Topic 4

1. Defining and using pointers
2. Arrays and pointers
3. C and C++ strings
4. <u>Dynamic memory allocation</u>
5. Arrays and vectors of pointers
6. Problem solving: draw a picture
7. Structures
8. Pointers and structures

Dynamic Memory Allocation

You may not know beforehand how many values you need in an array.

To solve this problem, use *dynamic memory allocation* and ask the C++ run-time system to create new values whenever you need them.

The run-time system keeps a large storage area, called the **free store** or **heap**, that can allocate values and arrays of any type:

```
double *p = new double[n];
```

allocates an array of size n, and yields a pointer to the starting element. (Here n need not be a constant.)

Dynamic Memory Allocation Examples

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

```
 //get a single variable
 double* account_pointer = new double;

//get an array variable
 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: `delete`

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).

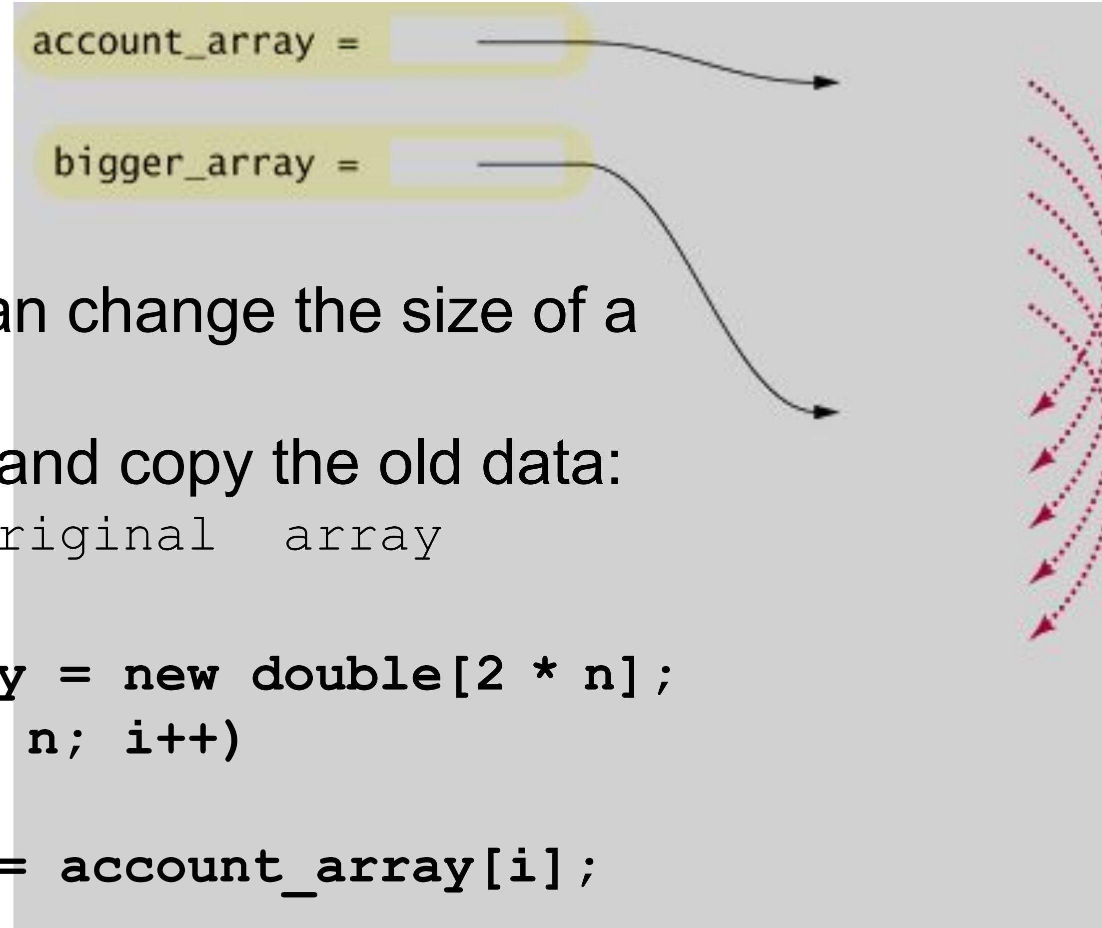
```
delete account_pointer;
delete[] account_array;
```

Don't Use a Pointer after `delete`

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 – Resizing an Array



Unlike static arrays, you can change the size of a dynamic array.

Make a new, bigger array and copy the old data:

```
//n = size of the original  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;
```

Dynamic Memory Allocation – THE RULES

1. **Every call to `new` *<u>must</u>* 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 (don't use the pointer) after it has been deleted.**

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

or worse…

Dynamic Memory Allocation – Common Errors: Table 5

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 Error: Dangling Pointers

It is a run-time error to use a pointer that points to memory that has already been deleted.

Such a pointer is called a **dangling pointer**.

Because the freed memory will be reused for other purposes, you can do real damage with a dangling pointer. For example:

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

delete[] values; //values now dangling

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

Avoiding Dangling Pointers

To prevent a **dangling pointer, assign the special value nullptr**
To any pointer that you delete:

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

delete[] values; //values now dangling

values = nullptr; //makes pointer safe
```

Common Error: Memory Leaks

A memory block that is never deallocated is called a memory leak.

If you allocate a few small blocks of memory and forget to deallocate them, this is not a huge problem.

When the program exits, all allocated memory is returned to the operating system.

Every call to `new` should have a matching call to **`delete`**.

But if your program runs for a long time, or if it allocates lots of memory (perhaps in a loop) without the `delete`s, then it can run out of memory and crash.