ARRAYS AND POINTERS

Today

- Today we will look at:
  - Arrays
  - Pointers
  - Arrays of objects
- This material is kind of covered in Chapter 3 by Pohl.
- All the examples in these notes are on the class website.

Overview of arrays and pointers

- Arrays and pointers are strongly related

```
int A[10];   // declare an array of 10 ints
int *pA;     // declare a pointer to an int
pA = &A[0];  // pA points to the 0th element
             // of of A
pA = A;      // this has the same effect
```

- Pointer arithmetic is meaningful with arrays:

  If we do

  `pA = &A[0];`

  then `*(pA + 1)` points to `A[1]`

- Remember the difference between `(*pA) + 1` and `*(pA + 1)` (which == `*pA + 1`)
- Note that an array name is a pointer, so we can also do `*(A + 1)` and in general:
  - `*(A + i) == A[i]` and so are `A + i == &A[i]`
- The difference:
  - An array name is a constant, and a pointer is not.
  - So we can do: `pA = A` and `pA++` but we can NOT do: `A = pA` or `A++`
- When an array name is passed to a function, what is really passed is a pointer to the array.

Arrays review

- A string is an *array* of characters
- An array is a “regular grouping or ordering”
- A data structure consisting of related elements of the same data type
- Arrays need:
  - Data type
  - Name
  - Length

- Length can be determined:
  - *statically* — at compile time.

    ```
            char str1[10];
    ```

  - *dynamically* — at run time

    ```
    char *str2;
    ```

- We’ll talk about how to do dynamic declaration in the next lecture.

Arrays and memory

- Defining a variable is called “allocating memory” to store that variable
- Defining an array means allocating memory for a group of bytes, i.e., assigning a label to the first byte in the group
- Individual array elements are *indexed*
  - Starting with 0
  - Ending with *length* $- 1$
- Indices follow array name, enclosed in square brackets (`[ ]`) e.g., `arr[25]`

Character array example

```
// example: arrays0c.cpp

#include <iostream>
using namespace std;

const int MAX = 6;

int main( void ) {
  char str[MAX] = "ABCDE";
  int i;
  for ( i=0; i<MAX-1; i++ ) {
    cout << str[i] << " ";
  }
  cout << endl;
} /* end of main() */
```

Integer array example

```
// example: arrays0i.cpp

#include <iostream>
using namespace std;

const int MAX = 6;

int main() {
  int arr[MAX] = { -45, 6, 0, 72, 1543, 62 };
  int i;
  for ( i=0; i<MAX; i++ ) {
    cout << arr[i] << " ";
  }
  cout << endl;
} /* end of main() */
```

Pointers overview

- A pointer contains the address of an element
- Allows one to access the element “indirectly”
- & is a unary operator that gives address of its argument
- `*` is a unary operator that fetches contents of its argument (i.e., its argument is an address)
- Note that & and `*` bind more tightly than arithmetic operators
- You can print the value of a pointer using `cout` with the pointer or using C-style printing (e.g., `printf()`) and the formatting character `%p`

Pointers and memory

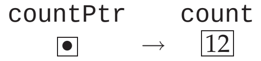
- Variables that contain memory addresses as their values
- Other data types we’ve learned about use *direct* addressing
- Pointers facilitate *indirect* addressing
- Declaring pointers:
  - Pointers indirectly address memory where data of the types we’ve already discussed is stored (e.g., `int`, `char`, `float`, etc.—even classes)
  - Declaration uses asterisks (*) to indicate a pointer to a memory location storing a particular data type
- Example:

```
int *count;
float *avg;
```

- Ampersand & is used to get the address of a variable
- Example:

```
int count = 12;
int *countPtr = &count;
```

- `&count` returns the *address* of `count` and stores it in the pointer variable `countPtr`
- A picture:



Here’s another example:

```
int i = 3, j = -99;
int count = 12;
int *countPtr = &count;
```

and here’s what the memory looks like:

variable name	memory location	value
`count`	0xbffff4f0	12
`i`	0xbffff4f4	3
`j`	0xbffff4f8	-99
...		
`countPtr`	0xbffff600	0xbffff4f0
...		

Address arithmetic

- An array is some number of contiguous memory locations
- An array definition is really a pointer to the starting memory location of the array
- And pointers are really (big) integers
- So you can perform integer arithmetic on them
- e.g., +1 increments a pointer, -1 decrements
- You can use this to move from one memory location to another
- Often this is used to access one array element after another

```
// pointers0.cpp

#include <iostream>
using namespace std;

int main() {

  int i, *j, arr[5];

  for ( i=0; i<5; i++ ) {
    arr[i] = i;
  }

  cout << "arr=" << arr << endl;
  cout << endl;
```

```
  for ( i=0; i<5; i++ ) {
    cout << "i=" << i  << " arr[i]=" << arr[i];
    cout << " &arr[i]=" << &arr[i] << endl;
  }

  cout << endl;

  j = &arr[0];
  cout << "j=" << j;
  cout << " *j=" << *j;
  cout << endl << endl;;

  j++;
  cout << "after adding 1 to j: j=" << j;
  cout << " *j=" << *j << endl;

}
```

