Common Error: Returning a Pointer to a Local Variable

What would it mean to
"return an array"
?

Common Error: Returning a Pointer to a Local Variable

Consider this function that tries to return
a pointer to an array containing two elements,
the first and last values of an array:

```
double* firstlast(double a[], int size)
{
   double result[2];
   result[0] = a[0];
   result[1] = a[size - 1];
   return result;
}
```

Local memory is invalid after the function call has ended!

What would the value the caller gets be pointing to?

Common Error: Returning a Pointer to a Local Variable

A solution would be to pass
in an array to hold the answer:

```
double* firstlast(double a[], int size,
                  double[] result)
{
   result[0] = a[0];
   result[1] = a[size - 1];
}
```


C and C++ Strings, POP QUIZ (7.3)

"Q: What?"

Really we mean:

"Q: What is this?"

A *C string*, of course!
(notice the double quotes: "Like this")

C and C++ Strings

C++ has two mechanisms for manipulating strings.

The `string` class
- Supports character sequences of arbitrary length.
- Provides convenient operations such as concatenation and string comparison.

we did this in CIS2

C strings
- Provide a more primitive level of string handling.
- Are from the C language (C++ was built from C).
- Are represented as arrays of `char` values.

char data type could only hold 1 single character

`char` Type and Some Famous Characters

The type `char` is used to store an individual character.

char Type and Some Famous Characters

Some of these characters are plain old letters and such:

```
char yes = 'y';
char no = 'n';
char maybe = '?';
```


char Type and Some Famous Characters

Some are numbers masquerading as digits:

```
char theThreeChar = '3';
```

That is not the number three – it's the *character* 3.
`'3'` is what is actually stored in a disk file when you write the `int` 3.
Writing the variable `theThreeChar` to a file would put the same `'3'` in a file.

char Type and Some Famous Characters

Recall that a stream is a sequence of characters – `chars`.

char Type and Some Famous Characters

So some characters are literally what they are:

`'A'`

Some represent digits:

`'3'`

Some are other things that can be typed:

`'C'`
`'+'`
`'+'`

but…

Some Famous Characters

Some of these characters are true individuals.
"Characters" you might say (if they were human).

They are quite "special":

`'\n'`
`'\t'`

These are still single (individual) characters:
the *escape sequence* characters.

the backslash turns off usual meaning and gives it its special meaning

Some Famous Characters

And one you can output to the screen in order to annoy those around you
(if you were naughty and didn't mute your computer when you entered the classroom)

`'\a'`

– the *alert* character.

Don't try this at home
– no we mean
ONLY try this at home!!!

Some Famous Characters

And there is one special character that is especially special to C strings:

The null terminator character:

`'\0'`

That is an escaped zero.
It's in ASCII position zero.
It is the value 0 (not the character zero, `'0'`)
If you output it to screen nothing will appear.

Some Famous Characters

Table 3 Character Literals

`'y'`	The character y
`'0'`	The character for the digit 0. In the ASCII code, `'0'` has the value 48.
`' '`	The space character
`'\n'`	The newline character acts like endl
`'\t'`	The tab character
`'\0'`	The null terminator of a string
`"y"`	**Error:** Not a char value

no double quotes for char type

The Null Terminator Character and C Strings

The null character is special to C strings because it is always the last character in them:

`"CAT"` is really this sequence of characters:

`'C' 'A' 'T' '\0'`

end of "CAT"

The null terminator character indicates the end of the C string

The Null Terminator Character and C Strings

The literal C string `"CAT"` is actually an array of ***four*** `chars` stored somewhere in the computer.

In the C programming language, literal strings are always stored as character arrays.

Now you know why C++ programmers often refer to arrays of char values as "C strings".

Pop Quiz #2.

Q:

Is `"C strings"` a string?

Yes
...wait...
No
...wait...

