C++ Dynamic data and pointers

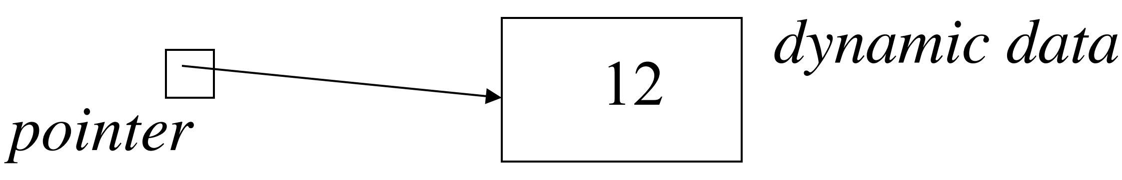
- definitions and motivation
- new
- NULL
- dangling pointers / address-of
- memory leaks
- delete
- pointers to objects
- aliasing
- introduction to linked lists (cont. next time)

Announcements

- Lab 12 this week based on today's material
- PA5: involves linked lists (coming soon)

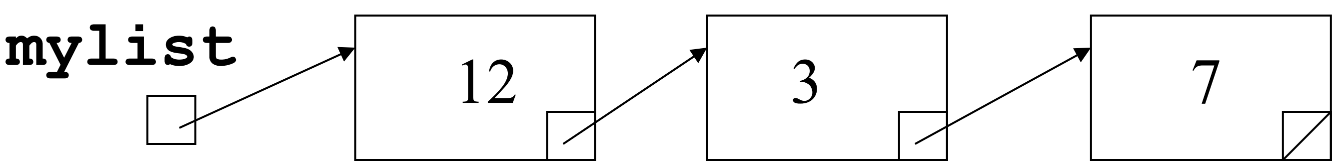
Definitions

- An *address* identifies a location in memory. All data in programs have addresses.
- A *pointer* is a variable that can hold an address
- *dynamic data* is memory that is allocated to your program at run-time.
  - has no name
  - run-time system tells us its address on allocation
  - access it through a pointer to that address
  - the pointer has a name (it's a variable), but the memory it points to does not
- Box and pointer diagram:



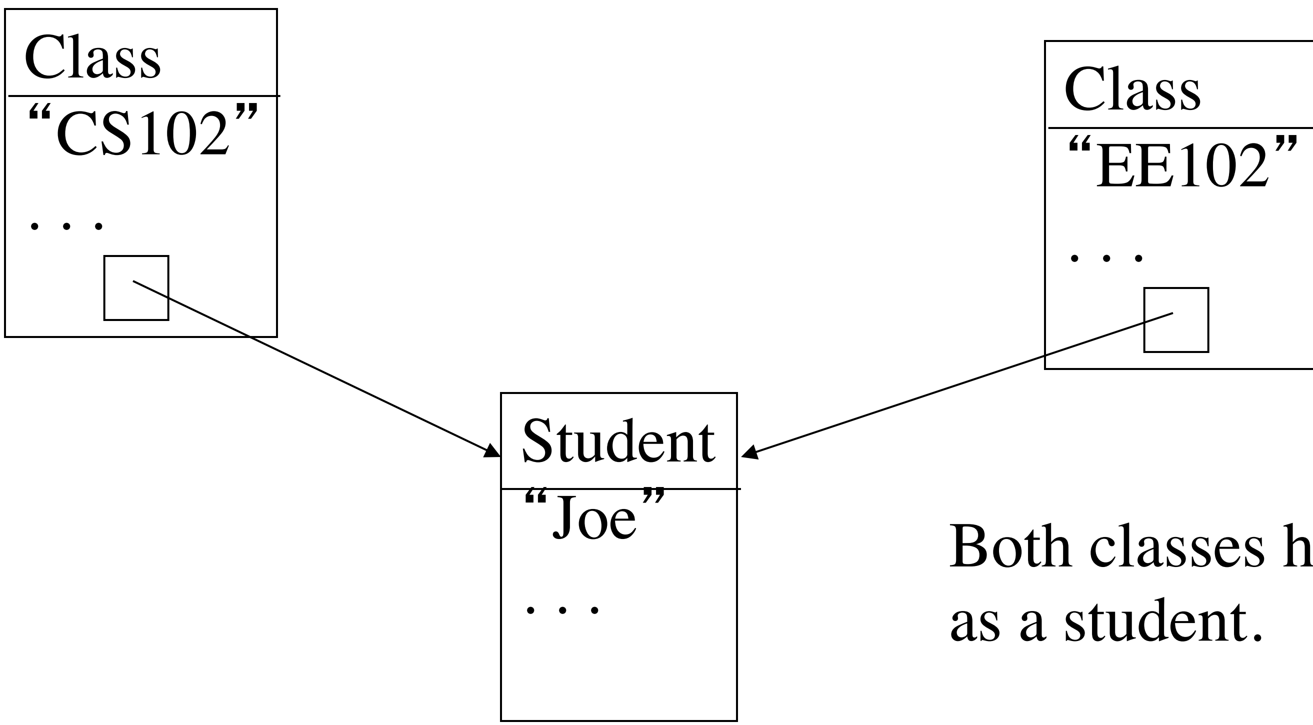
What are pointers for?

