Exceptions, Pointers to Structs, Pointers to Objects

CS 2308
Fall 2011

Jill Seaman

Lecture 15

Using content from textbook slides: Starting Out with C++, Gaddis, Pearson/Addison-Wesley

16.1 : (simple) Exceptions

- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner
- Can be as simple or complex as program design requires

Exceptions - Terminology

- Exception: object or value that signals an error
- Throw an exception: send a signal that an error has occurred
- Catch/Handle an exception: process the exception; interpret the signal

Exceptions: syntax

- throw with argument: used to signal exception:

```
throw <expression>;
```

- try/catch statement:

```
try
{
    /* statements */
}
catch (type param)
{
    /* statements */
}
...
catch (type param)
{
    /* statements */
}
```

Throw should occur during execution of these statements

param type should match type of thrown expression

these statements should process the matching exception

can have multiple catch blocks for a given try block

Exceptions – Semantics

- An exception is called during execution of the try block
- When an exception is thrown, control flow is immediately altered to find a catch block.
- The computer searches the catch blocks immediately after the containing try block for one with a parameter matching the type of the thrown expression.
- If a matching catch block is found, it is executed. Then flow continues after the try/catch stmt.
- If no matching catch block is found, the program terminates.
- If try block executes with no exceptions, then the catch blocks are skipped.

Exceptions: example

- Throw an exception from Time constructor

```
Time::Time(int hr, int min) {

   if (hr > 12 || hr < 1)
      throw "Hour value out of range";

   if (min > 59 || min < 0)
      throw "Minute value out of range";

   hour = hr;
   minute = min;
}
```

Exceptions: example

- Main function catches exception from Time constructor:

```
int main() {

   try {
      Time time1(13,33);
      cout << time1.display() << endl;
   }
   catch (char *msg) {
      cout << "Exception: " << msg << endl;
   }
   cout << "After the try/catch." << endl;

   return 0;
}
```

- Output:

```
Exception: Hour value out of range
After the try/catch.
```


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     // member of structure s1

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

Structure Pointer: example

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

```
void getStudent(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;
getStudent(&s1);
cout << s1.name << endl;
...
```


Dynamically Allocting 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

- 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 Allocting Objects

- Objects can be dynamically allocated with new:

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

You can pass arguments to a constuctor 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 that whenever an object is “destroyed” that its destructor is called.
  - Automatic/regular variables are destroyed at the end of their scope (end of block/function where they are defined).
  - Dynamically allocated objects are destroyed when they are “deleted”.
- If an object contains dynamically allocated variables that are deleted in its destructor (like in SomeClass), then they will be deleted when the containing object is deleted.

deleting Dynamically Allocated Objects

- Recall SomeClass, with dynamically allocated value.

```
class SomeClass
{
   private:
      int *value;
   public:
      SomeClass(int);
      ~SomeClass();
      int getVal();
      void setVal(int);
};

SomeClass::SomeClass(int val) {
   value = new int;
   *value = val;
}

SomeClass::~SomeClass() {
   delete value;
}
```


deleting Dynamically Allocated Objects

- driver that has a ptr to SomeClass:

```
#include "SomeClass.h"

int main() {

   SomeClass *scptr;
   scptr = new SomeClass(5);

   cout << scptr->getVal() << endl;

   delete scptr;

   //...

   return 0;
}
```

This calls the desctructor first, which deletes (deallocates) scptr->value.

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 called.
- When used inside a member function of the Time class (for example), it has this hidden declaration:

```
Time *this;
```


`this`: access hidden members

- You can use `this` to access members that may be hidden by parameters with the same name (especially in constructors/setters):

```
Time::Time(int hour, int minute) {

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

Output:
2:25 2:25 2:25
  ```