CS103 Unit 6 - Pointers

Mark Redekopp
Why Pointers

• Scenario: You write a paper and include a lot of large images. You can send the document as an attachment in the e-mail or upload it as a Google doc and simply e-mail the URL. What are the pros and cons of sending the URL?

• Pros
  – Less info to send (send link, not all data)
  – Reference to original (i.e. if original changes, you’ll see it)

• Cons
  – Can treat the copy as a scratch copy and modify freely
Why Use Pointers

• [All of these will be explained as we go...]
• To change a variable (or variables) local to one function in some other function
  – Requires pass-by-reference (i.e. passing a pointer to the other function)
• When large data structures are being passed (i.e. arrays, class objects, structs, etc.)
  – So the computer doesn’t waste time and memory making a copy
• When we need to ask for more memory as the program is running (i.e. dynamic memory allocation)
• To provide the ability to access specific location in the computer (i.e. hardware devices)
  – Useful for embedded systems programming
• Imagine a set of 18 safe deposit or PO boxes each with a number
• There are 8 boxes with gold and the other 10 do not contain gold but hold a piece of paper with another box number (i.e. a pointer to another box)
• Value of box 9 “points-to” box 7
• Value of box 17 “points-to” box 3
Pointers

• Pointers are references to other things
  – Really pointers are the address of some other variable in memory
  – “things” can be data (i.e. int’s, char’s, double’s) or other pointers

• The concept of a pointer is very common and used in many places in everyday life
  – Phone numbers, e-mail or mailing addresses are references or “pointers” to you or where you live
  – Excel workbook has cell names we can use to reference the data ( =A1 means get data in A1)
  – URL’s (www.usc.edu is a “pointer” to a physical HTML file) and can be used in any other page to “point to” USC’s website
Prerequisites: Data Sizes, Computer Memory

POINTER BASICS
Review Questions

• T/F: The elements of an array are stored contiguously in memory
  – True

• When an array is declared (i.e. int dat[10]) and its name is written by itself (e.g. x = dat;) in an expression, it evaluates to what?
  – The start address of the array
C++ Pointer Operators

- Two operators used to manipulate pointers (i.e. addresses) in C/C++: & and *
  - &variable evaluates to the "address-of" variable
    - Essentially you get a pointer to something by writing &something
  - *pointer evaluates to the data pointed to by pointer (data at the address given by pointer)
  - & and * are essentially inverse operations
    - We say ‘&’ returns a reference/address of some value while ‘*’ dereferences the address and returns the value
      - &value => address
      - *address => value
      - *(&value) => value
Pointers

- ‘&’ operator yields address of a variable in C (Tip: Read ‘&foo’ as ‘address of foo’)
  - int x = 30;  char y='a';
    float z = 5.375;
    int dat[2] = {107,43};
  - &x => ??,
  - &y => ??,
  - &z => ??,
  - &dat[1] = ??;
  - dat => ??
Pointers

- ‘&’ operator yields address of a variable in C (Tip: Read ‘&foo’ as ‘address of foo’)
  - int x = 30; char y='a';
    float z = 5.375;
    int dat[2] = {107,43};
  - &x => 0x20bc4,
  - &y => 0x20bc8,
    &z => 0x20bcc,
  - &dat[1] = 0x20bd4;
  - dat => 0x20bd0

- Number of bits used for an address depends on OS, etc.
  - 32-bit OS => 32-bit addresses
  - 64-bit OS => 64-bit addresses
Pointers

- Just as we declare variables to store int’s and double’s, we can declare a pointer variable to store the “address-of” (or “pointer-to”) another variable
  - Requires 4-bytes of storage in a 32-bit system or 8-bytes in a 64-bit systems
  - Use a ‘*’ after the type to indicate this a pointer variable to that type of data
    - Why did people choose '*' to declare a pointer variable?
    - Because you'd have to put a '*' in front of the variable to get an actual data item (i.e. to get the int that an int pointer points to, put a '*' in front of the pointer variable.

- Declare variables:
  - int x = 30; char y='a';
    float z = 5.375;
    int dat[2] = {107,43};
  - int *ptr1;
    ptr1 = &x;       // ptr1 = 0x20bc4
    ptr1 = &dat[0];  // Change ptr1 = 0x20bd0
    // (i.e. you can change what a pointer points to)
  - float *ptr2 = &z; // ptr2 = 0x20bcc
Pointers

• Just as we declare variables to store int’s and double’s, we can declare a pointer variable to store the “address-of” (or “pointer-to”) another variable
  – Requires 4-bytes of storage in a 32-bit system or 8-bytes in a 64-bit systems
  – Use a ‘*’ after the type to indicate this a pointer variable to that type of data
    • Why did people choose '*' to declare a pointer variable?
    • Because you'd have to put a '*1 in front of the variable to get an actual data item (i.e. to get the int that an int pointer points to, put a '*' in front of the pointer variable.

