INTRODUCTION TO POINTERS

CS 23200

Tips for a Successful Class

- **Learn the concepts completely**
  - **You** must build an accurate mental model of the concepts
  - Understand **how** and **why**
  - If you miss a question or don't understand an example, go back and study it until you know why
  - If you have doubts about your understanding, come by office hours
- Ask me questions
- Tell me to slow down or speed up
- Read carefully (especially the homework/project descriptions)

Some Motivation for Pointers

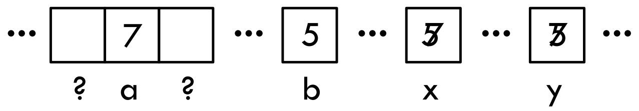
- Would like to call a function to swap two integers:

```
int x, y;
...
swap(x,y);
```

Some Motivation for Pointers

```
int main(){
   int a = 7;
   int b = 5;

⇨  swap(a,b);
   return a;
}
```

```
⇨ void swap(int x, int y){
     int temp = x;
     x = y;
     y = temp;
  }
```



- Memory
  - One large array of bytes
  - Each block in this diagram has enough bytes to hold an `int`

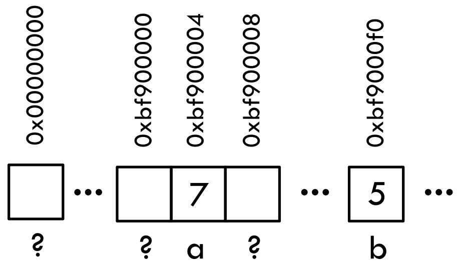
Some Motivation for Pointers

- We want `swap` to operate on the *particular 7* and 5 that correspond to `a` and `b` (not just a copy of 7 and 5)

- We can refer to the particular 7 and 5 by their locations (i.e., addresses) in memory

Memory Addresses

- Each byte of memory has an address
  - Address is the index in the large array of bytes that comprise memory
  - Typically written in hexadecimal
  - In this example, an `int` takes 4 bytes
- C has an operator to get the address of a variable: unary `&`
  - `&a` == 0xbf900004
  - `&b` == 0xbf9000f0



Pointers

- Pointers are variables that hold addresses
  - That is, the value of a pointer is an address
- The syntax for declaring a pointer variable:
  - `type *name;`
  - Declares *name* to be a pointer to a variable of type *type*
- Examples:

```
int x;
int *pInt;
pInt = &x;

char c;
char *pc;
c = '6';
pc = &c;
```

Dereferencing a Pointer

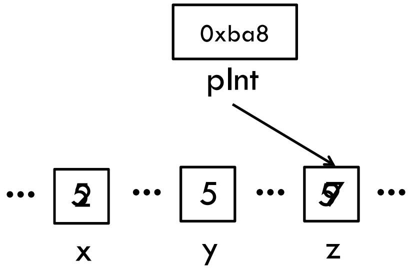
```
double *pDbl;
```

- `pDbl` is a pointer to a `double` (i.e., a `double*`)

- Dereferencing: accessing what the pointer refers to (i.e., what it points to)
  - The (unary) `*` operator dereferences a pointer
  - `*pDbl` is the double at address `pDbl`

Pointers Example

```
int x=2, y=5, z=7;
int *pInt;
pInt = &z;
*pInt = y;
x = *pInt;
*pInt += 2 * 2;
if(*pInt == 7){
   ...
}
```



The Many Uses of *

- For multiplication

```
int x=2, y=5;
x = y * 3;
```

- To specify a pointer type
  - When declaring a pointer
  - Does **not** dereference a pointer

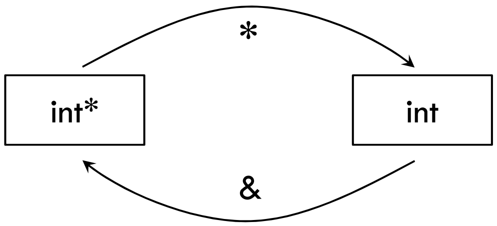
```
/* declare and
  initialize */
int* pInt = &y;

int *pTwo = &x;
```

- To dereference a pointer

```
/* wrong */
*pTwo = &x;

/* okay */
*pTwo = x;
```

Type Correctness

- For EVERY expression you write, think…
  - What type do I want? (e.g., int or int*)
  - Is the expression the correct type (e.g., int or int*)
- The key to * and &:



NOT CORRECT

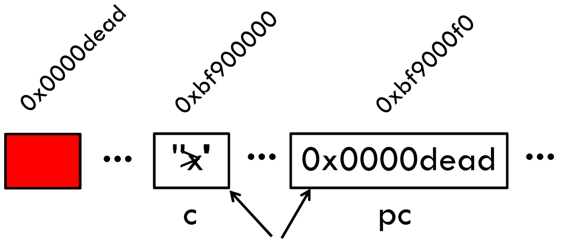
Type correctness is necessary,
but not sufficient!

```
double *pPi;
*pPi = 3.141592;
/* crash, boom, bang */
```

The Problem

- We declared double *pPi and then set *pPi to the double value for pi
- So what's the problem?
- Another example:

```
➡ char c;
➡ char *pc;
➡ c = 'x';
➡ *pc = 'x';
```



uninitialized memory (could contain anything)

- Access may be forbidden (**segmentation fault**)
- May overwrite important program data

Beware of Uninitialized Pointers

```
double *pPi;
*pPi = 3.141592;
/* crash, boom, bang */
```

- Need three steps:
  - Declare
  - Initialize (i.e., point to valid memory)
  - Use (i.e., deference)

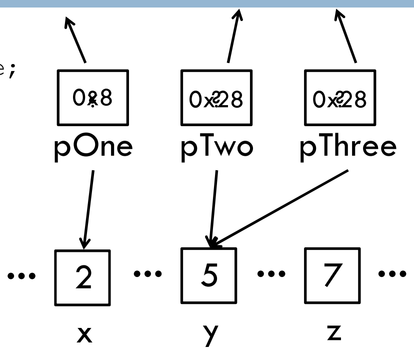
Beware of Uninitialized Pointers

- Convention: set pointers equal to `NULL` (the address 0x0) if they do not contain valid addresses
  - The seg fault occurs immediately, so at least you know where the problem is

```
double *pPi = NULL;

*pPi = 3.141592;
/* crash, boom, bang */
```

Examples

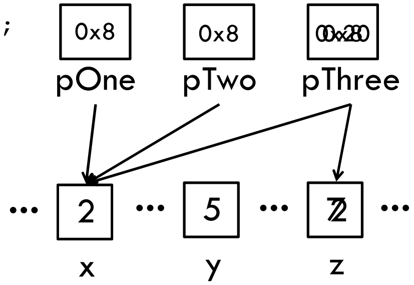
```
int x=2, y=5, z=7;
int *pOne, *pTwo, *pThree;

➡ pOne = &x;
➡ pTwo = &y;
➡ pThree = &y;

*pOne += 1;
*pTwo += 2;
*pThree += 3;
```



- What are the values of x, y, z at the end?
  - x==3, y==10, z==7

Examples

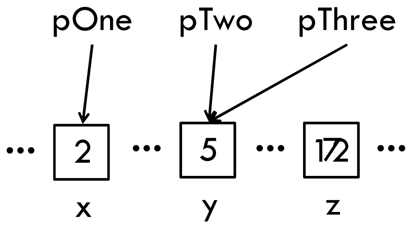
```
  int x=2, y=5, z=7;
  int *pOne, *pTwo, *pThree;

  pOne = &x;
  pTwo = pOne;
  pThree = &z;
⇨ *pThree = *pOne;
⇨ pThree = pOne;
⇨ *pThree = *pOne;
```



- □ What are the values of x, y, z at the end?
  - x==2, y==5, z==2

Examples

```
  int x=2, y=5, z=7;
  int *pOne, *pTwo, *pThree;

  pOne = &x;
  pTwo = &y;
⇨ if(*pOne > *pTwo){
        pThree = pOne;
  }
  else{
⇨       pThree = pTwo;
  }
⇨ z += *pThree;
```



- □ What are the values of x, y, z at the end?
  - x==2, y==5, z==12

Pointer Practice, Page 1

Back to Swap

```
void swap(int x, int y){
   int temp = x;
   x = y;
   y = temp;
}
int main(){
   int a = 7;
   int b = 5;

   swap(a,b);
   return a;
}
```

- □ This only exchanges the local variables `x` and `y` (not the `a` and `b` in `main`)

```
void swap(int *px, int *py){
   int temp = *px;
   *px = *py;
   *py = temp;
}
int main(){
   int a = 7;
   int b = 5;

   swap(&a,&b);
   return a;
}
```

- □ This swaps the integers in memory locations `px` and `py` (which are `&a` and `&b` in this example)

