CSC 533: Organization of Programming Languages
Spring 2005

Procedural and data abstraction
- control structures
  conditionals, loops, branches, …
- subprograms (procedures/functions/subroutines)
  subprogram linkage, parameter passing, implementation, …
- abstract data types (ADTs)
  data + functions, C++ classes, separate compilation, Java classes

We will focus on C++ and Java as example languages

Conditionals & loops

Early control structures were tied closely to machine architecture
e.g., FORTRAN arithmetic if: based on IBM 704 instruction

```
IF (expression) 10, 20, 30
10 code to execute if expression < 0
GO TO 40
20 code to execute if expression = 0
GO TO 40
30 code to execute if expression > 0
40 . . .
```

Later languages focused more on abstraction and machine independence

Some languages provide counter-controlled loops
e.g., in Pascal:
```
for i := 1 to 100 do
begin
  . . .
end;
```

- counter-controlled loops tend to be more efficient than logic-controlled
- C++ and Java don't have counter-controlled loops (for is syntactic sugar for while)
Branching

unconditional branching (i.e., GOTO statement) is very dangerous
  - leads to spaghetti code, raises tricky questions w.r.t. scope and lifetime
    what happens when you jump out of a function/block?
    what happens when you jump into a function/block?
    what happens when you jump into the middle of a control structure?

most languages that allow GOTO's restrict their use
  - in C++, can't jump into another function
    can jump into a block, but not past declarations

```cpp
void foo() {
  ...  
  goto label2; // illegal: skips declaration of str
  ...  
  label1:
  string str;
  ...  
  label2:
  goto label1; // legal: str's lifetime ends before branch
}
```

Branching (cont.)

why provide GOTO's at all? (Java doesn't)
  - backward compatibility
  - can be argued for in specific cases (e.g., jump out of deeply nested loops)

C++ and Java provide statements for more controlled loop branching
  - `break`: causes termination of a loop

```cpp
while (true) {
  cin >> num;
  if (num < 0) break;
  sum += num;
}
```

  - `continue`: causes control to pass to the loop test

```cpp
while (inputKey != 'Q') {
  if (keyPressed()) {
    inputKey = GetInput();
    continue;
  }  
  ...  
}
```
Procedural control

any implementation method for subprograms is based on the semantics of subprogram linkage (call & return)

in general, a subprogram call involves:
1. save execution status of the calling program unit
2. parameter passing
3. pass return address to subprogram
4. transfer control to subprogram
   possibly: allocate local variables, provide access to non-locals

in general, a subprogram return involves:
1. if out-mode parameters or return value, pass back value(s)
2. deallocate parameters, local variables
3. restore non-local variable environment
4. transfer control to the calling program unit

Parameters

in most languages, parameters are positional

- Ada also provides keyword parameters:

  AddEntry(dbase -> cds, new_entry -> mine);

  advantage: don’t have to remember parameter order
  disadvantage: do have to remember parameter names

C++ & Java allow for optional parameters (specify with …)

- no type checking performed!

  printf("Hello world\n");
  printf("%d, %d", num1, num2);
Parameters (cont.)

Ada and C++ allow for default values for parameters
- if value is passed in as argument, that value is assigned to parameter
- if not, default value is assigned

```cpp
void Display(const vector<int> & nums, ostream & ostr = cout) {
    for (int i = 0; i < nums.size(); i++) {
        ostr << nums[i] << endl;
    }
}
```

```cpp
ofstream ofstr("foo.out");
Display(numbers, ofstr); // displays to file
Display(numbers); // displays to cout
```

Note: default parameters must be rightmost in the parameter list WHY?