Pop Quiz #2

Answer:

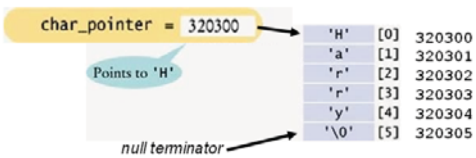
`"C strings"` is NOT an object of `string` type.
`"C strings"` IS an array of `chars` with a null terminator character at the end.

(and that English was correct!)

Character Arrays as Storage for C Strings

As with all arrays, a string literal can be assigned to a pointer variable that points to the initial character in the array:

```
char* char_pointer = "Harry";
     // Points to the 'H'
```



Using the Null Terminator Character

Functions that operate on C strings rely on this terminator.
The `strlen` function returns the length of a C string.

```
#include <cstring>
int strlen(const char s[])
{
   int i = 0;
   // Count characters before
   // the null terminator
   while (s[i] != '\0') { i++; }
   return i;
}
```


Using the Null Terminator Character

The call `strlen("Harry")` returns 5.

The null terminator character is not counted as part of the “length” of the C string – but it's there.

Really, it is.

Character Arrays

Literal C strings are considered constant.

You are not allowed to modify its characters.

Character Arrays

If you want to modify the characters in a C string, define a character array to hold the characters instead.

For example:

```
// An array of 6 characters
char char_array[] = "Harry";
```

Isn't something missing?

Character Arrays

The compiler counts the characters in the string that is used for initializing the array, including the null terminator.

```
char char_array[] = "Harry";
```

(6)

`I'm the compiler && I can count to 6`
`&& I wasn't fooled by that null terminator`

Character Arrays

The compiler counts the characters in the string that is used for initializing the array, including the null terminator.

```
char char_array[] = "Harry";
                    ↑
                   (6)
I'm the compiler && I put that 6 there
```


Character Arrays

You can modify the characters in the array:

```
char char_array[] = "Harry";
char_array[0] = 'L';

I'm the programmer && I changed Harry into Larry!
```


Converting Between C and C++ Strings

The cstdlib header declares a useful function:

`int atoi(const char s[])`

a-to-i or char-to-int

parameter is a char array

The `atoi` function converts a character array containing digits into its integer value:

```
char* year = "2012";
int y = atoi(year);
```

`y` is the integer 2012

Converting Between C and C++ Strings

Unfortunately there is nothing like this for the `string` class!
(can you believe that?!)

The `c_str` member function offers an "escape hatch":

converts a string variable to a Cstring array of characters then a-to-i works

```
string year = "2012";
int y = atoi(year.c_str());
```

Again, `y` is the integer 2012

Converting Between C and C++ Strings

Converting from a C string to a C++ string is very easy:

```
string name = "Harry";
```

`name` is initialized with the C string `"Harry"`.

Converting Between C and C++ Strings

Up to this point, we have always used the `substr` member function to access individual characters in a C++ `string`:

```
string name = "Harry";
```

`...name.substr(3, 1)...`

yields a string of length 1
containing the character at index 3
(the second 'r')

Converting Between C and C++ Strings

You can access individual characters with the [] operator:

↑ array notation

```
string name = "Harry";
name[3] = 'd';
```

b/c C++ strings are automatically also C strings

I'm the programmer && I changed Harry into Hardy!

Converting Between C and C++ Strings

You can write a function that will return the uppercase version of a `string`.

The `toupper` function is defined in the `cctype` header.

It converts lowercase characters to uppercase.

(The `tolower` function does the opposite.)

does one character at a time

```
char ch = toupper('a');
```

`ch` contains `'A'`

Converting Between C and C++ Strings

```
/**
   Makes an uppercase version of a string
   @param str a string
   @return a string with the characters in str converted to uppercase
*/
string uppercase(string str)
{
   string result = str; // Make a copy of str
   for (int i = 0; i < result.length(); i++)
   {
      // Convert each character to uppercase
      result[i] = toupper(result[i]);
   }
   return result;
}
```

↖ used in a loop to work w/each character at a time

C String Functions

Table 4 C String Functions ← char arrays

In this table, s and t are character arrays; n is an integer.

