EECS 280
DISCUSSION #5

Week of February 4

OUTLINE

- **Administrivia**
- Arrays
- Pointers
- Pointer-Array Duality
- Function Parameters

Administrivia

- Project 1
  - Test cases on CTools
- Project 2
  - Questions, comments, concerns?
  - Grades should be out in about a week
- Project 3
  - Due March 4 @ 11:59PM
  - Requires lots of time, debugging, and testing
  - Questions, comments, concerns?

OUTLINE

- Administrivia
- **Arrays**
  - Motivation
  - Usage
  - Strings
- Pointers
- Pointer-Array Duality
- Function Parameters

Array Motivation

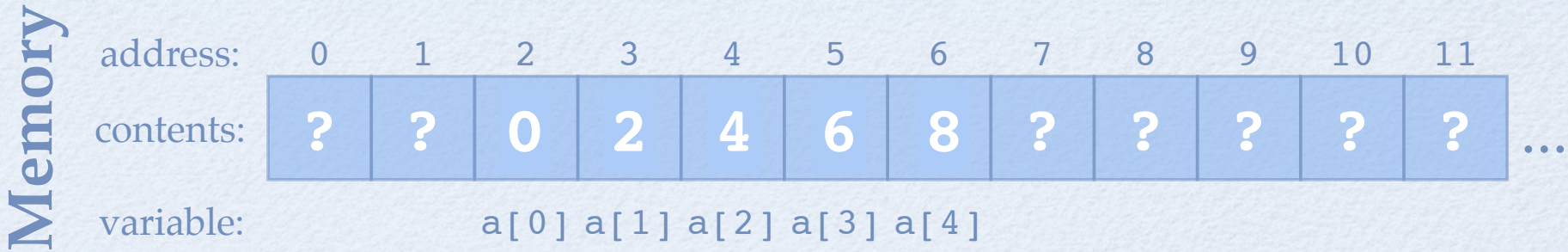
- Suppose we are writing a program to track mileage for 100 cars
- Why not use 100 variables
  - Messy, redundant code
  - Prone to bugs, hard to debug
  - What happens when we add/remove a car?

ARRAYS

A group of elements (each of the same data type) that are stored in contiguous memory and accessed by indexing

```
int car_mileage[100];
```

ARRAY REPRESENTATION




```
int a[5];
for ( int i=0; i<5; i++ )
  a[i] = ( i * 2 );

cout << a[2];
```

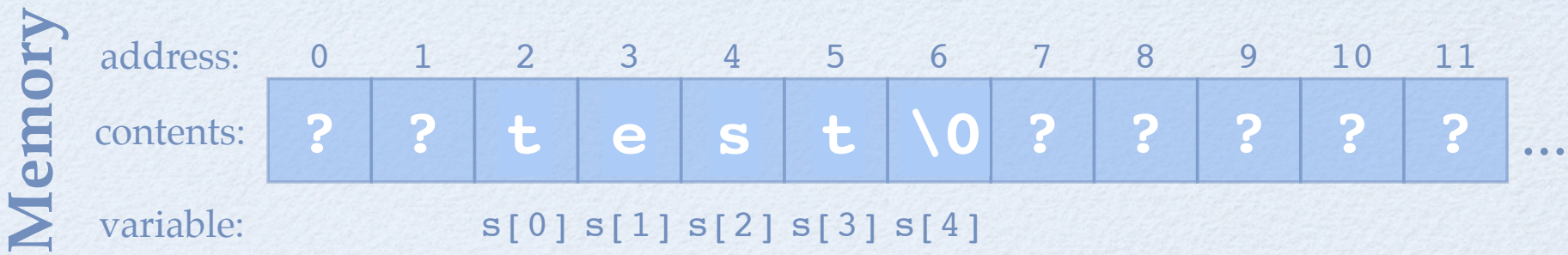
Notes on Arrays

- C++ does not restrict the access to array elements outside the range defined
  - If you access beyond the bounds of the array you may access/modify other variables or run into system errors
- You can declare an array of any type: char, double, float, self-made structures, etc.

Strings

- C++ Style Strings
  - Objects with useful functions
- C-Style Strings
  - Arrays of characters
  - To make use of most string functions, the final element should be `'\0'` (`NULL`)

String Representation




```
char s[5];
s[0] = 't';
s[1] = 'e';
s[2] = 's';
s[3] = 't';
s[4] = '\0';
cout << s;
```

