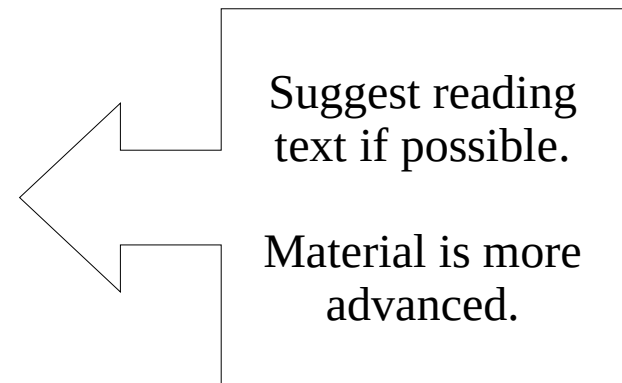


Arrays and Dynamic Memory

Readings Topics:
Pointers
Arrays (some)
Dynamic Memory



Dr. Donaldson's notes: <http://www.cs.sfu.ca/CourseCentral/130/tjd/chp9notes.html>

Topics

- 1) How can we store many elements (without a vector)?
- 2) How can we get and manage extra memory?

Arrays

Array
daysPerMonth

- Array Declaration:
 - Specify type of elements, and # elements.
`int daysPerMonth[12];`
 - Arrays are quite similar to vectors, but can be faster, and once created..
- Directly access an element:
 - For N elements use indices 0 to N-1
 - `daysPerMonth[0] = 31; // January`
- Ex:
 - `daysPerMonth[11] = 31; // December`
 - `int a = daysPerMonth[1]; // February`
 - `cout << daysPerMonth[1]; // Outputs 28`
 - `cin >> daysPerMonth[9]; // Read in oct.`

	Idx	Val
Jan	0	31
Feb	1	28
Mar	2	31
Apr	3	30
May	4	31
Jun	5	30
Jul	6	31
Aug	7	31
Sep	8	30
Oct	9	31
Nov	10	30
Dec	11	31

Array example

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    // Create the arrays for day names and hours per day.
    const int DAYS_PER_WEEK = 7;
    float hoursWorked[DAYS_PER_WEEK];

    // Ask user for time worked.
    for (int i = 0; i < DAYS_PER_WEEK; i++) {
        cout << "Hours worked on day " << i << ": ";
        cin >> hoursWorked[i];
    }

    // Calculate total hours
    cout << "Week summary:\n";
    cout << fixed << setprecision(1);
    float totalHours = 0;
    for (int i = 0; i < DAYS_PER_WEEK; i++) {
        cout << i << " = " << hoursWorked[i] << " hours.\n";
        totalHours += hoursWorked[i];
    }
    cout << "Total hours: " << totalHours << endl;
}
```

```
Hours worked on day 0: 0
Hours worked on day 1: 1.5
Hours worked on day 2: 26.9
Hours worked on day 3: 8.2
Hours worked on day 4: 1.6
Hours worked on day 5: 0
Hours worked on day 6: 1
Week summary:
0 = 0.0 hours.
1 = 1.5 hours.
2 = 26.9 hours.
3 = 8.2 hours.
4 = 1.6 hours.
5 = 0.0 hours.
6 = 1.0 hours.
Total hours: 39.2
```

In-Class Example

- Write a program which:
 - reads up to 10 floats from the keyboard
 - but which stops when the user enters a 0.
(Called a sentinel: a value which marks the end)
- It must then:
 - display the values to the screen

Possible solution

```
#include <iostream>
using namespace std;
int main()
{
    // Create the array
    const int MAX_SIZE = 10;
    float data[MAX_SIZE];

    // Populate the array
    cout << "Enter up to " << MAX_SIZE
         << " values (0 to exit):\n";
    int count = 0;
    for(count = 0; count < MAX_SIZE; count++) {
        // Get the next value
        float newValue = 0;
        cin >> newValue;

        // Are we done?
        if (newValue == 0) {
            break;
        }

        // Store in array:
        data[count] = newValue;
    }
```

```
    // Print out all the values:
    cout << "\nData:\n";
    for (int i = 0; i < count; i++) {
        cout << i << ": "
             << data[i] << endl;
    }
```

Enter up to 10 values (0 to exit):

10
15.112
20.222
0

Data:

0: 10
1: 15.112
2: 20.222

Passing a full array

- Need two things to pass an array to a function:

Function can handle any size of array.

Must tell it the size of the array separately.

When calling, pass in the array (no []!), and size.

