Control and subprogram implementation

- control structures
  conditionals, loops, branches, …
- subprograms (procedures/functions/subroutines)
  subprogram linkage, parameter passing, implementation, …

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

Conditionals & loops

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

```plaintext
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:

```plaintext
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
- some argue for its use 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) {
    num = input.nextInt();
    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 nonlocals

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

Ada and C++ allow for default values for parameters
C++ & Java allow for optional parameters (specify with ...)

public static double average(double... values) {
  double sum = 0;
  for (double v : values) { sum += v; }   
  return sum / values.length;
}

System.out.println( average(3.2, 3.6) );
System.out.println( average(1, 2, 4, 5, 8) );

- if multiple parameters, optional parameter must be rightmost  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
Parameter passing (cont.)

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

```c
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 → ???

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

- 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

```
public void messWith(ArrayList<String> lst)
{
    lst.add("okay");
    . . .
    lst = new ArrayList();
}

ArrayList<String> words = new ArrayList<String>(5);
messWith(words);
```

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/methods with the same name

- overloaded functions/methods must have different parameters to distinguish

  ```java
  public double doStuff(String str) { ... }
  public double doStuff(int x) { ... } // OK since param type is different
  public int doStuff(String str) { ... } // not OK, since only return differs
  ```

in C++, can overload operators for new classes

  ```java
  bool Date::operator==(const Date & d1, const Date & d2) {
    return (d1.day == d2.day &&
            d1.month == d2.month &&
            d1.year == d2.year);
  }
  ```

  - overloaded operators are NOT allowed in Java

## 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

  ```text
  Activation Record
  ┌───────────┐
  │ local variables │
  │ parameters │
  │ static link │
  │ dynamic link │
  │ return address │
  └───────────┘
  ```

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

```pascal
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?

```pascal
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

naive 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?

Wednesday: 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