Learn Four More Kinds of C Statements

We'll learn about statements for* ...  
◦ conditional decomposition:  
  if and switch;  
◦ iterative decomposition:  
  for, while, and do/while;  
◦ iteration control:  
  continue, and break; and  
◦ function control:  
  return.  

*Minor review: if and for covered in ECE120.

Statements Can Introduce Conditions

Simple statements in C can introduce conditional execution.  
Based on an expression, the computer executes one of two statements.

C’s if Statement Enables Conditional Execution

Conditional execution uses the if statement:  
if ( <expression> ) {  
  /* <expression> != 0:  
  execute "then" block */  
} else {  
  /* <expression> == 0:  
  execute "else" block */  
}  
<expression> can be replaced with any expression, and “else { ... }” can be omitted.
Examples of the *if* Statement

For example,

/* Calculate inverse of number. */
if (0 != number) {
    inverse = 1 / number;
} else {
    printf("Error!\n");
}

Examples of the *if* Statement

Or,

/* Limit size to 42. */
if (42 < size) {
    printf("Size set to 42.\n");
    size = 42;
}

switch Specifies Code Based on Expression Values

What if we have more than two choices?

For example, an operation in a simple calculator: +, −, ×, or ÷ (divide).

One answer: use
- a sequence of conditional constructs, or
- nested conditionals.

Another answer: if *choice based on values of an expression*, use a *switch* statement.

A Flow Chart with Multiple Choices

For example...

```
+     evaluate operator (as ASCII)     /
|     +     -     *     /          |
|     |     |     |     |          |
|     |     |     |     |          |
|     |     |     |     |          |
|     |     |     |     |          |
|     |     |     |     |          |
```

*add* *subtract* *multiply* *divide*
Multiple Choices Implemented with **switch**

In C, we write

```c
switch (operator) {
    case '+':     // add
        break;
    case '-':     // subtract
        break;
    case '*':     // multiply
        break;
    case '/':     // divide
        break;
}
```

**Constant Values, Break after Each Block of Code**

Switch allows any expression, but **values must be constant**. Normally, **use break at end of each case**.

- **No break** means keep going, such as when two values require the same code.
- **case 1**:  
  ```c
  case 1:
  // code for both 1 and 2
  break;
  ```
- **case 2**:  
  ```c
  case 2:
  // both cases execute this code!
  break;
  ```

**Pitfall: Be Sure Others Know Your Intent**

Leaving out **break** is usually an error.

- **case 1**:  
  ```c
  case 1:
  // do this first
  // code continues with next case
  break;
  ```
- **case 2**:  
  ```c
  case 2:
  // both cases execute this code!
  break;
  ```

People may “fix” the code. **Always comment!**

**Use `default` to Catch All Remaining Values**

```c
switch (<expression>) {
    case <value1>:
        break;
    ...
    default:
        // code for other values
        break;
}
```

**default** catches any other values (and should be the last case)
Simple Statements Can Also Be Iterations

Simple statements can also describe **iterative** execution. This type of execution repeats a statement until a test evaluates to false (0).

---

C's `for` Loop Enables Iterative Execution

The following is called a `for` loop:

```c
for (<init>; <test>; <update>) {
    /* loop body */
}
```

As shown on the previous slide, the computer:
1. Evaluates `<init>`.
2. Evaluates `<test>`, and stops if it is false (0).
3. Executes the **<body>**.
4. Evaluates `<update>` and returns to Step 2.

---

Iterations are Used for Repeated Behavior

/* Print multiples of 42 from 1 to 1000. */
int N;
for (N = 1; 1000 >= N; N = N + 1) {
    if (0 == (N % 42)) {
        printf("%d\n", N);
    }
}

---

Let's See How This Loop Works

/* Print 20 Fibonacci numbers. */
int A = 1; int B = 1; int C; int D;
for (D = 0; 20 > D; D = D + 1) {
    printf("%d\n", A);
    C = A + B;
    A = B;
    B = C;
}
Another Iterative Construct: the `while` Loop

A `while` loop
- only specifies a `<test>`
  and a loop body, but is
- otherwise equivalent to a `for` loop.

```java
while (<test>) {
    /* loop body */
}
```

Easy to Map `while` Loop into `for` Loop

```java
while (<test>) {
    /* loop body */
}
```
is completely equivalent to (with empty `<init>` and `<update>`):
```java
for ( ; <test> ; ) {
    /* loop body */
}
```

Execution of a `while` Loop

How does the computer execute a `while` loop?
```java
while (<test>) {
    /* loop body */
}
```

We can simplify the rules for a `for` loop...
1. Evaluates `<init>`. **Skip this step.**
2. Evaluates `<test>`, and stops if it is false (0).
3. Executes the loop body.
4. Evaluates `<update>` and returns to Step 2. **Skip this part.**

`while` Loop Performs the Iterative Decomposition

The `while` loop is identical to the iterative decomposition.

```java
while (<test>) {
    /* body */
}
```
Use **do/while** to Skip the First Test

What if we want to skip the first test?

```
while (<test>) {
    /* body */
} do {
    /* body */
} while (<test>);
```

**continue/break** Apply to Innermost Iteration

C supports two statements for iteration control:

- **continue**: current iteration is done, so
- **break**: current iterative construct is done, so

These apply to the innermost loop (or switch).

Where Do **continue/break** Go ...

... from switch?

... from extra2?
Where Do \textbf{continue} / \textbf{break} Go ...

\textbf{... from extra1?}

\textbf{continue} Goes to Test in \textbf{while} and \textbf{do} / \textbf{while}

Remember that
\begin{itemize}
  \item \textbf{while} and \textbf{do} / \textbf{while} are like \textbf{for} loops
  \item with no initialization nor update expressions, so
  \item \textbf{continue} goes to the test.
\end{itemize}

\textbf{return} Ends the Current Function (with a Return Value)

The \textbf{return} statement
\begin{itemize}
  \item provides a value (an expression) to be
  \item returned from the current function, and
  \item terminates function IMMEDIATELY.
\end{itemize}

In other words, in LC-3,
\begin{itemize}
  \item calculate the expression’s value
  \item copy the result into the return value slot
  \item tear down the stack frame
  \item RET
\end{itemize}