Chapter Seven:
Pointers (Part II) -
Dynamic Memory Allocation

C++ for Everyone by Cay Horstmann

Slides by Evan Gallagher

Dynamic Memory Allocation

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

You would normally use an array for this situation.

Dynamic Memory Allocation

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

So now can you use an array?

Dynamic Memory Allocation

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

Dynamic Memory Allocation

To solve this problem, you can use

dynamic allocation.

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!

Dynamic Memory Allocation

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

The OS *keeps*
a *heap*

Dynamic Memory Allocation

To ask for more memory,
say a **`double`**, you use the **`malloc()`** function
along with **`sizeof`** operator to get the memory
size in bytes to keep the given type.

```
(double*) malloc(sizeof(double))
```

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

This **`double`** location does not have a name.

(this is run-time)

Dynamic Memory Allocation

To request a dynamic array you use the same **`malloc`** function with some looks-like-an-array things added:

```
(double*) malloc(n * sizeof(double))
```

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

Dynamic Memory Allocation

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

```
double* account_pointer =
      (double*) malloc(sizeof(double));
double* account_array =
      (double*) malloc(n * sizeof(double));
```

Now you can use **`account_array`** as an array.

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 **`malloc`** function, you must return it to the heap using the **`free`** function.

```
free(account_pointer);
free(account_array);
```

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.

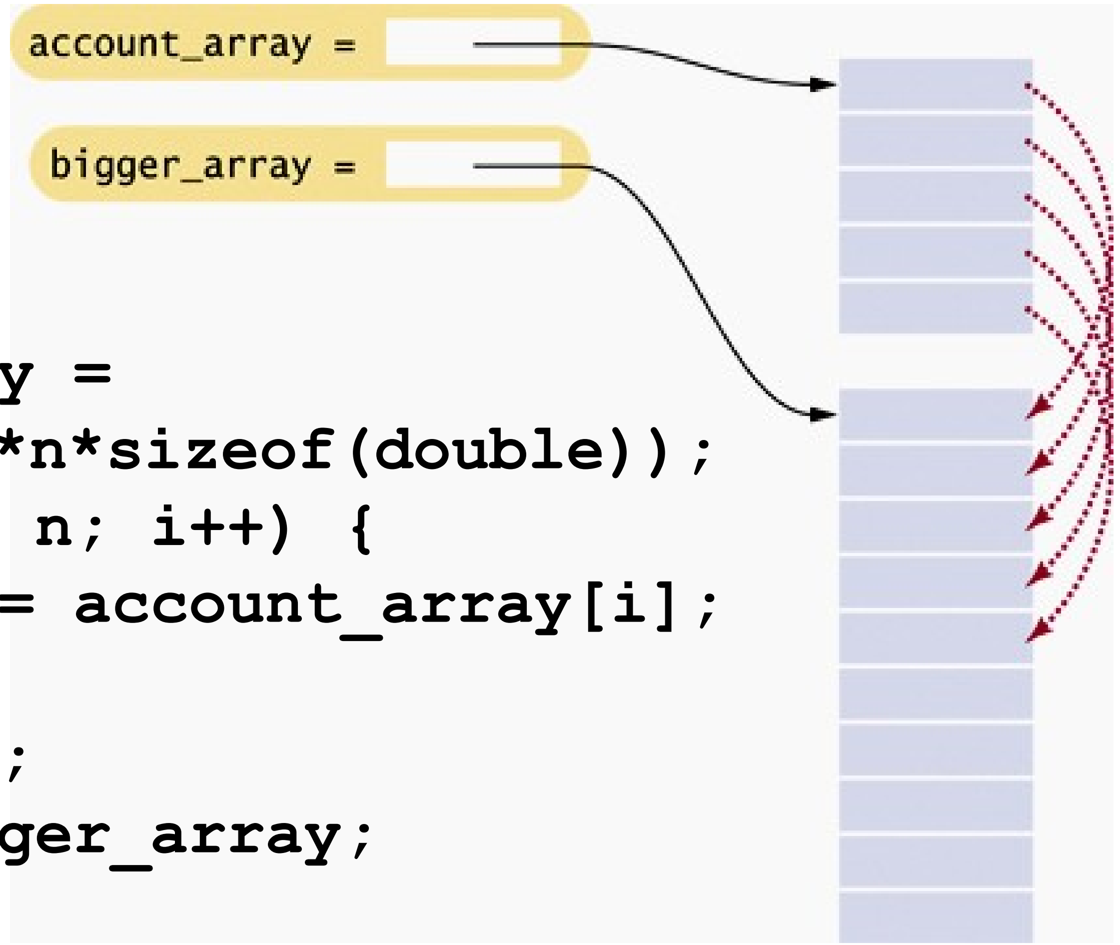
```
free(account_array);
account_array[0] = 1000;
      // NO! You no longer own the
      // memory of account_array
```

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

(**n** is the variable used with the array)

Dynamic Memory Allocation – THE RULES

1. **Every call to `malloc` *<u>must</u>* be matched by exactly one call to `free`.**

2. **Use `free` 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

Capture the pointer in a variable.

