PIC 10A

Lecture 24: The real point of pointers

Why use pointers?

Using pointers to manipulate values of int variables may be neat, but it is not what pointers were created for.

Pointers are ideally suited for creating and keeping track of relationships between objects (classes).

We will see how pointers streamline complicated database structures.

They eliminate the need for having multiple copies of the same object.

They make updating existing data structures easier and much more efficient.

Employee class take one

Recall the Employee class. Our Employee had a name and a salary.

We will update this class so that now Employee also has a boss which is also of type Employee.

```
class Employee
{
public:
    Employee();
    Employee(string n, double s, Employee b);
    string get_name() const;
    double get_salary() const;
private:
    string name;
    double salary;
    Employee boss;
};
```

Let's try to implement Employee

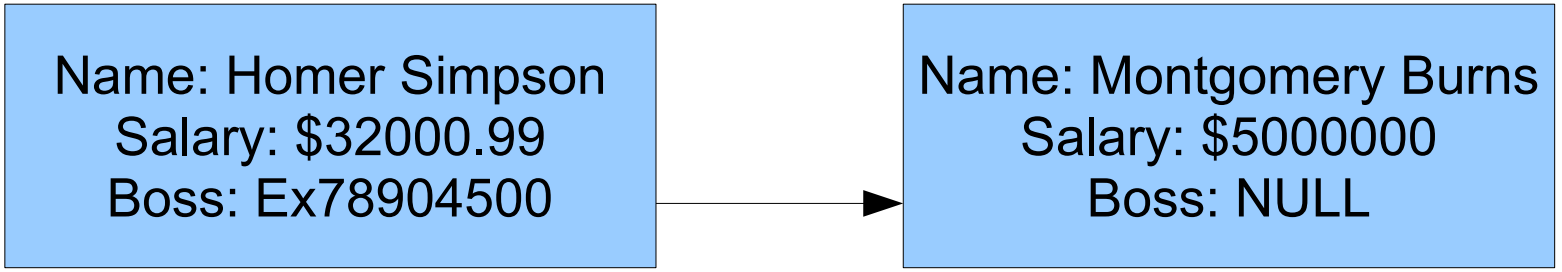
```
Employee::Employee()
{
   name="";
   salary=0;
   // How do we initialize boss??
}
```

Actually this turns out to be a big problem.

Compiler does not even let you write a class like this.

```
error C2460: 'Employee::boss' : uses 'Employee', which
is being defined
```

Employee class take two



Instead of having a class inside the class, lets have boss be a pointer to an Employee class!

Now things become much simpler.

Employee class take two

```
class Employee
{
public:
    Employee();
    Employee(string n, double s, Employee *b);
    string get_name() const;
    double get_salary() const;
    Employee * get_boss() const;
private:
    string name;
    double salary;
    Employee *boss;
};
```

Constructing Employee objects

```
Employee::Employee()
{
    name="";
    salary=0;
    boss = NULL;
}

Employee::Employee(string n, double s, Employee * b)
{
    name = n;
    salary = s;
    boss = b;
}

int main()
{
    Employee Burns("Mr. Burns", 5000000, NULL);
    Employee Homer("Homer Simpson", 32000.99, &Burns);
    return 0;
}
```

Other member functions

```
string Employee::get_name() const
{
   return name;
}

double Employee::get_salary() const
{
   return salary;
}

Employee * Employee::get_boss() const
{
   return boss;
}
```

How do we use a class like this?

```
int main()
{
    Employee Burns("Mr. Burns", 5000000, NULL);
    Employee Homer("Homer Simpson", 32000.99, &Burns);

    cout << "Homer's boss is: " << (*(Homer.get_boss()).get_name();

    return 0;
}
```

Arrow notation

It is such a common thing to have pointer to a class that there is special notation to access a member function through the pointer.

Say we have a pointer p to an object of class Person. To access the persons name we could write:

```
(*p).get_name()
```

Shortcut way is:

```
p->get_name()
```

Example:

```
cout << "Homer's boss is: " << (*(Homer.get_boss()).get_name();
```

becomes:

```
cout << "Homer's boss is: " << Homer.get_boss()->get_name();
```

Pointers to classes on the heap

We can create classes in the heap memory and have pointers point to them

```
Person * p = new Person("Homer", 5556677);
```

Modified book example

Let's create a class called Department.

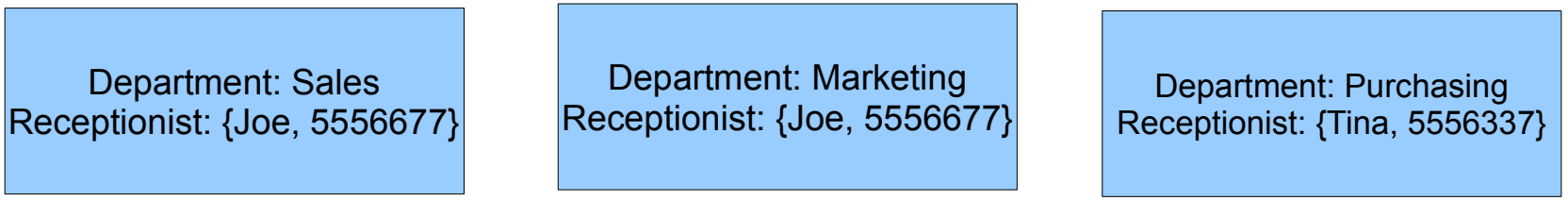
Department is made up of a department's name and a Person. Recall that our Person has a name and a phone number.

```
class department
{
....

private:
   string department_name;
   Person receptionist;
};
```