- A few different things. First . . .
- Pointers used in dynamic data structures
  - Data structures that can grow and shrink over time
    - E.g., vectors and strings use dynamic arrays
    - Linked lists are another one we will study presently
    - Binary search trees
- A box-and-pointer diagram for a linked list:



What are pointers for? (cont.)

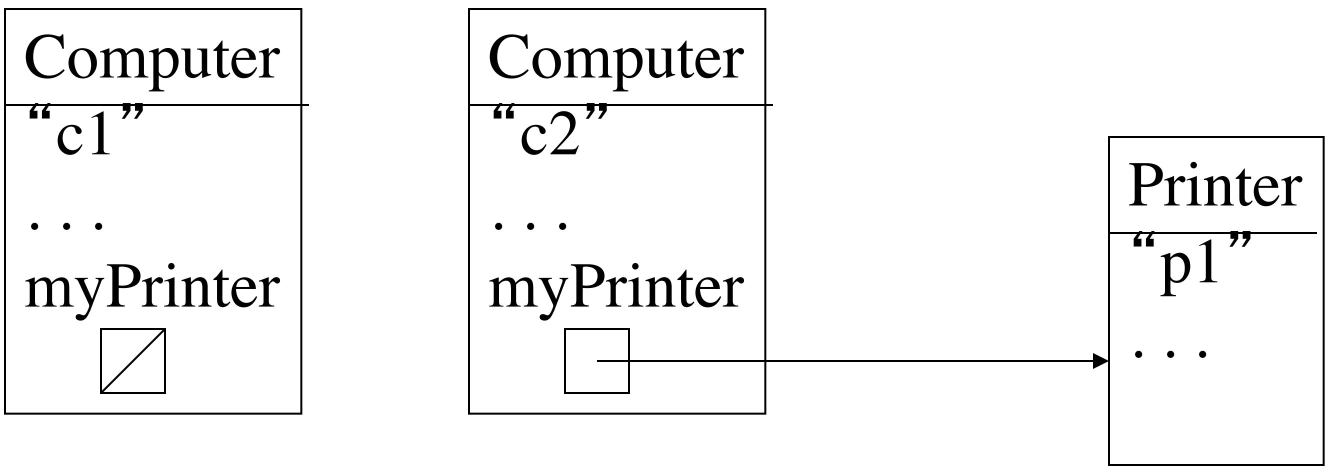
- Pointers used for shared data
  - Two different objects need access to the same data



Both classes have Joe as a student.

What are pointers for? (cont.)