Parameter passing

can be characterized by the direction of information flow

- **in mode:** pass by-value
- **out mode:** pass by-result
- **inout mode:** pass by-value-result, by-reference, by-name

by-value (in mode)
- parameter is treated as local variable, initialized to argument value

  - **advantage:** safe (function manipulates a copy of the argument)
  - **disadvantage:** time & space required for copying

used in ALGOL 60, ALGOL 68
default method in C++, Pascal, Modula-2
only method in C (and, technically, in Java)
Parameter passing (cont.)

by-result (out mode)
- parameter is treated as local variable, no initialization
- when function terminates, value of parameter is passed back to argument

potential problems:
- `ReadValues(x, x);`
- `Update(list[GLOBAL]);`

by-value-result (inout mode)
- combination of by-value and by-result methods
- treated as local variable, initialized to argument, passed back when done

same potential problems as by-result

used in ALGOL-W, later versions of FORTRAN

by-reference (inout mode)
- instead of passing a value, pass an access path (i.e., reference to argument)

  advantage: time and space efficient
  disadvantage: slower access to values (must dereference), alias confusion

void IncrementBoth(int & x, int & y)
{
    x++;
    y++;
}

requires care in implementation: arguments must be l-values (i.e., variables)

used in early FORTRAN
can specify in C++, Pascal, Modula-2
Java objects look like by-reference
Parameter passing (cont.)

by-name (inout mode)

- argument is textually substituted for parameter
- form of the argument dictates behavior
  - if argument is a:
    - variable ➔ by-reference
    - constant ➔ by-value
    - array element or expression ➔ ???

```plaintext
real procedure SUM(real ADDER, int INDEX, int LENGTH);
begin
    real TEMPSUM := 0;
    for INDEX := 1 step 1 until LENGTH do
        TEMPSUM := TEMPSUM + ADDER;
    end;

    SUM := TEMPSUM;
end;

SUM(X, I, 100) ➔ 100 * X
```

- flexible but tricky – used in ALGOL 60, replaced with by-reference in ALGOL 68

Parameters in Ada

in Ada, programmer specifies parameter mode

- implementation method is determined by the compiler

  - in ➔ by-value
  - out ➔ by-result
  - inout ➔ by-value-result (for non-structured types)
  ➔ by-value-result or by-reference (for structured types)

- choice of inout method for structured types is implementation dependent

  DANGER: IncrementBoth(a, a) yields different results for each method!
Parameters in Java

parameter passing is by-value, but looks like by-reference for objects

- recall, Java objects are implemented as pointers to dynamic data

```java
public static void Foo(ArrayList lst)
{
    lst.set(0, "okay");
    ...
    lst = new ArrayList();
}
```

```java
ArrayList numList = new ArrayList(5);
Foo(numList);
```

when pass an object, by-value makes a copy (here, copies the pointer)
pointer copy provides access to data fields, can change
but, can’t move the original

Polymorphism

in C++ & Java, can have different functions with the same name

- overloaded functions must have different parameters to distinguish

```java
void Display(string X) {
    cout << X << endl;
}
```

```java
void Display(string X, ostream & ostr) {
    ostr << X << endl;
}
```

in C++, could get same effect with default parameter

common use in OOP: different classes with same member function names

in C++, can overload operators for new classes

```java
bool Date::operator==(const Date & d1, const Date & d2)
// postcondition: returns true if d1 and d2 are same date, else false
{
    return (d1.day == d2.day &&
            d1.month == d2.month &&
            d1.year == d2.year);
}
```
Generic types

in C++ can parameterize classes/functions using templates

template <class Type>
class MyList {  template <class Item>
  public:
  . . .
  private:
  <Type> items[];
};

must specify Type when declare
an object

MyList<int> nums(20);

when called, Item is automatically
instantiated (must support <<)

<Type> items[];

Date day(9, 27, 2000);
Display(day);

can similarly write generic classes & methods in Java

public class MyList<T> { private T[] items; . . . }

public <T> void Display(T x) { System.out.println(x) }

Implementing subprograms

- some info about a subprogram is independent of invocation
  e.g., constants, instructions
  ➔ can store in static code segment

- some info is dependent upon the particular invocation
  e.g., return value, parameters, local variables (?)
  ➔ must store an activation record for each invocation

- local variables may be allocated when
  subprogram is called, or wait until
  declarations are reached (stack-dynamic)

