ICS103 Programming in C

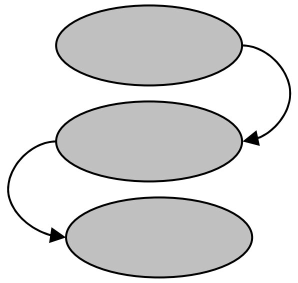
Lecture 6: Selection Structures

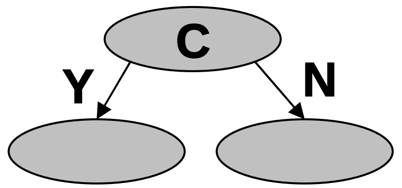
Outline

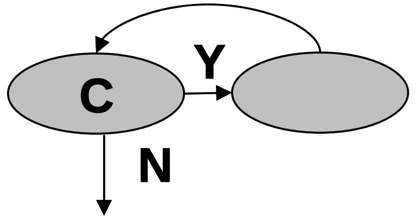
- Control Structures
- Conditions
  - Relational Operators
  - Logical Operators
- if statements
  - Two-Alternatives
  - One-Alternative
  - Nested If Statements
- switch Statement

Control Structures

- **Control structures** –control the flow of execution in a program or function.
- Three basic control structures:
  - **Sequential Flow** - this is written as a group of statements bracketed by { and }where one statement follows another.
  - **Selection control structure** - this chooses between multiple statements to execute based on some condition.
  - **Repetition –** this structure executes a block of code multiple times.







Compound Statements

- A **Compound statement** or a Code Block is written as a group of statements bracketed by { and } and is used to specify **sequential** flow.

```
{
    Statement_1;
    Statement_2;
    Statement_3;
}
```

  - Example: the `main` function is surrounded by {}, and its statements are executed sequentially.
  - Function body also uses compound statement.

Conditions

- A program chooses among alternative statements by testing the values of variables.
  - 0 means false
  - Any non-zero integer means true, Usually, we’ll use 1 as true.

```
if (a>=b)
     printf(“a is larger”);
else
     printf(“b is larger”);
```

- **Condition** - an expression that establishes a criterion for either executing or skipping a group of statements
  - `a>=b` is a condition that determines which `printf` statement we execute.

Relational and Equality Operators

- Most conditions that we use to perform comparisons will have one of these forms:
  - variable *relational*-*operator* variable e.g. a < b
  - variable *relational*-*operator* constant e.g. a > 3
  - variable *equality*-*operator* variable e.g. a == b
  - variable *equality*-*operator* constant e.g. a != 10

Relational and Equality Operators

Operator	**Meaning**	**Type**
<	less than	relational
>	greater than	relational
<=	less than or equal to	relational
>=	greater than or equal to	relational
==	equal to	equality
!=	not equal to	equality

Logical Operators

- **logical expressions** - expressions that use conditional statements and logical operators.
  - && (and)
    - ✓ A && B is true if and only if both A and B are true
  - || (or)
    - ✓ A || B is true if either A or B are true
  - ! (not)
    - ✓ !(condition) is true if condition is false, and false if condition is true
    - ✓ This is called the **logical complement** or **negation**
- Example
  - `(salary < 10000) || (dependents > 5)`
  - `(temperature > 90.0) && (humidity > 90)`
  - `!(temperature > 90.0)`

Truth Table && Operator

A	B	A && B
False (zero)	**False** (zero)	**False** (zero)
False (zero)	**True** (non-zero)	**False** (zero)
True (non-zero)	**False** (zero)	**False** (zero)
True (non-zero)	**True** (non-zero)	**True** (non-zero)

Truth Table || Operator

A	B	A || B
False (zero)	**False** (zero)	**False** (zero)
False (zero)	**True** (non-zero)	**True** (non-zero)
True (non-zero)	**False** (zero)	**True** (non-zero)
True (non-zero)	**True** (non-zero)	**True** (non-zero)

Operator Table ! Operator

A	!A
False (zero)	**True** (non-zero)
True (non-zero)	**False** (zero)

Remember!

- && operator yields a true result only when both its operands are true.
- || operator yields a false result only when both its operands are false.

