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
  
  \[
  \text{int} \ * \ a = \text{new int}[100]; \\
  \text{double} \ * \ b = \text{new double}[300];
  \]

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

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

- delete [] a; // storage returned

\[
\text{int} \ * \ a = \text{new int}[100];
\]

\[
\begin{array}{c}
\text{0} \quad 1 \quad 32 \quad 33 \quad 98 \quad 99
\end{array}
\]

*a is a pointer

*a is an int

\[a[0] \text{ is an int (same as } *a)\]

\[a[1] \text{ is an int}\]

*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
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

```
int * a = (int *) malloc(100*sizeof(int));

0  1  32  33  98  99
```

```
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
```
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 == \_\_\_\_\_\_\_\_\_\_

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

a[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(think 160, 164, 168)

double * d = new double[200];

*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, d+2 is 0x00c0
(think 176, 184, 192)
```
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?

```c
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 nul 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 <string.h>
C style strings/string functions

- `strlen` is the number of characters in a string
  - same as the number of elements in a 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 characters examined for a 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)// ???
```
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];
  ```
Microsoft question

- Write atoi, write itoa, which is harder?

- Questions? Issues? Problems?

```c
int atoi(const char * sp);
char * itoa(int num);
```

- Difference between `const char * p` and `char * const p`
  - one is a pointer to a constant character
  - one is a constant pointer to a character
# Classes, compilers, dependencies

```cpp
#include <string>
#include "day.h"

typedef string TimeRange;
class ostream;

class Appointment
{
    public:
        TimeRange duration();
        void print(ostream & output);
    private:
        Day myDay;
}
```

- **why use** class ostream instead of `#include <stream.h>`
- **what is a typedef and how is it used?**
- **make depend for Appointment/ostream?**
- **changes to Day force recompile for appointment clients?**
**.h guidelines, preprocessor in action**

- **minimize #includes in every .h file**
  - avoid circular dependencies
  - avoid re-compile by minimizing dependencies
- **class Foo in foo.h, class Bar in bar.h, client foobar.cpp**

```cpp
#ifndef _FOO_H
#define _FOO_H
define _FOO_H
#include "bar.h"
class Foo
{
    Bar getBar();
    Foo getFoo();
}
#endif // _FOO_H

#ifndef _BAR_H
#define _BAR_H
#define _BAR_H
class Bar
{
    Foo getFoo();
}
#endif // _BAR_H
```

```
// from foo.cpp
#include "bar.h"
#include "foo.h"
void Foo::doStuff(const Bar & b)...
```

- **Avoid #includes, use forward references, sometimes you must do this as shown above (even if you don’t want to)**
#include “foo.h”

- will be needed in .cpp file, e.g., foo.cpp and bar.cpp
- using pointers and references in .h files minimizes dependencies
  - minimize recompiles when .h changes
  - loose coupling: avoid implementation dependencies when possible
- avoid letting implementation leak into public view
  - what about private section?
  - opaque pointer: `FooImpl * myImpl;`
    - implementation of FooImpl is hidden, class can be implemented in foo.cpp (handle-body idiom)
  - factory: inheritance hierarchy, ABC
C++ idioms

- What happens with the statement `myDay = d;`?
  - assignment is memberwise unless operator `=` overloaded
  - copy constructor used in passing parameters by value
- If you need one of: destructor, assignment operator, copy constructor, you need all of them
  - heuristic only: managing resources other than memory
  - preventing objects from being copied
  - what about non-copyable state, e.g., stream

- In assignment operator, watch for self-assignment
- Study implementation of string/vector
copy constructor

- **Used for “first-time” creation**

  ```cpp
  Date d(1,1,2000);
  Date copy(d);
  ```

- **Used for pass-by-value**

  ```cpp
  DoStuff(Date d);
  //...
  Date first(1,1,2000);
  DoStuff(first);
  ```

- **what about use of myLength in code as opposed to length()?**

  ```cpp
  Template <class Item>
  Vector(const Vector<Item> & vec)
  // precondition: Item supports assignment
  // postcondition: return copy of vec
  {
    // allocate storage
    myList = new Item[myLength=vec.myLength];
    assert(myList != 0);
    // copy elements
    for(int k = 0; k < vec.myLength; k++)
    {
      myList[k] = vec.myList[k];
    }
  }
  ```