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

```
int* var_ptr = (int*) malloc(sizeof(int));
...
*var_ptr = 1000;
...
free (var_ptr);
```

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).

```
int* array_ptr = (int*) malloc(size*sizeof(int));
...
array_ptr[i] = 1000;
...
free (array_ptr);
```

Use the pointer as if it were an array.

Remember to use `free` when deallocating the array.

Dynamic Memory Allocation (C++ SYNTAX)

SYNTAX 7.2 Dynamic Memory Allocation

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

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

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.

Common Errors Dangling Pointers

DANGLING

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

Common Errors Dangling Pointers

```
int* values = (int*)malloc(n*sizeof(int));

// Process values

free(values);

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

Common Errors Dangling Pointers

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.

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

Common Errors Dangling Pointers

- **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

LEAKS

A *memory leak* is when use alolocate dynamic memory but you fail to free it when you are done.

Common Errors Memory Leaks

Remember Rule #1.

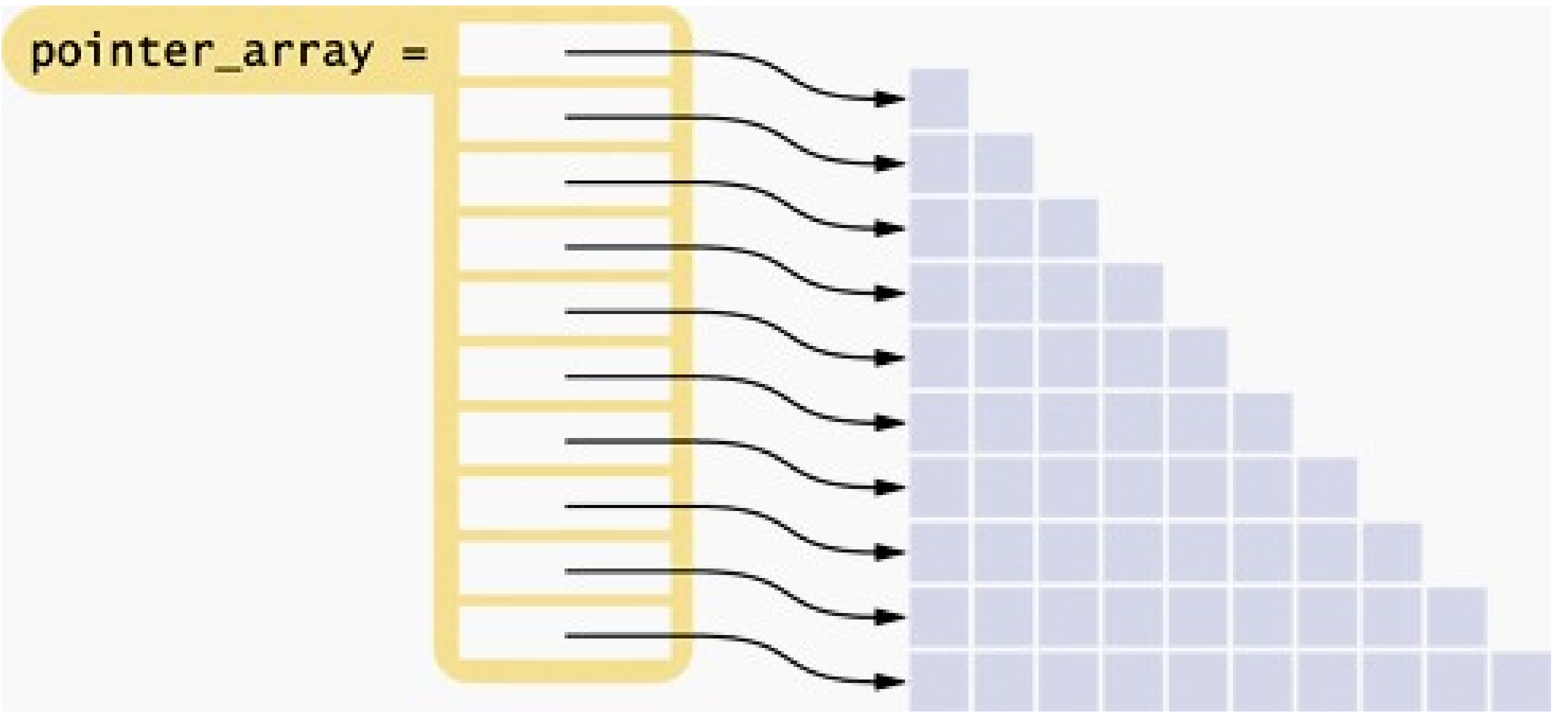
1. **Every call to `malloc` *must* be matched by exactly one call to `free`.**

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

Common Errors Dangling Pointers – Serious Business