• Declare variables:
  – int x = 30; char y='a';
    float z = 5.375;
    int dat[2] = {107,43};
  – int *ptr1;
    ptr1 = &x; // ptr1 = 0x20bc4
    ptr1 = &dat[0]; // Change ptr1 = 0x20bd0
    // (i.e.you can change what a pointer points to)
  – float *ptr2 = &z; // ptr2 = 0x20bcc

  Memory
  
  20bc0  00
  20bc4  30
  20bc8  ‘a’
  20bcc  5.375
  20bd0  107
  20bd4  43
  20bd8  20bc4
  20bdc  20bcc
  20be0  00
  ...
  ...
De-referencing / Indirection

• Once a pointer has been written with an address of some other object, we can use it to access that object (i.e. dereference the pointer) using the ‘*’ operator

• Read ‘*foo’ as...
  – ‘value pointed to by foo’
  – ‘value at the address given by foo’
    (not ‘value of foo’ or ‘value of address of foo’)

• Using URL analogy, using the * operator on a pointer is like “clicking on a URL” (follow the link)

• Examples:
  – ptr1 = dat;
    int a = *ptr1 + 5;
  – (*ptr1)++;    // *ptr = *ptr + 1;
  – *ptr2 = *ptr1 - *ptr2;
De-referencing / Indirection

- Once a pointer has been written with an address of some other object, we can use it to access that object (i.e. dereference the pointer) using the ‘*’ operator
- Read ‘*foo’ as...
  - ‘value pointed to by foo’
  - ‘value at the address stored in foo’
    (not ‘value of foo’ or ‘value of address of foo’)
- By the URL analogy, using the * operator on a pointer is like “clicking on a URL” (follow the link)
- Examples:
  - `int a = 5 + *ptr1; // a = 112 after exec.`
  - `(*ptr1)++; // dat[0] = 108`
  - `*ptr2 = *ptr1 - *ptr2; // z = 108 - 5.375 = 102.625`
- ‘*’ in a type declaration = declare/allocate a pointer
- ‘*’ in an expression/assignment = dereference
Cutting through the Syntax

- ‘*’ in a type declaration = declare/allocate a pointer
- ‘*’ in an expression/assignment = dereference

<table>
<thead>
<tr>
<th></th>
<th>Declaring a pointer</th>
<th>De-referencing a pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *p</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>*p + 1</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>int *ptr</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>*ptr = 5</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>*ptr++</td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>char *p1[ 10 ];</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Helpful tip to understand syntax: We declare an int pointer as:
- int *p because when we dereference it as *p we get an int
- char *x is a declaration of a pointer and thus *x in code yields a char
Pointer Questions

• Chapter 13, Question 6

```c
int x, y;
int* p = &x;
int* q = &y;
x = 35; y = 46;
p = q;
*p = 78;
cout << x << " " << y << endl;
cout << *p << " " << *q << endl;
```
Prerequisites: Pointer Basics, Data Sizes

POINTER ARITHMETIC
Review Questions

• The size of an 'int' is how many bytes?
  – 4

• The size of a 'double' is how many bytes?
  – 8

• What does the name of an array evaluate to?
  – It's start address...
  – Given the declaration int dat[10], dat is an int*
  – Given the declaration char str[6], str is a char*

• In an array of integers, if dat[0] lived at address 0x200, dat[1] would live at...?
  – 0x204
Pointer Arithmetic

• Pointers are variables storing addresses and addresses are just numbers
• We can perform addition or subtraction on those pointer variables (i.e. addresses) just like any other variable
• The number added/subtracted is implicitly multiplied by the size of the object so the pointer will point to a valid data item
  – \( \text{int } *\text{ptr} = \text{dat} ; \text{ptr} = \text{ptr} + 1 ; \)
  – // address in ptr was incremented by 4
• Examples:
  – ptr1 = dat;
  – \( x = x + *\text{ptr1} ; \) // \( x = 137 \)
  – ptr1++; // ptr1 now points at dat[1]
  – \( x = x + *\text{ptr1}++ ; \) // \( x = \text{dat}[1] = 137 + 43 \) then
    // inc. ptr1 to 0x20bd8
  – ptr1 = ptr1-2; // ptr1 now points back at dat[0]
• Array indexing and pointer arithmetic are very much related
• Array syntax:  data[i]
  – Says get the i-th value from the start of the data array
• Pointer syntax:  *(data + i)
  – Says the same thing as data[i]
• We can use pointers and array names interchangeably:
  – int data[10];  // data = 520;
  – *(data + 4) = 50;  // data[4] = 50;
  – int* ptr = data;  // ptr now points at 520 too
Arrays & Pointers

