CSC 533: Organization of Programming Languages

Spring 2008

Object-oriented programming in C++

- C++ design goals
- C++ reliability features
  by-reference, const, new & delete, bool, string
- ADTs in C++
  classes, memory management, templates
- OOP in C++
  ADT + inheritance + dynamic binding
  polymorphism, IS_A relationship

C++ design

C++ was developed by Bjarne Stroustrup at Bell Labs in 1984

- C++ is a superset of C, with language features added to support OOP

design goals:

1. support object-oriented programming (i.e., classes & inheritance)
2. retain the high performance of C
3. provide a smooth transition into OOP for procedural programmers

backward compatibility with C was key to the initial success of C++

- could continue to use existing C code; learn and add new features incrementally

however, backward compatibility had far-reaching ramifications

- C++ did add many features to improve reliability & support OOP
- but, couldn't remove undesirable features
  ➔ it is a large, complex, and sometimes redundant language

Added reliability features: pass by-reference

in C, all parameter passing was by-value

```
void reset(int num) {          |     int x = 9;
    num = 0;                   |     reset(x);
}                              |     printf("x = %d", x);
```

- but, could get the effect of by-reference via pointers

```
void reset(int* num) {         |     int x = 9;
    *num = 0;                  |     reset(&x);
}                              |     printf("x = %d", x);
```

C++ introduced cleaner by-reference passing (in addition to default by-value)

```
void reset(int & num) {        |     int x = 9;
    num = 0;                   |     reset(x);
}                              |     cout << "x = " << x;
```

3

Added reliability features: constants

in C, constants had to be defined as preprocessor directives

- weakened type checking, made debugging more difficult

```
#define MAX_SIZE 100
```

C++ introduced the `const` keyword

- can be applied to constant variables (similar to `final` in Java)
  the compiler will catch any attempt to reassign

  ```
  const int MAX_SIZE = 100;
  ```

- can also be applied to by-reference parameters to ensure no changes
  safe since const; efficient since by-reference

  ```
  void process(const ReallyBigObject & obj) {
      . . .
  }
  ```

4

Other reliability features

in C, memory was allocated & deallocated using low-level system calls

- C++ introduced typesafe operators for allocating & deallocating memory

```
int* a = (int*)malloc(20*sizeof(int));      int* a = new int[20];
…                                           …
free(a);                                    delete[] a;
```

in C, there was no boolean type – had to rely on user-defined constants

- C++ `bool` type still implemented as an int, but provided some level of abstraction

```
#define FALSE 0                             bool flag = true;
#define TRUE 1

int flag = TRUE;
```

in C, there was no string type – had to use char arrays & library functions

- C++ `string` type encapsulated basic operations inside a class

```
char* word = "foo";                         string word = "foo";
printf("%d", strlen(word));                 cout << word.length();
```

ADT's in C++

in order to allow for new *abstract data types*, a language must provide:

1. encapsulation of data + operations (to cleanly localize modifications)
2. information hiding (to hide internal details, lead to implementation-independence)

Simula 67 was first language to provide direct support for data abstraction

- class definition encapsulated data and operations; but no information hiding

C++ classes are based on Simula 67 classes, extend C struct types

- data known as *fields*, operations known as *member functions*
- each instance of a C++ class gets its own set of fields (unless declared `static`)
- all instances share a single set of member functions

data fields/member functions can be:

• *public*	visible to all
• *private*	invisible (except to class instances)
• *protected*	invisible (except to class instances & derived class instances)

can override protections by declaring a class/function to be a `friend`

C++ classes

C++classes followed the structure of structs (i.e., records)

- for backward compatiblity, structs remained
- but only difference: in a struct, fields/functions are `public` by default
  in a class, fields/functions are `private` by default

```
struct Point {
    int x;
    int y;
};


struct Point pt;
pt.x = 3;
pt.y = 4;
```

```
class Point {
  public:
    Point(int xCoord, int yCoord) {
      x = xCoord;  y = yCoord;
    }

    int getX() const { return x; }

    int getY() const { return y; }

  private:
    int x;
    int y;
};

Point pt(3, 4);
```

7

Memory management

as in C, variables in C++ are bound to memory stack-dynamically

- allocated when declaration is reached, stored on the stack
- this includes instances of classes as well as primitives

can use `new` & `delete` to create heap-dynamic memory

- requires diligence on the part of the programmer
- must explicitly delete any heap-dynamic memory, or else garbage references persist (there is no automatic garbage collection)
- in order to copy a class instance with heap-dynamic fields, must define a special copy constructor
- in order to reclaim heap-dynamic fields, must define a special destructor

8

Example: card game

can separate class definition into 2 files

- allows for separate (smart) compilation

```
// Card.h
//////////////////////////////////////

#ifndef _CARD_H
#define _CARD_H

using namespace std;

const string SUITS = "SHDC";
const string RANKS = "23456789TJQKA";

class Card {
  public:
    Card(char r = '?', char s = '?');
    char GetSuit() const;
    char GetRank() const;
    int GetValue() const;
  private:
    char rank;
    char suit;
};

#endif
```

```
// Card.cpp
////////////////////////////////////////////////

#include <iostream>
#include <string>
#include "Die.h"
#include "Card.h"
using namespace std;

Card::Card(char r, char s) {
    rank = r;
    suit = s;
}

char Card::GetRank() const {
    return rank;
}

char Card::GetSuit() const {
    return suit;
}

int Card::GetValue() const {
    for (int i = 0; i < RANKS.length(); i++) {
        if (rank == RANKS.at(i)) {
            return i+2;
        }
    }
    return -1;
}
```

9

Example (cont.)

classes/functions can be templated

- idea later adopted by Java generics
- here, vector class is similar to Java ArrayList

```
// DeckOfCards.h
//////////////////////////

#ifndef _DECKOFCARDS_H
#define _DECKOFCARDS_H

#include <vector>
#include "Card.h"
using namespace std;

class DeckOfCards {
  public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    bool IsEmpty() const;
  private:
    vector<Card> cards;
};

#endif
```

```
// DeckOfCards.cpp
//////////////////////////////////////////////////////////

#include <string>
#include "Die.h"
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

DeckOfCards::DeckOfCards() {
  for (int suitNum = 0; suitNum < SUITS.length(); suitNum++) {
    for (int rankNum = 0; rankNum < RANKS.length(); rankNum++) {
      Card card(RANKS.at(rankNum), SUITS.at(suitNum));
      cards.push_back(card);
    }
  }
}

void DeckOfCards::Shuffle() {
  Die shuffleDie(cards.size());

  for (int i = 0; i < cards.size(); i++) {
    int randPos = shuffleDie.Roll()-1;
    Card temp = cards[i];
    cards[i] = cards[randPos];
    cards[randPos] = temp;
  }
}

Card DeckOfCards::DrawFromTop() {
  Card top = cards.back();
  cards.pop_back();
  return top;
}

bool DeckOfCards::IsEmpty() const {
  return (cards.size() == 0);
}
```

10

Example (cont.)

following the convention from C:

`main` is a stand-alone function, automatically called if present in the file

```
#include <iostream>
#include <string>
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

int main() {
    DeckOfCards deck1, deck2;

    deck1.Shuffle();
    deck2.Shuffle();

    int player1 = 0, player2 = 0;
    while (!deck1.IsEmpty()) {
        Card card1 = deck1.DrawFromTop();
        Card card2 = deck2.DrawFromTop();

        cout << card1.GetRank() << card1.GetSuit() << " vs. "
             << card2.GetRank() << card2.GetSuit();

        if (card1.GetValue() > card2.GetValue()) {
            cout << ":  Player 1 wins" << endl;
            player1++;
        }
        else if (card2.GetValue() > card1.GetValue()) {
            cout << ":  Player 2 wins" << endl;
            player2++;
        }
        else {
            cout << ": Nobody wins" << endl;
        }
    }
    cout << endl <<"Player 1: " << player1
         << "  Player2: " << player2 << endl;

    return 0;
}
```

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

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();
```

Student class: extending Person

```
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;
};
```

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

can override a function from the parent class, but still access using the scope resolution operator ::

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();
```

note: `private` data fields are hidden even from derived classes (e.g., name cannot be accessed in Person)
- can access if defined to be `protected` instead

15

IS_A relationship

important relationship that makes inheritance work:

- 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);
```

- since by-reference parameters are really just pointers to objects, this means you can write generic functions that work for a family of objects

```
void Foo(Person & p) {            // can call with a Person or Student
    . . .
    p.Birthday();                 // calls Person::Birthday on either
    . . .
}
```

16

IS_A relationship & dynamic binding

BUT… for the IS_A relationship to work in general, member functions must be bound *dynamically*

- if the parent & derived classes both have member functions with the same name, how can the function know which type of object until it is passed in?

```
void Foo(Person & p)
{
    . . .
    p.Birthday();        // calls Person::Birthday
    . . .
    p.Display();         // calls ???
}

---------------------------

Foo(somePerson);         // would like for Foo to call Person::Display
Foo(someStudent);        // would like for Foo to call Student::Display
```

UNFORTUNATELY… *by default* C++ binds member functions statically

- compiler looks at the parameter type in the function, `Person`, and decides that `p.Display()` must be calling `Person::Display`

17

Dynamic binding & virtual

to specify dynamic binding, individual member functions in the parent class must be declared "virtual"

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

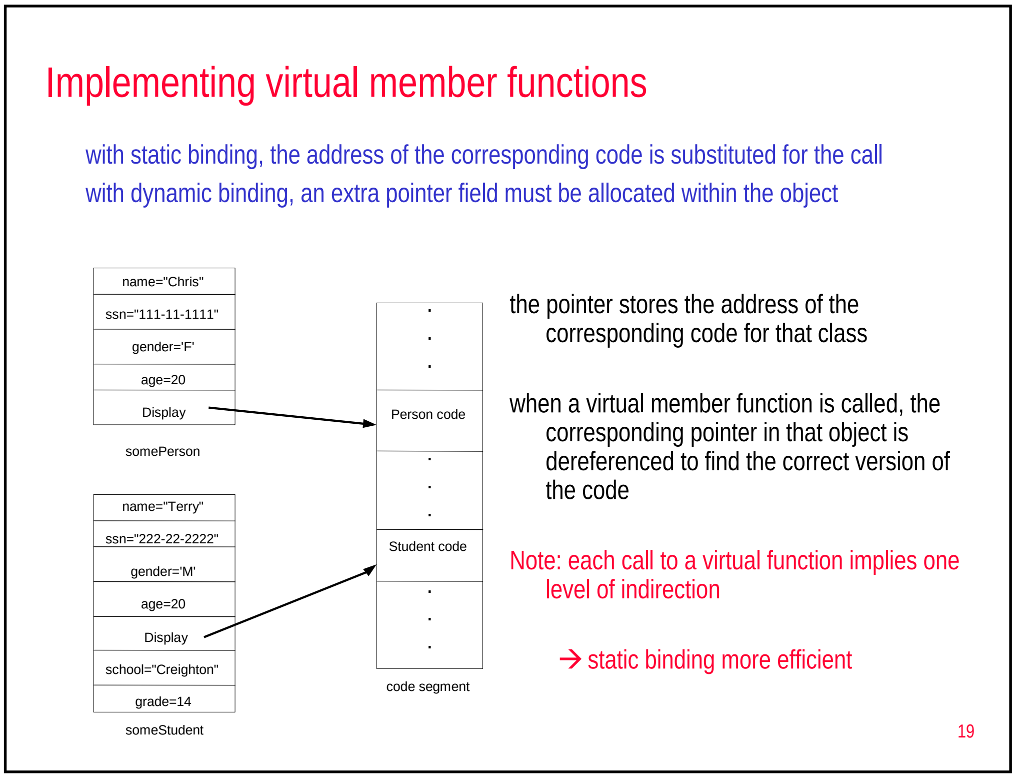
```
Foo(somePerson);         // calls Person::Display in Foo

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

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

note: Java performs dynamic binding automatically

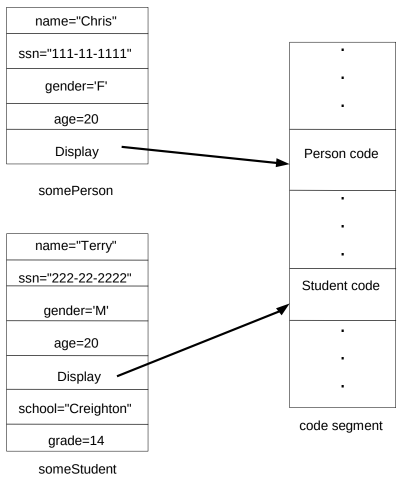
18



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

19