```
int* values = malloc(n * sizeof(int));

// Process values

free(values);
values = NULL;

...

if (values == NULL) ...
```

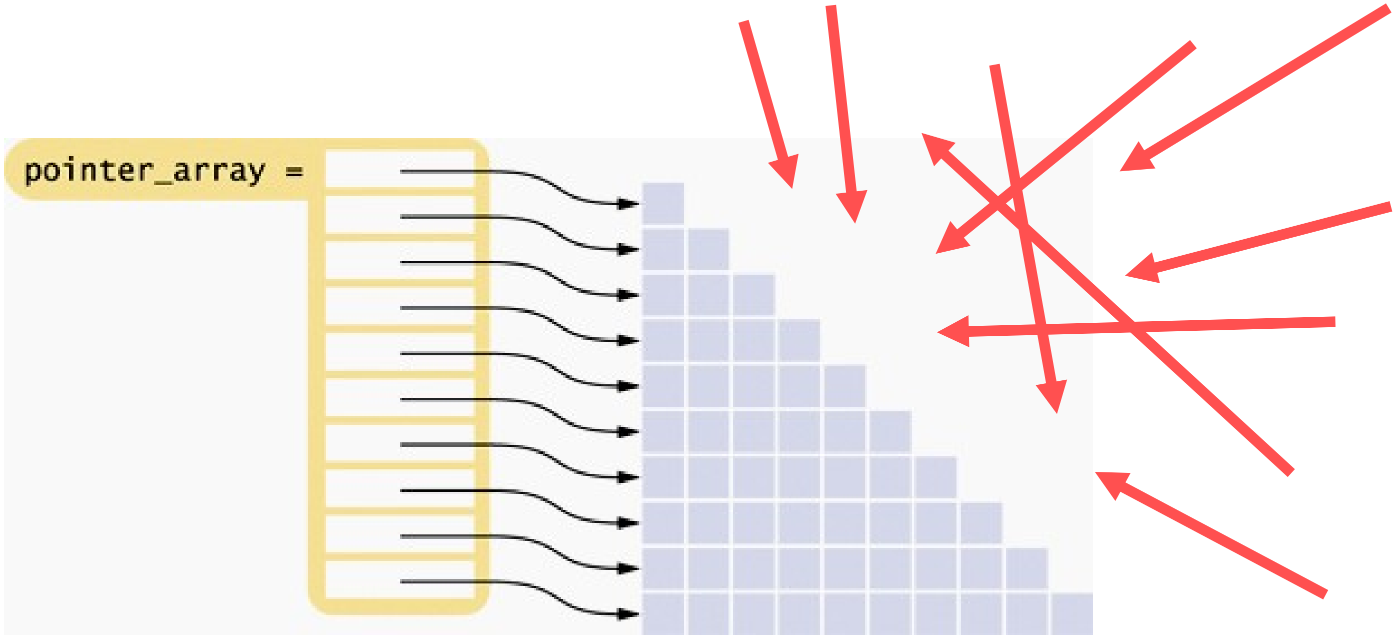
Arrays of Pointers – A Triangular Array



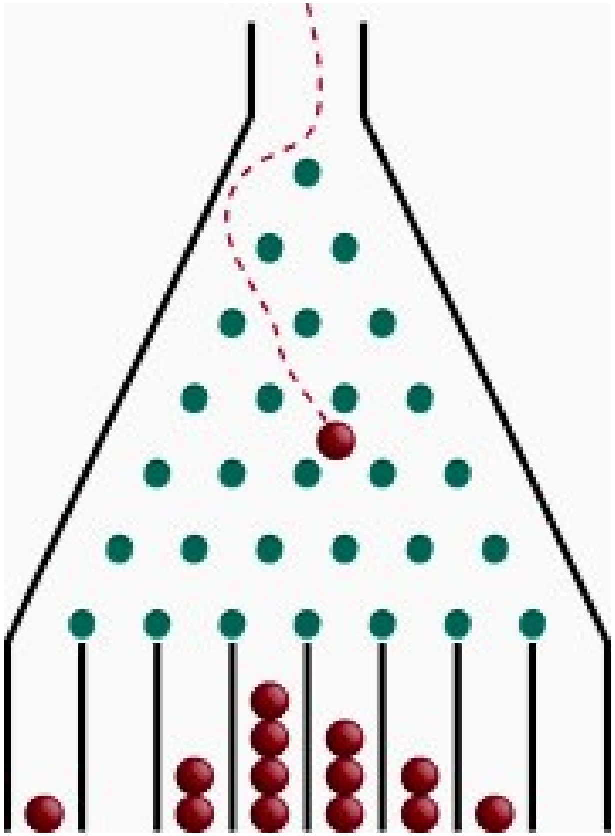
In this array, each row is a different length.

Arrays 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.

