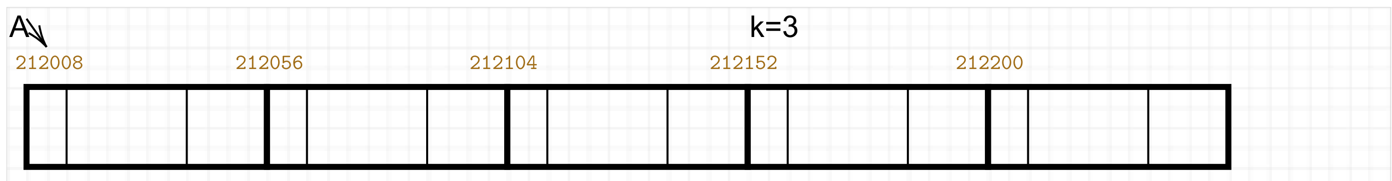
CSE305 Week 8 Tue.: Arrays Versus Lists / Records / Other Types

[First, following my old notes, which were based on Sebesta's older notes.]

Arrays

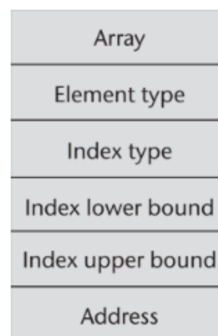
First fact: *Layout is Linear*, even for nested objects in the array. Using Ada's syntax, which is basically the most general possible:

```
A: array(lo..hi) of T ypeIKnowHowToLayOut;
```



Separate from this is the **compile-time descriptor** of the array---snipping Sebesta's figure:

Figure 6.4 Compile-time descriptor for single-dimensioned arrays



Ada does not require `lo` = 0 and makes `hi` inclusive. Most languages fix 0 as the lower index and have you declare a length `n`, which as an index is exclusive. **`T`** can be any type whose layout is known at the time the declaration of the array is **elaborated**. Let:

- **b** stand for the "base address" or "binding address" of the whole array storage object. Usually this is the same as the address of the first element, **`A(0)`** in Ada and Scala.
- **e** = `sizeof(T)`, the size of an element of type **`T`** in bytes. Above, **e** = 48.

The array must be **homogeneous** as far as the element size **e** is concerned. However, if the stored elements are all *pointers*, they can point to objects of different sizes in a hierarchy. The array is typed as the base class of the hierarchy.

The formula for accessing an element **`A(k)`**, where **`k`** is "between" `lo` and `hi` in the sense appropriate to the language (inclusive versus exclusive upper bound), is

b + (k - lo)*e = k*e + (b - lo*e).

The reason for arranging it the latter way is that the expression **(b - lo*e)** does not depend on **k** and so can be *precomputed*. Then locating **A(k)** in memory requires just one ***** and one **+** operation. Note also that if `lo` = 0 like in C, then the expression becomes simply

b + k*e.

This is still one * and one + operation, but it counts as *constant time*, and there is less pre-computation. For k=3 and e=48 above and b = 212008 we get address 212008 + 144 = 212152. (Earlier editions of Sebesta gave this formula, calling **b** the "base" address of the array.)

Arrays in OCaml

OCaml arrays use similar notation to lists, but without :: and with extra | ... | on the square brackets.

```
# let arr = [|3;4;5|];;
val arr : int array = [|3; 4; 5|]
```

Unlike with lists, OCaml arrays are mutable for individual entries. (!)

```
# arr.(1) <- 7;;
- : unit = ()
# arr;;
- : int array = [|3; 7; 5|]
```

The indexing operation uses parens and a dot. If you write **arr.(1) = 7** it is a Boolean test with an array access, not an assignment. The assignment as an operation returns **()**, called `unit`.

OCaml arrays are not natively resizable or appendable. Once created, their length is fixed. Nowadays that may seem a restriction after the general experience of using C++ `vector` and arrays in Java and C# and Python and Scala etc. Actually, the restriction is true of *native arrays* in basically all languages. The richer kinds of arrays come from library implementations that simulate the extra operations. Whether you consider `NumPy` arrays to be part of core Python is a matter of taste, but it is not the bedrock of the language design. OCaml has the `Array` module, which among several other things allows appending to create a *new* array:

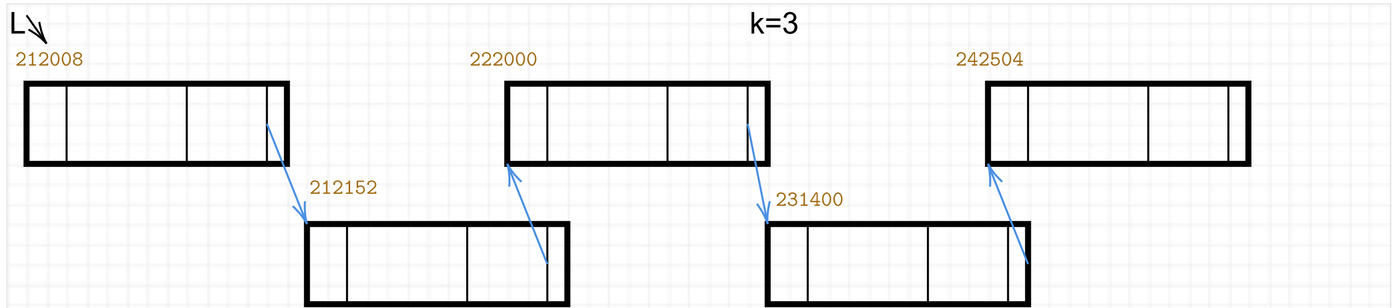
```
# let arr2 = Array.append arr arr;;
val arr2 : int array = [|3; 7; 5; 3; 7; 5|]
```

OCaml does not have "**+=**" like Scala does to append to an array in place---you have to create a new copy. Which can get expensive with new arrays... The Array module has conversion functions

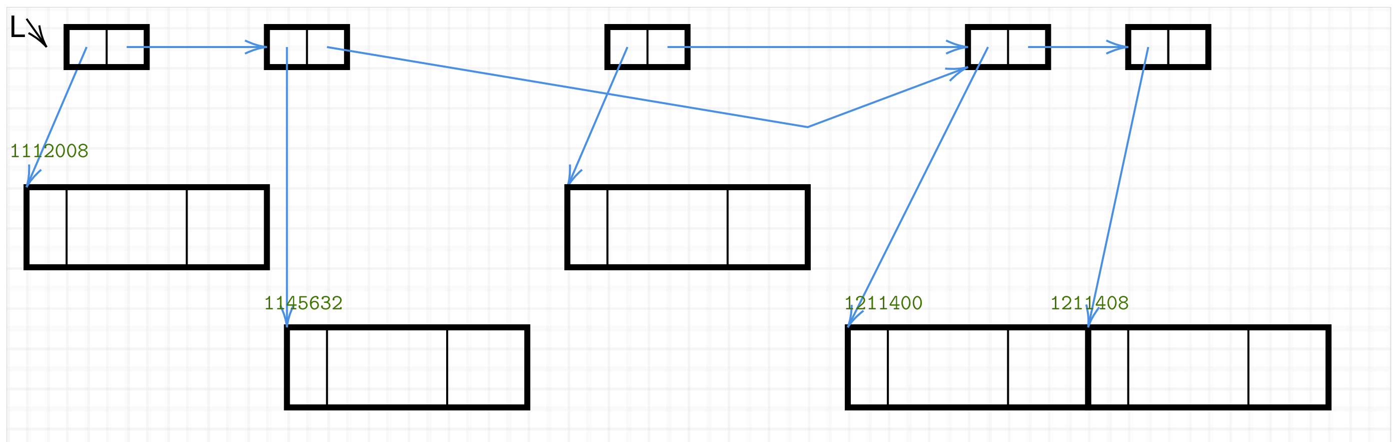
`Array.to_list` and `Array.from_list` so you can (expensively) emulate list operations that way. What's the real difference? In terms from CSE250:

- Arrays provide constant-time access to any indexed element.
- Lists provide constant-time appending, prepending, removal, and "splicing."

Under the hood, the memory layout of a list usually looks like a *linked list*:



or more usually via pointers into nodes strewn in the *system heap*:



Traversing the latter kind of list requires extra pointer accesses. But if you already have a pointer to the list's object, there is no extra cost. And nodes can be spliced in or out in $O(1)$ time. The diagram shows the third node being spliced out. Only a single pointer jump of the second node's **`next`** pointer needs to be executed to do this---the actual deletion of the bypassed node and two orphaned pointers can be executed later by **garbage collection** (or not done at all).

Added: Python calls its [...]-enclosed data structures "lists", but they allow indexing so can be treated as if they were arrays. So can lists in Scala. Another way Python lists are like arrays is that individual elements can be changed---whereas Scala lists (and Python tuples) have immutable items. However, neither Python's nor Scala's lists guarantee that indexed lookup occurs in $O(1)$ time, which is what the `NumPy` array gives. So in that sense, Python's "lists" really are more like lists than arrays. In a third sense they are *neither*---their objects need not have the same base type. Python can get away with this by using pointers to all the objects, per above diagram too.

