arrays and strings: what’s a char *?

- Why not rely solely on string and vector classes?
  - how are string and vector implemented?
  - lower level access can be more efficient (but be leery of claims that C-style arrays/strings required for efficiency)
  - real understanding comes when more levels of abstraction are understood

- string and vector classes insulate programmers from inadvertent attempts to access memory that’s not accessible
  - what is the value of a pointer?
  - what is a segmentation violation?
Contiguous chunks of memory

- In C++ allocate using array form of `new`
  ```
  int * a = new int[100];
  double * b = new double[300];
  ```

- `new []` returns a pointer to a block of memory
  - how big? where?

- size of chunk can be set at runtime, not the case with
  ```
  int a[100];
  cin >> howBig;
  int a[howBig];
  ```

- delete [] a; // storage returned

```
int * a = new int[100];
```

a is a pointer
*`a` is an int
a[0] is an int (same as `*a`)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
a+100 is valid for comparison of pointer values
C-style contiguous chunks of memory

- In C, malloc is used to allocate memory
  ```c
  int * a = (int *) malloc(100 * sizeof(int));
  double * d = (double *) malloc(200 * sizeof(double));
  ```

- malloc must be cast, is NOT type-safe (returns void *)
  - void * is 'generic' type, can be cast to any pointer type

- free(d); // return storage
- We WILL NOT USE malloc/free
Address calculations, what is sizeof(…)?

- x is a pointer, what is x+33?
  - a pointer, but where?
  - what does calculation depend on?

- result of adding an int to a pointer depends on size of object pointed to

- result of subtracting two pointers is an int:

\[(d + 3) - d = \_\_\_\_\_\_\_\_]
More pointer arithmetic

- address one past the end of an array is ok for pointer comparison only

- what about *(begin+44)?

- what does begin++ mean?

- how are pointers compared using < and using ==?

- what is value of end - begin?

```cpp
char * a = new int[44];
char * begin = a;
char * end = a + 44;
while (begin < end)
{
    *begin = 'z';
    begin++;
    // *begin++ = 'z'
}
```
What is a C-style string?

- **array of char terminated by sentinel ‘\0’ char**
  - sentinel char facilitates string functions
  - ‘\0’ is null char, unfortunate terminology
  - how big an array is needed for string “hello”? 

- **a string is a pointer to the first character just as an array is a pointer to the first element**
  - `char * s = new char[6];`
  - what is the value of s? of s[0]?

- **char * string functions in <cstring> (or <string.h>)**
C style strings/string functions

- strlen is the # of characters in a string
  - same as # elements in char array?

```c
int strlen(char * s)
// pre: '\0' terminated
// post: returns # chars
{
    int count=0;
    while (*s++) count++;
    return count;
}
```

- Are these less cryptic?

```c
while (s[count]) count++;  // OR, is this right?
char * t = s;
while (*t++);
return t-s;
```

- what’s “wrong” with this code?

```c
int countQs(char * s)
// pre: '\0' terminated
// post: returns # q’s
{
    int count=0;
    for(k=0;k < strlen(s);k++)
        if (s[k]==‘q’) count++;
    return count;
}
```

- how many chars examined for 10 character string?
- solution?
More string functions (from <string.h>)

- `strcpy` copies strings
  - who supplies storage?
  - what’s wrong with `s = t`?

```c
char s[5];
char t[6];
char * h = "hello";
strcpy(s,h); // trouble!
strcpy(t,h); // ok
```

```c
char * strcpy(char* t, char* s)
//pre: t, target, has space
//post: copies s to t, returns t
{
    int k=0;
    while (t[k] = s[k]) k++;
    return t;
}
```

- `strncpy` copies n chars (safer?)
- what about relational operators `<`, `==`, etc.?
- can’t overload operators for pointers, no overloaded operators in C
- `strcmp` (also `strncmp`)
  - return 0 if equal
  - return `neg` if `lhs < rhs`
  - return `pos` if `lhs > rhs`

```c
if (strcmp(s,t)==0) // equal
if (strcmp(s,t) < 0)// less
if (strcmp(s,t) > 0)// greater
```
Arrays and pointers

- These definitions are related, but not the same
  
  ```
  int a[100];
  int * ap = new int[10];
  ```

- both a and ap represent ‘arrays’, but ap is an lvalue

- arrays converted to pointers for function calls:
  
  ```
  char s[] = "hello";
  // prototype: int strlen(char * sp);
  cout << strlen(s) << endl;
  ```

- multidimensional arrays and arrays of arrays
  
  ```
  int a[20][5];
  int * b[10]; for(k=0; k < 10; k++) b[k] = new int[30];
  ```
Coping with C instead of C++

- Only pass-by-value, no reference parameters

```c
void changeIt(int * x) {
    *x = 3;
}
int val = 13;
changeIt(&val);
cout << val << endl;

void changeIt(int & x) {
    x = 3;
}
int val = 13;
changeIt(val);
cout << val << endl;
```

- To make an array (allocate and pass back)

```c
void fillUp(int **x) {
    *x = new int[100];
}
int * x;
fillUp(&x);
x[0] = 13;
```

what about int *x[] as param? address-of operator is evil, purposefully left out of java
Pointers Idioms in C

- **Typical purposes for pointers**
  - sharing (just like C++)
  - to represent arrays (multi-dimensional arrays are possible but different, why??)
  - to simulate reference parameters (syntax is uglier)

- **Initializing pointers**
  - use new (or new [])
    \[ \text{int } \ast x = \text{new int}; \]
  - use another pointer
    \[ \text{int } \ast y = x; \]
  - use 0 (i.e., NULL)
    \[ \text{int } \ast z = 0; \]
  - use address-of operator \( \& \)
    \[ \text{int } \ast zz = \&i; \]

- **Thus, in C, pointers pile up**

\[
\begin{align*}
\text{void fillUp (int **x)} & \quad // ?? \\
\text{void fillUp (int ***x)} & \quad // ?? \\
\text{void fillUp (int ****x)} & \quad // ?? \\
\end{align*}
\]
C++ and C-isms

- In C++ a struct is a class in which public is default
  - supports everything a class does, including inheritance
  - in C, a struct requires the word *struct*, typedef often used

```c
typedef struct node_s
{
    int value;
    struct node_s * next;
} Node;
```

- **pointers to functions**: use right-left-right rule (watch parens)

```c
void foo(int (*df) (double,double));
typedef int (*iddfunc) (double,double);
void foo(iddfunc df);
```