Overloading Operators

and Dynamic Memory Allocation

Week 5

Gaddis: 14.5

CS 5301
Fall 2013

Jill Seaman

9.8 Dynamic Memory Allocation

- When a function is called, memory for local variables is automatically allocated.
- When a function exits, memory for local variables automatically disappears.
- Must know ahead of time the maximum number of variables you may need.
- Dynamic Memory allocation allows your program to create variables on demand, during run-time.

The new operator

- "new" operator requests dynamically allocated memory for a certain data type:

```
int *iptr;
iptr = new int;
```

- new operator returns address of newly created anonymous variable.
- use dereferencing operator to access it:

```
*iptr = 11;
cin >> *iptr;
int value = *iptr / 3;
```


Dynamically allocated arrays

- dynamically allocate arrays with new:

```
int *iptr;  //for dynamically allocated array
int size;

cout << "Enter number of ints: ";
cin >> size;
iptr = new int[size];

for (int i=1; i<size; i++) {
   iptr[i] = i;
}
```

- Program will throw an exception and terminate if not enough memory available to allocate

delete!

- When you are finished using a variable created with new, use the delete operator to destroy it:

```
int *ptr;
double *array;

ptr = new int;
array = new double[25];
. . .
delete ptr;
delete [] array;  // note [] required for dynamic arrays!
```

- Do not "delete" pointers whose values were NOT dynamically allocated using new!
- Do not forget to delete dynamically allocated variables (Memory Leaks!!).

9.9 Returning Pointers from Functions

- functions may return pointers:

```
int * findZero (int arr[]) {
   int *ptr;
   ptr = arr;
   while (*ptr != 0)
      ptr++;
   return ptr;
}
```

NOTE: the return type of this function is (int *) or pointer to an int.

- The returned pointer must point to
  - dynamically allocated memory OR
  - an item passed in via an argument

NOTE: if the function returns dynamically allocated memory, then it is the responsibility of the calling function to delete it.

Returning Pointers from Functions: **duplicateArray**

```
int *duplicateArray (int *arr, int size) {

   int *newArray;
   if (size <= 0)          //size must be positive
      return NULL;         //NULL is 0, an invalid address

   newArray = new int [size];  //allocate new array

   for (int index = 0; index < size; index++)
      newArray[index] = arr[index];   //copy to new array

   return newArray;
}
```

```
int a [5] = {11, 22, 33, 44, 55};
int *b = duplicateArray(a, 5);
for (int i=0; i<5; i++)
   if (a[i] == b[i])
      cout << i << " ok" << endl;
delete [] b;  //caller deletes mem
```

Output

```
0 ok
1 ok
2 ok
3 ok
4 ok
```


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
```


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;
```


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.

IntCell declaration

- Problem with the default copy constructor: what if object contains a pointer?

```
class IntCell
{
   private:
      int *storedValue;    //ptr to int

   public:
      IntCell (int initialValue);
      ~IntCell();
      int read () const;
      void write (int x);
};
```


IntCell Implementation

```
#include "IntCell.h"

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

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

int IntCell::read () const {
  return *storedValue;
}

void IntCell::write (int x) {
  *storedValue = x;
}
```


Problem with member-wise assignment

- What we get from member-wise assignment in objects containing dynamic memory (ptrs):

```
IntCell object1(5);
IntCell object2 = object1; // calls copy constructor

  //object2.storedValue=object1.storedValue

object2.write(13);
cout << object1.read() << endl;
cout << object2.read() << endl;
```

What is output?

```
5
13
```

or

```
13
13
```

Programmer-Defined Copy Constructor

- Prototype and definition of copy constructor:

```
IntCell(const IntCell &obj);
```

Add to class declaration

```
IntCell::IntCell(const IntCell &obj) {
   storedValue = new int;
   *storedValue = obj.read();   //or *(obj.storedValue)
}
```

- Copy constructor takes a **reference** parameter to an object of the class
  - otherwise it would use the copy constructor to initialize the obj parameter, which would call the copy constructor: this is an infinite loop