Function	Description
`strlen(s)`	Returns the length of s.
`strcpy(t, s)`	Copies the characters from s into t.
`strncpy(t, s, n)`	Copies at most n characters from s into t.
`strcat(t, s)`	Appends the characters from s after the end of the characters in t.
`strncat(t, s, n)`	Appends at most n characters from s after the end of the characters in t.
`strcmp(s, t)`	Returns 0 if s and t have the same contents, a negative integer if s comes before t in lexicographic order, a positive integer otherwise.

#include < check >

Dynamic Memory Allocation (7.4)

In many programming situations, you know you will be working with several values.

You would normally use an array for this situation, right?

(yes)

Dynamic Memory Allocation

But suppose you do not know beforehand how many values you need.

So now can you use an array?

(oh dear!)

Dynamic Memory Allocation

The size of a *static array* must be known when you define it.

can define it w/a CAPACITY value, but how do you know if CAPACITY is large enough?

Dynamic Memory Allocation

To solve this problem, you can use

dynamic allocation.

Dynamic arrays are not static.

(Static, like all facts.)

Dynamic Memory Allocation

To use dynamic arrays, you ask the C++ run-time system to create new space for an array whenever you need it.

This is at RUN-TIME?

On the fly?

Arrays on demand!

(cool)

Dynamic Memory Allocation

Where does this memory for my on-demand arrays come from?

The OS *keeps*

a *heap*:

a *Heap*`O´RAM

(to give to good little programmers like you)
(and poets)

Dynamic Memory Allocation

Yes, it's really called:

The Heap

(or sometimes the *freestore*
– and it really is free!
All you have to do is ask)

Dynamic Memory Allocation

To ask for more memory, say a `double`, you use the `new` operator:

`new double`

the runtime system seeks out room for a `double` on the heap, reserves it just for your use and returns a pointer to it.

This `double` location does not have a name.

(this is run-time)

Dynamic Memory Allocation

But just how useful is one single `double`?

(Not very)

Dynamic Memory Allocation

How about a brand new array from that Heap'O'RAM?

(Yes, please)

Dynamic Memory Allocation

To request a dynamic array you use the same `new` operator with some looks-like-an-array things added:

```
new double[n]
```

where `n` is the number of `double`s you want and, again, you get a pointer to the array.

an array of `double`s on demand!

Dynamic Memory Allocation

You need a pointer variable to hold the pointer you get:

Example of how to set it up

```
double* account_pointer = new double;
double* account_array = new double[n];
```

Now you can use `account_array` as an array.

The magic of array/pointer duality lets you use the array notation `account_array[i]` to access the `i`th element.

Dynamic Memory Allocation

When your program no longer needs the memory that you asked for with the `new` operator, you must return it to the heap using the `delete` operator for single areas of memory (which you would probably never use anyway).

```
delete account_pointer;
```


Dynamic Memory Allocation

Or more likely, you allocated an array. So you must use the `delete[]` operator.

```
delete[] account_array;
```

to delete dynamic array memory

do this as soon as you are done w/the array (to be efficient)

Dynamic Memory Allocation

After you delete a memory block,
you can no longer use it.
The OS is very efficient – and quick – "your" storage space may already be used elsewhere.

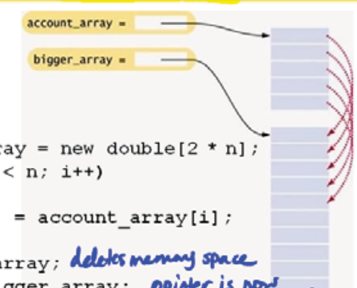
```
delete[] account_array;
account_array[0] = 1000;
      // NO! You no longer own the
      // memory of account_array
```


Dynamic Memory Allocation

Unlike static arrays,
which you are stuck with after you create them,
you can change the size of a dynamic array.