Modified book Example

Objects of the Department classes can be graphically represented as below:



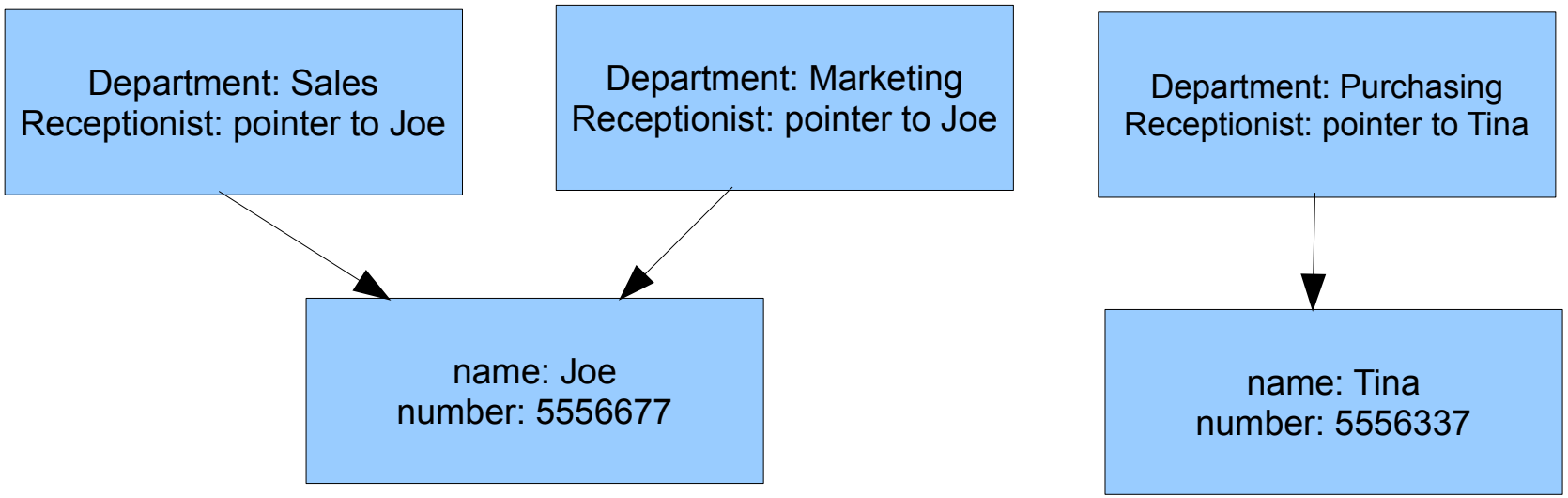
In this example the sales department and the marketing department have the same receptionist Joe.

We are storing in these two objects two copies of identical Person object.

What happens if we need to update Joe's phone number?

Modified book example

Better method is for the department to store a pointer to a person class rather than a whole person class.



Now the sales object and the marketing object are sharing the object containing Joe's data. So now we can update a single object and the changes are recorded everywhere.

Pointers allow us to keep a single copy of an object!

Classes in vectors

We already had a homework assignment where we stored classes in a vector. We had a vector called phone_book where every element of the vector was an instance of the Person class.

Moving classes around in memory is very cpu intensive. Every time we rearrange the vector, delete elements or add elements we are creating copies of classes or moving them around in memory.

Better idea: Instead of storing classes in a vector, store pointers to classes in a vector.

Pointers are just memory addresses they don't take lot of memory. So they can be rearranged deleted or added from a vector with very little effort.

How does a phone_book work with pointers?

```
int main()
{

    vector<Person *> phone_book;

    phone_book.push_back(new Person("Bruin, Joe", 5556456));
    phone_book.push_back(new Person("Simpson, Homer", 5557471));
    phone_book.push_back(new Person("Duffman, Barry", 5533331));

...
```

How does a phone_book work with pointers?

```
void print(vector<Person> phone_book)
{
    for (int i=0; i < phone_book.size(); i++)
    {

        phone_book[i].print();
        cout << "\n";
    }
}

void print(vector<Person *> phone_book)
{
    for (int i=0; i < phone_book.size(); i++)
    {

        phone_book[i]->print();
        cout << "\n";
    }
}
```

How does a phone_book work with pointers?

```
int find (const vector<Person> phone_book, string name)
{
    int i = 0;
    while ( i < phone_book.size() && phone_book[i].get_name() != name)
    {
        i++;
    }
    if (i < phone_book.size())
        return i;
    else return -1;
}

int find (const vector<Person*> phone_book, string name)
{
    int i = 0;
    while ( i < phone_book.size() && phone_book[i]->get_name() != name)
    {
        i++;
    }
    if (i < phone_book.size())
        return i;
    else return -1;
}
```

Linked lists

Another structure that is easy to manage via pointers is linked list of objects.

Every object has a pointer to a previous and next objects.

We usually have a pointer that is pointing to the current object. This pointer might be stored in a variable called current.

To move to the next object or previous object we can write:

```
current = current->get_next();
current = current->get_previous();
```

Importance of NULL pointers

When using pointers it is always important to check that the pointer you are about to access is not NULL.

For example in the linked list case the last element in the list does not have a next element. So the next pointer should be set to NULL.

By checking if the next pointer is NULL we can tell if we are at the end of the list!

Similarly first element of the list does not have a previous element and the previous pointer of the first element should be set to NULL.