Today's lecture

- Midterm Thursday, October 25, 6:10-7:30pm
  general information, conflicts
- Object oriented programming
  - Abstract data types (ADT)
  - Object oriented design
  - C++ classes

Midterm

date: October 25, 6:10-7:30pm.

location: TBA

Closed book, closed notes, 80 minutes exam. **You need to bring a picture ID to the exam!!!**

There will be no class on Thursday, October 25.

CONFLICT (see p.8 in Undergraduate Schedule):

- Another Common Hour Exam at the same time.
- Another regularly scheduled class at the same time.
- Another regularly scheduled recitation at the same time.

YOU NEED TO ASK YOUR PROFESSOR TO SEND EMAIL TO RYDER@CS TO CONFIRM THE CONFLICT

If you have a conflict, you need to sign-up in the **conflict sheet**. Give the exact reason for your conflict (course number, regularly scheduled hour or common hour exam, ...).

Historic Progression of Data Types

1. User-defined types
   - Can define arbitrary operations
   - Transparent: whole structure of type visible
   - Cannot control access to structure

2. Abstract data types
   - Encapsulation, information hiding
   - Opaque: hides data representation
   - Access restricted to well-defined interface functions

3. Object-orientation
   - Inheritance
   - Code re-use
   - Polymorphic behavior

Data Abstraction

Specification rather than implementation:

- Define *behavior* of data type (through interface functions)
- Hide *implementation* of data type
  $\Rightarrow$ hide details irrelevant to the use of the data type

User-defined Stack Type in C subset of C++

```
#include <stream.h>
typedef int bool;
typedef int elt;
#define MAX  20
#define EMPTY  -1

typedef struct {elt s[MAX]; int top; } stack;

stack * create() {
 stack * newstack = (stack *)malloc(sizeof(stack));
 newstack->top = EMPTY;
 return (newstack);
}
void push(stack* stk,elt data)
 {stk->s[++stk->top]=data;}
void pop(stack* stk) {stk->top--;}
elt peek(stack* stk) {return (stk->s[stk->top]);}

int main() { /**** using the stack ****/
stack *x;    /* stack of indefinite lifetime */
    x = create();
    push(x,2); push(x,3);
    cout << peek(x) << "\n";}
```

Problems with Data Types

- Implementation of the type can be seen
  (E.g., the array inside the stack)

- "Users" of the type can change its value arbitrarily
  E.g., `x->s[5] = 10;`
  - doesn't respect push/top access pattern
  - doesn't respect `elt` typedef

- "Users" of the data type can write operations that create inconsistent states
  (E.g., adding an entry without changing the top index)

- "Users" cannot extend the set of operations in a reliably safe fashion

Abstract Data Types (ADTs)

User *may only* manipulate objects of the type through use of provided functions *without* knowing internal representation

- Encapsulation: may only use provided functions
- Information hiding: cannot see internal representation

Advantages of Abstract Data Types

- Easier to use: as if only type names and function headers were visible
- Safety through access control
  - User can't make inconsistent states
  - User can't make assumptions about data representation
- Designer of ADT can modify implementation without affecting users
- Encourages modularity in programs, facilitating larger, more complex systems

Designing an Abstract Data Type

1. Specify interface
2. Identify and maintain invariants

Example: bounded stack

Interface:

- Stack of some kind of element `elt`
- `create` makes a new, empty stack
- `push` pushes new element on stack; cannot push onto full stack
- `pop` removes an element; cannot pop from empty stack
- `peek` returns the top element on stack
- `is_empty` determines if the stack is empty
- `is_full` determines if the stack is empty

Invariants

- peek(push(S, e)) = e
- pop(push(S, e)) = S
- is_empty(create())
- not is_empty(push(S, e))
- not is_full(pop(S))

Object-Oriented Programming (OOP)

- Similar to abstract data types
  - Allows users to build new types
  - Encapsulation
  - Information hiding
- Allows code sharing or reuse between related types: *inheritance*
- Theory of object-oriented programming is *not* finished
  - "First" object oriented language: Simula'67
  - Different languages work differently
  - Syntax can be complicated
  - Semantics may be ill-defined

Object-Oriented Design

Design:
What components?
How do they interact?

