Lecture 4: Introduction to C: Control Flow
How to solve problems via programming

• In this class we are learning how to solve problems by writing computer (in particular C) programs

• Even for simple problems such as:
  – “given two numbers, print the larger one”
  – “given a set of numbers, compute their sum”

producing the program that achieves the given task requires some intermediate steps
“given two numbers, print the larger one”

To write a program that computes this task we need to answer questions such as:

- How will the input numbers given?
- How are we going to print the output?
  - These are the questions about the specification of the problem (this is called requirements specification)
- How will we compute the sum?
  - This is a question about how we can solve the problem using a computer
  - It is about algorithm design
The process suggested in the Textbook

1. Problem statement
   • start with clearly defining the problem to solve
2. Input/output description
   • treat solution like a “black box” – focus on what goes in and what comes out
3. Work the problem by hand on an example
   • two purposes: (a) verify you know exactly what to do; (b) have known results to check solution later
4. Algorithm development and implementation
   • includes designing the solution and implementing it
5. Testing
   • making sure solution works correctly
The process suggested in the Textbook

1. Problem statement
2. Input/output description
3. Work the problem by hand on an example
4. Algorithm development and implementation
5. Testing

- Steps 1, 2 and 3 are all called requirements analysis.
- Step 4 consists of software design and implementation.
- Step 5 is for verification and validation and it is necessary for catching and removing bugs in both our algorithm development and implementation.
Modern software development

• Software development for large and complex software projects requires more careful thinking about software development process

• Modern software development practices include
  – **Iterative (agile) software development**: A program is developed incrementally by starting from the most basic functionality and then gradually adding new functionality
  – **Test driven software development**: Before writing the program a set of test cases are identified and the program development is guided by tests
  – **Object oriented design**: Designing software in a way that encapsulates the data and the computation together
What is algorithm?

• To solve a programming problem we first need to develop an algorithm.
  – But, what is an algorithm?
• An algorithm is like a recipe
  – A recipe is a step by step description of how to convert some ingredients to a dish you can eat
  – An algorithm is a step by step description of how to convert a given input to a computer to an output
• This is a very hard question to answer
What is an algorithm?

• An algorithm is like a recipe
  – A recipe is a step by step description of how to convert some ingredients to a dish you can eat
  – An algorithm is a step by step description of how to convert a given input to a computer to the desired output

• If you write a program that implements the algorithm, then the implementation (the program) will produce the desired output for the given inputs
How do you come up with an algorithm?

- This is a very hard question to answer

- Algorithm design is a creative activity and like any creative activity it cannot be converted to a step-by-step description
  - We do not have an algorithm for algorithm design!

- Computer scientists developed many algorithms for many general problems, so knowing them helps

- Computer scientists also developed some general strategies for algorithm design that can be used to solve different problems
  - One general strategy: Stepwise refinement
Stepwise refinement for software development

• Stepwise refinement is a divide and conquer approach
  1. Clearly state the intended task

  2. Divide the task to a set of subtasks and re-express the intended task as an equivalent structure of properly connected subtasks, each solving part of the problem

  3. Divide each subtasks far enough until the complexity of each subtask is manageable (i.e., you know how to write a program segment for that subtask)

• So, if we do algorithm development using stepwise refinement, then the resulting algorithm is a collection of subtasks
How do we write the algorithm we designed?

• We can do this in two basic ways:
  – We can write **pseudocode**
    • It is description of an algorithm that looks like a code but it is not in any programming language
    • It is written for humans and it is not machine readable
    • There is no standard way of writing pseudocode
  – We can use **flowcharts**
    • Flowcharts are an almost standard block diagram representation for computational tasks
Basic Operation | Pseudocode Notation | Flowchart Symbol
---|---|---
Input | read radius | read radius

Computation | set area to $\pi \cdot \text{radius}^2$ | area $= \pi \cdot \text{radius}^2$

Output | print radius, area | print radius, area

Comparisons | if radius < 0 then ... | if radius < 0 ?

