DYNAMIC MEMORY

Today

- Today we will start to look at pointers
- The reason for doing this is that we want to be able to use dynamic memory.
- The idea is that rather than declaring how much memory we will need at *compile* time, we can say at *run time*.
- This material is kind of covered in Chapter 3 by Pohl.
- All the examples in these notes are on the class website.

Overview of pointers

- By now you should be happy with the following (see `pointers.cpp`):

```
int a;         // declare an integer
int b[10];     // declare an array of 10 ints
```

- New today is the idea that you can declare:

```
int *aptr;     // declare a pointer to an int
```

  can also be written:

```
int* aptr;
```

  the whitespace makes no difference.

- A pointer contains the address of an element
- Allows one to access the element “indirectly”

- What can we do with pointers?
- Two new operations & and `*`.
- & is a unary operator that gives address of its argument

```
aptr = &a
```

- The pointer now contains the address of `a`.
- When we want `a` we use `*`.

- `*` is a unary operator that fetches contents of its argument (i.e., its argument is an address)
- We call this *dereferencing* the pointer.
- Whatever we do to `aptr` we do to `a`, so

```
*aptr = 6;
```

  sets the value of `a` to 6.
- Since `*aptr` is an integer, we can do any integer thing to it:

```
*aptr = *aptr + 1;
```

- Note that & and `*` bind more tightly than arithmetic operators.

Pointers and memory

- What we covered so far tells us how to *use* pointers.
- Now let's think about what actually happens.
- Pointers are 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:

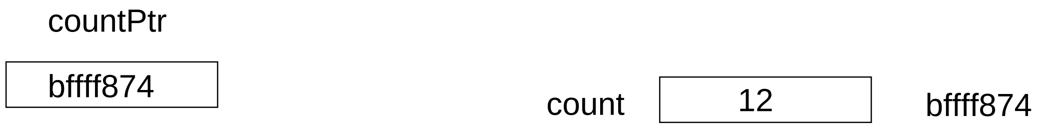
```
int *count;
float *avg;
```

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

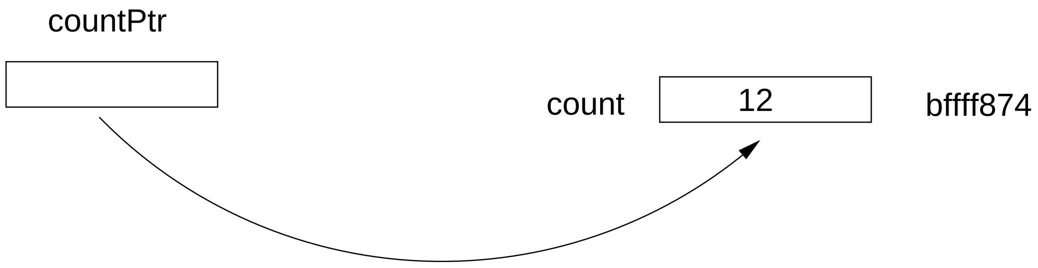
```
int count = 12;
int *countPtr = &count;
```

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

- What happens is something like this:



- Which we usually draw like this:



- When we write

  `*countPtr`

  we are saying "go to the address in `countPtr`".

Dynamic memory allocation

- One of the main reasons we need pointers is to support *dynamic memory allocation*.
- In C++, there are two functions that handle dynamic memory allocation: `new` and `delete`
- For example:

```
int *p, *q, *r;
p = new int(5);  // allocation and initialization
q = new int[10]; // allocation, but uninitialized
r = new int;     // allocation, but uninitialized
```

- Some compilers initialize values to 0 by default, but not all—that is not part of the language specification, so don't rely on it!

- More abstractly he syntax for `new` is:

  `new` *type-name*

  `new` *type-name initializer*

  `new` *type-name[expression]*

- The point of dynamic memory allocation is to allow your program to decide, while running, how much data it needs to store.
- You can, therefore, tailor the size of an array to the problem you are trying to solve.

- Here's an example (modified from the book, p139).

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

int main() {
 int *data;
 int  size;

 cout << "enter array size: ";
 cin >> size;

 data = new int[size]; // allocate array of ints

 for ( int j=0; j<size; j++ ) {
   cout << (data[j]=j) << '\t';
 }
 cout << endl;

} // end of main()
```

- We declare `data` as a pointer to the kind of data we want to store in the array.
- `new` returns an address — the address of the first element of the array.
- After we assign this to `data`, we use `data` as the name of the array.
- Note that we declare the size of the array while the program is running.
- (Just don't try to declare the array *before* you set the value that determines the size.)

- The syntax for `delete` is:

  `delete` *expression*

  `delete []` *expression*

- The first form is for non-arrays; the second form is for arrays
- We use delete to make sure our programs don't have *memory leaks*. where we declare memory and don't "give it back" when we are done with it.
- The next slide gives the example from before with a `delete`.
- Other examples of the use of `new` and `delete` can be found in the two stack handling programs `stack-with-ctors.cpp` (from Unit II) and `dynamic-stack.cpp`.

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

int main() {
 int *data;
 int  size;

 cout << "enter array size: ";
 cin >> size;

 data = new int[size]; // allocate array of ints

 for ( int j=0; j<size; j++ ) {
   cout << (data[j]=j) << '\t';
 }
 cout << endl;

 delete [] data;

} // end of main()
```

- In general, pointers go well with dynamic memory allocation.
- If you don't know how often you will call `new`, then you can't specify the size of an array, and you can't give every new piece of allocated memory a name.
- But you can have a pointer that knows its location in memory.

```
int *pToInt;

pToInt = new int;
```

- To keep track of lots of dynamically allocated memory, we often created linked datastructures, like that in `dynamic-stack.cpp`.
- Other structures like this are covered in CIS 22 Datastructures.

Arrays of objects

- You can create arrays of objects (see `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 more dynamic memory allocation.
- We declare a pointer to an array and then LATER declare the memory for the array using the `new` function.

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

```
int main() {
  point *triagain = new point[3];

  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;
}
```

- You can use pointers to objects in simpler ways also (see `pointers.cpp`):

```
point p;
point* pptr;

pptr = &p;
```

- Having set the pointer to point to the object, we can access the members of the object.
- We can do this by dereferencing the pointer:

```
(*pptr).set(1.2,3.4);
```

- We can also do this using the special operator `->`:

```
pptr->print();
```

Summary

- This lecture looked at pointers.
- We saw how to use pointers.
- We also talked about what pointers do, how they handle memory.
- The reason for talking about pointers is to be able to handle dynamic memory, and we talked about that.
- We also looked at arrays of objects, and pointers to objects.