Operator Precedence

- Operator’s precedence determine the order of execution. Use parenthesis to clarify the meaning of expression.
- Relational operator has higher precedence than the logical operators.
- Ex: followings are different.

```
 (x<y || x<z) && (x>0.0)
  x<y || x<z && x>0.0
```

function calls

! + - & (unary operations)

*, /, %

+, -

<, >, <=, >=

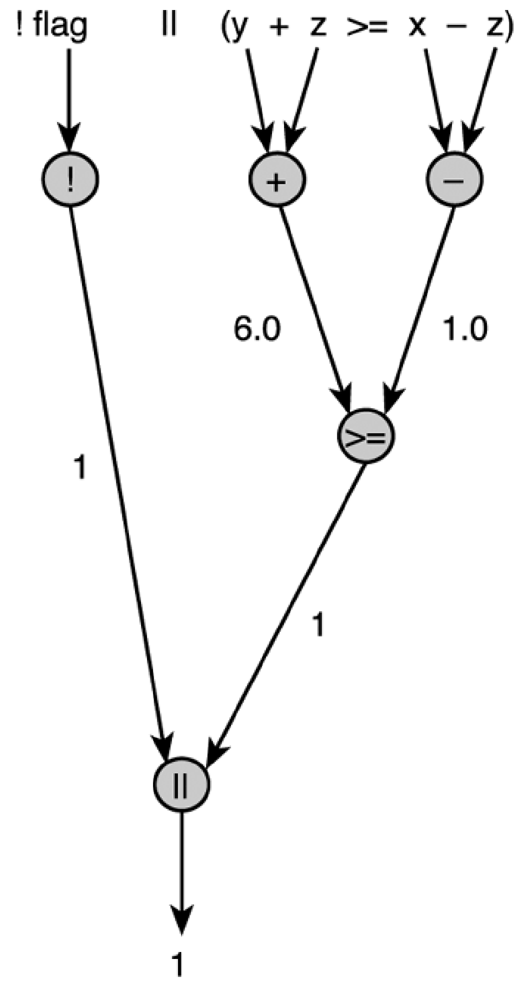
==, !=

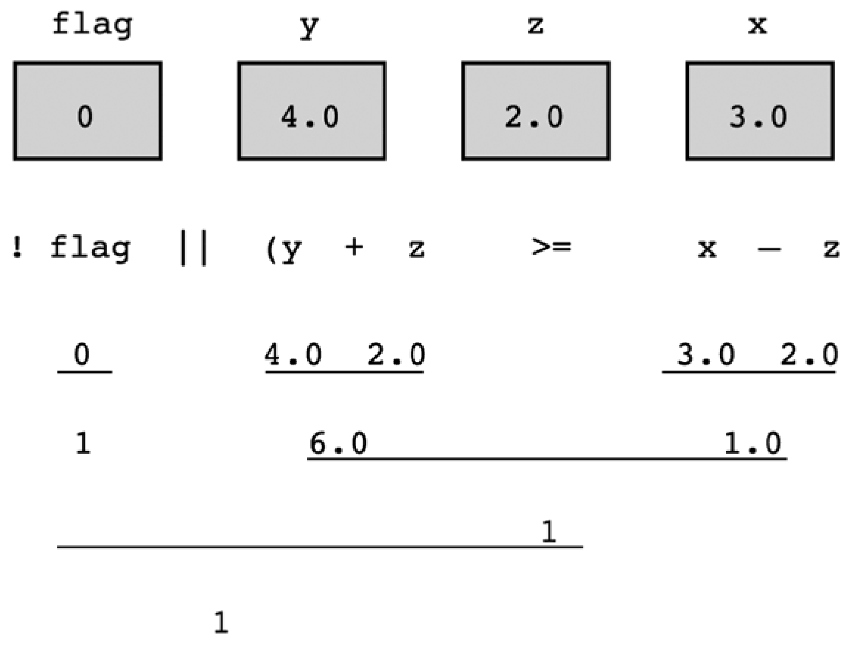
&&

||

=

Evaluation Tree and Step-by-Step Evaluation for !flag || (y + z >= x - z)





Writing English Conditions in C

- Make sure your C condition is logically equivalent to the English statement.
  - "x and y are greater than z"

    `(x > z) && (y > z)` (valid)

    `x && y > z` (invalid)

Character Comparison

- C allows character comparison using relational and equality operators.
- During comparison Lexicographic (alphabetical) order is followed. (See Appendix A for a complete list of ASCII values).

```
‘9’ >= ‘0’  // True
‘a’ < ‘e’   // True
‘a’ <= ch && ch <= ‘z’ /* True if ch is a char type
                          variable that contains a
                          lower case letter.*/
```

Logical Assignment

- You can assign an `int` type variable a non zero value for true or zero for false.

```
 Ex: even = (n%2 == 0)
         if (even) { do something }
```

- Some people prefer following for better readability.

```
#define FALSE 0
#define TRUE !FALSE
 even = (n%2 == 0)
     if (even == TRUE) { do something }
```

Complementing a condition

- We can complement a logical expression by preceding it with the symbol !.
- We can also complement a single condition by changing its operator.
  - Example : The complements of (`age == 50`) are `!(age == 50) , (age != 50)`
  - The relational operator should change as follows
    <= to >, < to >= and so on

DeMorgan’s Theorem

- DeMorgan’s theorem gives us a way of simplifying logical expressions.
- The theorem states that the complement of a conjunction is the disjunction of the complements or vice versa. In C, the two theorems are

```
1. !(x || y) == !x && !y
2. !(x && y) == !x || !y
```

  Example: If it is not the case that I am tall and thin, then I am either short or fat (or both)

- The theorem can be extended to combinations of more than two terms in the obvious way.