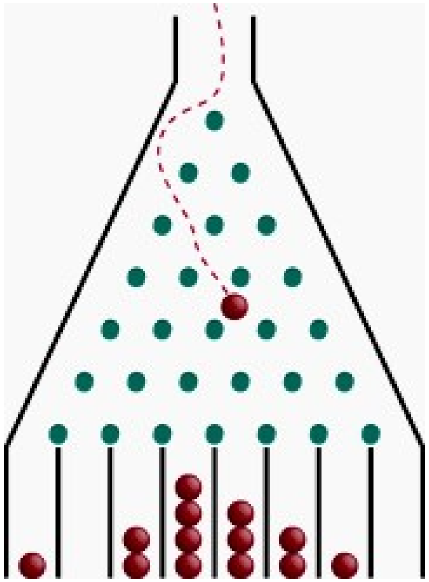


A Galton Board



A Galton Board Simulation

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

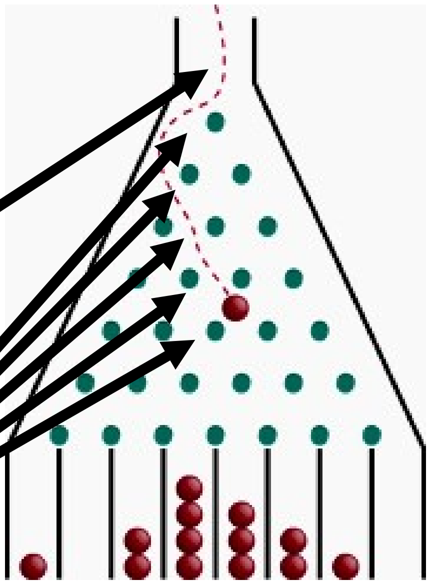


A Galton Board Simulation

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.



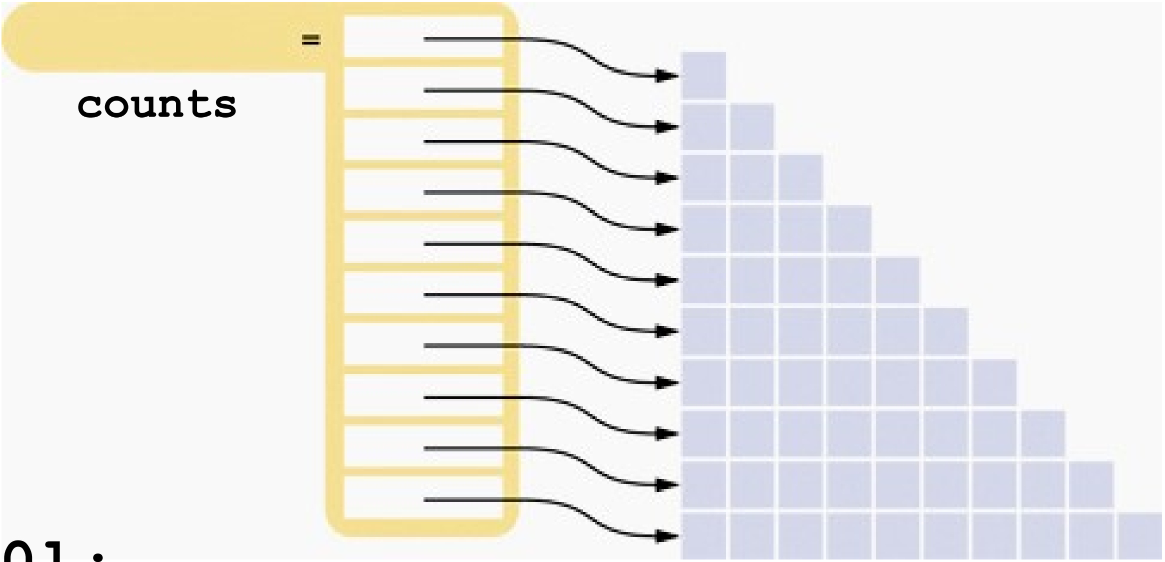
The balls in the bins approximate a bell-curve distribution.

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

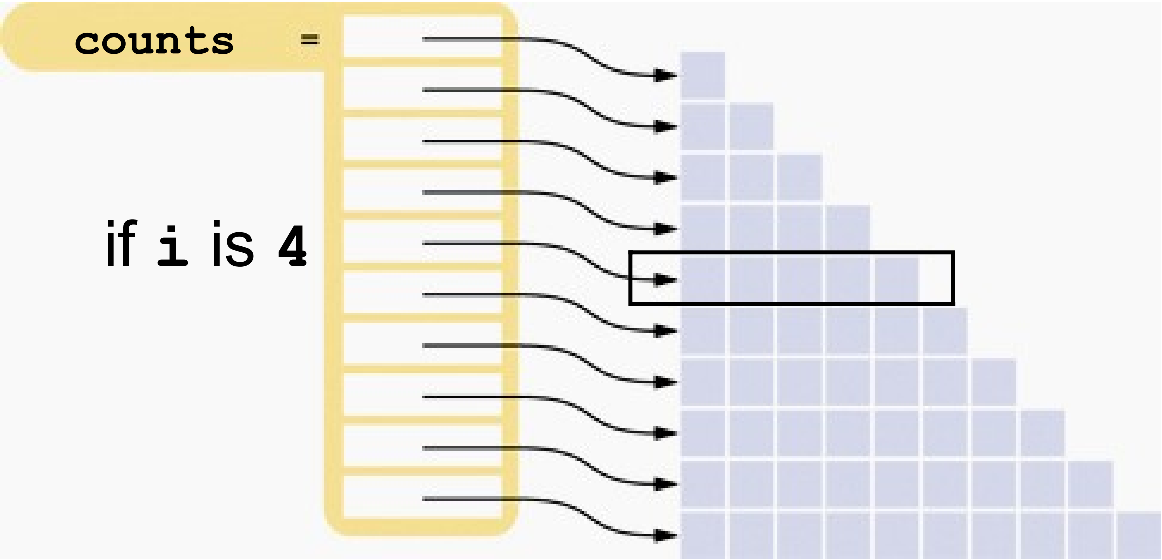
We will simulate a board with ten rows of pegs.
Each row requires an array of counters.
The following statements initialize the triangular array:




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

A Galton Board Simulation

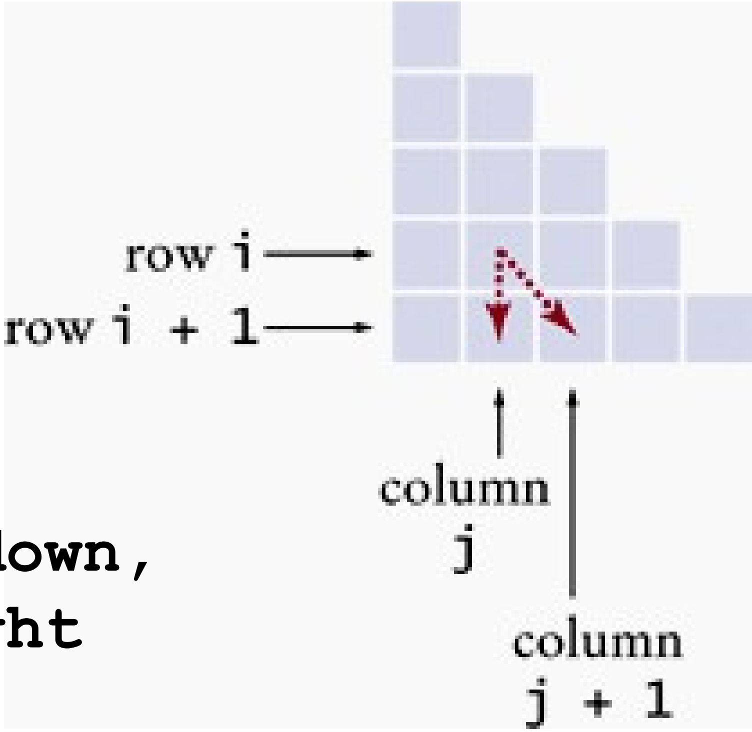
We will need to print each row:




```
// print all elements in the ith row
for (int j = 0; j <= i; j++) {
   printf("%5d", counts[i][j]);
}
printf("\n");
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   const int RUNS = 1000;   // Simulate 1,000 ball
   int* counts[10];
   srand(time(0));
   // allocate rows and init first two with zero
   for (int i = 0; i < 10; i++) {
      counts[i] = (int*) malloc(sizeof(int)*(i + 1));
      for (int j = 0; j <= 1; j++) {
         counts[i][j] = 0;
      }
   }
```

A Galton Board Simulation

```
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]++;
    }
}
```

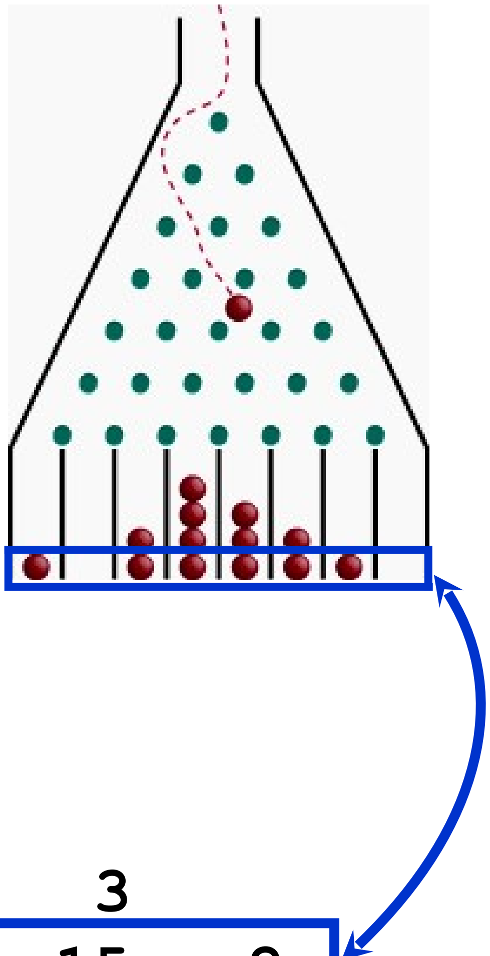
A Galton Board Simulation

```
    // Print all counts
    for (int i = 0; i < 10; i++) {
       for (int j = 0; j <= i; j++) {
          printf("%5d",counts[i][j]);
       }
       printf("\n");
    }

    // Deallocate the rows
    for (int i = 0; i < 10; i++) {
       free(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
```



Memory Reallocation

To change/extend the size of the memory previously allocated

```
int size = 10 * sizeof(int);
int* ptr = (int*) malloc(size);
size = size * 2;
int* ptr_new = (int*) realloc(ptr, size);
```

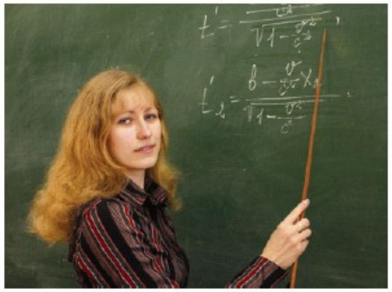
`realloc` changes the size of the object pointed to by **`ptr`** to **`size`**. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. **`realloc`** returns a pointer to the new space, or **`NULL`** if the request cannot be satisfied, in which case **`ptr`** is unchanged.

Memory Reallocation

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int* ptr = (int*) malloc(sizeof(int) * 2);
   int* ptr_new;

   *ptr = 10;
   *(ptr + 1) = 20;

   ptr_new = (int*) realloc(ptr, sizeof(int) * 3);
   *(ptr_new + 2) = 30;
   for(int i = 0; i < 3; i++) {
       printf("%d ", *(ptr_new + i));
   }
   free(ptr_new);
   return 0;
}
```

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

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`.

End Chapter Seven, Part II
Memory Allocation

Slides by Evan Gallagher

C++ for Everyone by Cay Horstmann