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 or 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 a specific location in the computer (i.e. hardware devices)
  – Useful for embedded systems programming
### Pointer Analogy

- Imagine a set of 18 safe deposit or PO boxes each with a number.
- There are 8 boxes with gold jewelry 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.

![Pointer Analogy: A hand is reaching for a safe deposit box with numbers on it.](image)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>3</td>
</tr>
</tbody>
</table>
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)
  – URLs (www.usc.edu is a pointer to a physical HTML file on some server) and can be used in any other page to "point to" USC’s website

Memory

520 is a “pointer” to the integer 9
536 is a “pointer” to the integer 5
Prerequisites: Data Sizes, Computer Memory

POINTER BASICS
Review Questions

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

• When an array is declared (i.e. int dat[10]) and its name is written by itself (e.g. cout << dat;) in an expression, it evaluates to what?
  – ____________________________
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
    • More on why this syntax was chosen in a few slides...

• Declare variables:
  – int x = 30;  char y='a';  
    float z = 5.375;
    int dat[2] = {107,43};
  – int *ptr1;
    ptr1 = &x;   // ptr1 = ______________
    ptr1 = &dat[0];   // Change ptr1 = ______________
    // i.e. you can change what a pointer points to
  – float *ptr2 = &z; // ptr2 = ____________
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
    - More on why this syntax was chosen in a few slides...

- 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
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 += 1; // *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:
  - `ptr1 = dat;`
    - `int a = *ptr1 + 5;` // `a = 112` after exec.
  - `*ptr1 += 1;` // `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>Yes</td>
<td></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
• 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?
  – ____

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

• What does the name of an array evaluate to?
  – __________________
    – Given the declaration int dat[10], dat is an _____
    – Given the declaration char str[6], str is a _____

• In an array of integers, if dat[0] lived at address 0x200, dat[1] would live at...?
  – ___________
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.
  - `int *ptr1 = dat; ptr1 = ptr1 + 1;` // address in ptr was incremented by 4
- Examples:
  - `ptr1 = dat;`
  - `x = x + *ptr1;` // x = 137
  - `ptr1 = ptr1 + 1;` // ptr1 now points at dat[1]
  - `x = x + *ptr1++;` // x = dat[1] = 137+43 then // inc. ptr1 to 0x20bd8
  - `ptr1 = ptr1-2;` // ptr1 now points back at dat[0]
Pointer Arithmetic and Array Indexing

• Array indexing and pointer arithmetic are very much related
• Array syntax: \texttt{data[i]}
  – Says get the \texttt{i}-th value from the start of the data array
• Pointer syntax: \texttt{*(data + i) <=> data[i]}
  – Both of these get the \texttt{i}-th value in an array
• We can use pointers and array names interchangeably:
  – \texttt{int data[10]; // data = 520;}
  – \texttt{*(data + 4) = 50; // data[4] = 50;}
  – \texttt{int* ptr = data; // ptr now points at 520 too}

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>520 524 528 532 536 540</td>
</tr>
</tbody>
</table>

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
520 & 524 & 528 & 532 & 536 & 540 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
09 & 08 & 07 & 06 & 50 & 04 & ... \\
\hline
\end{tabular}
\end{center}

\begin{center}
\textbf{Memory}
\end{center}
Arrays & Pointers

• Array names and pointers have a unique relationship
  – 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;
}
```
Pass by Reference

Prerequisites: Pointer Basics
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).

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Code for all functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000</td>
<td>Data for decrement_it</td>
</tr>
<tr>
<td></td>
<td>(y=3 then 2) and return link</td>
</tr>
<tr>
<td>0xffff ffff</td>
<td>Data for main (a, y=3) and return link</td>
</tr>
<tr>
<td></td>
<td>System stack area</td>
</tr>
</tbody>
</table>
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 doit() operates on
- Thus we say we are passing-by-reference
- The value of y is CHANGED by doit() and that change is visible when we return.

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

// Remember * in a type
// declaration means "pointer"
// variable
void decrement_it(int* x)
{
    *x = *x - 1;
}
```

Resulting Output:

```
a=3, y=2
```

Address 0x00000000

System Memory (RAM)

Data for main (a=3, y=2) and return link

Data for doit (x=0x20bd4) and return link

Code for all functions

System stack area

0xffff ffff
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
```
Correct Usage of Pointers

- Commonly functions will take some inputs and produce some outputs
  - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
  - We could use the return value from the function but let's practice with pointers
- Can use a pointer to have a function modify the variable of another

```c
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);

int main()
{
  int wid = 8, len = 5, a;
  mul2(wid, len, &a);
  cout << "Ans. is " << a << endl;
  return 0;
}

int mul1(int in1, int in2)
{
  return in1 * in2;
}

void mul(int in1, int in2, int* out)
{
  *out = in1 * in2;
}
```
Misuse of Pointers/References

- Make sure you don't return a pointer to a dead variable
- You might get lucky and find that old value still there, but likely you won't

```c++
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);
int& badmul2(int in1, int in2);

int main()
{
    int wid = 8, len = 5;
    int *a = badmul1(wid, len);
    cout << "Ans. is " << *a << endl;
    return 0;
}

// Bad! Returns a pointer to a var. that will go out of scope
int* badmul1(int in1, int in2)
{
    int out = in1 * in2;
    return &out;
}
```
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_v1(int [], int);
void add_1_to_array_v2(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_v1(data);
    cout << “data[0]” << data[0] << endl;
    add_1_to_array_v2(data);
    cout << “data[0]” << data[0] << endl;
    return 0;
}
void add_1_to_array_v1(int my_array[], int size)
{
    int i=0;
    for(i=0; i < 10; i++){
        my_array[i]++;
    }
}
void add_1_to_array_v2(int *my_array, int size)
{
    int i=0;
    for(i=0; i < size; i++){
        my_array[i]++;
    }
}
```
Argument Passing Example

```cpp
#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] << " ";
    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;
}
```

Memory (RAM)

Address

0

Code

... 0xbf004

Globals

... 0xbf008

Heap

... 0xbf00c

fill_data (array=0xbf008, max = 100 val=0, i = 0)

main:
(len = 0
  data[0] = ?
  data[1] = ?
  data[2] = ?
  ...
)

fill_data (array=0xbf008, max = 100 val = -1, i = 2)

main:
(len = 2
  data[0] = 4
  data[1] = 3
  data[2] = ?
  ...
)
Exercises

• In class exercises
  – 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)
• 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=?
  – k = (**ourptr) + 1; // k=?
  – k = (**ourptr + 1); // k+?
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

To figure out the type 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>
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>Orig. Type</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>
<tr>
<td>k = int</td>
<td>&amp;k</td>
<td>int*</td>
</tr>
<tr>
<td></td>
<td>&amp;myptr</td>
<td>int**</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;x[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>&amp;k</td>
<td></td>
</tr>
<tr>
<td>myptr</td>
<td></td>
</tr>
<tr>
<td>*myptr</td>
<td></td>
</tr>
<tr>
<td>(*ourptr) + 1</td>
<td></td>
</tr>
<tr>
<td>myptr + 2</td>
<td></td>
</tr>
<tr>
<td>&amp;ourptr</td>
<td></td>
</tr>
</tbody>
</table>

To figure out the type of data a pointer expression will yield...
- Each * in the expression cancels a * from the variable type.
- Each & in the expression adds a * to the variable type.
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>x[0]</td>
<td>int</td>
</tr>
<tr>
<td>x</td>
<td>int*</td>
</tr>
<tr>
<td>&amp;k</td>
<td>int*</td>
</tr>
<tr>
<td>myptr</td>
<td>int*</td>
</tr>
<tr>
<td>*myptr</td>
<td>int</td>
</tr>
<tr>
<td>&amp;myptr</td>
<td>int**</td>
</tr>
<tr>
<td>ourptr</td>
<td>int**</td>
</tr>
<tr>
<td>*ourptr</td>
<td>int*</td>
</tr>
<tr>
<td>myptr + 1</td>
<td>int*</td>
</tr>
</tbody>
</table>
ARRAYS OF POINTERS AND C-STRINGS
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
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

```c
#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;
    ...
}
```

```c
int main(int argc, char *argv[])
{
    char name[40];
    strcpy(name, "Tommy");
}
```
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

- `str1` = 240, 244
- `str2` = 288, 292
- `str3` = 300, 305
- `str4` = 196, 199
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
char *names[4] ={"Bill", 
    "Suzy", 
    "Pedro", 
    "Ann"};

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]);
...
```
cin/cout & char*s

- cin/cout determine everything they do based on the type of data passed
- cin/cout have a unique relationship with char*s
- When cout is given a variable of any type it will print the value stored in that exact variable
  - Exception: When cout is given a char* it will assume it is pointing at a C-string, go to that address, and loop through each character, printing them out
- When cin is given a variable it will store the input data in that exact variable
  - Exception: When cin is given a char* it will assume it is pointing at a C-string, go to that address, and place the typed characters in that memory

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

int main()
{
    int x = 5, dat[10]; // dat is like an int*
    char word[10] = "Hello";
    char *name = word;

    cout << x << endl;    /* 5 */
    cout << dat << endl;   /* 400 */
    cout << word << endl;  /* Hello */
    cout << name << endl;  /* Hello */
    cout << name[0] << endl; /* H */
    cout << (void*) name << endl; /* 440 */

    cin >> x;      /* Store into x (@396) */
    cin >> name;   /* Store string starting at 440 */

    return 0;
}
```
Exercises

- Cmdargs_sum
- Cmdargs_smartsum
- Cmdargs_smartsum_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 \textit{dynamically} (i.e. at run-time)
  – \textbf{If we want memory to live beyond the end of a functions} (i.e. we want to control when memory is allocated and deallocated)
    • This is the primary reason we use dynamic allocation
  – If we don't know how much we'll need until run-time
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 & 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 *dptr = 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 new 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)
- This code fails to save a pointer to the new int once area() finishes

```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 (HEX)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbe0</td>
<td>0x93c</td>
</tr>
<tr>
<td>0xbe4</td>
<td>8</td>
</tr>
<tr>
<td>0xbe8</td>
<td>5</td>
</tr>
<tr>
<td>0xbec</td>
<td>0x04000ca0</td>
</tr>
</tbody>
</table>

Heap Area of RAM

<table>
<thead>
<tr>
<th>Address (HEX)</th>
<th>Contents</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>0x0400120</td>
</tr>
</tbody>
</table>

This Photo by Unknown Author is licensed under CC BY-SA
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)

- This code fails to save a pointer to the new int once area() finishes

```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

- Main
- Wid
- Len
- A

Heap Area of RAM

- 0x93c
- 40

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)
- I must keep at least 1 pointer to dynamic memory at all times until I delete it

```
// 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; // 40
}
```

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

**Never** return a pointer to a local variable

---

```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;
}

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

---

**Stack Area of RAM**

<table>
<thead>
<tr>
<th>0xbe0</th>
<th>0xbe4</th>
<th>0xbe8</th>
<th>0xbcfc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans</td>
<td>w</td>
<td>l</td>
<td>Return link</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>5</td>
<td>004000ca0</td>
</tr>
</tbody>
</table>

**Heap Area of RAM**

<table>
<thead>
<tr>
<th>0xbf0</th>
<th>0xbf4</th>
<th>0xbf8</th>
<th>0xbfc</th>
</tr>
</thead>
<tbody>
<tr>
<td>wid</td>
<td>len</td>
<td>a</td>
<td>Return link</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>-73249515</td>
<td>00400120</td>
</tr>
</tbody>
</table>
Pointer Mistake

- **Never** return a pointer to a local variable
- Pointer will now point to dead memory and the value it was pointing at will be soon corrupted/overwritten
- We call this a dangling pointer (i.e. a pointer to bad or dead memory)

```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;
}
```

```
int* area(int w, int l)
{
    int ans;
    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

```c++
#include <iostream>
using namespace std;

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

    names[0] = “Tim”;
    names[1] = “Christopher”;
    ...
```

- 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
    char *names[10];

    for(int i=0; i < 10; i++){
        /* read in and store each name */
    }
}
```

```plaintext
names[0] "Tim"
names[1] "Christopher"
...
"Jennifer"
```
More Dealing with Text Strings

- Will this code work to store 10 names?
  - Exercise: 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](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* 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];

    // One "scratchpad" array to read in a name
    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
    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, ‘=’?

```
#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, ‘=‘?

#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, ‘=’?

```
#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!*

---

Jennifer

```
temp_buf: 0x1c0: “Jennifer”
0x8a4
```

Timothy

```
0x1c0:
0x980
```

Christopher

```
0x980
```
Shallow Copy vs. Deep Copy

- Pointers are references... assigning a pointer doesn’t make a copy of what its pointing at
- Deleting the same memory **twice** will cause the program to **crash**

```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

- Can we use `strcpy()` instead?

```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;
}
```
Shallow Copy vs. Deep Copy

- Can we use strcpy() instead?
- **No!** Because what if the new name is **longer** than the array allocated for the old name...we'd write off the end of the array and corrupt 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”
    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 = allocate new memory AND then copy the original data (1 by 1) to the 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