→ `char s[] = "test";`

→ `test`

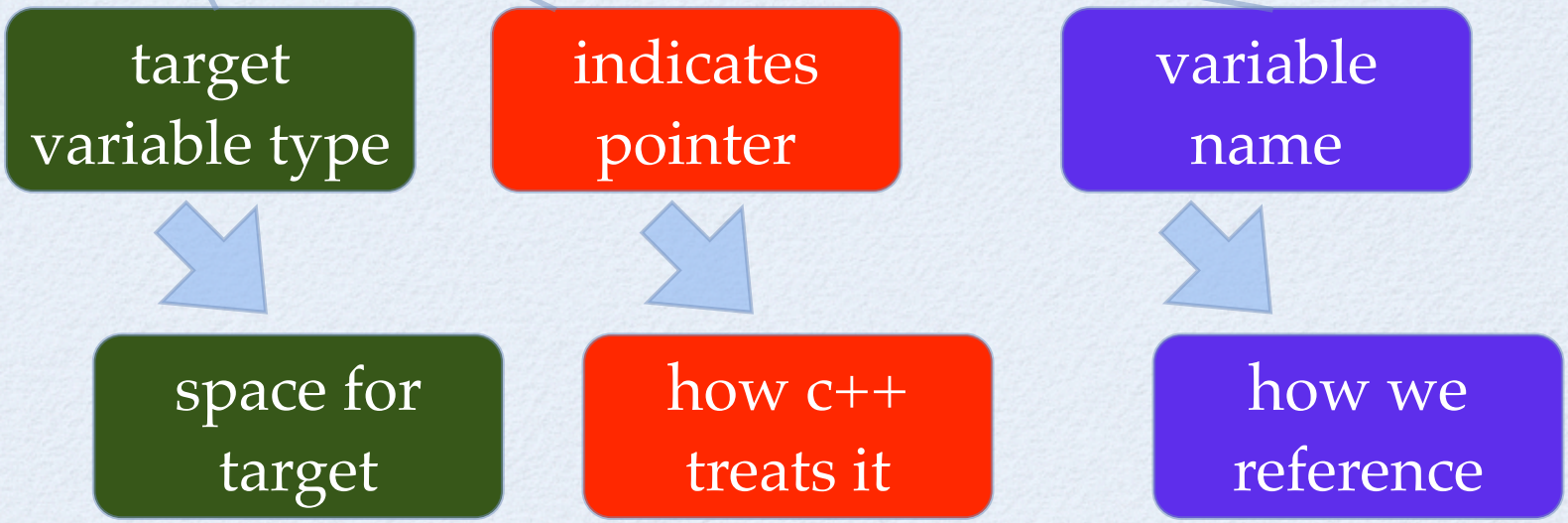
OUTLINE

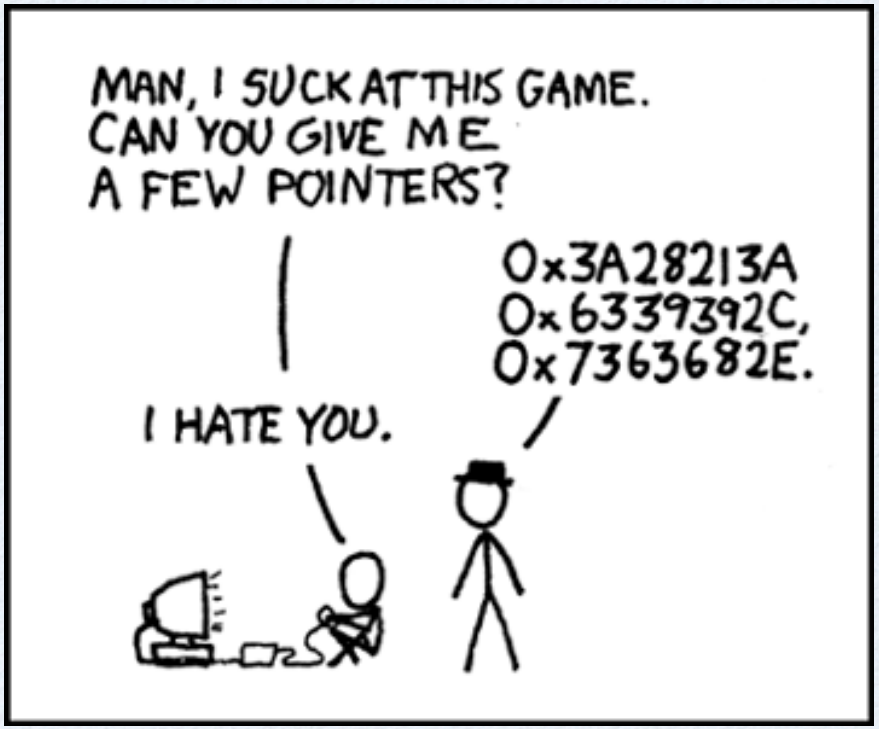
- Administrivia
- Arrays
- **Pointers**
  - Definition & Usage
  - Examples
- Pointer-Array Duality
- Function Parameters

POINTERS

A pointer is a variable that holds a memory address [of another variable]

```
int *my_pointer;
```





REINFORCING HUMOR

POINTER OPERATORS

`&variable` - “address of” variable

`*pointer` - “value at” address

POINTER REPRESENTATION


```
int x;
x = 5;
cout << x;   5
cout << &x;  2
```

```
int *p;
p = &x;
cout << p;   2
cout << &p;  8
cout << *p;  5
```

what's in p's box

the address of p

box pointed to by p

POINTER MANIPULATION


```
x = 7;
cout << x;   7
cout << &x;  2
cout << p;   2
cout << &p;  8
cout << *p;  7
```

```
*p = 4;
cout << x;   4
cout << &x;  2
cout << p;   2
cout << &p;  8
cout << *p;  4
```

OUTLINE

- Administrivia
- Arrays
- Pointers
- **Pointer-Array Duality**
  - Array Notation in Pointer Terms
  - Examples
- Function Parameters

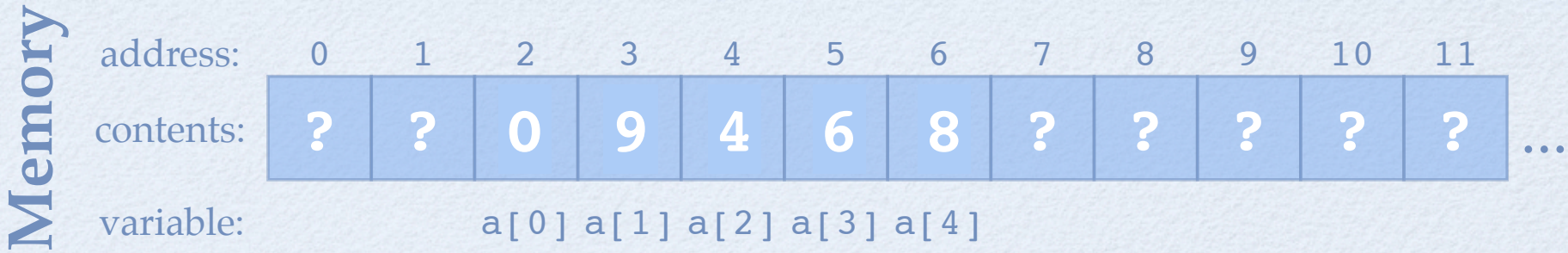
Array element addresses

- Each array element has a unique address:
  - `&a[0]` for element `a[0]`
  - `&a[1]` for element `a[1]`
  - ...
- C++ has a shortcut:
  - `a` is the same as `&a[0]`
  - `(a+1)` is the same as `&a[1]`
  - ...

Array element values

- The bracket (“[]”) notation serves as a shortcut to array element values:
  - a[0] for *a
  - a[1] for *(a + 1)
  - a[2] for *(a + 2)
  - ...

ARRAYS AS POINTERS




```
cout << a;          2
cout << (a+3);      5
cout << *(a+4);     8
*(a+1) = 9;
cout << a[1];       9
cout << (a+1);      3
cout << *(a+1);     9
```

OUTLINE

- Administrivia
- Arrays
- Pointers
- Pointer-Array Duality
- **Function Parameters**
  - Motivation
  - Passing Pointers
  - Passing References

Motivation

```
int add_one( int x )
{
    return ( x + 1 );
}
```

- Consider `add_one`
- When we call this function, it makes a copy of the parameter in the activation record, regardless of size
- What if we were passing large data (think databases) instead of just numbers?
  - Pass pointers!

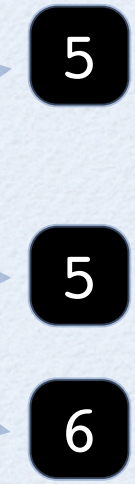
PASSING VALUES

```
int add_one( int number )
{
   number++;
   return number;
}

int main()
{
   int x = 5;
   int y = 0;

   cout << x;
   y = add_one( x );
   cout << x;
   cout << y;

   return 0;
}
```



PASSING POINTERS

```
void add_one( int *number_pointer )
{
   (*number_pointer)++;
}

int main()
{
   int x = 5;
   int *p = &x;

   cout << x;
   add_one( p );
   cout << x;

   return 0;
}
```

PASSING REFERENCES

```
void add_one( int &number_reference )
{
   number_reference++;
}

int main()
{
   int x = 5;

   cout << x;
   add_one( x );
   cout << x;

   return 0;
}
```

Function Parameters

- Pass-by-Value
  - Can't "clobber" passed variables
  - Activation record size depends upon size of parameters
  - 1 return value
- Pass-by-Reference
  - Passed variables can be changed
  - Activation record size is constant
  - Potentially multiple "return" values

Final Thoughts

- Play with pointers - they can be tricky at first but if you master the concepts, they are a very powerful tool
- Effective array use is about exploiting patterns in stored data
- Get started on Project 3 early
  - Think, plan, and develop test cases before coding - it WILL save you time later