Pointers and Memory

Ch 9: Pointers
Ch 7: Arrays (some)

Suggest reading text if possible.
Material is more advanced.
Topics

1) How can a program work with addresses?
2) How can we use pointers with functions?
3) How can we store many elements without a vector?
4) How can we get and manage extra memory?
Pointers
Address Of

• Variables in Memory
  – Each variable is stored in..
  – Get the address of a variable with..
    int answer = 42;
    cout << "Value: " << answer << endl;
    cout << "Address: " << &answer << endl;

<table>
<thead>
<tr>
<th>Value:</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address:</td>
<td>0x7fffe3bd9fac</td>
</tr>
</tbody>
</table>

• Can a program work with addresses?
  ..
Pointers

- Declare a pointer by adding a star after the type:
  \[
  \text{int}\ast \ p\text{StudentNum};
  \]

- You should always initialize the pointer:
  \[
  \text{float}\ast \ p\text{Height} = \text{nullptr}; \quad \text{// Point to nothing.}
  \]
  \[
  \text{int} \ \text{answer} = 42;
  \]
  \[
  \text{int}\ast \ p\text{Answer} = \&\text{answer};
  \]

- All pointers are the same size.
Using a Pointer

• Dereference
  – Add a * before the variable name.

• Example
  int answer = 1;
  int* pAns = nullptr;

  pAns = &answer;  
  *pAns = 23;

  cout << "Answer: " << answer << endl;
  cout << "*pAns: " << *pAns << endl;
Example

// Create a variable
float myPi = 3.14;
cout << "@1: myPi = " << myPi << endl;

// Create pointer; point to the variable.
float* pPi = &myPi;
cout << "@2: pPi = " << pPi << endl;
cout << "@2: &myPi= " << &myPi << endl;
cout << "@2: *pPi = " << *pPi << endl;

// Change via the variable
myPi = 13.9;
cout << "@3: myPi = " << myPi << endl;
cout << "@3: *pPi = " << *pPi << endl;

// Change via the pointer
*pPi = 999.2;
cout << "@4: myPi = " << myPi << endl;
cout << "@4: *pPi = " << *pPi << endl;
Pointer Operator Recap

- **Uses of ***
  - Multiplication: `int x = 10 * 2;`
  - Pointer declaration: `int *ptr = nullptr;`
  - Pointer dereferencing: `*ptr = (*ptr) / 2 + 1;`

- **Uses of &**
  - AND: `if (x > 1 && x < 10) {...}
  - Memory Address: `ptr = &x;`
  - Pass by reference: `int foo(string &str);`

---

code

```
// Working with Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>2047</th>
<th>2048</th>
<th>2049</th>
<th>2050</th>
<th>2051</th>
<th>2052</th>
<th>2053</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

---

code

```
// Working with Variables
```
// Complete questions a) through d)
void bustedCode()
{
    int age = 75;
    int* pPointer = nullptr;

    // a) What's wrong with this?
    *pPointer = &age;

    // b) What's wrong with this?
    pPointer = 42;

    // c) Change the value to which pPointer points to zero:

    // d) Change pPointer to point to zero:

}
Pass by Pointer
Pass by...

- **Pass-by-value**
  - Copies of the arguments are passed to the function.

- **Pass-by-reference**
  - Function works on actual argument (reference).

- **Pass-by-pointer**
  - ..
    - is passed to function.
  - Changes in the function to that location in memory..

- Pass-by-reference is similar to pass-by-pointer, but it handles the pointer access for you.
void swapByVal(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

void swapByRef(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

void swapByPtr(int* pA, int* pB) {
    int temp = *pA;
    *pA = *pB;
    *pB = temp;
}

int main() {
    int mine = 1;
    int yours = 99;
    swapByVal(mine, yours);
    cout << mine << " " << yours << endl;

    int mine = 1;
    int yours = 99;
    swapByRef(mine, yours);
    cout << mine << " " << yours << endl;

    int mine = 1;
    int yours = 99;
    swapByPtr(&mine, &yours);
    cout << mine << " " << yours << endl;
}
Review

- Write a void function named sort() which:
  - takes two pointers to float's (pX and pY).
  - if *pX > *pY, then swap their values.

```cpp
// Sort calling example:
float one = 100;
float two = 12.5;
sort(&one, &two);
cout << "One = " << one << " , Two = " << two
    << endl;
```

One = 12.5
Two = 100
Arrays and Pointers
Arrays

- 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.


<table>
<thead>
<tr>
<th>Idx</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>0</td>
</tr>
<tr>
<td>Feb</td>
<td>1</td>
</tr>
<tr>
<td>Mar</td>
<td>2</td>
</tr>
<tr>
<td>Apr</td>
<td>3</td>
</tr>
<tr>
<td>May</td>
<td>4</td>
</tr>
<tr>
<td>Jun</td>
<td>5</td>
</tr>
<tr>
<td>Jul</td>
<td>6</td>
</tr>
<tr>
<td>Aug</td>
<td>7</td>
</tr>
<tr>
<td>Sep</td>
<td>8</td>
</tr>
<tr>
<td>Oct</td>
<td>9</td>
</tr>
<tr>
<td>Nov</td>
<td>10</td>
</tr>
<tr>
<td>Dec</td>
<td>11</td>
</tr>
</tbody>
</table>
#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

= hoursWorked.cpp
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
```cpp
#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 << "Data:\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:

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

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.

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

```c
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..

```cpp
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] <<", ";
}
```

<table>
<thead>
<tr>
<th>Array:</th>
<th>0x7fff87968010</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer:</td>
<td>0x7fff87968010</td>
</tr>
<tr>
<td>costs[0]:</td>
<td>0</td>
</tr>
<tr>
<td>*costs:</td>
<td>0</td>
</tr>
<tr>
<td>pValue[0]:</td>
<td>0</td>
</tr>
<tr>
<td>*pValue:</td>
<td>0</td>
</tr>
<tr>
<td>0, 10, 20, 30, 40,</td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Memory
Memory

- **Code Storage**
  - Also called "text"
  - Stores the..

- **Data Storage**
  - Stores the..
  - Types: Static, Dynamic, Automatic.

- **Static Memory**
  - Holds..
  - Values initialized when program starts.
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

```cpp
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!

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

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);
}
```

0 1 2 3 4
0 4.59163e-41 0 0 0
What went wrong?

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

```c
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

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

float* makeDynamicArrayArrayOfNumbers(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 *myDynamicArr =
        makeDynamicArrayArrayOfNumbers(SIZE);
    displayArray(myDynamicArr, SIZE);
    delete[] myDynamicArr;
}
```

- 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!
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;
}
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

• Pointers point to memory locations.
  - Declare:    int* pHeight = 0;
  - Set:       pHeight = &someVariable;
  - Use:       cout << *pHeight;

• Pass-by-pointer allows changes to the arguments.

• 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.