Object-oriented programming

Lecture #10: Overloading

Outline

- Function Overloading
- Operator Overloading

Function Overloading

- The ability to give several functions the *same name*, provided the parameters for each of the functions differ in either:
  - number
  - type
  - order

Function Overloading

- (A form of polymorphism, as we discussed in Lecture #9)
- In C++ (and in many programming languages), a function is identified by not only the **name** but also the **number**, the **order** and the **types of the parameters**, which is called the **<u>signature</u>**

Examples

- void swap (unsigned long &, unsigned long &)
- void swap (double &, double &)
- void swap (char &, char &)
- void swap (Point &, Point &)

Each is a different function!!!

Poor Practice

```
class Student {
   public:
     unsigned credits (); // get the credits
     unsigned credits (unsigned n); // set the credits with n
};
```

Functions with the same names should have similar functionality

Overloading Constructors

```
class Point {
  int x, y;
  public:
    Point (int xx = 0, int yy = 0) {
      x = xx; y = yy;
      cout << “Point Constructor.\n”;
    }
};
```

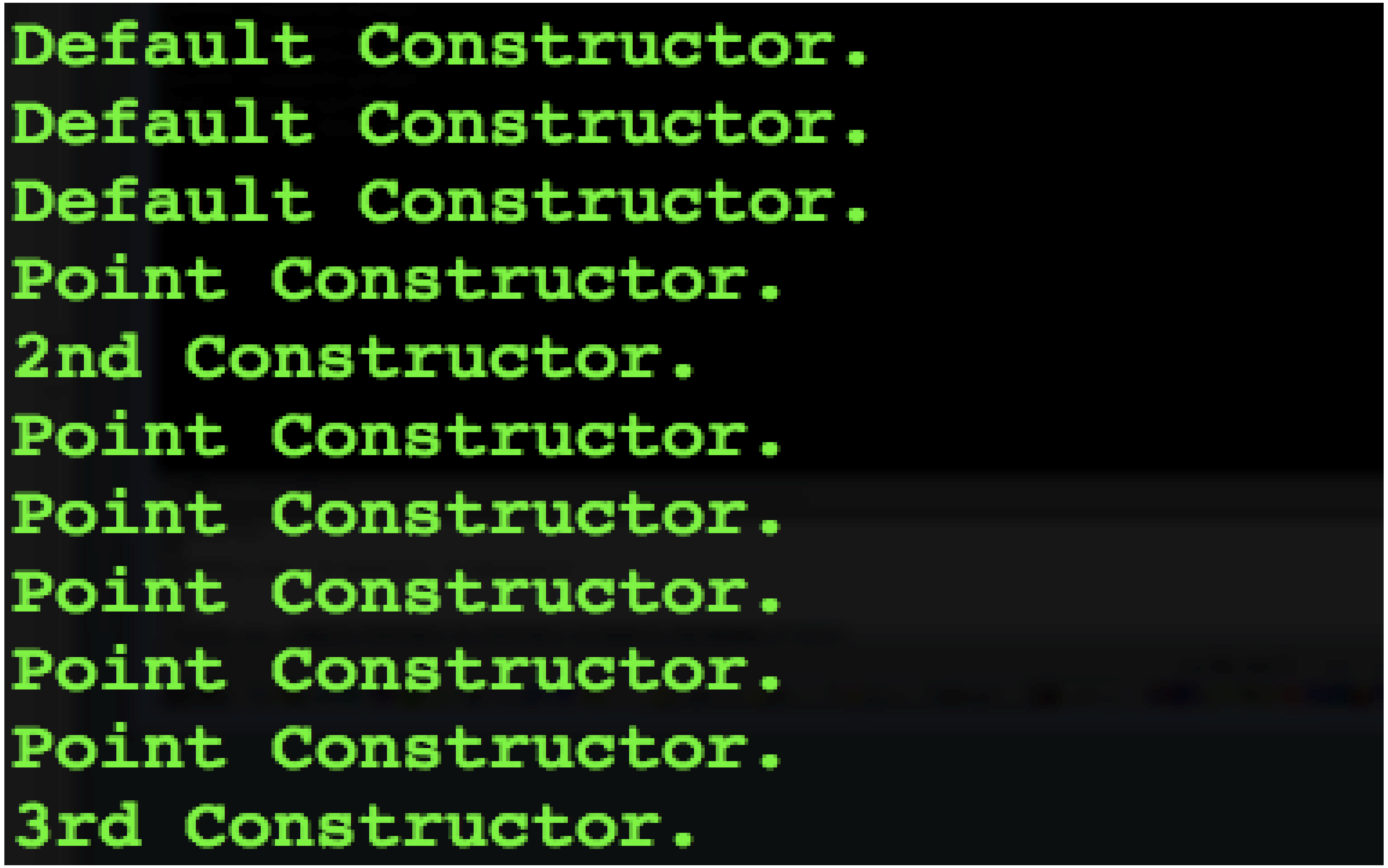
Overloading Constructors

```
class Figure {
   public:
     Figure() { cout << “Default Constructor.\n”; }
     Figure(const Point & center) {
       cout << “2nd Constructor.\n”;
     }
     Figure(const Point vertices[], int count) {
       cout << “3rd Constructor.\n”;
     }
};
```

