CENG 242 - Chapter 9: Control Flow

Burak Metehan Tunçel - May 2022

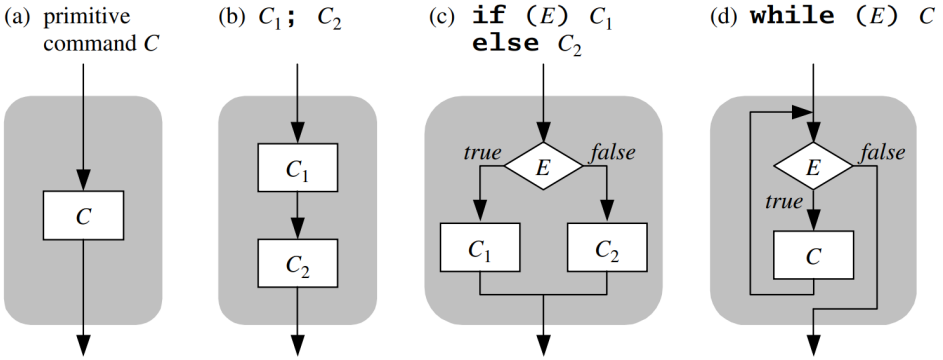

Figure 1: Flowcharts of primitive, sequential, if-, and while-commands.

1 Sequencers

Figure 1 shows four flowcharts: a *simple command*, a *sequential subcommand*, an *if-command*, and a *while-command*. Each of these flowcharts has a single entry and a single exit. The same is true for other conditional commands, such as case commands, and other iterative commands, such as for-commands. It follows that *any* command formed by composing simple, sequential, conditional, and iterative commands has a single-entry single-exit control flow.

Sometimes we need to implement more general control flows. In particular, single-entry multi-exit control flows are often desirable.

A ***sequencer*** is a construct that transfers control to some other point in the program, which is called the sequencer's ***destination***. Using sequencers we can implement a variety of control flows, with multiple entries and/or multiple exits. We shall examine several kinds of sequencers:

- jumps
- escapes
- exceptions

This order of presentation follows the trend in language design from low-level sequencers (jumps) towards higher-level sequencers (escapes and exceptions).

Some kinds of sequencers are able to ***carry*** values. Such values are computed at the place where the sequencer is executed, and are available for use at the sequencer's destination.

2 Jumps

A ***jump*** is a sequencer that transfers control to a specified program point. A jump typically has the form "`goto` L`;`", and this transfers control directly to the program point denoted by L, which is a ***label***.

```
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");
```

Code 1

Unrestricted jumps allow any command to have multiple entries and multiple exits. They tend to give rise to "spaghetti" code, so called because its flowchart is tangled. Also, "spaghetti" code tend to be hard to understand and is prone to errors.

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

Code 2

Also, when we consider the jumps, lifetime and values of local variables and values of index variables become a problem.

In `C`, labels are local to enclosing block. No jumps allowed into the block or out the block. Newer languages avoid jumps.

Jump is avoided but single entrance multiple exit is still desirable. It can be satisfied by **escapes**.

3 Escapes

An ***escape*** is a sequencer that terminates execution of a textually enclosing command or procedure. In terms of a flowchart, the destination of an escape is always the exit point of an enclosing subchart. With escapes we can program *single-entry multi-exit control flows*.

Depending on which enclosing block to jump out of:

- loop: break sequencer.
- loops: exit sequencer.
- function: return sequencer.
- program: halt sequencer.

3.1 Break Sequencer

break sequencer in C, C++, Java terminates the innermost enclosing loop block.

The break sequencer of `C`, `C++`, and `JAVA` allows any composite command (typically a loop or switch command) to be terminated immediately.

- 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:

```
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;
      }
    }
```

Code 3

- 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:

```
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
    }
```

Code 4: Jump out of a function block, jump inside of a function block

Jumps update current instruction pointer. But what about environment, activation record (run-time stack)? Jumping outside or inside a function block is possible only for one direction if stack position can be recovered. Called *non-local jumps*.