Make a new, improved, bigger array
and copy over the old data – but remember
to delete what you no longer need.

Dynamic Memory Allocation – Resizing an Array




```
double* bigger_array = new double[2 * n];
for (int i = 0; i < n; i++)
{
   bigger_array[i] = account_array[i];
}
delete[] account_array;
account_array = bigger_array;
n = 2 * n;
```

deletes memory space

pointer is now assigned a new value

(n is the variable used with the array)

Dynamic Memory Allocation – Serious Business

Son,
we need to talk.

We need to have a serious discussion about *safety*.

Safety and security are very important issues.

Really – THIS IS SERIOUS
Sit down!

Dynamic Memory Allocation – Serious Business

Son, heap allocation is a powerful feature,
and you have proven yourself to be a responsible enough programmer to begin using dynamic arrays
but you must be very careful to

follow these rules precisely:

Dynamic Memory Allocation – THE RULES

1. Every call to `new` *must* be matched by exactly one call to `delete`.
2. Use `delete[]` to delete arrays. And always assign `NULL` to the pointer after that.
3. Don't access a memory block after it has been deleted.

If you don't follow these rules, your program can
crash or *run unpredictably*

Dynamic Memory Allocation

SYNTAX 7.2 Dynamic Memory Allocation

```
int* var_ptr = new int;
...
*var_ptr = 1000;
...
delete var_ptr;

int* array_ptr = new int[size];
...
array_ptr[i] = 1000;
...
delete[] array_ptr;
```

Capture the pointer in a variable.

The new operator yields a pointer to a memory block of the given type.

Use the memory.

Delete the memory when you are done.

Use this form to allocate an array of the given size (size need not be a constant).

Use the pointer as if it were an array.

Remember to use delete[] when deallocating the array.

Dynamic Memory Allocation – Common Errors

Table 5 Common Memory Allocation Errors

Statements	Error
`int* p;` `*p = 5;` `delete p;`	There is no call to new int.
`int* p = new int;` `*p = 5;` `p = new int;`	The first allocated memory block was never deleted.
`int* p = new int[10];` `*p = 5;` `delete p;`	The delete[] operator should have been used.
`int* p = new int[10];` `int* q = p;` `q[0] = 5;` `delete p;` `delete q;`	The same memory block was deleted twice.
`int n = 4;` `int* p = &n;` `*p = 5;` `delete p;`	You can only delete memory blocks that you obtained from calling new.

Common Errors Dangling Pointers – Serious Business

Son, there's more:

DANGLING

Dangling pointers are when you use a pointer that has already been deleted or was never initialized.

Common Errors Dangling Pointers – Serious Business

```
int* values = new int[n];
// Process values

delete[] values;

// Some other work
values[0] = 42;
```

Good, son.
Being responsible!

Son!
NO!!!

trying to point to something that does not exist anymore

Common Errors Dangling Pointers – Serious Business

The value in an uninitialized or deleted pointer might point somewhere in the program you have no right to be accessing.

You can create real damage by writing to the location to which it points.

It's not yours to play with, son.

Common Errors Dangling Pointers – Serious Business

Even just *reading* from that location can crash your program.

You've seen what's happened to other programs.

Common Errors Dangling Pointers – Serious Business

Remember what happened to Jimmy?
A dialog box with a bomb icon.

And Ralph?
"General protection fault."

And poor Henry's son?
"Segmentation fault" came up,
and the program ***was terminated.***

Common Errors Dangling Pointers – Serious Business

Or worse, son – you could hurt *yourself!*

If that dangling pointer points at your own data,
and you write to it –

you may very well have messed up your own
future,
your own data!

Just don't do it, son!