• Have a unique relationship
• Array name evaluates to start address of array
  – Thus, the name of an integer array has type: int*
  – The name of character array / text string has type: char*
• Array indexing is same as pointer arithmetic

```c
int main(int argc, char *argv[])
{
    int data[10] = {9,8,7,6,5,4,3,2,1,0};
    int* ptr, *another;  // * needed for each
                        // ptr var. you
                        // declare
    ptr = data;         // ptr = start address
                        // of data
    another = data;     // another = start addr.

    for(int i=0; i < 10; i++){
        data[i] = 99;
        ptr[i] = 99;   // same as line above
        *another = 99; // same as line above
        another++;
    }

    int x = data[5];
    x = *(ptr+5);   // same as line above
    return 0;
}
```
Prerequisites: Pointer Basics

PASS BY REFERENCE
Pass by Value

• Notice that actual arguments are different memory locations/variables than the formal arguments
• When arguments are passed a copy of the actual argument value (e.g. 3) is placed in the formal parameter (x)
• The value of y cannot be changed by any other function (remember it is local)

```c
void decrement_it(int);
int main()
{
    int a, y = 3;
    decrement_it(y);
    cout << "y = " << y << endl;
}
void decrement_it(int y)
{
    y--;  
}
```

Address 0x00000000

System Memory (RAM)

Code for all functions

Data for decrement_it (y=3 then 2) and return link

Data for main (a, y=3) and return link

System stack area
Pass by Reference

- Pointer value (i.e. the address) is still passed-by-value (i.e. a copy is made)
- However, the value of the pointer is a reference to \( y \) (i.e. \( y \)'s address) and it is really the value of \( y \) that \( \textit{doit}() \) operates on
- Thus we say we are passing-by-reference
- The value of \( y \) is CHANGED by \( \textit{doit}() \) and that change is visible when we return.

```c
int main()
{
    int a, y = 3;
    // assume \( y \) @ 0x20bd4
    // assume \( \texttt{ptr} \)
    a = y;
    decrement_it(&y);
    cout << "a=" << a;
    cout << "y=" << y << endl;
    return 0;
}

void decrement_it(int* x)
{
    *x = *x - 1;
}
```

Resulting Output:
\( a=3, y=2 \)
Swap Two Variables

- Classic example of issues with local variables:
  - Write a function to swap two variables
- Pass-by-value doesn’t work
  - Copy is made of x,y from main and passed to x,y of swapit...Swap is performed on the copies
- Pass-by-reference (pointers) does work
  - Addresses of the actual x,y variables in main are passed
  - Use those address to change those physical memory locations

```c++
int main()
{
    int x=5, y=7;
    swapit(x, y);
    cout << "x=" << x << " y=";
    cout << y << endl;
}
void swapit(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Program output: x=5, y=7

```c++
int main()
{
    int x=5, y=7;
    swapit(&x, &y);
    cout << "x=" << x << " y=";
    cout << y << endl;
}
void swapit(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Program output: x=7, y=5
Passing Arrays as Arguments

- In function declaration / prototype for the **formal** parameter use
  - “type []” or “type *” to indicate an array is being passed

- When calling the function, simply provide the name of the array as the **actual** argument
  - In C/C++ using an array name without any index evaluates to the starting address of the array

- C does NOT implicitly keep track of the size of the array
  - Thus either need to have the function only accept arrays of a certain size
  - Or need to pass the size (length) of the array as another argument

```c
void add_1_to_array_of_10(int []);
void add_1_to_array(int *, int);

int main(int argc, char *argv[])
{
    int data[10] = {9,8,7,6,5,4,3,2,1,0};
    add_1_to_array_of_10(data);
    cout << "data[0]" << data[0] << endl;
    add_1_to_array(data,10);
    cout << "data[9]" << data[9] << endl;
    return 0;
}

void add_1_to_array_of_10(int my_array[])
{
    int i=0;
    for(i=0; i < 10; i++){
        my_array[i]++;
    }
}

void add_1_to_array(int *my_array, int size)
{
    int i=0;
    for(i=0; i < size; i++){
        my_array[i]++;
    }
}
```
#include <iostream>
using namespace std;

int main()
{
    int len = 0;
    int data[100];
    len = fill_data(data, 100);
    for(int i = 0; i < len; i++)
    {
        cout << data[i] << " ";
    }
    cout << endl;
    return 0;
}

// fills in integer array w/ int’s from user until -1 is entered
int fill_data(int *array, int max)
{
    int val = 0;
    int i = 0;
    while(i < max){
        cin >> val;
        if (val != -1)
            array[i++] = val;
        else
            break;
    }
    return i;
}
Exercises

