POINTERS AND ARRAYS

Today

- Today we continue with pointers.
- In particular we look at the relationship between pointers and arrays.
- Again this material is kind of covered in Chapter 3 by Pohl.
- Most of the examples in these notes are on the class website.

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;
str2 = new char [10];
```

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 $\mathit{length} - 1$
- 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() */
```

- Now we will go back and recall some things about pointers.
- Consider this:

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

- Here's what the memory looks like:

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

- The next slides give some more complex examples.

```
#include <iostream>   // pointers1.cpp
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 to x

  y = *px;      // set y to the value stored at the address
                // pointed to by px; that is the value of x

  cout << "x=" << x << " px=" << px << " y=" << y << endl;
```

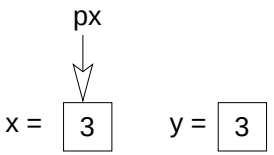
```
  x++;           // increment x

  cout << "x=" << x << " px=" << px << " y=" << y << endl;

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

  cout << "x=" << x << " px=" << px << " y=" << y << endl;

  *px++;         // take away the parens

  cout << "x=" << x << " px=" << px << " y=" << y << endl;

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

  cout << "*px= " << *px << endl;

}
```

- The output is...

```
x=3 px=0xbffff874 y=3
x=4 px=0xbffff874 y=3
x=5 px=0xbffff874 y=3
x=5 px=0xbffff878 y=3
*px=3
```

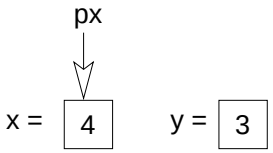
- Here's a picture of what's going on:

step 0:



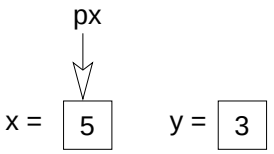
this is the initial situation:
x is initialized to the value 3
px is initialized to point to x
y is initialized to the value pointed to by x

step 1:



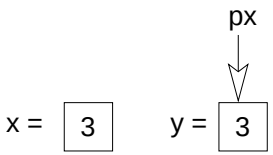
here is the situation after incrementing x

step 2:



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

step 3:



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

Pointer arithmetic

- Incrementing pointers moves the pointer through memory.
- Increasing a pointers' value by 1 increases the address it contains by some multiple of 1.
- The number of bytes in that kind of data.
- In the example above, it is a bit of a party trick — flashy but with no obvious purpose.
- But it does have a serious use.
- Pointer arithmetic is meaningful with arrays:

- Imagine we have:

```
int A[10];
int* pA;
```

- If we do

```
pA = &A[0];
```

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

- We can use pointer arithmetic to access different elements of an array.

```
// 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.

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

Generic pointers

- Last class, we talked about pointers to specific data types, e.g.,:

```
int  *pToInt, *pToInt2;
char *pToChar;
```

- You can also have a pointer to a `void`:

```
void *pToVoid;
```

- Clearly this is not a pointer *to* anything (what is a `void`?).
- A "pointer to a `void`" is a *generic* pointer.
- You can use it to point to different kinds of object.
- When you *dereference* the pointer, it is like converting it to that data type

- Below are all legal statements, given the definitions above:

```
pToVoid = pToInt;
pToInt2 = reinterpret_cast<int*>(pToVoid);
pToChar = &C;
pToVoid = pToChar;
pToVoid = &A;
```

  But you can't do this:

```
pToInt2 = pToVoid;
```

- See `generic-pointer.cpp`
- You can use a generic pointer, for example, as an argument to a function to which you might need to pass different kinds of object.

Summary

- This lecture recapped pointers and arrays.
- But the main topics of the lecture were:
  - Pointer arithmetic; and
  - The relationship between pointers and arrays.
- (These are basically the same topic).
- We also covered generic pointers.