Common Errors Dangling Pointers – Serious Business

Son, programming with pointers requires *iron discipline.*

- Always initialize pointer variables.
- If you can't initialize them with the return value of `new` or the `&` operator, then set them to `NULL`.
- Never use a pointer that has been deleted.

Common Errors Memory Leaks – Serious Business

And Son, I'm sorry to say, there's even more:

LEAKS

A *memory leak* is when use new to get dynamic memory but you fail to delete it when you are done.

Common Errors Memory Leaks – Serious Business

I know, I know, you think that a few `doubles`
and a couple of `strings` left on the heap
now and then doesn't really hurt anyone.

But son, what if everyone did this?
Think of a loop – 10,000 times you grab just a few bytes
from the heap and don't give them back!

What happens when there's no more heap
for the OS to give you?

Just give it up, son – give back what you no longer need.

Common Errors Memory Leaks – Serious Business

Remember Rule #1.

1. Every call to `new` *must* be matched by exactly one call to `delete`.

And after deleting, set it to NULL so that it can be tested for danger later.

Common Errors Dangling Pointers – Serious Business

```
int* values = new int[n];
// Process values

delete[] values;
values = NULL;

later...
if values = NULL ...
```

Very good, son.
Being very
responsible!

much better!

Great!

Arrays and Vectors of Pointers (7.5)

When you have a sequence of pointers,
you can place them into an array or vector.

An array and a vector of ten `int*` pointers are defined as

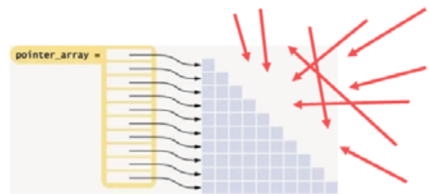
```
int* pointer_array[10];

vector<int*> pointer_vector(10);
```


Arrays and Vectors of Pointers – A Triangular Array

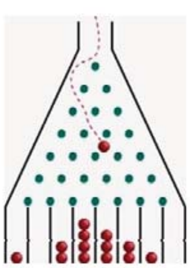
In this array, each row is a different length.

Arrays and Vectors of Pointers – A Triangular Array

In this situation, it would not be very efficient
to use a two-dimensional array,
because almost half of the elements would be wasted.

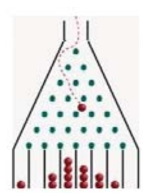


A Galton Board (7.6)



A Galton Board Simulation

We will develop a program that
uses a triangular array to simulate
a Galton board.



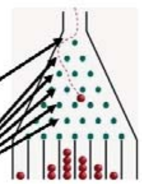
A Galton Board Simulation

A Galton board consists of a pyramidal arrangement of pegs and a row of bins at the bottom.

Balls are dropped onto the top peg and travel toward the bins.

At each peg, there is a 50 percent chance of moving left or right.

The balls in the bins approximate a bell-curve distribution.

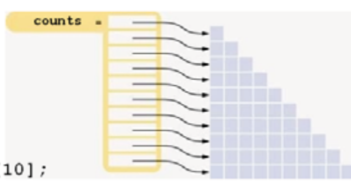


A Galton Board Simulation

The Galton board can only show the balls in the bins, but we can do better by keeping a counter for *each* peg, incrementing it as a ball travels past it.

A Galton Board Simulation

We will simulate a board with ten rows of pegs.
Each row requires an array of counters.
The following statements initialize the triangular array:




```
int* counts[10];
for (int i = 0; i < 10; i++)
{
   counts[i] = new int[i + 1];
}
```

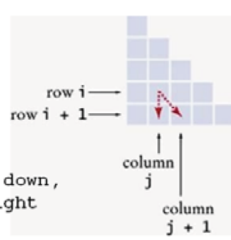

A Galton Board Simulation

We will need to print each row:


```
// print all elements in the ith row
for (int j = 0; j <= i; j++)
{
   cout << setw(4) << counts[i][j];
}
cout << endl;
```


