CSC 533: Organization of Programming Languages

Spring 2005

Object-Oriented Programming

- object-based vs. object-oriented programming
- OOP = ADT + inheritance + dynamic binding
- IS_A relationship
- private vs. protected vs. public
- virtual member functions
- example: HW3 OOP version

Object-based programming

object-based programming (OBP):

- solve problems by modeling real-world objects (using ADTs)
- a program is a collection of interacting objects

when designing a program, first focus on the data objects involved, understand and model their interactions

advantages:

- natural approach
- modular, good for reuse
  usually, functionality changes more often than the objects involved

OBP languages: must provide support for ADTs

e.g., C++, Java, JavaScript, Visual Basic, Object Pascal

Object-oriented programming

OOP extends OBP by providing for inheritance

- can derive new classes from existing classes
- derived classes inherit data & operations from parent class,
  can add additional data & operations

advantage: easier to reuse classes,
don't even need access to source for parent class

pure OOP languages: all computation is based on message passing (method calls)

e.g., Smalltalk, Eiffel, Java

hybrid OOP languages: provide for interacting objects, but also stand-alone functions

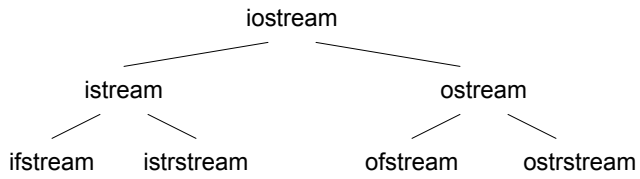
e.g., C++, JavaScript

OOP properties

necessary (but not sufficient) for OOP:

ADTs + inheritance + dynamic (late) binding

example: C++ iostreams



ifstream & ostream are derived from (subclasses of) iostream
ifstream & istrstream are derived from (subclasses of) istream
ofstream & ostrstream are derived from (subclasses of) ostream

the derived class inherits all of the properties (data & methods) of the parent class

- an object of the derived class *IS_A* object of the parent class

IS_A relation & dynamic binding

because an object of the derived class IS A member of the parent class, can pass it (by-reference) anywhere a parent object is expected.

```
void PrintHello(ostream & ostr)
{
    ostr << "Hello" << endl;
}
```

```
PrintHello(cout);

ofstream ostr("greet.out");
PrintHello(ostr);
```

- in a sense, another way of providing generic functions

note: parameter type cannot be determined at compile time – must be bound dynamically

- at run-time, can determine which type of object is being passed in
- select the appropriate function

C++ example: Person class

data: name
social security number
gender
age
...

operations: create a person
have a birthday
display person info
...

```
class Person
{
  public:
    Person(string nm, string id, char sex, int yrs) {
        name = nm;  ssn = id;  gender = sex;  age = yrs;
    }

    void Birthday() {
        age++;
    }

    void Display() {
        cout << "Name: " << name << endl << "SSN : << ssn << endl
             << "Gender: " << gender << endl << "Age: " << age << endl;
    }

  private:
    string name, ssn;
    char gender;
    int age;
};
```

```
Person somePerson("Bjarne", "123-45-6789", 'M', 19);
somePerson.Birthday();
somePerson.Display();
```

Extending a class

now suppose we want to represent information about students

Student = Person + additional attributes/capabilities

(1) could copy the Person class definition, rename as Student, add new features

advantages? disadvantages?

(2) could define the Student class so that it has a Person object embedded inside it

```
class Student
{
  public:
    // STUDENT FEATURES (MEMBER FUNCTIONS)
  private:
    Person self;
    // NEW DATA FIELDS
};
```

advantages? disadvantages?

Extending via inheritance

(3) better solution – use inheritance

- define a Student class that is *derived* from Person
- a derived class inherits all data & functionality from its parent class

- in effect, a Student IS_A Person (with extra)

example: extending person

- data: *inherited person data*
  school
  grade (1-12 high school, 13-16 college, 17- grad school)
  . . .

- operations: *inherited person member functions*
  constructor (with data values)
  advance a grade
  . . .

Student class

```
class Student : public Person
{
  public:
    Student(string nm, string id, char sex, int yrs,
            string sch, int lvl) : Person(nm, id, sex, yrs) {
        school = sch;  grade = lvl;
    }

    void Advance() {
        grade++;
    }

  private:
    string school;
    int grade;
};
```

specifies that Student is derived from Person, public fields stay public

Student constructor initializes its own data fields, but must call the Person constructor to initialize inherited data

Note: only new data fields and member functions are listed, all data/functions from Person are automatically inherited

```
Student someStudent("Bjarne", "123-45-6789", 'M', 19, "Creighton", 13);
someStudent.Birthday();
someStudent.Advance();
```


private vs. protected

recall, data/functions in a class can be

- `private`: accessible only to member functions of the class
- `public`: accessible to any program that includes the class

with inheritance, may want a level of protection in between

- `protected`: accessible to member functions of the class
  AND member functions of derived classes

in our example, Person data fields were declared private

→ Student class cannot directly access those data fields
must instead go through the Person member functions
(just like any other class or program)

serious drawback: when you design/implement a class, have to plan for inheritance