Beginning of algorithm | main: | start main

End of algorithm | | stop main
How are the tasks in an algorithm combined?

- They can be combined as a sequence:
  - first execute task1, then execute task2
- They can be combined using selection:
  - check a condition, if the condition is true execute task1, if not execute task 2
- They can be combined using repetition (aka iteration):
  - repeat task1 until a condition becomes true

- Structured programming languages (like C) have language constructs for composing tasks using sequence, selection and repetition
  - We call these control structures since they control the execution order of the statements
C has 7 basic control structures

• **1**\(^{st}\) is trivial: sequence structure: statements listed sequentially are executed sequentially
  
  \[ a = 4; \]
  \[ b = 5; \]

• 3 choices of selection structures:
  
  - if
  - if/else
  - switch

• 3 choices of repetition structures:
  
  - while
  - for
  - do/while
Sequence in flowchart
if selection structure as a flowchart?
if statement syntax

Either

    if  (Boolean expression)
        statement;

Or

    if  (Boolean expression) {
        statement1;
        statement2;
        ...
    }

• **Boolean** (aka **conditional**) expression:
  – evaluates to 1 (true) or 0 (false)

• Note the indentation to help readability of the code
## Relational Operators

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>is less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>is less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>is greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>is greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>is equal to</td>
</tr>
<tr>
<td>!=</td>
<td>is not equal to</td>
</tr>
</tbody>
</table>
Simple Boolean expressions, relational operators

- Relational operators: <, >, <=, >=, ==, !=
- e.g., int x=1, y=2, z=3;
  x > y /* false */
  x >= z - y /* true */
  z != x + y /* false */

- **Note precedence:** relational operators have lower precedence than arithmetic operators
  x == z + y /* false */
- Note, not same as
  x = z + y /* makes x be 5 */
Boolean (aka logical) operators: &&, ||, !

- **Boolean (aka logical) operators** are used for combining simple Boolean expressions into more complex expressions
  - && means logical and
  - || means logical or
  - ! means logical negation
- Operands are Boolean expressions
  - e.g., grade == ‘A’ && weight > 10
- **Note precedence**: relational operators have higher precedence than Boolean operators
- Rules (see Table 3.1, p. 89, next page)
  - op1 && op2 - true if both operands are true
  - op1 || op2 - true if either operand is true
  - !op - true if operand is false
- **Note precedence**: ! then && then ||
## Boolean (logical) operators

<p>|   |   | A &amp;&amp; B | A || B | !A | !B |
|---|---|--------|--------|----|----|
| False | False | False | False | True | True |
| False | True  | False | True  | True | False |
| True  | False | False | True  | False | True |
| True  | True  | True  | True  | False | False |</p>
<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operation</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( )</td>
<td>Innermost first</td>
</tr>
<tr>
<td>2</td>
<td>++ -- + - ! (type)</td>
<td>Right to left (unary)</td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Left to right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td>Left to right</td>
</tr>
<tr>
<td>5</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>Left to right</td>
</tr>
<tr>
<td>6</td>
<td>== !=</td>
<td>Left to right</td>
</tr>
<tr>
<td>7</td>
<td>&amp;&amp;</td>
<td>Left to right</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>= += -= *= /= %=</td>
<td>Right to left</td>
</tr>
</tbody>
</table>
if/else Selection Structure
if/else

if  (Boolean expression) {
    statements;
} else {
    statements;
}

• Example:
  if (grade >= 60)
      printf("%s", "Pass");
  else
      printf("%s", "Fail");

  See .../demos/richter.c
Conditional operator

- Conditional operator can be used in place of simple if then else statements
- Example:
  ```c
  if (a < b)
    count++;
  else
    c = a+b;
  ```
- equivalent code with **conditional operator**
  ```c
  a<b ? count++ : c = a + b;
  ```
- Conditional operator does not allow block or nesting but returns a value

  ```c
  printf("%s",(grade>=60 ? "Pass" : "Fail"));
  ```