• In class exercises
  – Swap
  – Roll2
  – Product
Prerequisites: Pointer Basics

POINTERS TO POINTERS
Pointers to Pointers Analogy

• We can actually have multiple levels of indirection (de-referencing)
• Using C/C++ pointer terminology:
  – *9 = gold in box 7  (9 => 7)
  – **16 = gold in box 3  (16 => 5 => 3)
  – ***0 = gold in box 3  (0 => 8 => 5 => 3)
Pointer Analogy

• What if now rather than holding gold, those boxes simply held other numbers
• How would you differentiate whether the number in the box was a “pointer” to another box or a simple data value?
  – You can’t really. Context is needed
• This is why we have to declare something as a pointer and give a type as well:
  – int *p; // pointer to an integer one hop (one level of indirection) away
  – double **q; // pointer to a double two hops (two levels of indirection) away
Pointers to Pointers to...

- Pointers can point to other pointers
  - Essentially a chain of “links”
- Example
  - int k, x[3] = {5, 7, 9};
  - int *myptr, **ourptr;
  - myptr = x;
  - ourptr = &myptr;
  - k = *myptr;
  - k = (**ourptr) + 1;
  - k = *(*ourptr + 1);

To figure out the type of data a pointer expression will yield…Take the type of pointer in the declaration and let each * in the expression 'cancel' one of the *'s in the declaration.

<table>
<thead>
<tr>
<th>Type Decl.</th>
<th>Expr</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>myptr = int*</td>
<td>*myptr</td>
<td>int</td>
</tr>
<tr>
<td>ourptr = int**</td>
<td>**ourptr</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>*ourptr</td>
<td>int*</td>
</tr>
</tbody>
</table>
Pointers to Pointers to…

• Pointers can point to other pointers
  – Essentially a chain of “links”
• Example
  – int k, x[3] = {5, 7, 9};
  – int *myptr, **ourptr;
  – myptr = x;
  – ourptr = &myptr;
  – k = *myptr; // k = 5
  – k = (**ourptr) + 1; // k = 6
  – k = *(*(ourptr + 1); // k = 7
Check Yourself

- Consider these declarations:
  - `int k, x[3] = {5, 7, 9};`
  - `int *myptr = x;`
  - `int **ourptr = &myptr;`

- Indicate the formal type that each expression evaluates to (i.e. `int, int *, int **`)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x[0]</code></td>
<td></td>
</tr>
<tr>
<td><code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>myptr</code></td>
<td></td>
</tr>
<tr>
<td><code>*myptr</code></td>
<td></td>
</tr>
<tr>
<td><code>*ourptr</code></td>
<td></td>
</tr>
<tr>
<td><code>myptr + 1</code></td>
<td></td>
</tr>
<tr>
<td><code>ourptr</code></td>
<td></td>
</tr>
</tbody>
</table>
Check Yourself

• Consider these declarations:
  – int k,x[3] = {5, 7, 9};
  – int *mypi = x;
  – int **ourptr = &mypi;

• Indicate the formal type that each expression evaluates to (i.e. int, int *, int **)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td>Int</td>
</tr>
<tr>
<td>x</td>
<td>Int *</td>
</tr>
<tr>
<td>mypi</td>
<td>Int *</td>
</tr>
<tr>
<td>*mypi</td>
<td>Int</td>
</tr>
<tr>
<td>*ourptr</td>
<td>Int *</td>
</tr>
<tr>
<td>mypi + 1</td>
<td>Int *</td>
</tr>
<tr>
<td>ourptr</td>
<td>Int **</td>
</tr>
</tbody>
</table>
ARRAYS OF POINTERS AND C-STRINGS
C-String Constants

• C-String constants are the things we type in "..." and are stored somewhere in memory (chosen by the compiler)

• When you pass a C-string constant to a function it passes the start address and it's type is known as a **const char ***
  – char* because you are passing the address
  – const because you cannot/should not change this array's contents

```cpp
#include <cstring>

//cstring library includes
//void strcpy (char * dest, const char* src);
int main(int argc, char *argv[])
{
    // These are examples of C-String constants
    cout << "Hello" << endl;
    cout << "Bye!" << endl;
    ...
}
```

```cpp
name = 240 279 300 305
T o m m y \0
```

```cpp
#include <cstring>

//cstring library includes
//void strcpy (char * dest, const char* src);
int main(int argc, char *argv[])
{
    char name[40];
    strcpy(name, "Tommy");
}
```

```cpp
name = 240 279 300 305
T o m m y \0
```
Arrays of pointers