Design Issues For Arrays

Here is Sebesta's list---the answers are a mix of mine and his:

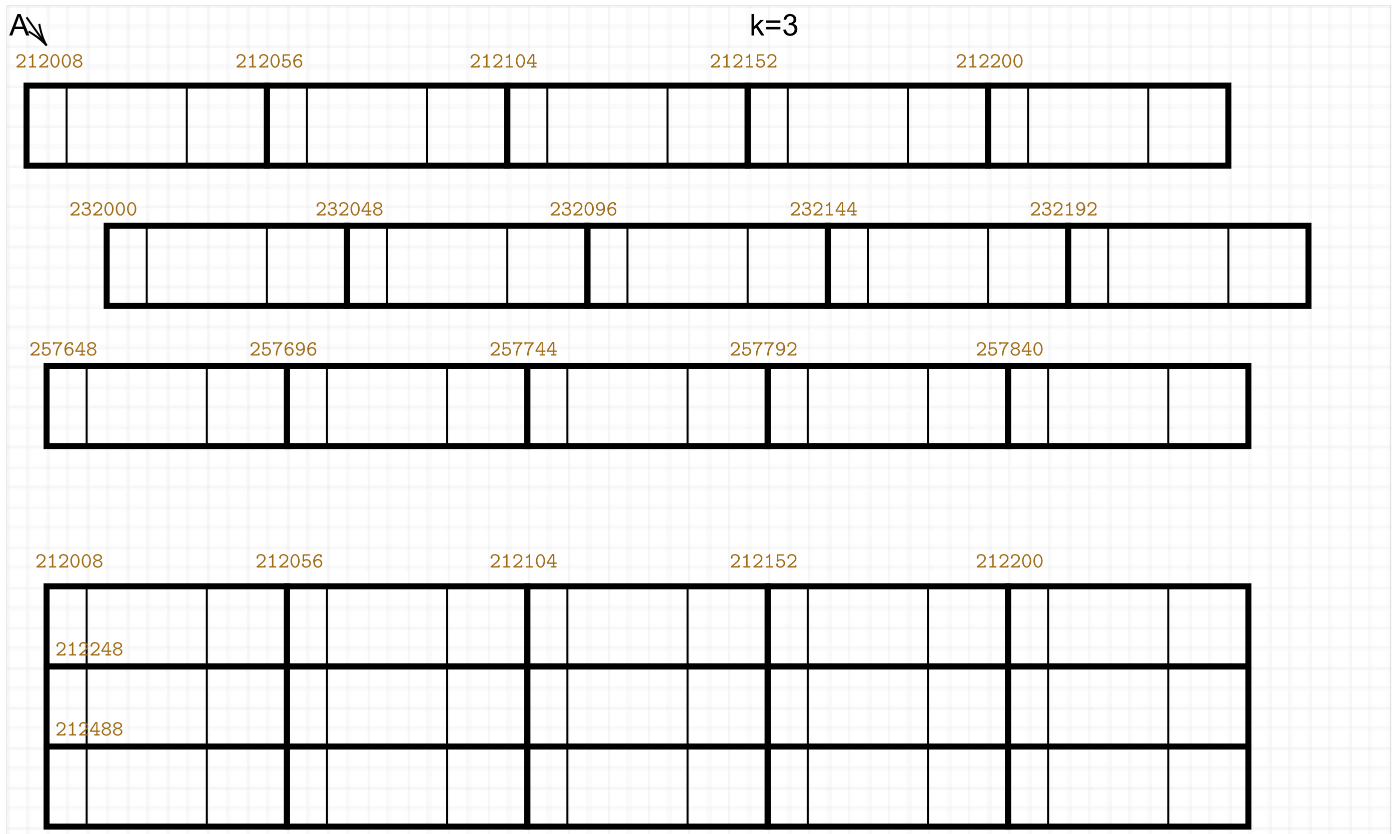
- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

1. Subscripts: Must they be `int`? or allow types convertible to int, like `size_t` in C++? Allowing an enumeration to index an array is a little weird, because the enum has compile-time fixed size, whereas the array has stack-dynamic size.
2. This used to be a bigger issue. Still optional in C/C++ but mandated in most newer languages. E.g. OCaml specifies at core language level that an index out of bounds raises the `Invalid_argument` exception.
3. In old FORTRAN etc., the bounds for all arrays had to be fixed at compile time. The real question is, "do array objects know their length?" Nowadays the answer is yes, so that variable-length arrays can be passed as parameters and are generally created "at activation time." Arrays allocated on the system heap have lifetimes beyond an activation frame, so their bounds are fixed at creation time. A resizable array, however, has to be fully (non-fixed) **heap-dynamic** in Sebesta's terms.
4. Static arrays (in C/C++ and older languages) are allocated at compile time, but otherwise the answers are generally "at activation time" and "at object creation time."
5. Yes automatically if an array type is legal as the type **`T`**. But access is less efficient than with true multidimensional arrays, which must be uniform.
6. There used to be a global max number of dimensions in FORTRAN etc.
7. Most languages allow both initialization to an "array literal" and allocation to a given size before initializing. For safety, the latter often requires filling in by a given element, e.g. via `make` in OCaml.
8. Slices are a nice selling point of Python, provided also in Scala via its integration of **range types**. The OCaml Array module has a less-rich notion of `sub`-arrays.

[goto Sebesta slides]

Summary of remaining part of lecture from Sebesta's slides:

In the later part of the lecture from Sebesta's slides, the first diagram for an array above was morphed into one of a multidimensional array. I have morphed it further to show an array-of-arrays first. Again, the array type **T** is shown as a record/`struct` totaling 48 bytes. My notational convention (orange 6-digit addresses) suggests that this is a value array allocated in the system stack memory---as one gets in C++ and Scala when arrays are created without using **new**. For an array on the system heap, the addresses are green 7-digit numbers but the relationships in the diagram would not differ much. [NB: The "system stack" really is a stack data structure in an over-arching sense, but the "system heap" is not strictly a *heap* in the data-structures sense.]



An ordinary array-of-arrays could be "ragged" in two senses: the constituent arrays can have different lengths, and they can be---indeed almost certainly will be---separate array storage-objects in memory. Whereas, a true multidimensional array must have uniform row as well as column bounds, and is usually allocated in one linear sequence in **row-major order**. This is shown in the second part of the diagram---note that the leftmost addresses for the second and third rows add $5 \times 48 = 240$ to the base address of the previous row. (FORTRAN uses column-major order because that goes better with some code for matrix multiplication.) The row-major formula is:

address of `A[i,j]` = **b + (i-1)*n*e + j*e**

where $0 \leq i < m$ and $0 \leq j < n$ for the $m \times n$ matrix A, where again **e** is the size in bytes of the element type.

Associative arrays are a core language element in Python and Perl and Ruby. In C++ and Java and Scala they are a library type called **map** or **Map** which can in turn be set to one of various implementations, say via a hash table or binary search tree. They have a **key type** K and a **value type** V. The key type can be any **discrete** type, meaning any type for which equality makes sense---usually excluding `float`/`double` and function types. If M is a map, k is a key, and v is a value, then the assignment

M[k] = v

has these consequences:

1. Use the equality test to see if **M** already has a key equal to **k**.
2. If not, add **k** as a key and make **v** the associated value.
3. If yes, then overwrite the previously associated value by **v**.

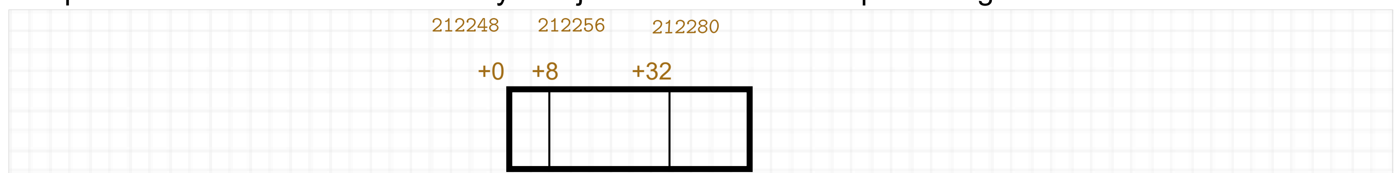
The behavior in point 3 enforces that **M** never has duplicate keys and has only one value for each key. Steps 2 and 3 only need $O(1)$ time but the key search in step 1 needs an efficient data structure to work quickly. Step 2 says that the map's size grows by 1---Sebesta mentions the idea of a "statically sized" map but that's not a major idea now. A long-burning question is:

Can an associative array ever approach a real array (in the special case where $K = \{0, \ldots, n-1\}$ is the key type) in time-critical performance?

Theoretically, the implementation by hash table is supposed to give $O(1)$ time for step 1 as well---which is why associative arrays are called "hashes" in Perl and Ruby---but there are practical hitches. (In Python they are called "dictionaries.")

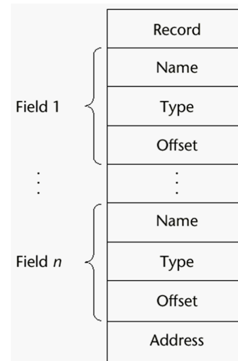
Records have largely been superseded by classes nowadays. They can be nested---i.e., a field of a record can be another record. Nesting was a visionary feature of the original 1958 COBOL implementation of records---curiously, the syntax for that used indentation the way Python does now for statements. C calles a record a **struct** but does not provide inheritance for them; in C++, a **struct** is fully equivalent to a **class** that makes **public** rather than **private** the default visibility.

The main point to mention now is that their fields are named storage objects in themselves, with their own binding addresses. Before a record object is created, you can't know the absolute addresses of the fields, but one can know the **offsets** from the base address at compile time by summing the sizes of the fields. Each field must be of a known type---which can be an already-declared record type. Here is a picture of the offsets for the 48-byte object used as an example throughout.



Access to a record field is direct and immediate just like with a simple variable. It does *not* add the offset to the base address the way an array adds the index (times the element size) to the base

address. This difference in efficiency remains tangible even today. [Sebesta does mention the idea of "dynamic subscripts" for records, but that's another historical item that we can put in the fuhgeddaboudit category.] The offsets are used only in the compile-time descriptor of a record type (copying Sebesta's diagram):



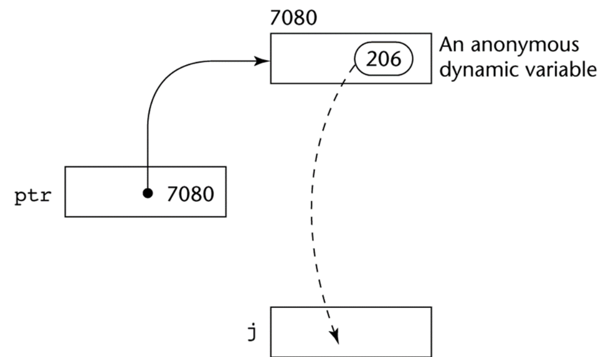
Tuples we have already seen. The main difference from lists is that they can be **heterogenous**, like records, meaning can have elements of different types. In Standard ML, a tuple of size t is formally the same as a record with fields named **#1**, **#2**, through **#t**. But OCaml allows tuple values to be read only via **match**, except that pairs `(x,y)` have functions **fst** and **snd** defined to get their fields. The text's mentions of Standard ML and F# are basically equivalent to what we've seen in OCaml.

Union Types can be **free** or **discriminated**. Free unions are available in C but---as far as I know---the erstwhile mission-critical time and storage reasons to use them (without type checking) no longer apply or are not thought worth the safety risks. The closest experience to free unions I have is that Perl allows a number to be treated as a string or number depending on context---and even then I often have to resort to calling the Perl `looks_like_number` function explicitly in my chess data scripts. Whereas, Python insists on explicit conversions **str** and **int**. (See also note on "Type Conversions" in section 7.4, next lecture.) Discriminated unions were gee-whiz features in Modula-2 and Ada 40 years ago but we've been using them in OCaml for weeks with no fanfare. The example

```
type intOrString = Int of int | Str of string
```

from recitations is the most simply typical one. The effect here is similar to Python. The other way that the polymorphism of union types enters in naturally is with object hierarchies, so we will revisit this later.

Pointers and Reference Types: Sebesta goes into mechanics of pointer and reference assignment that we already covered in connection with storage objects before break as part of treating the Chapter 5 material. Here's his diagram I briefly showed of assigning `j = *ptr`:



Whether a programming language should have **automatic garbage collection** was a major question in the last millennium, but the answer nowadays is almost always *yes* even in systems-level applications. As for **dangling pointers**, we'll hit them after reviewing the code constructs in chapters 8--10 that can potentially create them.

Option Types are mentioned in the one-page section 6.12 and on one of Sebesta's slides. We've already seen them in OCaml and have covered the benefit of having a type-checked `None` value rather than using null pointers which could blow up into segmentation faults. Our attitude will not be to care about the various syntaxes for this in certain other languages but to focus on their implementation in OCaml as the springboard for ideas of how to use them. If Sebesta's coverage of these and unions and tuples etc. helps round out your understanding of how they work in OCaml, so much the better, but for core course material we're doing "single-focus" as much as possible.

Type Checking---we've already touched on this as a main part of the next phase of compilation after parsing. And OCaml's automatic type inferencing has provided royal examples of it. **Type equivalence** we can fold under the later, broader topic of **type compatibility** in OOP. And as regards the last section 6.16 on type theory, I mentioned it at the start, alongside literally singing its praises. That's a wrap on Chapter 6.