Non-local jumps can be useful in unexpected error occuring inside of many levels of recursion in order to jump to the outer-most related caller function. Instead of jumping, it can be satisfied by **exceptions**.

4 Exceptions

An ***abnormal situation*** is one in which a program cannot continue normally. Typical examples are the situations that arise when an arithmetic operation overflows, or an input/output operation cannot be completed.

What should happen when such an abnormal situation arises? Too often, the program simply halts with a diagnostic message. It is much better if the program transfers control to a *handler*, a piece of code that enables the program to recover from the situation. A program that recovers reasonably from such situations is said to be *robust*.

Exceptions are a superior technique for handling abnormal situations. An ***exception*** is an entity that represents an abnormal situation (or a family of abnormal situations). Any code that detects an abnormal situation can ***throw*** (or *raise*) an appropriate exception. That exception may subsequently be ***caught*** in another part of the program, where a construct called an ***exception handler*** (or *just handler*) recovers from the abnormal situation.

Every exception can be caught and handled, and the programmer has complete control over where and how each exception is handled. Exceptions cannot be ignored: if an exception is thrown, the program will halt unless it catches the exception.

The first major programming language with a general form of exception handling was `PL/I`. Then, a better exception concept has been designed into more modern languages such as `C++` and `JAVA`.

A handler for any exception may be attached to any command. If that command (or any procedure called by it) throws the exception, execution of that command is terminated and control is transferred to the handler. The command that threw the exception is never resumed.

We can attach different handlers for the same exception to different commands in the program. We can also attach handlers for several different exceptions to the same command.

Note the following important properties of exceptions:

- If a subcommand throws an exception, the enclosing command also throws that exception, unless it is an exception-handling command able to catch that particular exception. If a procedure's body throws an exception, the corresponding procedure call also throws that exception.
- A command that throws an exception is terminated abruptly (and will never be resumed).
- Certain exceptions are built-in, and may be thrown by built-in operations. Examples are arithmetic overflow and out-of-range array indexing.
- Further exceptions can be declared by the programmer, and can be thrown explicitly when the program itself detects an abnormal situation.

`C++` and `JAVA`, being object-oriented languages, treat exceptions as objects. `JAVA`, for instance, has a built-in `Exception` class, and every exception is an object of a subclass of `Exception`. Each subclass of `Exception` represents a different abnormal situation. An exception object contains an explanatory message (and possibly other values), which will be carried to the handler. Being first-class values, exceptions can be stored and passed as parameters as well as being thrown and caught.

The `JAVA` sequencer "`throw` E`;`" throws the exception yielded by expression E. The `JAVA` exception-handling command has the form:

```
try { C0; }
catch { (T1 I1) C1; }
...
catch { (Tn In) Cn; }
finally { Cf; }
```

this is able to catch any exception of class T_1 or ... or T_n. If the subcommand C_0 throws an exception of type T_i, then the exception handler C_i is executed with the identifier I_i bound to that exception. Just before the exception-handling command terminates (whether normally or abruptly), the subcommand C_f is executed unconditionally. (The `finally` clause is optional.)

4.1 From Lecture Slides

4.1.1 Exception

Exceptions are 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. However, `C++` and `Java` has "`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.

```
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 */
}
```

Code 5

Error handling with `try-catch`. (based on run-time stack)

```
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 */
  } ...
}
```

Code 6

Nested exceptions are handled based on types. C++:

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

Code 7

In case no handlers found a run time error generated. Program halts.

4.1.2 Co-routines

- ***Sequential flow:*** local jumps, subroutine calls, exceptions
- ***Concurrent flow:*** multiple contexes (stack and instruction pointer). Execution switches between them.
- ***Multiple uses:*** callbacks, generators (iterators), threads, fibers, asynchronous, event based, or concurrent programming
- Non-local jumps to different environments guided coordinated programs or a global scheduling mechanism: