by the runtime system for the programming language). This is the case in Java and Scheme, where the runtime system must determine when memory cells become garbage and so can be deallocated.

There are two primary approaches to implicitly deallocating storage (**garbage collection**):

- **reference counting**
  - each memory cell contains a count of the number of pointers or references that refer to it (the reference count)
  - each time a pointer value is changed (assigned to), the reference count of its "old" value is decremented and of its "new value" is incremented
  - when the reference count of a cell drops to 0, that cell is garbage and can be deallocated
  - this approach is **eager** - a cell is deallocated as soon as it becomes garbage
- **mark-scan** (or **mark and sweep**)

  - each cell contains 1 extra bit (the **mark bit**) that is used to indicate whether or not the cell is garbage
  - during program execution, memory is allocated as needed with no attempts at deallocation
  - when the heap is exhausted (or nearly so), the following algorithm is executed:
    1. all cells have their mark bit unset (to indicate that they are potentially garbage)
    2. the **mark** phase: starting with all pointer/reference variables on the stack, all pointers/references are followed and all cells reached have their mark bit set (to indicate that they are not garbage)
    3. the **scan** (or **sweep** phase): all unmarked cells are deallocated

- this approach is **lazy** - garbage is not deallocated until more memory is needed