Activation Record

<table>
<thead>
<tr>
<th>local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
</tr>
<tr>
<td>static link</td>
</tr>
<tr>
<td>dynamic link</td>
</tr>
<tr>
<td>return address</td>
</tr>
</tbody>
</table>

16
Run-time stack

when a subroutine is called, an instance of its activation record is pushed

```
program MAIN;
  var a : integer;

procedure P1;
  begin
    print a;
  end; {of P1}

procedure P2;
  var a : integer;
  begin
    a := 0;
    P1;
  end; {of P2}

begin
  a := 7;
  P2;
end. {of MAIN}
```

when accessing a non-local variable
- follow static links for static scoping
- follow dynamic links for dynamic scoping

Run-time stack (cont.)

when a subroutine terminates, its activation record is popped (note LIFO behavior)

```
program MAIN;
  var a : integer;

procedure P1;
  begin
    print a;
  end; {of P1}

procedure P2;
  var a : integer;
  begin
    a := 0;
    P1;
  end; {of P2}

begin
  a := 7;
  P2;
end. {of MAIN}
```

when the last activation record is popped, control returns to the operating system
Run-time stack (cont.)

Note: the same subroutine may be called from different points in the program.

```plaintext
program MAIN;
var a : integer;

procedure P1;
begin
  print a;
end; {of P1}

procedure P2;
var a : integer;
begin
  a := 0;
P1;
end; {of P2}

begin
  a := 7;
P2;
P1;
end. {of MAIN}
```

Using dynamic scoping, the same variable in a subroutine may refer to a different addresses at different times.

In-class exercise

run-time stack?

output using static scoping?

output using dynamic scoping?

```plaintext
program MAIN;
var a : integer;

procedure P1(x : integer);
procedure P3;
begin
  print x, a;
end; {of P3}
begin
  P3;
end; {of P1}

procedure P2;
var a : integer;
begin
  a := 0;
P1(a+1);
end; {of P2}

begin
  a := 7;
P1(10);
P2;
end. {of MAIN}
```
Optimizing scoping

 naïve implementation:
  - if variable is not local, follow chain of static/dynamic links until found

in reality, can implement static scoping more efficiently (displays)
  - block nesting is known at compile-time, so can determine number of links that must be traversed to reach desired variable
  - can also determine the offset within the activation record for that variable

→ can build separate data structure that provides immediate access

can’t predetermine # links or offset for dynamic scoping
  - subroutine may be called from different points in the same program

can’t even perform type checking statically  why not?

Data abstraction

pre 80’s: focus on process abstraction
recently: data abstraction increasingly important
  Object-Oriented Programming (OOP) is an outgrowth of data abstraction in software development

an abstract data type (ADT) requires
  1. encapsulation of data and operations
     cleanly localizes modifications
  2. information hiding (hide internal representation, access through operations)
     makes programs independent of implementation, increases reliability

Simula 67: first to provide direct support for data abstraction
  - class definition encapsulated data and operations
  - no information hiding
ADT’s in Modula-2

Modula-2 provides encapsulation via modules

- **definition module**: partial specification of types, plus subprogram headers
- **implementation module**: completed definitions of types, subprograms

Can be defined in separate files, compiled separately

Modula-2 provides information hiding via opaque types

- **transparent type**: complete definition of type in definition module
  - underlying data is visible and accessible
- **opaque type**: no implementation details in definition module
  - underlying data is hidden

Client program imports definition module (implementation is linked later):
- **Problem**: compiler must know size of an object when declared
- **Solution**: opaque types must be implemented as pointers to structures

