CSCI 135
Pointers and Their Applications

Pointers and Arrays

C arrays are disguised pointers!

Ex: `int arr[4] = {10,11,12,13}`

Assume arr is allocated to address 1000, and ints require 4 bytes.

RAM:

Address	**Data**	**Name**
...	...	...
1012	13	arr[3]
1008	12	arr[2]
1004	11	arr[1]
1000	10	arr[0]
...	...	...

Equivalences:

```
int arr[4];                  const int *arr;
arr[2] = arr[3];     ≡       *(arr+2) = *(arr+3);
arr[4] = 7;                  *(arr+4) = 7; BOUNDS ERROR!
```

Returning Array from Functions?

Suppose we had something like this:

```
typedef int[8] ArrType;
ArrType foo(...) {      don't do this!
}
int main() {
  ArrType x = foo(...)
  cout << x[3];
}
```

? What would the above print

Returning Array from Functions?

Suppose we had something like this:

```
typedef int[8] ArrType;
ArrType foo(...) {      don't do this!
}
int main() {
  ArrType x = foo(...)
  cout << x[3];
}
```

? What would the above print

1. x is assigned a pointer to an int
2. x[3] is *(x+3), which points to 3 ints past x in RAM
3. But 3 ints past x is a variable with lifetime local to foo!

Returning Arrays From Functions

Not allowed to be a return value! Alternatives if you need to:

1. Pass array to function (implicitly by reference since its a pointer)

   ```
   foo(int arr[], int size) {...}
   ```

   Function foo can modify arr freely since the actual corresponding to the formal arr has scope/lifetime of caller (at least).
2. Return a struct where one field is an array.

   ```
   struct SomeStruct {
     int     blah1[32];
     double  blah2;
   };
   SomeStruct foo(...) {...}
   ```

3. Return a pointer, but ensure that the variable pointed to has eternal lifetime (probably overkill for most programs).

Dynamic Arrays - Motivation

Recall:

- [Static] Array dimension must be specified at compile time (so compiler can allocate storage).
- [Static] Array parameters do not indicate dimension, since they are just abbreviations for pointers.

```
const int DIM=16;
void foo(int arr[], int size) {
  arr[2]=7;
}
int main() {
  int x[DIM]; space allocated here
  foo(x,DIM);
}
```

$\equiv$

```
const int DIM=16;
void foo(int *arrp, int size) {
  *(arrp+2) = 7;
}
int main() {
  const int *xp = (int *)
    malloc((sizeof int)*DIM);
  foo(xp,DIM);
}
```

Dynamic Arrays - Motivation

Recall:

- [Static] Array dimension must be specified at compile time (so compiler can allocate storage).
- [Static] Array parameters do not indicate dimension, since they are just abbreviations for pointers.

```
const int DIM=16;
void foo(int arr[], int size) {
  arr[2]=7;
}
int main() {
  int x[DIM]; space allocated here
  foo(x,DIM);
}
```

≡

```
const int DIM=16;
void foo(int *arrp, int size) {
  *(arrp+2) = 7;
}
int main() {
  const int *xp = (int *)
    malloc((sizeof int)*DIM);
  foo(xp,DIM);
}
```

? What if we don't know the size of the array at compile time?

Dynamic Arrays

A dynamic array is an array whose size is not specified statically.
⇒ Size can be dynamically determined.

C++ approach:

```
int *arr;                  allocates space for 1 pointer
arr = new int[16];         allocates space for 16 ints
```

C approach (normally not used in C++):

```
int *arr;
arr = (int *) malloc((sizeof int) * 16);
```

In C, dynamic memory allocation required the exact number of bytes that needed to be allocated.

Changing Size Of Dynamic Array

```
int *arr;
arr = new int[16];
...
cin >> newSize;
int *tempp = new int[newSize];     allocate new space
for (int i=0; i<16; i++)           copy element by element
  tempp[i]=arr[i];
delete [] arr;                     avoid garbage!
arr = tempp;
```

⚠The delete tells the OS that the RAM area used to store the old contents of arr are free to be used for future allocates (new, malloc), NOT that the arr variable is deleted or that anything in the old arr is immediately overwritten.

Returning an Array From a Function

Dynamic arrays ⇒ Return pointer to base type!

```
int * foo(int & size, ...) {
  int * arr;
  size = 32;
  arr = new int[size];       Allocated on heap
  <do something to arr>
  return arr;                Pointer to arr[0]
}
```

More About `malloc`

```
arr = (int *) malloc((sizeof int) * 16);
```

- sizeof operator returns the number of bytes needed to store its argument type (*e.g.*, 4)
- malloc(n) allocates storage for n bytes on the heap, like new (64 bytes in above example if ints need 4 bytes)
- Malloc is defined in stdlib.h as:
  `void * malloc(size_t size)`
  where size_t is an unsigned type (with automatic type conversion from ints).
- ⇒ Need to cast from `void *` to `int *`

⚠Better to use new over malloc (in C++)

Revisiting Deep and Shallow Copying (in C/C++)

- Arrays are copied shallowly.
  `arr1 = arr2` only makes arr1 point to the same data as arr2.

```
int arr1[10];
int arr2[10];
int *ptr = new int[10];

ptr = arr1;      valid code, address of the array
                 is saved in the pointer

arr1 = arr2;     invalid in C++
arr1 = ptr;      invalid in C++
```

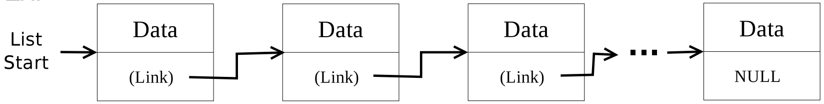
- All struct fields are copied, including pointers, but entities, to which they point, are not copied.
- Exception: struct fields that are arrays are copied entirely, element by element (*i.e.*, deep copy).

Dynamic Data Structures

Static data structures: fixed topology/shape and size

Dynamic data structures: either/both topology and size can change *at run time*

Ex:



Key points:

- Each element has a link field (in addition to the data field/s)
- Links are implemented as pointers, and manipulated using standard pointer operators
- Final link is NULL (so we can determine end)
- Program only has variable pointing to start; other elements are retrieved by traversing links.
- Data structures differ by number and type of links, allowing for different types of efficient access (CSCI 235, mainly).
- Allocated on heap, unlike static data structures.

Building Block: Node

```
typedef ... SomeType;
struct Node
{
  SomeType data;
  Node *   nextp;   NULL if end of list
};
Node * myList;      points to start of list
```

- SomeType could be just an int, or something huge
- nextp pointer is storage overhead, but allows us to implement insertion/deletion efficiently
- Don't forget to set link to NULL at end of list (or we won't know where the list ends!)
- Variations of Node used for other (non-list) topologies.

Traversing a List

```
struct Node {
  string  key;
  int     id
  Node *  nextp;
};
Node * find(Node * list, string name) {
  Node * head = list;
  while ((head != NULL) && (name != head->key))
    head = head->nextp;
  return head;
}
```

Exercises

1. Take one of your 1-D array examples from earlier, and rewrite using pointers (explicitly).
2. Using the Node definition given with the list example, modify the given traversal code assuming the list is sorted by key. If the element is not found, return a NULL. Of course, you should be as efficient as possible.
3. Using the Node definition given with the list example, write a function that inserts a key,id pair into the list assuming the list is sorted by key. You probably want to draw a picture first.
4. A polynomial may be represented as an array of coefficients (*e.g.*, $x^3 + 2x^2 + 4 = [1, 2, 0, 4]$. Write the following functions:
   - Node * makePoly(double coefs[])
   - int degree(Node * poly)
   - double eval(Node * poly, double x)
5. Write a loop that creates garbage, and run it as many iterations as needed for something bad to happen. What happens? Then, rewrite the code to avoid creating garbage.