# Pointers to Structs and Objects, and the “this” pointer

Sections: 11.9, 13.3, & 14.5

CS 2308
Fall 2013

Jill Seaman



## 11.9: Pointers to Structures

- Given the following Structure:

```
struct Student {
   string name;        // Student’s name
   int idNum;          // Student ID number
   int creditHours;    // Credit hours enrolled
   float gpa;          // Current GPA
};
```

- We can define a pointer to a structure

```
Student s1 = {“Jane Doe”, 12345, 15, 3.3};
Student *studentPtr;
studentPtr = &s1;
```

- Now studentPtr points to the s1 structure.



## Pointers to Structures

- How to access a member through the pointer?

```
Student s1 = {“Jane Doe”, 12345, 15, 3.3};
Student *studentPtr;
studentPtr = &s1;

cout << *studentPtr.name << end;          // ERROR
```

- dot operator has higher precedence than the dereferencing operator, so:

studentPtr is not a structure!

`*studentPtr.name` is equivalent to `*(studentPtr.name)`

- So this will work:

```
cout << (*studentPtr).name << end;       // WORKS
```



## structure pointer operator: ->

- Due to the “awkwardness” of the notation, C has provided an operator for dereferencing structure pointers:

`studentPtr->name` is equivalent to `(*studentPtr).name`

- The **structure pointer operator** is the hyphen (-) followed by the greater than (>), like an arrow.
- In summary:

```
s1.name      // a member of structure s1

sptr->name   // a member of a structure pointed to by sptr
```

## Structure Pointer: example

- Function to input a student, using a ptr to struct

```
void inputStudent(Student *s) {
   cout << "Enter Student name: ";
   getline(cin,s->name);

   cout << "Enter studentID: ";
   cin >> s->idNum;

   cout << "Enter credit hours: ";
   cin >> s->creditHours;

   cout << "Enter GPA: ";
   cin >> s->gpa;
}
```

- Call:

```
Student s1;
inputStudent(&s1);
cout << s1.name << endl;
...
```



## Dynamically Allocating Structures

- Structures can be dynamically allocated with new:

```
Student *sptr;
sptr = new Student;

sptr->name = "Jane Doe";
sptr->idNum = 12345;
...
delete sptr;
```

- Arrays of structures can also be dynamically allocated:

```
Student *sptr;
sptr = new Student[100];
sptr[0].name = "John Deer";
...
delete [] sptr;
```



## Structures and Pointers: syntax

- Expressions:

| | |
|---|---|
| `s->m` | s is a structure pointer, m is a member |
| `*a.p` | a is a structure, p (a pointer) is a member. This expr is the value pointed to by p: `*(a.p)` |
| `(*s).m` | s is a structure pointer, m is a member. Equivalent to `s->m` |
| `*s->p` | s is a structure pointer, and p (a pointer) is in the structure pointed to by s. Equiv to `*(s->p)`. |
| `*(*s).p` | s is a structure pointer, and p (a pointer) is in the structure pointed to by s. Equiv to `*(s->p)`. |



## in 13.3: Pointers to Objects

- We can define pointers to objects, just like pointers to structures

```
Time t1(12,20);
Time *timePtr;
timePtr = &t1;
```

- We can access public members of the object using the structure pointer operator (->)

```
timePtr->addMinute();
cout << timePtr->display() << endl;
```

```
Output:
12:21
```

## Dynamically Allocating Objects

- Objects can be dynamically allocated with new:

```
Time *tptr;
tptr = new Time(12,20);
...
delete tptr;
```

You can pass arguments to a constructor using this syntax.

- Arrays of objects can also be dynamically allocated:

```
Time *tptr;
tptr = new Time[100];
tptr[0].addMinute();
...
delete [] tptr;
```

It can use only the default constructor to initialize the elements in the new array.



## deleting Dynamically Allocated Objects

- Recall IntCell, with dynamically allocated member.

```
class IntCell
{
   private:
      int *storedValue;
   public:
      IntClass(int);
      ~IntClass();
      int read();
      void write(int);
};
```

```
IntCell::IntCell(int val) {
   storedValue = new int;
   *storedValue = val;
}

IntCell::~IntCell() {
   delete storedValue;
}
```



## deleting Dynamically Allocated Objects

When is the storedValue deallocated?

```
#include "IntCell.h"

int main() {

   IntCell *icptr;
   icptr = new IntCell(5);

   cout << icptr->read()
        << endl;

   delete icptr;

   //...

   return 0;
}
```

This calls icptr->~IntCell() first, which deletes (deallocates) icptr->storedValue. Then it deallocates icptr.

```
#include "IntCell.h"

int main() {

   IntCell ic(5);

   cout << ic.read()
        << endl;

   //...

   return 0;
}
```

ic.~IntCell() is called here, which deletes (deallocates) ic.storedValue. Then ic is destroyed.



## in 14.5 The `this` pointer

- `this`: a predefined pointer available to a class's member functions
- `this` always points to the instance (object) of the class whose function is being executed.
- Use `this` to access member vars that may be hidden by parameters with the same name:

```
Time::Time(int hour, int minute) {
   // Time *this; implicit decl

   this->hour = hour;
   this->minute = minute;
}
```

# `this`: an object can return itself

- Often, an object will return itself as the result of a binary operation, like assignment:

  `v1 = v2 = x;` is equivalent to `v1 = (v2 = x);`

- because associativity of = is right to left.
- But what is the result of (v2 = x)?

- It is the left-hand operand, v2.

  `v1 = v2 = x;` is equivalent to
  ```
  v2 = x;
  v1 = v2;
  ```



# Returning `this`

```
class Time {
   private:
     int hour, minute;
   public:
      const Time operator= (const Time &right);
};

const Time Time::operator= (const Time &right) {
   hour = right.hour;
   minute = right.minute;
   return *this;
}

Time time1, time2, time3(2,25);
time1 = time2 = time3;
cout << time1.display() << " "
     << time2.display() << " "
     << time3.display() << endl;
```

Output:
2:25 2:25 2:25