- future extensions to a class are not always obvious

Overriding member functions

when a derived class adds new data,existing functionality may need to be updated

- can override existing member functions with new versions

  e.g., Student class has additional data fields

  → Display member function must be overridden to include these

```
class Student : public Person
{
  public:
    Student(string nm, string id, char sex, int yrs,
            string sch, int lvl) : Person(nm, id, sex, yrs) {
        school = sch;  grade = lvl;
    }

    void Advance() {
        grade++;
    }

    void Display() {
        Person::Display();
        cout << "School: " << school << endl << "Grade: " << grade << endl;
    }

  private:
    string school;
    int grade;
};
```

uses scope resolution operator :: to call the Display that belongs to the Person class

Note: must do this since Person data is private

Polymorphism

different classes can have member functions with the same names

- since member functions *belong to* instances of the class, the compiler does not have any trouble determining which code to execute

```
Person somePerson("Chris", "111-11-1111", 'F', 20);
somePerson.Birthday();              // calls Person::Birthday
somePerson.Display();               // calls Person::Display


Student someStudent("Terry", "222-22-2222", 'M', 20, "Creighton", 14);
someStudent.Birthday();             // calls Student::Birthday (inherited from Person)
someStudent.Advance();              // calls Student::Advance
someStudent.Display();              // calls Student::Display (overriding Person)
```

IS_A relationship

important feature of inheritance:

an instance of a derived class is considered to be an instance of the parent class

a `Student` IS_A `Person`
an `ifstream` IS_A `istream` IS_A `iostream`

thus, a pointer to a parent object can point to a derived object

```
Person * ptr = new Student("Terry", "222-22-2222", 'M', 20, "Creighton", 14);
```

similarly, a derived object can be passed by-reference to a function that expects an object of the parent class

```
void Foo(Person & p)
{
    . . .
    p.Birthday();
    . . .
}
```

```
Person somePerson("Chris", "111-11-1111", 'F', 20);
Foo(somePerson);

Student someStudent("Terry", "222-22-2222", 'M', 20,
                    "Creighton", 14);
Foo(someStudent);
```

note: if you pass a derived object by-value, the copy will be truncated to the parent type

Benefits of IS_A

the IS_A relationship is central to the utility of inheritance

- can define generic functions that work for a family of objects

```
void Foo(Person & p)
{
    . . .
    p.Birthday();
    . . .
}
```

if Foo only utilizes common functionality, then a single function suffices for both classes

PROBLEM: what if the function calls member functions that differ?

```
void Foo(Person & p)
{
    . . .
    p.Display();
    . . .
}
```

```
Foo(somePerson);
Foo(someStudent);
```

```
void Hello(ostream & ostr)
{
    ostr << "Hello" << endl;
}
```

```
Hello(cout);
Hello(outputFile);
```

Dynamic (late) binding

at compile time, can't know if the argument to the function will be of the parent class or a derived class

- by default, the compiler will use the member function from the parent class
- that is, the function call is statically bound to the corresponding code

```
void Foo(Person & p)
{
    . . .
    p.Display();
    . . .
}
```

```
Foo(somePerson);    // Foo calls Person::Display to
                    // display data fields

Foo(someStudent);   // Foo calls Person::Display to
                    // display only those fields
                    // defined in Person
```

in order to call the appropriate (most specific) member function

- must have dynamic binding of the function call

  for `Foo(somePerson)`, bind `Display` call to `Person::Display`

  for `Foo(someStudent)`, bind `Display` call to `Student::Display`

Virtual member functions

in C++, member functions calls are bound statically by default

- to specify dynamic binding, must declare the member function to be "virtual" in the parent class

```
class Person
{
  public:
    . . .
    virtual void Display() {
        cout << "Name: " << name << endl << "SSN : << ssn << endl
             << "Gender: " << gender << endl << "Age: " << age << endl;
    }
  private:
    . . .
};
```

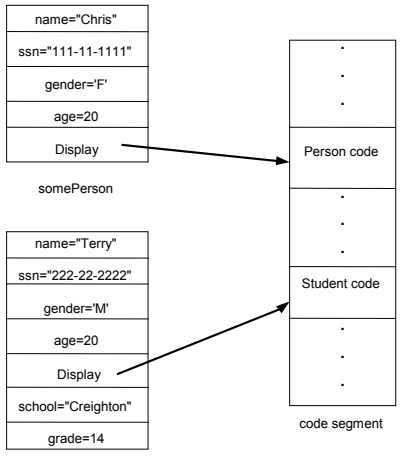
```
Foo(somePerson);         // calls Person::Display in Foo

Foo(someStudent);        // calls Student::Display in Foo
```

Serious drawback: when you design/implement a class, have to plan for inheritance

Implementing virtual member functions

with static binding, the address of the corresponding code is substituted for the call

with dynamic binding, an extra pointer field must be allocated within the object



the pointer stores the address of the corresponding code for that class

when a virtual member function is called, the corresponding pointer in that object is dereferenced to find the correct version of the code

Note: each call to a virtual function implies one level of indirection

→ static binding more efficient

Removing member functions

can even remove existing member functions in the derived class

- put prototype for member function in the private section, don't implement
- since prototype exists, original is overridden
- since not implemented, can't be called

generally, removing member functions is frowned upon

- destroys the IS_A relationship
  if the derived object IS_A parent object, then any operation allowed on a parent object should be allowable on a derived object

why isn't overriding/redefining member functions considered bad?