Overloading Constructors

```
int main() {
    Figure fig1[3];
    Point center(25, 50);
    Figure fig2(center);
    const int VCount = 5;
    Point verts[VCount];
    Figure fig3(verts, VCount);
    return 0;
}
```

Overloading Constructors



```
Default Constructor.
Default Constructor.
Default Constructor.
Point Constructor.
2nd Constructor.
Point Constructor.
Point Constructor.
Point Constructor.
Point Constructor.
Point Constructor.
3rd Constructor.
```

Coercion (revisited)

```
void calculate (long p1, long p2, double p3, double p4);
long a1 = 12345678;
int a2 = 1;
double a3 = 2.3455555;
float a4 = 3.1;
calculate(a1, a2, a3, a4); // OK
Student s;
calculate(s, 10, 5.5, 6) // Incompatible
```

Overloading Resolution

- Best-matching function principle:
  - For each argument, the compiler finds the set of all functions that best match the parameter
  - If resulted in zero or more than one function, an error is reported

Example

```
void display (int x); // version 1
void display (float y); // version 2
int i;
float f;
double d;
display(i); // version 1
display(f); // version 2
display(d); // do not know which one!!!
```

Another Example

```
void print (float a, float b) { cout << "version 1\n”; }
void print(float a, int b)  { cout << "version 2\n”; }
int main() {
  int i = 0, j = 0; float f = 0.0; double d = 0.0;
  print(i, j); // version 2
  print(i, f); // version 1
  print(d, f); // version 1
} // ex4-coercion.cpp
```

Another Example

```
print (d,3.5);   // error
print (i,4.5);   // error
print (d,3.0);   // error
print (i,d);     // error
```

<u>Explicit Casting makes it work</u>

```
print (d,float(3.5)); // version 1
print (i,int(4.5));   // version 2
print (d,float(3.0)); // version 1
print (i,int(d));               // version 2
```

Operator Overloading

- Refers to the technique of ascribing new meaning to standard operators such as +, >>, =, ..., when used with class operands
- In fact, it is a way to name a function
- **Using the same name with some normal operators, make the program more readable**

Operator Overloading

- Define an overloaded operator in class AClass

```
class AClass {
   public:
     int operator +(AClass &a) { return  1; }
   };
   int main() {
     AClass a, b; int i;
     i = a+b; //i = a.operator +(b);
} // Lec10_ex5-overloading-op.cpp
```

Example of Operator Overloading

(Already in C++)

- The ‘+’ symbol has been overloaded to represent:
  - integer addition
  - floating-point addition
  - pointer addition

Operators Allowing Overloading

- Unary Operators
  - new, delete, new[], delete[],
  - ++, --, (), [], +, -, *, &, !, ~,
- Binary operators
  - +, -, *, /, %, =, +=, -=, *=, /=, %=, &, |, ^, ^=, &=, |=, ==, !=, >, <, >=, <=, ||, &&, <<, >>, >>=, <<=, ->, ->*

Operators that do not Allow Overloading

- ‘.’ member access
- ‘.*’ member access-dereference
- ‘::’ scope resolution
- ‘?:’ arithmetic-IF

Restrictions

- Neither the precedence nor the associativity of an operator can be changed
- Default arguments cannot be used
- The “arity” of the operator cannot be changed
- Only existing operators may be overloaded

The Time Class

```
//Lec10_ex6-time
```

The Time Class

- When the compiler sees **++a,** it generates a call to **Time::operator++();** When it sees **b++** it calls **Time::operator++(int);**
- All the user sees is that a different function gets called for the prefix and postfix versions. However, the two functions calls have different signatures, so they link to two different function bodies. The compiler passes a dummy constant value for the **int** argument (which is never given an identifier because the value is never used) to generate the different signature for the postfix version.

“=” Operator and Copy Constructor

- The “=” (Assignment) can also be overloaded.

Copy Constructor

```
Transcript::Transcript( const Transcript & T ) {
      count = T.count;
      courses = new string[MAXCOURSE];
      for(unsigned i = 0; i < count; i++)
        courses[i] = T.courses[i];
      cout <<"copy constructor."<<endl;
}
```

Assignment operator

```
Transcript & Transcript::operator =( const Transcript & T )
{
    if( this != &T ) { // not the same object?
        delete [] courses;
        courses = new string[MAXCOURSE];
        count = T.count;
        for(int i = 0; i < count; i++)
          courses[i] = T.courses[i];
    }
    cout << "= operator.” << endl;
    return *this;
}
```

Assignment Constructor

Lec10_ex7-transcript.cpp