Control structures in C

Lecture Topics
- Conditional constructs
- Iterative constructs
- Examples
- Style

Lecture materials
Textbook § 13.3-13.5

Homework
None

Machine problem
MP1.1 due February 2 at 5pm submitted electronically

MP1.2 due February 17 at 5pm submitted electronically
Conditional constructs

- In C, conditional constructs can be implemented using *if*, *if-else*, or *switch* statements
- In the last lecture we covered *if* and *if-else* constructs; we will now look at the *switch* statement

**switch statement**

- consider example shown in the left column; it also can be implemented as shown on the right:

<table>
<thead>
<tr>
<th>Using cascaded <em>if-else</em> statements</th>
<th>Using <em>switch</em> statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>if</em> (expression == const1)</td>
<td><em>switch</em> (expression) {</td>
</tr>
<tr>
<td>action1;</td>
<td><em>case</em> const1:</td>
</tr>
<tr>
<td><em>else if</em> (expression == const2)</td>
<td>action1;</td>
</tr>
<tr>
<td>action2;</td>
<td><em>break</em>;</td>
</tr>
<tr>
<td><em>else if</em> (expression == const3)</td>
<td><em>case</em> const2:</td>
</tr>
<tr>
<td>action3;</td>
<td>action2;</td>
</tr>
<tr>
<td>...</td>
<td><em>break</em>;</td>
</tr>
<tr>
<td><em>else</em></td>
<td><em>case</em> const3:</td>
</tr>
<tr>
<td>actionN;</td>
<td>action3;</td>
</tr>
<tr>
<td></td>
<td><em>break</em>;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td><em>default</em>:</td>
</tr>
<tr>
<td></td>
<td>actionN;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

- this only works when we consider some discrete values to which *expression* is evaluated, const1, const2, ...
Iterative constructs

- Iterative construct means that some statements will be executed multiple times until some condition is met:

```
while (condition) {
    subtask;
}
```

- Such construct implements a loop structure in which action is executed multiple times, as long as some condition is true
  - action is also called loop body
- In C, iterative constructs can be implemented using while, do-while, or for loop statements

**while and do-while statements**

- while (condition) {
  subtask;
}

- do {
  subtask
} while (condition);

- For while loop, loop body may or may not be executed even once
- For do-while loop, loop body will be executed at least once

```
subtask
```
while | do-while
--- | ---
x = 0;
while (x < 10) {
    printf(“x=%d
”, x);
    x = x + 1;
}
x = 0;
do {
    printf(“x=%d
”, x);
    x = x + 1;
}while (x < 10);

for statement

- for (init; test; reinit) {
  subtask;
}

- Example

while | for
--- | ---
x = 0;
while (x < 10) {
    printf(“x=%d
”, x);
    x = x + 1;
}
for (x = 0; x < 10; x++)
printf(“x=%d
”, x);

break and continue

- break will cause the loop to be terminated
- continue will cause to skip the rest of code in the loop and start executing next loop iteration
Examples

Simple calculator

- **Problem statement:** write a program that lets user enter a simple expression consisting of two operands and one operation, e.g., ‘2 + 3’, performs the entered calculation, and prints the result.
- Using systematic decomposition, we first derive a flowchart that shows all the main steps in the program that need to be implemented
  - Get input (using scanf)
  - Recognize which operation is to be implemented (using switch construct)
  - Output results (using printf)

```c
#include <stdio.h> /* needed for printf and scanf */

int main()
{
    int operand1, operand2; /* two operands */
    char operation; /* operation to be performed */
    int result; /* result of the operation */
    /* simple calculator
        Input: an expression to be evaluated, for example, 4 / 6
        Output: value to which the expression evaluates,
                or an error message if the operation is not supported */
```
/* get input */
printf("Enter expression operand1 operation operand 2: ");
scanf("%d %c %d", &operand1, &operation, &operand2);

/* calculate expression */
switch (operation)
{
   case "+": result = operand1 + operand2; break;
   case "-": result = operand1 - operand2; break;
   case "/": result = operand1 / operand2; break;
   case "*": result = operand1 * operand2; break;
   default: printf("Invalid operation \%c\n", operation);
}

/* print result */
printf("result=%i\n", result);

return 0;
}

• Two problems with this implementation
  o What if user enters 10 / 0?
  o The program will still print out “result” even if the operator was not supported. How do we fix this?

Character counter
• Problem statement: read characters from the keyboard and convert them to lower case until '0' (sentinel) is entered

#include <stdio.h>    /* needed for printf and scanf */

int main()
{
   char inchar, outchar;
   scanf("%c", &inchar);
   while (inchar != '0')
   {
      if ((inchar >= 'A') && (inchar <+'Z'))
         outchar = ('a' - 'A') + inchar;
      else
         outchar = inchar;

      printf("%c\n", outchar);
      scanf("%c", &inchar);
   }
   return 0;
}

Riemann integral
• Problem statement: write a program to compute integral of a function f(x) on an interval [a,b].
- **Algorithm:** use integral definition as an area under a function \( f(x) \) on an interval \([a,b]\)

\[
\int_a^b f(x)\,dx = \lim_{n \to \infty} \sum_{i=0}^{n-1} f\left(a + \frac{b-a}{n} i\right) \frac{b-a}{n}
\]

- Using systematic decomposition, we first derive a flowchart

```c
#include <stdio.h>

int main()
{
    int n = 100;  /* hardcoded number of Reimann sum terms */
    float a = -1.0f;  /* hardcoded [a,b] */
    float b = 1.0f;
```
```c
float s = 0.0f; /* computed integral value */
int i; /* loop counter */
float x, y; /* x and y=f(x) */
float dx = (b - a) / n; /* width of rectangles */

for (i = 0; i < n; i++)
{
    x = a + dx * i;
    y = x * x + 2 * x + 3;
    s += y * dx;
}

printf("%f\n", s);
return 0;
```

**Style**

- Style is what separates a good program from not so good
- Once the program is written, a lot of time will be spent maintaining it, thus, it is important to make the maintenance task as simple as possible
  - Documentation
    - Program should be well-documented, it should start with an opening comment describing the purpose, input, output, authors, revision history, etc.
    - Each function must be documented as well
    - Variables should be documented
    - Code sections should be documented
  - Clarity
    - program should read like a technical paper
    - should be organized into sections based on functions implemented
    - code inside functions should be organized into paragraphs, each paragraph starting with a topic-specific comment and be separated from other paragraph by space
    - indentation should be used to identify code inside blocks or conditionals
    - variables should be named to have intuitive enough meaning
    - and so should be functions
  - Simplicity
    - The program should be made as simple and easy to understand as possible
    - Functions should be not extensively long
    - Avoid complex constructs, such as nested ifs
    - Statements should be short
- Refer to ECE 190 C Coding Conventions at [http://courses.engr.illinois.edu/ECE190/info/conventions.html](http://courses.engr.illinois.edu/ECE190/info/conventions.html)