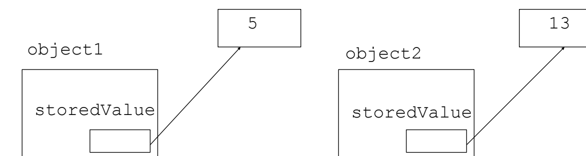
Programmer-Defined Copy Constructor

Each object now points to separate dynamic memory:

```
IntCell object1(5);
IntCell object2 = object1;   //now calls MY copy constr

object2.write(13);
cout << object1.read() << endl;
cout << object2.read() << endl;
```

Output:
5
13



Example class: Time

class declaration with functions defined inline

We will use this for operator overloading examples:

```
class Time {              //new data type
  private:
    int hour;
    int minute;
  public:
    Time()  { hour = 12; minute = 0; }
    Time(int hr,int min) { hour = hr; minute = min; }
    void setHour(int hr) { hour = hr; }
    void setMinute(int min) { minute = min; }
    int getHour() const { return hour; }
    int getMinute() const { return minute; }
    void display() const { cout << hour << ":" << minute; }
};
```


14.5 Operator Overloading

- Operators such as =, +, <, and others can be defined to work for objects of a user-defined class
- The name of the function defining the over-loaded operator is `operator` followed by the operator symbol:

  `operator+` to define the + operator, and

  `operator=` to define the = operator
- Just like a regular member function:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file

Invoking an Overloaded Operator

- Operator can be invoked (called) as a member function:

```
int minutes = object1.operator-(object2);
```

- It can also be invoked using the more conventional syntax:

```
int minutes = object1 - object2;
```

This is the main reason to overload operators, so you can use this syntax for objects of your class

- Both call the same operator- function, from the perspective of object1

Example: minus for Time objects

Subtraction

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

int Time::operator- (Time right) {     //Note: 12%12 = 0
   return (hour%12)*60 + minute -
          ((right.hour%12)*60 + right.minute);
}

//in a driver:
Time time1(12,20), time2(4,40);
int minutesDiff = time2 - time1;
cout << minutesDiff << endl;
```

Output: 260

Overloading == and < for Time

```
bool Time::operator== (Time right) {
   if (hour == right.hour &&
       minute == right.minute)
      return true;
   else
      return false;
}

bool Time::operator< (Time right) {
   if (hour == right.hour)
      return (minute < right.minute);
   return (hour%12) < (right.hour%12);
}

//in a driver:
Time time1(12,20), time2(12,21);
if (time1<time2) cout << "correct" << endl;
if (time1==time2) cout << "correct again"<< endl;
```


Overloading + for Time

```
class Time {
  private:
    int hour, minute;
  public:
    Time operator+ (Time right);
};
Time Time::operator+ (Time right) { //Note: 12%12 = 0
  int totalMin = (hour%12)*60 + (right.hour%12)*60
               + minute + right.minute;
  int h = totalMin / 60;
  if (h==0) h = 12;       //convert 0:xx to 12:xx
  Time result(h, totalMin % 60);
  return result;
}
//in a driver:
  Time t1(12,5);
  Time t2(2,50);
  Time t3 = t1+t2;
  t3.display();
```

Output: 2:55

Overloading Prefix ++ for Time

```
class Time {
  private:
    int hour, minute;
  public:
    Time operator++ ();
};
Time Time::operator++ (Time right) { //Note: 12%12 = 0
  if (minute == 59) {
    minute = 0;
    if (hour == 12)
      hour = 0;
  } else {
    minute++;
  }
  return *this;  //this points to the calling instance
}
//in a driver:
  Time t1(12,55);
  Time t2 = ++t1;
  t1.display(); cout << " ";  t2.display();
```

Output: 12:56 12:56

Overload = for IntCell

Now = for IntCell will not use member-wise assignment

```
class IntCell {
   private:
      int *value;
   public:
      IntCell(const IntCell &obj);
      IntCell(int);
      ~IntCell();
      int read() const;
      void write(int);
      void operator= (IntCell rhs);
};

void IntCell::operator= (IntCell rhs) {
   write(rhs.read());
}

//in a driver:
IntCell object1(5), object2(0);
object2 = object1;
object2.write(13);
cout << object1.read() << endl;
```

Output: 5