- Pointers used for optional data
  - Some computers have a printer

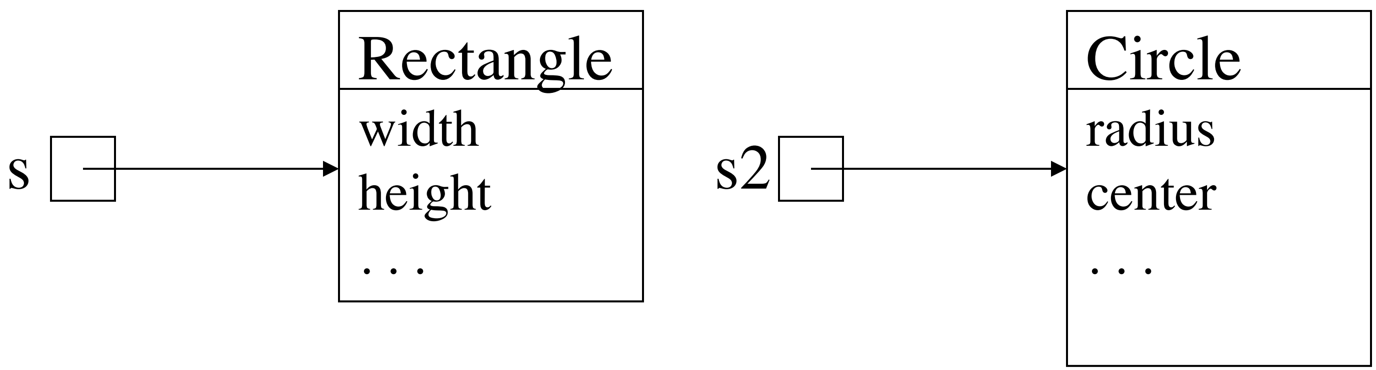


What are pointers for? (cont.)

- Pointers used for object-oriented programming (inheritance / polymorphism)
- E.g., in Java:

```
Shape s = new Rectangle(…);
s.draw();
```

version of **`draw`** called depends on run-time type of object



- in C++: only works with *pointers* to objects

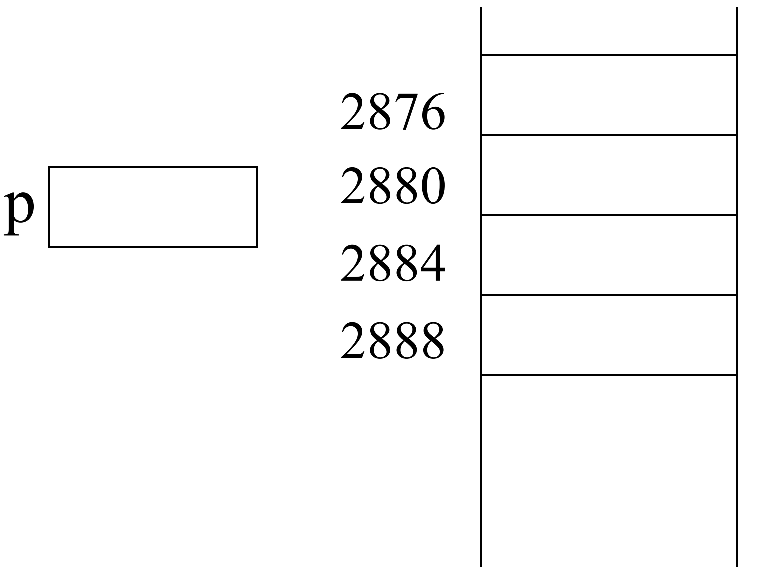
Pointer example

- A pointer variable can only point to memory holding one type of data. E.g.,

```
char * p;     // p is a pointer to char
int * q;    // q is a pointer to int
```

- An example and what's going on behind the scenes:

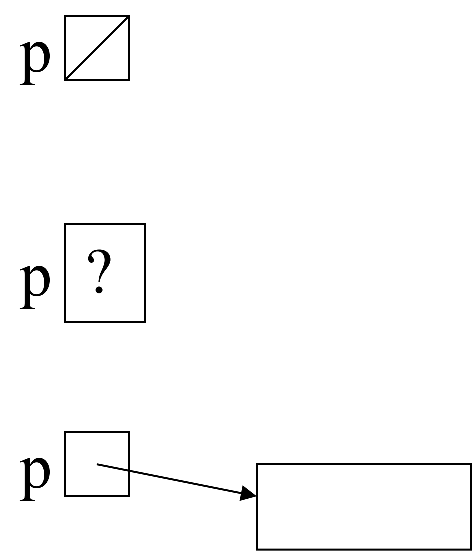
```
char * p;

p = new char;

*p = 'x';

cout << *p;
```



- **`*p`** is called *dereferencing* the pointer

NULL

- NULL is a special pointer value which means the pointer is not pointing to anything.
- You can test if a pointer is NULL to see if it points anywhere.
- 3 states for a pointer:
    1. NULL
    2. uninitialized (unknown)
    3. points to valid memory



See **`testNull.cpp`**

testNull.cpp

```
int * c;
cout << c;
cout << *c;
c = new int;
cout << *c;
*c = 327;
cout << *c;
c = NULL;
cout << *c;
if (c != NULL) {
  cout << *c;
}
```

Dangling pointers and "address of"

- Can also make pointers to non-dynamic data

```
int n;

int * p;

p = &n;   // & is address-of operator
```

- This is a dangerous practice. The pointer might last longer than the variable it points to.
  - suppose **`n`** local, but **`p`** isn't (e.g., instance variable)
  - **`n`**'s memory is later reused for something else
  - **`p`** is now a *dangling pointer*
  - Cause of many bugs in programs (only a run-time error sometimes)

See **`dangle.cpp`**

memory leaks

- Allocating data that we then lose access to. Can't get it back.
- Ex 1 (run out of memory eventually)

```
char * p;
for (int i = 0; i < 100000; i++) {
  p = new char;
}
```

- Ex 2 (common beginner mistake)

```
char * p = new char;
char * tmp = new char;
tmp = p;  // just wasted old *tmp
```

memory leaks (cont.)

- Ex 3

```
void myFunc() {
  char * c = new char;

  // use pointer and dyn data here

}  // c goes out of scope
```

- This is a valid thing to do; but how do we not lose the memory…(and risk running out of space later)?

delete

- We allocate data dynamically with **`new`**.
- We can deallocate the same data when we are done with it with **`delete`**.
- Update of example from last slide:

```
void myFunc() {
  char * c = new char;

  // use pointer and dyn data here

  delete c;
        // release mem pointed to by c
}
```

Dangling pointers from delete

- Use delete very carefully.
- Not safe to release memory when there's still another pointer pointing to it.
- Dangling pointer below:

```
char * p = new char;
. . .
char * tmp = p;
. . .
delete p;
. . .
cout << *tmp;
```

- See example in **`dangle2.cpp`**

Reminder: how to call methods

- Call a method on an object (local var, allocated on the stack)

```
class Student {
 public:
  Student();
  Student(string aName, int aScore);
  int getScore() const;
  . . .
 private:
  string name;
  int score;
};
```

```
Student stud2;        // calls default constructor

Student stud2("joe", 54);

cout << stud2.getScore();
```

Pointer syntax with classes

- Often create pointers to objects and then want to call a method of the object.

```
class Student {
 public:
  Student();
  Student(string aName, int aScore);
  int getScore() const;
  . . .
 private:
  string name;
  int score;
};
```

```
Student * s;
s = new Student();
Student *t = new Student("joe", 54);

cout << (*s).getScore();

cout << *s.getScore();       // does not compile

cout << s->getScore();        // a shortcut
```

Aliasing

- With pointers easy to get two ways to refer to the same object.

```
class Student {
   . . .
   string name;
   int score;
};

Student *p, *r;
p = new Student();
p->setScore(10)'
r = new Student();
r->setScore(50);
```