• We often want to have several arrays to store data
  – Store several text strings
• Those arrays may be related (i.e. all names of students in a class)

```c
int main(int argc, char *argv[])
{
    int i;
    char str1[] = "Bill";
    char str2[] = "Suzy";
    char str3[] = "Pedro";
    char str4[] = "Ann";
    // I would like to print out each name
    cout << str1 << endl;
    cout << str2 << endl;
    ...
}
```

Painful

\[
\begin{array}{c}
\text{str1=240} & 244 \\
\text{str2=288} & 292 \\
\text{str3=300} & 305 \\
\text{str4=196} & 199
\end{array}
\]

\[
\begin{array}{c}
\text{Bill}\backslash 0 \\
\text{Suzy}\backslash 0 \\
\text{Pedro}\backslash 0 \\
\text{Ann}\backslash 0
\end{array}
\]
Arrays of pointers

• We often want to have several arrays to store data
  – Store several text strings
• Those arrays may be related (i.e. all names of students in a class)
• What type is 'names'?  
  – The address of the 0-th char* in the array
  – The address of a char* is really just a char**

```c
int main(int argc, char *argv[])
{
    int i;
    char str1[] = “Bill”;
    char str2[] = “Suzy”;
    char str3[] = “Pedro”;
    char str4[] = “Ann”;
    char *names[4];

    names[0] = str1; ...; names[3] = str4;
    for(i=0; i < 4; i++){
        cout << names[i] << endl;
    }
    ...  
}
```

Still painful
Arrays of pointers

• We can have arrays of pointers just like we have arrays of other data types

• Usually each value of the array is a pointer to a collection of “related” data
  – Could be to another array

```c

int main(int argc, char *argv[])
{
    int i;
    for(i=0; i < 4; i++){
        cout << names[i] << endl;
    }
    return 0;
}
```
Command Line Arguments

• Now we can understand the arguments passed to the main function (int argc, char *argv[])

• At the command prompt we can give inputs to our program rather than making querying the user interactively:
  – $ ./prog1  4  0.5  100000
  – $ cp broke.c broke2.c

• Command line string is broken at whitespaces and copied into individual strings and then packaged into an array (argv)
  – Each entry is a pointer to a string (char *)

• Argc indicates how long that arrays is (argv[0] is always the executable name)
Command Line Arguments

- **Recommended usage:**
  - Upon startup check argc to make sure the user has input the desired number of args (remember the executable counts as one of the args.)

- **Problem:**
  - Each argument is a text string...for numbers we want its numeric representation not its ASCII representation
  - `cstdlib` defines: `atoi()` [ASCII to Integer] and `atof()` [ASCII to float/double]
  - Each of these functions expects a pointer to the string to convert

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

// char **argv is the same as char *argv[]
int main(int argc, char **argv)
{
    int init, num_sims;
    double p;
    if(argc < 4){
        cout << "usage: prog1 init p sims" << endl;
        return 1;
    }
    init = atoi(argv[1]);
    p = atof(argv[2]);
    num_sims = atoi(argv[3]);
    ...
```
Review: String Function/Library
(#include <cstring>)

- int strlen(char *dest)
- int strcmp(char *str1, char *str2);
  - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
- char *strcpy(char *dest, char *src);
  - strncpy(char *dest, char *src, int n);
  - Maximum of n characters copied
- char *strcat(char *dest, char *src);
  - strncat(char *dest, char *src, int n);
  - Maximum of n characters concatenated plus a NULL
- char *strchr(char *str, char c);
  - Finds first occurrence of character ‘c’ in str returning a pointer to that character or NULL if the character is not found
Exercises

• Cmdargs_sum
• Cmdargs_smartsun
• Cmdargs_smartsun_str
• toi
Recap: Why Use Pointers

• To change a variable (or variables) local to one function in some other function
  – Requires pass-by-reference (i.e. passing a pointer to the other function)
• When large data structures are being passed (i.e. arrays, class objects, structs, etc.)
  – So the computer doesn’t waste time and memory making a copy
• To provide the ability to access specific location in the computer (i.e. hardware devices)
  – Useful for embedded systems programming
• When we need a variable address (i.e. we don’t or could not know the address of some desired memory location BEFORE runtime)
Pointer Basics

DYNAMIC MEMORY ALLOCATION
Dynamic Memory Allocation

• I want an array for student scores but I don’t know how many students we have until the user tells me

• What size should I use to declare my array?
  – int scores[??]

• Doing the following is not supported by all C/C++ compilers:
  
  ```
  int num;
  cin >> num;
  int scores[num];  // Some compilers require the array size to be statically known
  ```

• Also, recall local variables die when a function returns

• We can allocate memory dynamically (i.e. at run-time)
  – If we don't know how much we'll need until run-time
  – If we want memory to live beyond the end of a functions (i.e. we want to control when memory is allocated and deallocated)
Dynamic Memory & the Heap

- Code usually sits at low addresses
- Global variables somewhere after code
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
- Heap grows downward, stack grows upward...
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error
C Dynamic Memory Allocation

- **malloc**(int *num_bytes*) function in stdlib.h
  - Allocates the number of bytes requested and returns a pointer to the block of memory
- **free**(void *ptr) function
  - Given the pointer to the (starting location of the) block of memory, free returns it to the system for re-use by subsequent malloc calls
C++ new & delete operators

- **new** allocates memory from heap
  - replaces “malloc”
  - followed with the type of the variable you want or an array type declaration
    - double *dpotr = new double;
    - int *myarray = new int[100];
  - can obviously use a variable to indicate array size
    - **returns a pointer of the appropriate type**
      - if you ask for a new int, you get an int * in return
      - if you ask for an array (new int[10]), you get an int * in return
  - **delete** returns memory to heap
    - Replaces “free”
    - followed by the pointer to the data you want to de-allocate
      - delete dptr;
    - **use delete [] for arrays**
      - delete [] myarray;
Dynamic Memory Analogy

- Dynamic Memory is “ON-Demand Memory”
- Analogy: Public storage rentals
  - Need extra space, just ask for some storage and indicate how much you need (‘new’ statement with space allocated from the heap)
  - You get back the “address”/storage room number (‘new’ returns a pointer to the allocated storage)
  - Use the storage/memory until you are done with it
  - Need to return it when done or else no one else will ever be able to re-use it
Dynamic Memory Allocation

```c
int main(int argc, char *argv[]) {
    int num;
    cout << "How many students?" << endl;
    cin >> num;
    int *scores = new int[num];
    // can now access scores[0] .. scores[num-1];
    return 0;
}
```

```c
int main(int argc, char *argv[]) {
    int num;
    cout << "How many students?" << endl;
    cin >> num;
    int *scores = new int[num];
    // can now access scores[0] .. scores[num-1];
    delete [] scores
    return 0;
}
```
Fill in the Blanks

• ________ data = new int;

• ________ data = new char;

• ________ data = new char[100];

• ________ data = new char*[20];

• ________ data = new string;
Fill in the Blanks

• ________ data = new int;
  – int*
• ________ data = new char;
  – char*
• ________ data = new char[100];
  – char*
• ________ data = new char*[20];
  – char**
• ________ data = new string;
  – string*
Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw these as boxes in the heap area

```plaintext
// Computes rectangle area, // prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
  int wid = 8, len = 5, a;
  area(wid, len);
}

int* area(int w, int l)
{
  int* ans = new int;
  *ans = w * l;
  return ans;
}
```
Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw these as boxes in the heap area

```c
// Computes rectangle area, // prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
    int wid = 8, len = 5, a;
    area(wid, len);
}

int* area(int w, int l)
{
    int* ans = new int;
    *ans = w * l;
    return ans;
}
```

Stack Area of RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbf0</td>
<td>8</td>
</tr>
<tr>
<td>0xbf4</td>
<td>5</td>
</tr>
<tr>
<td>0xbf8</td>
<td>-73249515</td>
</tr>
<tr>
<td>0xbfc</td>
<td>0x00400120</td>
</tr>
</tbody>
</table>

Heap Area of RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x93c</td>
<td>40</td>
</tr>
</tbody>
</table>

MEMORY LEAK
No one saved a pointer to this data
Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw these as boxes in the heap area

```c
// Computes rectangle area, // prints it, & returns it
int* area(int, int);
void print(int);
int main()
{
    int wid = 8, len = 5, *a;
    a = area(wid, len);
    cout << *a << endl;
    delete a;
}
```

```c
int* area(int w, int l)
{
    int* ans = new int;
    *ans = w * l;
    return ans;
}
```
Exercises

• In-class-exercises
  – ordered_array
SHALLOW VS. DEEP COPY
Dealing with Text Strings

• What’s the best way to store text strings for data that we will not know until run time and that could be short or long?

• Statically:
  – Bad! Either wastes space or some user will enter a string just a little too long

```cpp
#include <iostream>
using namespace std;

int main()
{
    // store 10 user names of up to 40 chars
    char names[10][40];

    for(int i=0; i < 10; i++){
        cin >> names[i];
    }
}
```

```
names[0]  “Tim”
names[1]  “Christopher”
...
```
Jagged 2D-Arrays

• What we want is just enough storage for each text string
• This is known as a jagged 2D-Array since each array is a different length
• To achieve this we will need an array of pointers
  – Each pointer will point to an array of different length

```cpp
#include <iostream>
using namespace std;

int main()
{
    // store 10 user names of up to 40 chars
    char *names[10];

    for(int i=0; i < 10; i++){
        cin >> names[i];
    }
}
```

<table>
<thead>
<tr>
<th>names[0]</th>
<th>&quot;Tim&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>names[1]</td>
<td>&quot;Christopher&quot;</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Jennifer&quot;</td>
</tr>
</tbody>
</table>
More Dealing with Text Strings

• Will this code work to store 10 names?
  – http://cs103.usc.edu/websheets/#deepnames

• No!! You must allocate storage (i.e. an actual array) before you have pointers pointing to things...
  – Just because I make up a URL like: http://docs.google.com/uR45y781 doesn't mean there's a document there...

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    for(int i=0; i < 10; i++){
        cin >> names[i];
    }
    // Do stuff with names

    return 0;
}
```
More Dealing with Text Strings

- Will this code work to store 10 names?

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];

    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = temp_buf;
    }

    // Do stuff with names

    for(int i=0; i < 10; i++){
        delete [] names[i];
    }

    return 0;
}
```
More Dealing with Text Strings

• What’s the best way to store text strings for data that we will not know until run time and that could be short or long?
  • Dynamically:
    – Better memory usage
    – Requires a bit more coding

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];

    char temp_buf[40];
    for(int i=0; i < 10; i++)
    {
        cin >> temp_buf;
        // Find length of strings
        int len = strlen(temp_buf);
        names[i] = new char[len + 1];
        strcpy(names[i], temp_buf);
    }

    // Do stuff with names
    for(int i=0; i < 10; i++)
    {
        delete [] names[i];
    }
    return 0;
}
```
More Dealing with Text Strings

• What’s the best way to store text strings for data that we will not know until run time and that could be short or long?

• Dynamically:
  – Better memory usage
  – Requires a bit more coding

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        // Find length of strings
        int len = strlen(temp_buf);
        names[i] = new char[len + 1];
        strcpy(names[i], temp_buf);
    }
    // Do stuff with names
    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```
Shallow Copy vs. Deep Copy

• If we want to change the name, what do we have to do?
• Can we just use the assignment operator, ‘=’?

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }
    // What if I want to change names[0] & [1]
    cin >> temp_buf; // user enters “Allison”
    names[0] = temp_buf;
    cin >> temp_buf; // user enters “Jennifer”
    names[1] = temp_buf;
    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```
Shallow Copy vs. Deep Copy

- If we want to change the name, what do we have to do?
- Can we just use the assignment operator, ‘=’?

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }
    // What if I want to change names[0]&[1]
    cin >> temp_buf; // user enters “Allison”
    names[0] = temp_buf;
    cin >> temp_buf; // user enters “Jennifer”
    names[1] = temp_buf;
    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```
Shallow Copy vs. Deep Copy

- If we want to change the name, what do we have to do?
- Can we just use the assignment operator, ‘=’?

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];

    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }

    // What if I want to change names[0]&[1]
    cin >> temp_buf; // user enters “Allison”
    names[0] = temp_buf;
    cin >> temp_buf; // user enters “Jennifer”
    names[1] = temp_buf;

    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```

`temp_buf` evaluates to address of array. So `names[0] = temp_buf` simply copies address of array into names[0]...It does not make a copy of the array.
Shallow Copy vs. Deep Copy

- Pointers are references... assigning a pointer doesn’t make a copy of what its pointing at it makes a copy of the pointer (a.k.a. “shallow copy”)
  - Shallow copy = copy of pointers to data rather than copy of actual data

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }
    // What if I want to change names[0]&[1]
    cin >> temp_buf; // user enters “Allison”
    names[0] = temp_buf;
    cin >> temp_buf; // user enters “Jennifer”
    names[1] = temp_buf;
    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```

Same problem with assignment of temp_buf to names[1]. Now we have two things pointing at one array and we have lost track of memory allocated for Timothy and Christopher... memory leak!
Shallow Copy vs. Deep Copy

- Pointers are references... assigning a pointer doesn’t make a copy of what its pointing at.

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char *
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++)
    {
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }
    // What if I want to change names[0]&[1]
    cin >> temp_buf; // user enters “Allison”
    names[0] = temp_buf;
    cin >> temp_buf; // user enters “Jennifer”
    names[1] = temp_buf;
    for(int i=0; i < 10; i++)
    {
        delete [] names[i];
    }
    return 0;
}
```

When we try to “delete” or free the memory pointed to by names[i], it will now try to return memory it didn’t even allocate (i.e. temp_buf) and cause the program to crash!
Shallow Copy vs. Deep Copy

• If we want to change the name, what do we have to do?
• Can we just use the assignment operator, ‘=’? NO!
• Can we use strcpy()?
If we want to change the name, what do we have to do?

Can we just use the assignment operator, ‘=’? NO!

Can we use strcpy()? Not alone... what if name is longer.

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char* names[10];
    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }
    // What if I want to change names[0] & [1]
    cin >> temp_buf;  // user enters “Allison”
    strcpy(names[0], temp_buf);
    cin >> temp_buf;  // user enters “Jennifer”
    strcpy(names[1], temp_buf);
    for(int i=0; i < 10; i++){
        delete [] names[i];
    }
    return 0;
}
```
Deep Copies

• If we want to change the name, what do we have to do?

• Must allocate new storage and copy original data into new memory (a.k.a. “deep copy”)
  – Deep copy = Copy of data being pointed to into new memory

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    // store 10 user names
    // names type is still char **
    char *names[10];

    char temp_buf[40];
    for(int i=0; i < 10; i++){
        cin >> temp_buf;
        names[i] = new char[strlen(temp_buf)+1];
        strcpy(names[i], temp_buf);
    }

    // What if I want to change names[0]&[1]
    cin >> temp_buf; // user enters “Allison”
    delete [] names[0];
    names[0] = new char[strlen(temp_buf)+1];
    strcpy(names[0], temp_buf);

    cin >> temp_buf; // user enters “Jennifer”
    delete [] names[1];
    names[1] = new char[strlen(temp_buf)+1];
    strcpy(names[1], temp_buf);
    ...}
```
Exercise

• In-class-exercises
  – nxmboard
END LECTURE
• 8 Index Cards:
  – Number 800,804,808,...832 in upper left
  – int x = 1298;  char y='a';
    float z = 5.375;
    int dat[2] = {107,43};
    • Variable name in upper right, value in middle
• Practice '&' operator
  – &x => ??,
  – &y => ??,
  – &z => ??,
  – &dat[1] = ??;
  – dat => ??
• Practice '*' operator
  – *(800),
  – *(812),
• Pointer variable decl.
  – Take cards for 820,824
  – int *ptr1 = &x;
    ptr1 = &dat[1];
  – double *ptr2;
    ptr2 = &z;
• Dereference practice
  – int a = 5;
    a = a + *ptr1;
  – (*ptr1)++;
  – *ptr2 = *ptr1 - *ptr2;
• Pointer Arithmetic
  – int *ptr = dat; ptr = ptr + 1;
  – // address in ptr was incremented by 4
  – ptr1++;  // ptr1 now points at dat[1]
  – (*ptr1)++;  // dat[0] = 108
  – x=*ptr1++;  // x = dat[1] = 43 then inc. ptr1 to
    *(ptr1-2)++;  // dat[0] = 109
  – 0x20bd8
• Pointer Arithmetic
  – int *ptr1 = dat; ptr1 = ptr1 + 1;
  – // address in ptr was incremented by 4
  – ptr1++;   // ptr1 now points at dat[1]
  – (*ptr1)++;  // dat[0] = 108
  – x=*ptr1++;  // x = dat[1] = 43 then inc. ptr1 to
  – Ptr1 = ptr1-2;

• Pointers to Pointers
• 2 New Cards
  – int *mynptr = &dat[0];
  – int **ourptr = &mynptr;
  – x = *mynptr;
  – x = (**ourptr) + 1;
  – x = *(*ourptr + 1);
Arrays of pointers

- We often want to have several arrays to store data
  - Store several text strings
- Those arrays may be related (i.e. scores of students in a class)

```c
int stu1scores[5] = {0,0,0,0,0};
int stu2scores[5] = {0,0,0,0,0};
int stu3scores[5] = {0,0,0,0,0};
int stu4scores[5] = {0,0,0,0,0};

int main(int argc, char *argv[])
{
    int avg = 0;
    for(i=0;i < 5;i++) { avg += stu1scores[i]; }  
    for(i=0;i < 5;i++) { avg += stu2scores[i]; }  
    for(i=0;i < 5;i++) { avg += stu3scores[i]; }  
    for(i=0;i < 5;i++) { avg += stu4scores[i]; }  
    avg /= 4*5;
}
```

Painful

```plaintext
stu1scores = 240
0 0 0 0 0

stu2scores = 300
0 0 0 0 0

stu3scores = 480
0 0 0 0 0

stu4scores = 800
0 0 0 0 0
```
Activity

• Write a program that
  – Defines a function that accepts an integer, `len` as input and then generates an array of `len` random integers in the range [00-99] and returns it to the caller
  – From main ask the user for `len` via `cin`, call your function and "store" the return result
  – Iterate over the array returned by your function and average the values.
  – Print that average...is that value close to ______________ (expected value)?
  – Should the array be locally allocated or dynamically?
  – Go back & have `len` be entered from the command line