A Galton Board Simulation

We will simulate a ball bouncing through the pegs:




```
int r = rand() % 2;
// If r is even, move down,
// otherwise to the right
if (r == 1)
{
   j++;
}
counts[i][j]++;
```


A Galton Board Simulation

ch07/galton.cpp

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
   srand(time(0));
   int* counts[10];

   // Allocate the rows
   for (int i = 0; i < 10; i++)
   {
      counts[i] = new int[i + 1];
      for (int j = 0; j <= i; j++)
      {
         counts[i][j] = 0;
      }
   }
```

A Galton Board Simulation

ch07/galton.cpp

```
const int RUNS = 1000;
// Simulate 1,000 balls
for (int run = 0; run < RUNS; run++)
{
   // Add a ball to the top
   counts[0][0]++;
   // Have the ball run to the bottom
   int j = 0;
   for (int i = 1; i < 10; i++)
   {
      int r = rand() % 2;
      // If r is even, move down,
      // otherwise to the right
      if (r == 1)
      {
         j++;
      }
      counts[i][j]++;
   }
}
```


A Galton Board Simulation

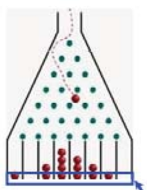
ch07/galton.cpp

```
   // Print all counts
   for (int i = 0; i < 10; i++)
   {
      for (int j = 0; j <= i; j++)
      {
         cout << setw(4) << counts[i][j];
      }
      cout << endl;
   }

   // Deallocate the rows
   for (int i = 0; i < 10; i++)
   {
      delete[] counts[i];
   }

   return 0;
}
```


A Galton Board Simulation

This is the output
from a run of the program:

```
1000
 480 520
 241 500 259
 124 345 411 120
  68 232 365 271  64
  32 164 283 329 161  31
  16  88 229 303 254  88  22
   9  47 147 277 273 190  44  13
   5  24 103 203 288 228 113  33   3
   1  18  64 149 239 265 186  61  15   2
```



Chapter Summary

Define and use pointer variables.

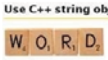
- A pointer denotes the location of a variable in memory.
- The type `T*` denotes a pointer to a variable of type `T`.
- The `&` operator yields the location of a variable.
- The `*` operator accesses the variable to which a pointer points.
- It is an error to use an uninitialized pointer.
- The `NULL` pointer does not point to any object.

Understand the relationship between arrays and pointers in C++.

- The name of an array variable is a pointer to the starting element of the array.
- Pointer arithmetic means adding an integer offset to an array pointer, yielding a pointer that skips past the given number of elements.
- The array/pointer duality law states that `a[n]` is identical to `*(a + n)`, where `a` is a pointer into an array and `n` is an integer offset.
- When passing an array to a function, only the starting address is passed.

Chapter Summary

Use C++ string objects with functions that process character arrays.



- A value of type `char` denotes an individual character. Character literals are enclosed in single quotes.
- A literal string (enclosed in double quotes) is an array of `char` values with a zero terminator.
- Many library functions use pointers of type `char*`.

- The `c_str` member function yields a `char*` pointer from a `string` object.
- You can initialize C++ `string` variables with C strings.
- You can access characters in a C++ `string` object with the `[]` operator.

Chapter Summary

Allocate and deallocate memory in programs whose memory requirements aren't known until run time.

- Use dynamic memory allocation if you do not know in advance how many values you need.
- The `new` operator allocates memory from the heap.
- You must reclaim dynamically allocated objects with the `delete` or `delete[]` operator.
- Using a dangling pointer (a pointer that points to memory that has been deleted) is a serious programming error.
- Every call to `new` should have a matching call to `delete`.

Chapter 7 Homework

Review Exercises

R7.2, R7.3, R7.7, R7.13,
R7.16, R7.19

Programming Exercises

P7.2, P7.8, P7.10

due Fri.
Oct. 26th