Example: an elevator control system

Elevator

- control panel
  - buttons
  - lights
- door
- speaker

Floor

- control panel
  - buttons
  - lights
- door
- indicator lights

Control system

- multiple floors
- multiple elevators
- location optimizer

Example — Observe

- Different parts of system are independent, with limited interfaces
- Implementation of any one portion can easily be changed
- Objects can be composed of other objects
- Same kind of object may appear in lots of places

Guidelines of OOP

Access to data should be as restricted as possible:

- Each class controls its data
- A class should have access to all and only the data it needs to perform its work

C++

Classes

- Describe abstract data types
- Encapsulate data and define operations on it

Class definitions

- Define *data members* (variables)
- Define *member functions* (or methods)
- Access restriction: `public` and `private`

Objects are instances of classes

ADT Stack in C++

```
// statically allocated stack ADT

#define  MAX  20 // default stack size

typedef int elt ;
typedef int boolean;

class stack{             // encapsulated data type
 private:
    elt s[MAX];             //
    int top;                // hidden data representation
    const int EMPTY = -1;   //

 public:
    stack() { top = EMPTY; }   // constructor => create()

    boolean isempty() { return (top == EMPTY); }

    boolean isfull() { return (top == MAX - 1); }

    void push(elt data)
         {  if (!isfull()) s[++top]=data;
            else cout<<" stack is full; cannot push\n";  }

    void pop()
         {  if (!isempty()) top--;
            else cout<<" stack is empty; cannot pop\n";  }

    elt peek()
        {  if (!isempty()) return s[top];
           else cout<<" stack is empty; cannot peek\n";  }
};
```

Constructors and Destructors

- Define a *constructor*, called to initialize objects (object instances) of the class (constructors may take arguments)
- May define a *destructor*, called to free heap memory used by objects
- Constructors and destructors for class `X`:
  constructor: `X(...)`
  destructor: `~X()`
- Constructor called implicitly when object is allocated (created)
- Destructor called implicitly when control leaves scope of object (end of object's lifetime).

ADT Stack in C++

```
// dynamically allocated stack ADT

typedef struct cell {
    elt info;
    struct cell* link; } CellType;

class stack{
 private:
   CellType * top;
 public:
   stack() {top=NULL;}

   ~stack() { while (top != NULL) pop(); }

   boolean isempty() { return (top == NULL); }

   boolean isfull() { return  0; }

   void push(elt data)
     { CellType* add = new CellType;
       add->info = data;
       add->link = top;
       top = add;  }

   void pop() { CellType* tmp;
                tmp = top;
                top = top->link;      // no error check
                delete tmp; }

   elt peek() { return (top->info); } // no error check
};
```

Functions (and Operators) in C++

- Function body can be defined outside class definition.
  Still need function interface (signature) declaration in class definition.
  (Somewhat similar idea: foo.h and foo.c file)
- Functions can have optional parameters.
- Functions and operators can be overloaded:
  - Have different implementations on different types
  - Must be distinguishable by type signature
  - Like `+` on `int` and `float`

ADT Stack in C++

```
class stack{
 private:
    CellType * top;
 public:
    stack();
    ~stack();
    boolean isempty();
    boolean isfull();
    void push(elt data);
    void pop();
    elt peek();
};
stack::stack() { top=NULL; }
stack::~stack() { while (top != NULL) pop(); }

boolean stack::isempty() { return (top == NULL); }

boolean stack::isfull() { return  0; }

void stack::push(elt data)
     { CellType* add = new CellType;
       add->info = data;
       add->link = top;
       top = add;  }

void stack::pop() { CellType* tmp;
                    tmp = top;
                    top = top->link;     // no error check
                    delete tmp; }

elt stack::peek() { return (top->info); }// no error check
```

Efficiency in OO Code

Encapsulation and information hiding imply many function calls.

Function calls have high run-time overhead.

Efficient compilers *inline* calls where possible:

- Function code is expanded at point of call
- Like a macro

  $\Rightarrow$ larger, unreadable machine code

  $\Rightarrow$ faster machine code

How to deal with large software systems?

Imperative, top–down structured programming — Involves writing the program in very high level descriptions and successively refining these into lower level descriptions, always maintaining a "correct" program. Abstraction levels are driven by task granularities. Often the same or very similar pieces of code are used in different parts of the program.

Object oriented design — based on simulation of the *application world*. The basic entities of the application domain become the entities in the *solution domain* from which the program is build up. Object oriented design begins from the data and builds programs from the bottom up. Separation of specification (abstraction) and implementation.

Object oriented design

Advantages:

- Allows reuse of abstractions in many different settings since typical application domains have commonly used abstractions.
- Abstractions can be reimplemented using different data structures without affecting the design of the program at a higher level (abstract data types (ADTs), representation independence).

Object oriented paradigm

Objects — basic entity in the solution domain

- object is a *set of services* that correspond to an abstraction from the application domain.
- objects simulate real–world entities that they are supposed to model, for instance, a lion, a machine, a stack, an employee.
- objects that provide the same set of services are in the same *class* (have same type) in the solution domain. All objects in the same class have the same *interface*.

Terminology

Object oriented analysis — analysis of the application domain to understand the basic entities, their characteristics, and the relationships among them.

Object oriented design — process of designing a collection of classes and objects to simulate the entities in the application domain.

Object oriented programming — process of implementing an object–oriented design in a suitable programming language, such as C++, Java, Smalltalk, ...

Object oriented programming

How to write an object oriented program in C++?

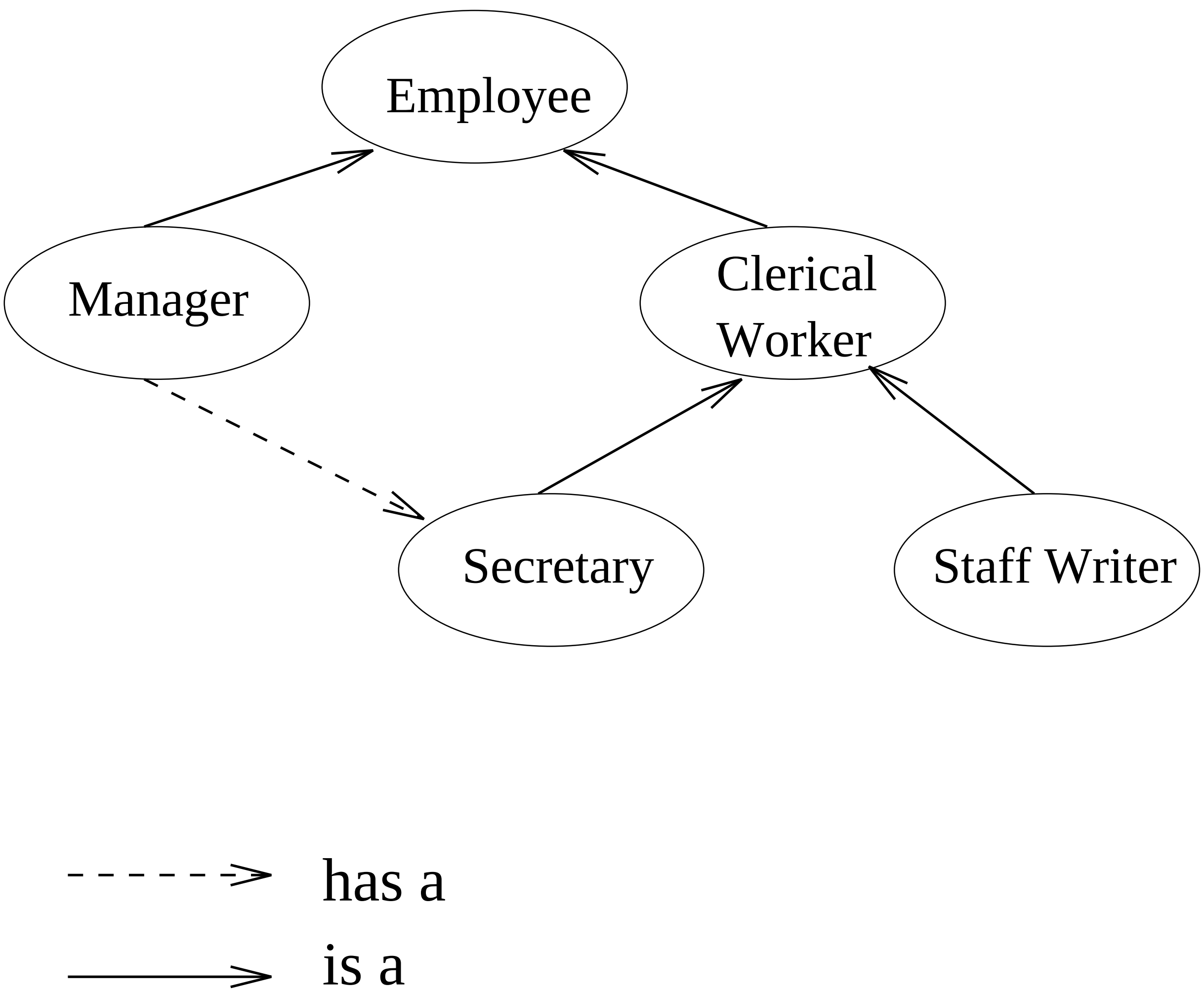
- Identify the entities — What entity should be an object in the solution world (level of abstraction, different implementations, potential for reuse)?
- Identify the behavior of entities — What are the services?
- Identify the relationships between entities — Example: is one entity a specialization of another?
- Create a C++ design structure for the entities — What is the public interface of C++ classes to represent entity types?

Relationships among entities

There are a number of important relationships among entities that are useful in object–oriented design, e.g.:

- is–a — An entity type T1 is in the *is–a* relationship to another entity type T2 if every entity of type T1 is a member of type T2. In the solution domain, this is represented as a relationship between classes implemented using *(public) inheritance* (T2 corresponds to *base class* or *superclass*, and T1 corresponds to *derived class* or *subclass*).
- has–a — An entity e_1 is in the *has–a* relationship with entity e_2, if e_2 is part of e_1 or e_1 uses e_2 for implementation. There are two "types" of the has–a relationship: class level (complete containment) or instance level (share instance via pointer).
- uses–a — Occurs when one class instance takes another class instance as a parameter. For example, a manager might use a particular company facility. In this case, facility is not a manager, nor it is owned by manager.

Example: “Is–a” and “has–a” relationships



Inheritance in Object-Oriented Languages

Subtypes:

- A subtype S of type T:
  any operation that can apply to object t of type T
  can apply to object s of type S.
- Any object s of type S can be used in any context
  that an object of type T can.

Inheritance:

- Provides a means of subtyping and sharing code.
- Allows redefinition of operations.
- In C++, terminology is base classes
  and derived classes.

C++ Derived Classes

- Inherit (data and function) members from base class.
- We use public inheritance.
- May have its own constructors and/or destructors.
- Its own constructors/destructors may explicitly or implicitly call base class constructors/destructors.

Example: “Is–a” and “has–a” relationships

```
#include <stream.h>

class Employee
{ public:
    int ID;
    Employee(int id) {ID = id;} };

class ClericalWorker : public Employee
{ public:
    int group;
    ClericalWorker(int id, int grp) : Employee(id) {group = grp;} };

class Secretary : public ClericalWorker
{ public:
    char *name;
    Secretary(int id, int grp) : ClericalWorker(id, grp) {} };

class StaffWriter : public ClericalWorker
{ public:
    char *name;
    StaffWriter(int id, int grp) : ClericalWorker(id, grp) {} };

class Manager : public Employee
{ public:
    int devision;
    Secretary *assistant;
    Manager(int id, char *nm) : Employee(id)
         {assistant = new Secretary(++id,1);
          assistant->name = nm;} };

main()
{
  Manager Bob(123, "Steve");

  cout << "Manager Bob (ID " << Bob.ID << ") works with "
       << Bob.assistant->name << " (ID " << Bob.assistant->ID << ")\n";
}
```