Programming Languages: Control Flow

Onur Tolga Şehitoğlu

Computer Engineering, METU

3 April 2008
1. Control Flow

2. Jumps

3. Escapes

4. Exceptions
Control Flow

- Usual control flow: a command followed by the other. Executed in sequence. **single entrance - single exit**
- Commands to change control flow and transfer execution to another point: **sequencers**
  - Jumps
  - Escapes
  - Exceptions
Jumps transfer control to a point in the code. The destination is marked with labels.

When jumps to arbitrary positions are possible:

```java
L1:  x++;  
     if (x>10) goto L2;  
     j++;  
     for (i=0; i<j; j++) {
        x=x*2;  
     }  
L2:  if (x>1000) goto L3;  
     else goto L1;  
L3:  printf("out\n");
```

Called spaghetti coding
- Unrestricted jumps ⇒ spaghetti coding.

- Dream of a PL where labels are first order values. 😊

- Further problems. Which jumps have problems?:

```plaintext
L1: ....
    goto L2; ①
    ....
    for (i=0; i<10; i++) {
        int x=t;
    }
L2: ....
    goto L1; ②
    ...
    goto L2: ③
```

- Lifetime and values of local variables? Values of index variables?

- C: Labels are local to enclosing block. No jumps allowed into the block. Newer languages avoid jumps.

- Single entrance multiple exit is still desirable. → escapes
Escapes

- Restricted jumps to out of textually enclosing block(s)
- Depending on which enclosing block to jump out of:
  - loop: `break` sequencer.
  - loops: `exit` sequencer.
  - function: `return` sequencer.
  - program: `halt` sequencer.
- **break sequencer** in C, C++, Java terminates the innermost enclosing loop block.

- **continue** in C, C++ stays in the same block but ends current iteration.

- **exit sequencer** in Ada or labeled break in Java can terminate multiple levels of blocks by specifying labels. Java code:

```java
L1: for (i=0; i<10; i++) {
    for (j=i; j<i; j++) {
        if (...) break;
        else if (...) continue;
        else if (...) break L1;
        else if (...) continue L1;
        s+=i*j;
    }
}
```
- **return sequencer** exist in most languages for terminating the innermost function block.

- **halt sequencer** either provided by operating system or PL terminates the program.

Consider jump inside of a block or jump out of a block for the function case:

```c
int f(int n) {
    int a;

    L1: if (n<0) goto L2; ①
        else if (n=1) return 1;
        else return f(n-1)*n;
}
int main() {  
    ...  
    f(12);
L2: ....
    goto L1: ②  
}
```
- Jump out of a function block, jump inside of a function block
- Activation record, run-time stack? Possible only for one direction if stack position can be recovered.
- **Non-local jumps**
- unexpected error occurring inside of many levels of recursion. Jump to the outer-most related caller function. **Exceptions**
Exceptions

- Controlled jumps out of multiple levels of function calls to an outer control point (handler or catch)
- C does not have exceptions but non-local jumps possible via setjmp(), longjmp() library calls.
- C++ and Java: try {...} catch(...) {...}
- Each try-catch block introduces a non-local jump point. try block is executed and whenever a throw expr command is called in any functions called (even indirectly) inside try block execution jumps to the catch() part.
- try-catch blocks can be nested. Execution jumps to closes catch block with a matching type in the parameters with the thrown expression.
Conventional error handling. Propagate errors with return values.

```c
... int searchopen(char *f) { ... /* if search fails error occurs here */ return -5; ... }
int openparse(char *f) { ... if ((r = searchopen(f))<0) return r; else ...
} int main() { ... if ((rv=openparse("file.txt"))<0) { /*handle error here */ ... }
```
Error handling with try–catch. (based on run-time stack)

```c
enum Exception { NOTFOUND, ..., PERMS};
void searchopen(char *f) { ...
    /* if open fails error occurs here */
    throw PERMS;
...}
void openparse(char *f) { ...
    searchopen(f);
    ...
}
int main() { ...
    try {...
        openparse("file.txt");
        ...
    } catch(Exception e) {
        /*handle error here */
    }
    ...
}
```
Nested exceptions are handled based on types. C++:

```c++
int main() { ... try { C1; f(); C2 } catch (double a) { ... } }
void f() { ...; try { ...; g(); ... } catch (int a) { ... } }
void g() { ...; throw 4; ...; throw 1.5; ... }
```

In case no handlers found a runtime error generated. Program halts.