cis15
advanced programming techniques, using c++
fall 2007
lecture # V.1

topics:

- arrays
- pointers
- arrays of objects

resources:

- some of this lecture is covered in parts of Pohl, chapter 3

arrays and pointers overview

- arrays and pointers are strongly related

```
int A[10];   // declare an array of size 10 ints (consecutive in memory)
int *pA;     // declare a pointer to an int
pA = &A[0];  // pA points to the 0th element of A i.e., the address of A[0]
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 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 $A++$ or $p = \&a$
- when an array name is passed to a function, what is passed is the beginning of 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
- an array has a length associated with it
- arrays need:
  - data type
  - name
  - length
- length can be determined:
  - *statically* — at compile time
    e.g., `char str1[10];`
  - *dynamically* — at run time
    e.g., `char *str2;`
    (we'll talk about how to do this in our 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 $length - 1$
- indeces follow array name, enclosed in square brackets (`[ ]`)
  e.g., `arr[25]`

arrays: 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() */
```


arrays: 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"
- & = unary operator that gives address of its argument
- * = 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: memory addresses (1)

- 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;
```

pointers: memory addresses (2)

- 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:

```
countPtr     count
  [•]    →   [12]
```


pointers: memory addresses (3)

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
...		

pointers: 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 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 array element to another

pointers: example

```
// 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;
    cout << " arr[i]=" << arr[i];
    cout << " &arr[i]=" << &arr[i];
    cout << 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;

}
```


and 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.

pointers: another example

```
// 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 to x

  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 );
  // printing them (above) produces something like:
  // x=3 px=0xbffffce0 y=3
  // note that the precise value of px will depend on the machine
  // and may change each time the program is run, because its value
  // depends on what portion of memory is allocated to the program
  // by the operating system at the time that the program is run

  x++;          // increment x

  printf( "x=%d px=%p y=%d\n",x,px,y );
  // printing them (above) produces something like:
  // x=4 px=0xbffffce0 y=3
  // note that the value of x changes, but not px or y

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

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

```
// printing them (above) produces something like:
// x=5 px=0xbffffce0 y=3
// note that the value of x changes, because px contains the
// address of x

// what happens if we take away the parens?
*px++;

printf( "x=%d px=%p y=%d\n",x,px,y );
// printing them (above) produces something like:
// x=5 px=0xbffffce4 y=3
// the value of px changes -- is that what you expected?
// also note that it goes up by 4 -- because it is an integer pointer
// and integers take up 4 bytes

// since px has changed, what does it point to now?
printf( "*px=%d\n",*px );
// the output is:
// *px=3
// because px now points to y's address -- this is because y was
```



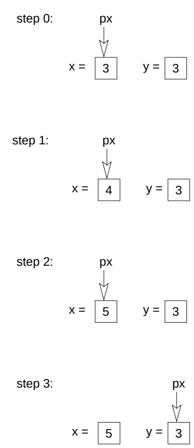
```
  // declared right after x was declared. note that this is usually
  // the case, but not necessarily. use an array to ensure contiguity of
  // addresses.

}
```


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, e.g.:

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

- the difference between them:

```
// refs.cpp

#include <iostream>
#include <cstdio>
using namespace std;
```



```
int main() {

  int A = 1;       // declare and define A
  int B = 2;       // declare and define next memory location after A
  int *p = &A;     // p points to A
  int &refA = A;   // alias (reference) for A
  *p = 3;          // *p points to A, so A is assigned 3

  cout << "initially:      ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
  cout << endl;

  A = *p + 1;      // A is assigned value of *p=3 plus 1
  cout << "after A=*p+1:   ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
  cout << endl;

  A = refA + 1;      // A is assigned value of refA, now 4, plus 1
  cout << "after A=refA+1: ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
```



```
  cout << endl;

  A++;
  cout << "after A++       ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
  cout << endl;

  p++;
  cout << "after p++       ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
  cout << endl;

  refA++;
  cout << "after refA++    ";
  cout << " A=" << A << " p=" << p << " *p=" << *p << " refA=" << refA;
  cout << " &A=" << &A << " &p=" << &p << " &refA=" << &refA;
  cout << endl;

}
```


and the output is:

```
initially:       A=3 p=0xbffff864 *p=3 refA=3 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
after A=*p+1:    A=4 p=0xbffff864 *p=4 refA=4 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
after A=refA+1:  A=5 p=0xbffff864 *p=5 refA=5 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
after A++        A=6 p=0xbffff864 *p=6 refA=6 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
after p++        A=6 p=0xbffff868 *p=2 refA=6 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
after refA++     A=7 p=0xbffff868 *p=2 refA=7 &A=0xbffff864 &p=0xbffff860 &refA=0xbffff864
```

arrays of objects

- you can create arrays of objects just as you create arrays of primitive data types
- example:

```
/* 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; }
  int getX() { return x; }
  int getY() { return y; }
```



```
  void print() const { cout << "(" << x << "," << y << ") "; }
};

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...
- example:

```
/* arrayso1.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; }
  int getX() { return x; }
  int getY() { return y; }
  void print() const { cout << "(" << x << "," << y << ") "; }
};

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;
}
```