Pointers and Functions

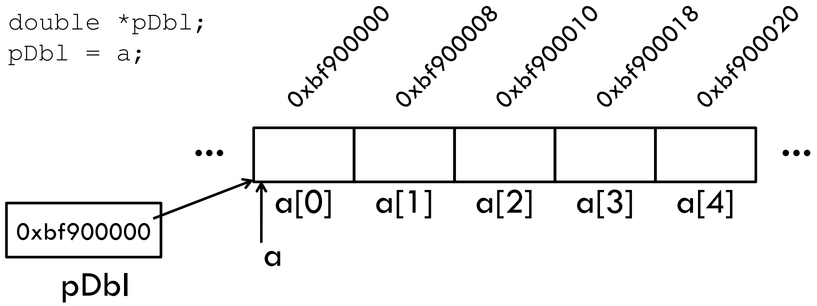
- Pointers allow a function to manipulate multiple variables from the calling function
- The argument lists use the same syntax as pointer declarations
  - int swap(int *px, int *py);
  - void someFunction(const int x, int *xSquared,
                                        int *xCubed);

Pointer Practice, Page 2

Pointers and Arrays

- Array variables ***are pointers***! (essentially)

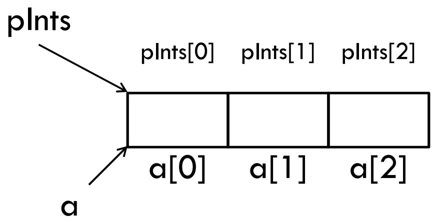
```
double a[5];
/* a  is a pointer to the first element in the array */
/* a == &a[0] */

double *pDbl;
pDbl = a;
```



Pointers and Arrays

- Pointers can be indexed like arrays
- pInts[i] is the item that is i spots after *pInts

```
int a[3];
int *pInts = a;

pInts[0] = 3;
pInts[1] = 4;
pInts[2] = 5;

/* a contains ...
                     {3,4,5} */
```



Pointers and Arrays

- Two noteworthy differences between array variables and pointers:

  1. Declaring an array allocates space for the given number of items, but declaring a pointer does not

```
/* allocates space for 5 ints;
   ids is the address of the first int */
int ids[5];


/* does not allocate space for any ints */
int *pInts;
```

Pointers and Arrays

2. Can assign different addresses to a pointer, but not to an array variable...

```
int ids[5];
int *pInts;
...
ids = pInts; /* not allowed */
pInts = ids; /* allowed */
```

Other differences exist, but are not as practically important.
See http://c-faq.com/aryptr/aryptrequiv.html for more information

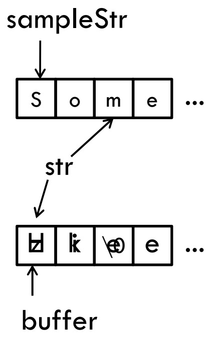
Arrays as Function Arguments

```
/* these prototypes are equivalent! */

int findSpace(char buffer[]);

int findSpace(char buffer[10]);

int findSpace(char buffer[10000]);

int findSpace(char *buffer);

/* the last one is preferred, because it captures
   what the compiler is actually doing */
```

Pointer as a Function Argument

```
/* replaces every occurrence of oldc in str with newc */
void replaceChar(const char oldc, const char newc,
                  char* str){

   int i;
   for(i=0; str[i] != '\0'; i++){
      if(str[i] == oldc){
         str[i] = newc;
      }
   }

}
```

```
void replaceChar(const char oldc, const char newc,
                 char* str);
int main(){
   char sampleStr[] = "Some Sample Phrase";
   const int bufferSize = 100;
   char buffer[bufferSize];

   replaceChar('s', 'n', &sampleStr[2]);
   printf("%s\n", sampleStr);

   int i;
   for(i=0; i<bufferSize-1; i++){
      int x = getchar();
      if(x == EOF || (char)x == '\n')
         break;
      buffer[i] = (char)x;
   }
   buffer[i] = '\0';
   replaceChar('s', 'n', buffer);
   printf("%s\n", buffer);
   return 0;
}
```



Summary of Pointers and Arrays

- `int arr[5];`
- `int *pArr = arr;`

- Array name is essentially a pointer
- Pointer can be indexed like an array
  - `pArr[i] == arr[i]`
  - Be careful with pArr[i]: i spots after pArr must be valid data

Pointer Practice

Summary

- Pointers specify locations in memory, addresses of other variables
- Array variables are essentially pointers (and vice versa)
- Motivation for pointers:
  - Allow functions to manipulate multiple variables from the calling function
  - Further motivation next time

- Next subject: moving beyond basic data types to structured data