```
void showAllElements(char arr[], int size) {  
    cout << "Displaying all elements:\n";  
    for (int i = 0; i < size; i++) {  
        cout << arr[i] << " ";  
    }  
    cout << endl;  
}
```

Display all elements:
H e l l o

```
int main () {  
    const int N = 5;  
    char myArray[] =  
        {'H', 'e', 'l', 'l', 'o'};  
  
    // Pass the whole array.  
    showAllElements(myArray, N);  
    ...  
}
```

Pass array by Pointer

- Passing an array to a function passes..
 - It is not a copy of the array:
it is the address of the real thing.
 -

```
void zAllElements(char arr[], int size) {  
    for (int i = 0; i < size; i++) {  
        arr[i] = 'z';  
    }  
}
```

```
int main () {  
    const int N = 5;  
    char myArray[] =  
        {'H', 'e', 'l', 'l', 'o'};  
  
    // Pass the whole array.  
    zAllElements(myArray, N);  
    showAllElements(myArray, N);  
    ...  
}
```


Arrays and Pointers

- Arrays & pointers are similar:
 - Array names can be..
 - Pointers can be..

```
int costs[] = {0, 10, 20, 30, 40};
int *pValue = costs;    //..

cout << "Array:  " << costs << endl;
cout << "Pointer: " << pValue << endl;

cout << "costs[0]: " << costs[0] << endl;
cout << "*costs:  " << *costs << endl;

cout << "pValue[0]:" << pValue[0] << endl;
cout << "*pValue:  " << *pValue << endl;

for (int i = 0; i < 5; i++) {
    cout << pValue[i] << ", ";
}
```

```
Array:      0x7fff87968010
Pointer:    0x7fff87968010
```

```
costs[0]:   0
*costs:     0
```

```
PValue[0]:  0
*pValue:    0
```

```
0, 10, 20, 30, 40,
```

Arrays vs Vectors

- Arrays and Vectors have a similar purpose...
 - Many problems which could be solved with one can also be solved with the other.

	Array	Vector
Data Type	Fundamental type to C++ (and C) language	A class in the C++ standard library (“using namespace std;”)
Change Size?	Fixed size	Dynamically resizes
Code to create	<code>int myArray[10];</code>	<code>vector<int> myVect;</code>
Set element	<code>myArray[0] = 42;</code>	<code>myVect.at(0) = 42</code> or <code>myVect[0] = 42</code>
Add extra element	Impossible	<code>myVect.push_back(101);</code>
Access element	<code>cout << myArray[0];</code>	<code>cout << myVect.at(0);</code> <code>cout << myVect[0];</code>
Ask it for its size	Impossible	<code>cout << myVect.size();</code>
Pass to function	Pass as a pointer (array)	By value, by ref, or by pointer

Dynamic Memory

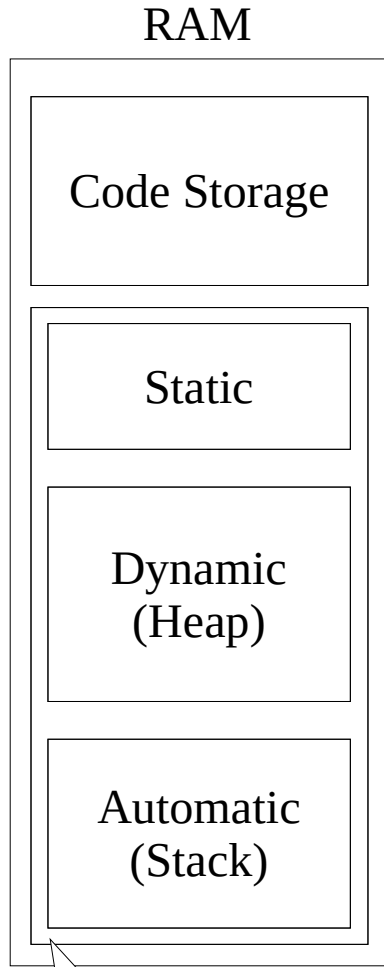
Why do we need this?

Doesn't vector do everything we need?

Vector's great! However...

- There's more to software development than vector
- Vector had to be implemented using something!

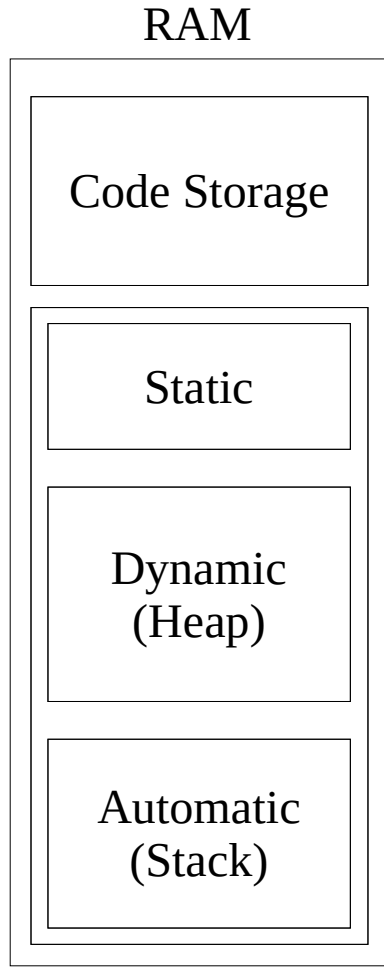
Memory



- Code Storage
 - Also called "text"
 - Stores the..
- Data Storage
 - Stores the..
 - Types: Static, Dynamic, Automatic.
- Static Memory
 - Holds..
 - Values initialized when program starts.

Data
Storage

Memory: Automatic



- Automatic Storage
 - Local variables allocated..
 - When function exits, it pops its local variables off the stack..
 - Space reused for next function call.
 - Calling a huge number of functions will overflow the stack (crash the program).

Example: Bad Recursion

```
void crashProgram() {  
    crashProgram();  
}
```

A function that ..

Returning a new array

- How can a function return a new array?
 - You can't return an array, but you can return a pointer
 - Here's the first (bad) try!

```
float* makeArrayOfNumbers(int size) {  
    float arr[size];  
  
    for (int i = 0; i < size; i++) {  
        arr[i] = i;  
    }  
    displayArray(arr, size);  
    return arr;  
}
```

```
0 1 2 3 4  
0 4.59163e-41 0 0 0
```

```
void displayArray(float arr[], int size) {  
    for (int i = 0; i < size; i++) {  
        cout << arr[i] << " ";  
    }  
    cout << endl;  
}  
  
int main() {  
    const int SIZE = 5;  
    float *myArr =  
        makeArrayOfNumbers(SIZE);  
    displayArray(myArr, SIZE);  
}
```

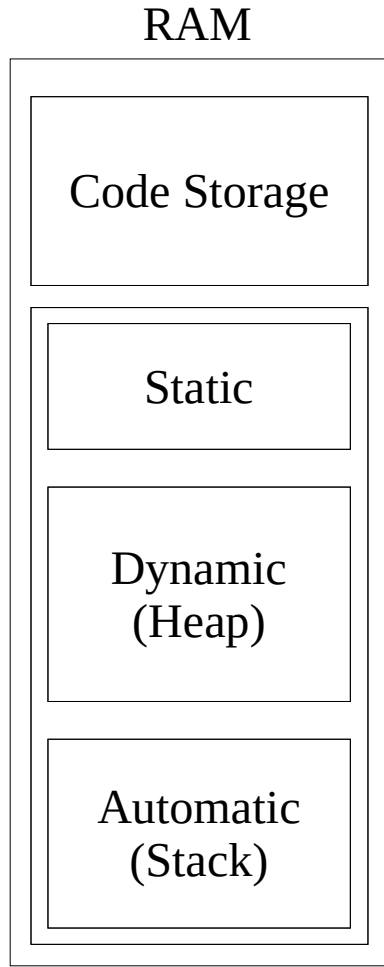
What went wrong?

- Never..
 - Local variables are popped off the stack when the function finishes.
 - All pointers to popped-locals become..

```
float* badIdea(int size) {  
    float arr[size];  
    // ...  
    return arr;  
}
```

- How can we get some memory which is not on the stack?
 - So it will not be popped when the function exits?

Memory: Dynamic



- Dynamic Storage
 - Allows for:
 -
 - gives program control over..
 - Allocate using..
Deallocate using..
 - In separate memory region..

What would happen if we "dynamically" allocated on the stack instead?

Dynamic Arrays

```
void displayArray(float arr[], int size);
```

```
float* makeDynamicArray(int size)
{
    float *arr = new float[size];    ..
    for (int i = 0; i < size; i++) {
        arr[i] = i;
    }
    displayArray(arr, size);
    return arr;                      ..
}
```

```
int main()
{
    float *myArr = makeDynamicArray(SIZE);    ..
    displayArray(myArr, SIZE);
    delete[] myArr;                          ..
}
```

- Use dynamic allocation to create an array in the heap.
- Return a pointer to this array.
- Use the array like a normal array.
- Later, we must free the memory using delete.

Dynamic Allocation

- new

```
double *heightArr = new double[100];
```

- new allocates space from heap.

-

- delete

```
delete[] heightArr;
```

- delete releases (frees) memory.

- Must free memory..

- Can only free it once!

Returning allocated space

```
int* getRandArray(int n)
{
    // Allocate space
    int* pArr = new int[n];

    // Initialize data
    for (int i = 0; i < n; i++) {
        pArr[i] = rand() % 100 + 1;
    }

    return pArr;
}
```

```
int main()
{
    const int SIZE = 10;

    // Get the array of data
    int* pData = getRandArray(SIZE);

    // Use the allocated memory
    cout << "Data: ";
    for (int i = 0; i < SIZE; i++) {
        cout << pData[i] << " ";
    }
    cout << endl;

    // Free the memory to
    // avoid memory leaks.
    delete[] pData;
    pData = nullptr;
}
```

Pointers

- Pointers:
 - Pointers are often allocated..
 - Pointer destroyed when it goes out of scope.
 - When pointer destroyed, data it points to..
- Dynamic Array
 - Allocated on the heap, pointed to by a pointer.
 - Must call `delete[]` on the dynamic array regardless of when pointers are destroyed.

Summary

- Arrays are like vectors, but you manage the memory.
 - Arrays are pointers; pointers are arrays.
- Dynamic memory
 - Use `new` to allocate array on heap;
 - Use `delete[]` to free the memory.