Modula-2 example

```pascal
IMPLEMENTATION MODULE stackmod;
FROMInOut IMPORT WriteString, WriteLn;
FROM Storage IMPORT ALLOCATE;
cost max = 100;

type stacktype = POINTER TO
  RECORD
    list : ARRAY[1..max] OF INTEGER;
    topsub : [0..max]
  END;

PROCEDURE create(VAR stk:stacktype);
BEGIN
  NEW(stk);
  stk^.topsub := 0
END create;

PROCEDURE push(VAR stk:stacktype; ele:INTEGER);
BEGIN
  IF stk^.topsub = max THEN
    WriteString("ERROR – Stack overflow");
    WriteLn
  ELSE
    INC(stk^.topsub);
    stk^.list[stk^.topsub] := ele
END

PROCEDURE pop(VAR stk:stacktype);
BEGIN
  IF stk^.topsub > 0 THEN
    stk^.topsub := stk^.topsub - 1;
    IF stk^.topsub = 0 THEN
      WriteString("ERROR – Stack underflow");
      WriteLn
    ELSE
      WriteString("WARNING – Stack underflow");
      WriteLn
    END
  END;
END stackmod;
```

Here, stacktype is opaque
- no details in definition module
- defined as a pointer to a record in the implementation module
- memory must be dynamically allocated
- lots of pointer dereferencing
ADT's in C++

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

- in Modula-2, modules export type definitions and applicable functions
- in C++, classes export an ADT that contains its own member functions

All instances of a C++ class share a single set of member functions
Each instance gets its own set of data fields (unless declared static)

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++ example

#ifdef _STACK_H
#define _STACK_H

#include <vector>
using namespace std;

template <class Item>
class Stack {
public:
    Stack() {
        // nothing more needed
    }

    void push(Item x) {
        vec.push_back(x);
    }

    void pop() {
        vec.pop_back();
    }

    Item top() const {
        return vec.back();
    }

    bool isEmpty() const {
        return (vec.size() == 0);
    }

private:
    vector<Item> vec;
};
#endif
C++ example (cont.)

the client program must:
- include the .h file

once included, the user-defined class is indistinguishable from primitives
- can declare objects of that type
- can access/modify using member functions
  
  object.memberFunc(params)

C++ example (cont.)

the Standard Template Library (STL) contains many useful class definitions:
- stack
- queue
- priority_queue
- set
- map
Separate compilation

as in Modula-2, can split non-templated class definitions into:

- interface (.h) file
- implementation (.cpp) file

```cpp
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
 Die sixSided(6), eightSided(8);
 int roll16 = -1, roll18 = -2;
 while (roll16 != roll18) {
 roll16 = sixSided.Roll();
 roll18 = eightSided.Roll();
 cout << sixSided.numRolls() << "": "
 << roll16 << " " << roll18 << endl;
 } cout << "DOUBLES!!" << endl;
 return 0;
}
```

Separate compilation (cont.)

the client program must:

- include the .h file
- add the .cpp file to the project (Visual C++)

advantages:

- compiler only compiles files that have been changed
- .h file is a readable reference
- can distribute compiled .obj file, hide .cpp source code from user
ADTs in Java

Java classes look very similar to C++ classes
- each field/method has its own visibility specifier
- must be defined in one file, can’t split into header/implementation
- javadoc facility allows automatic generation of documentation
- extensive library of data structures and algorithms
  - List: ArrayList, LinkedList
  - Set: HashSet, TreeSet
  - Map: HashMap, TreeMap
  - Queue, Stack, …
- load libraries using `import`

```java
public class Die {
    private int numSides;
    private int numRolls;

    public Die() {
        numSides = 6;
        numRolls = 0;
    }

    public Die(int sides) {
        numSides = sides;
        numRolls = 0;
    }

    public int getNumberOfSides() {
        return numSides;
    }

    public int getNumberOfRolls() {
        return numRolls;
    }

    public int roll() {
        numRolls = numRolls + 1;
        return (int)(Math.random() * getNumberOfSides()) + 1;
    }
}
```

Tuesday: TEST 1

types of questions:
- factual knowledge: TRUE/FALSE
- conceptual understanding: short answer, discussion
- synthesis and application: parse trees, heap trace, scoping rules, ...

the test will include extra points (Mistakes Happen!)
- e.g., 52 or 53 points, but graded on a scale of 50

study advice:
- review online lecture notes (if not mentioned in class, won’t be on test)
- review text
- reference other sources for examples, different perspectives
- look over quizzes