Multiple inheritance

it is possible for a derived class to inherit from multiple classes

```
class Dean : public Teacher, public Administrator
{
  . . .
}
```

- what if parent classes have different member functions with the same name?

  any call in the derived class must be clarified using the scope resolution operator

  ```
  austin.Administrator::GetSchedule();
  ```

- what if the same parent class appears as multiple ancestors?

  ```
  class Teacher : public Employee
  {
    . . .
  }
  ```

  ```
  class Administrator : public Employee
  {
    . . .
  }
  ```

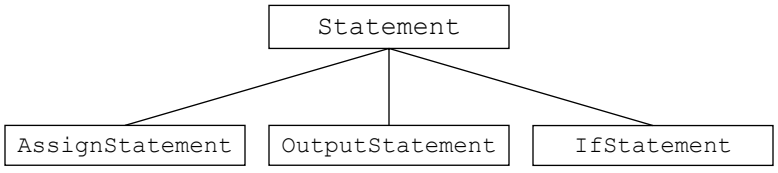
  multiple copies of data fields are made (one for each inheritance path), use :: to clarify

  ```
  austin.Teacher::IDNumber = 12345;
  ```


Example: HW3

consider the SILLY interpreter from HW3

- have different kinds of statements
  assignment statement, output statement, if statement
- each statement has similar functionality
  read a statement, execute a statement, display a statement
- perfect example for inheritance:
  define general statement class, derive classes for each type of statement



Statement class

```
enum STATEMENT_TYPE {ASSIGN, OUTPUT, IF, ERROR};

class Statement
{
  public:
    Statement() { }

    virtual void Read(Tokenizer & program) = 0;
    virtual void Execute(VarTable & variables) const = 0;
    virtual STATEMENT_TYPE GetType() const = 0;
    virtual void Display() const = 0;

    static Statement * GetNext(Tokenizer & program);

};
```

Statement class provides framework for derived classes
- assigning 0 to member functions makes them *abstract*
- abstract functions must be overridden

GetNext reads an arbitrary statement, returns a pointer
- a *static* member function is shared by the class – can be called by itself

need derived classes for assignment, output, and if statements
- each will have its own private data fields
- each will implement the member functions appropriately

Derived classes

```
Statement * Statement::GetNext(Tokenizer & program)
{
    Token next = program.PeekAhead();
    if (next.GetType() == UNKNOWN) {
        return NULL;
    }

    Statement * stmt;

    if (next.GetValue() == "output") {
        stmt = new OutputStatement();
    }
    else if (next.GetValue() == "if") {
        stmt = new IfStatement();
    }
    else if (next.GetType() == IDENTIFIER) {
        stmt = new AssignStatement();
    }
    else {
        cout << "ERROR: unrecognized statement"
             << endl;
        exit(1);
    }

    stmt->Read(program);

    return stmt;
}
```

```
class AssignStatement : public Statement
{
  public:
    AssignStatement() { }
    void Read(Tokenizer & program);
    void Execute(VarTable & vars) const;
    STATEMENT_TYPE GetType() const;
    void Display() const;
  private:
    string lhs;
    Expression rhs;
};
```

```
class OutputStatement : public Statement
{
  public:
    OutputStatement() { }
    void Read(Tokenizer & program);
    void Execute(VarTable & vars) const;
    STATEMENT_TYPE GetType() const;
    void Display() const;
  private:
    Expression rhs;
};
```

```
class IfStatement : public Statement
{
  public:
    IfStatement() { }
    void Read(Tokenizer & program);
    void Execute(VarTable & vars) const;
    STATEMENT_TYPE GetType() const;
    void Display() const;
  private:
    Expression test;
    vector<Statement *> stmts;
};
```

interp.cpp

```
#include <iostream>
#include <string>
#include "Token.h"
#include "Tokenizer.h"
#include "VarTable.h"
#include "Statement.h"
using namespace std;


int main()
{
    string filename;
    cout << "Enter the program file name: ";
    cin >> filename;

    Tokenizer program(filename);
    VarTable variables;

    Statement * stmt = Statement::GetNext(program);
    while (stmt != NULL) {
          stmt->Execute(variables);
          stmt = Statement::GetNext(program);
    }

    return 0;
}
```

prompt the user for the program file

initialize the string tokenizer and the variable table

repeatedly get the next statement and execute

for HW4, will extend existing statements & add new types of statements

- requires no change to main program!