The output is:

```
arr=0xbffff864

i=0 arr[i]=0 &arr[i]=0xbffff864
i=1 arr[i]=1 &arr[i]=0xbffff868
i=2 arr[i]=2 &arr[i]=0xbffff86c
i=3 arr[i]=3 &arr[i]=0xbffff870
i=4 arr[i]=4 &arr[i]=0xbffff874

j=0xbffff864 *j=0

after adding 1 to j: j=0xbffff868 *j=1
```

NOTE that the absolute pointer values can change each time you run the program! BUT the relative values will stay the same.

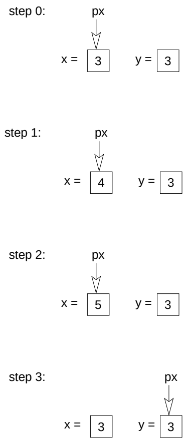
```
// pointers1.cpp

#include <iostream>
using namespace std;

int main() {

  int x, y;      // declare two ints
  int *px;       // declare a pointer to an int

  x = 3;         // initialize x

  px = &x;       // set px to the value of the address of x; i.e., to point

  y = *px;       // set y to the value stored at the address pointed
                 // to by px; in other words, the value of x

  printf( "x=%d px=%p y=%d\n",x,px,y );
```

```
  x++;           // increment x

  printf( "x=%d px=%p y=%d\n",x,px,y );

  (*px)++;       // increment the value stored at the address
                 // pointed to by px

  printf( "x=%d px=%p y=%d\n",x,px,y );

  *px++;         // take away the parens

  printf( "x=%d px=%p y=%d\n",x,px,y );

  // since px has changed, what does it point to now?

  printf( "*px=%d\n",*px );

}
```

and the output is...

```
step 0: here is what we start with: x=3 px=0xbffff874 y=3
step 1: after incrementing x:       x=4 px=0xbffff874 y=3
step 2: after incrementing (*px):   x=5 px=0xbffff874 y=3
step 3: after incrementing *px:     x=5 px=0xbffff878 y=3
       and *px=3
```

and here's a picture of what's going on:



this is the initial situation:
x is initialized to the value 3
px is initialized to point to x
y is initialized to the value pointed to by x

here is the situation after incrementing x

*here is the situation after incrementing (*px),*
i.e., the value that px points to, in other words, x

here is the situation after incrementing px
i.e., the POINTER increments, in other words,
it moves to point to the next contiguous item in
memory, in this case, y

Pointers and references

- *Pointers* (same as in C):
  - `int *p` means "pointer to int"
  - `p = &i` means p gets the address of object i
- *References* (not in C):
  - They are basically aliases — alternative names — for the values stored at the indicated memory locations,

```
int     n;
int     &nn = n;
double  arr[10];
double  &last = arr[9];
```

- The difference between them is shown by `refs.cpp` on the class website.

Arrays of objects

- You can create arrays of objects.

```
/* arrayso.cpp */

#include <iostream>
using namespace std;

class Point {
private:
  int x, y;
public:
  Point() { }
  Point( int x0, int y0 ) : x(x0), y(y0) { }
  void set( int x0, int y0 ) { x = x0; y = y0; }
  void print() const { cout << "(" << x << "," << y << ") "; }
};
```

- Each element of the array is an object, and is handled in the usual way.

```
int main() {
  Point triangle[3];
  triangle[0].set( 0,0 );
  triangle[1].set( 0,3 );
  triangle[2].set( 3,0 );
  cout << "here is the triangle: ";
  for ( int i=0; i<3; i++ ) {
    triangle[i].print();
  }
  cout << endl;
}
```

Pointers to objects

- You can also create pointers to objects just as you create pointers to primitive data types
- In the example below, we demonstrate *dynamic memory allocation* by declaring a pointer to an array and then LATER declaring the memory for the array using the `new` function.
- At the end of the program, we call the `delete` function to de-allocate the memory (it's not really necessary at the end of a program, but you might want to use it inside a program to keep your memory management clean).
- We'll talk more about dynamic memory allocation and memory management in the next lecture...

- Assuming the same definition of `point` as before.

```
int main() {
  Point *triagain = new Point[3];
  assert( triagain != 0 );
  triagain[0].set( 0,0 );
  triagain[1].set( 0,3 );
  triagain[2].set( 3,0 );
  cout << "tri-ing again: ";
  for ( int i=0; i<3; i++ ) {
    triagain[i].print();
  }
  cout << endl;
  delete[] triagain;
}
```

Summary

- This lecture has looked at
  - Pointers
  - Arrays
  - References

  and it began to explore the notion of dynamic memory allocation.
- The next lecture will look at dynamic memory allocation in more detail.