See **sharedObj.cpp** and **copiedObj.cpp**

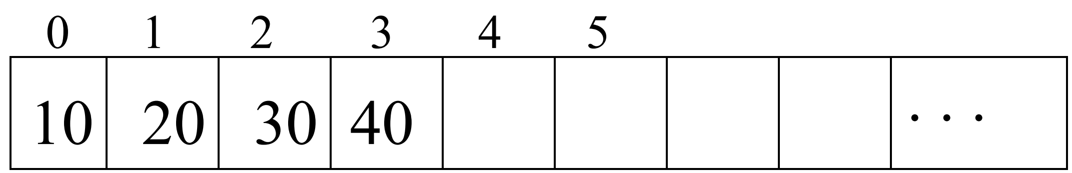
next step

```
p = r;
cout << p->getScore();
r->setScore(0);
cout << p->getScore();
```

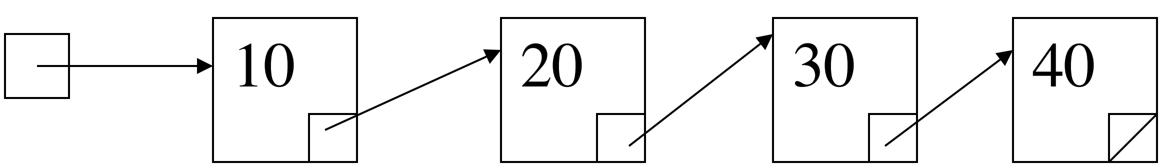
```
*p = *r;
cout << p->getScore();
r->setScore(0);
cout << p->getScore();
```

Introduction to Linked lists

- Want to store a collection of things (elements).
- All elements are the same type
- Can use an array/vector:



- Alternate: linked list
  – Only use as much space as you need at a time.
  – Can insert and delete from middle without shifting values left or right by one.



Linked list types

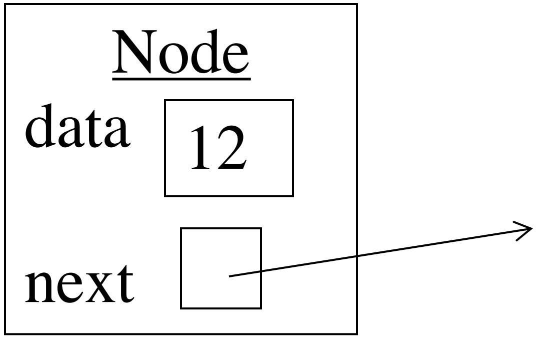
- initially do linked list code outside of a class.
- One “box” is a linked list node:

  store data value together with a pointer.

- we’ll use a **`struct`** for our node type
- **`struct`** is just like **`class`**, but default visibility for fields and member functions is **`public`** instead of **`private`**
- Use **`struct`** to lump some data together without much functionality to go with it.
- Violates idea of encapsulation – use **`struct`** sparingly.

Node with constructors

```
struct Node {
   int data;
   Node *next;
   Node (int item);
   Node (int item, Node *n);
};
Node::Node(int item) {
   data = item;
   next = NULL;
}
Node::Node(int item, Node *n) {
   data = item;
   next = n;
}
```



Example calls:

```
Node *p = new Node(3);
Node *q = new Node(5, p);
Node *r = new Node(12, NULL);
```

Linked list types (cont)

- The type for a linked list variable itself will be
  `Node *`
- Can make a type for this for clarity:
  `typedef Node* ListType;`
- Some examples:

```
ListType list;
list = NULL;  // empty list


insertFront(list, 3);


insertFront(list, 7);


insertFront(list, 4);
```

Traversing a linked list

```
void printList(ListType list) {
  Node *p = list;
  while (p != NULL) {
    cout << p->data << " ";
    p = p->next;
  }
  cout << endl;
}
```