DeMorgan’s Theorem …

- DeMorgan’s Theorems are extremely useful in simplifying expressions in which an AND/OR of variables is inverted.
- A C programmer may use this to re-write

  ```
  if (!a && !b) ...
  ```

  as

  ```
  if(!(a || b) ...
  ```

  Thus saving one operator per statement.
- Good, optimizing compiler should do the same automatically and allow the programmer to use whatever form seemed clear to them.

if statement : Two alternatives

```
if (condition)
  {compound_statement_1 } // if condition is true
else
  { compound_statement_2 } // if condition is false
```

Example:

```
if (crash_test_rating_index <= MAX_SAFE_CTRI) {
  printf("Car #%d: safe\n", auto_id);
  safe = safe + 1;
}
else {
  printf("Car #%d: unsafe\n", auto_id);
  unsafe = unsafe + 1;
}
```

if statement : Two alternatives…

- When the symbol { follows a condition or else, the C complier either executes or skips all statements through the matching }
- In the example of the previous slide, if you omit the braces enclosing the compound statements, the if statement would end after the first `printf` call.
- The `safe = safe + 1;` statement would always be executed.
- You MUST use braces if you want to execute a compound statement in an `if` statement.
- To be safe, you may want to always use braces, even if there is only a single statement.

No {}?

```
if (rest_heart_rate > 56)
  printf("Keep up your exercise program!\n");
else
  printf("Your heart is in excellent health!\n");
```

- If there is only one statement between the {} braces, you can omit the braces.

One Alternative?

- You can also write the `if` statement with a single alternative that executes only when the condition is true.

```
if ( a <= b )
   statement_1;
```

Nested if Statements

- So far we have used if statements to code decisions with one or two alternatives.
- A compound statement may contain more if statements.
- In this section we use nested if statements (one if statement inside another) to code decisions with multiple alternatives.

```
if (x > 0)
  num_pos = num_pos + 1;
else
  if (x < 0)
     num_neg = num_neg + 1;
  else
     num_zero = num_zero + 1;
```

Comparison of Nested if and Sequences of ifs

- Beginning programmers sometime prefer to use a sequence of if statements rather than a single nested if statement

```
if (x > 0)
  num_pos = num_pos + 1;
if (x < 0)
  num_neg = num_neg + 1;
if (x == 0)
  num_zero = num_zero +1;
```

- This is less efficient because all three of the conditions are always tested.
- In the nested if statement, only the first condition is tested when `x` is positive.

Multiple-Alternative Decision Form of Nested if

- Nested if statements can become quite complex. If there are more than three alternatives and indentation is not consistent, it may be difficult for you to determine the logical structure of the if statement.
- You can code the nested if as the multiple-alternative decision described below:

```
if ( condition_1 )
   statement_1
else if ( condition_2 )
   statement_2
   .
   .
   .
else if ( condition_n )
   statement_n
else
   statement_e
```

Example

- Given a person’s salary, we want to calculate the tax due by adding the base tax to the product of the percentage times the excess salary over the minimum salary for that range.

Salary Range	**Base tax**	**Percentage of Excess**
0.00 – 14,999.99	0.00	15
15,000.00 – 29,999.99	2,250.00	18
30,000.00 – 49,999.99	5,400.00	22
50,000.00 – 79,999.99	11,000,00	27
80,000.00 – 150,000.00	21,600.00	33

```
if ( salary < 0.0 )
  tax = -1.0;
else if ( salary < 15000.00 )
  tax = 0.15 * salary;
else if ( salary < 30000.00 )
  tax = (salary – 15000.00)*0.18 + 2250.00;
else if ( salary < 50000.00)
  tax = (salary – 30000.00)*0.22 + 5400.00;
else if ( salary < 80000.00)
  tax = (salary – 50000.00)*0.27 + 11000.00;
else if ( salary <= 150000.00)
  tax = (salary – 80000.00)*0.33 + 21600.00;
else
  tax = -1.0;
```

Order of Conditions in a Multiple-Alternative Decision

- When more than one condition in a multiple-alternative decision is true, only the task following the first true condition executes.
- Therefore, the order of the conditions can affect the outcome.
- The order of conditions can also have an effect on program efficiency.
- If we know that salary range 30,000 - 49,999 are much more likely than the others, it would be more efficient to test first for that salary range. For example,

```
 if ((salary>30,000.00) && (salary<=49,999.00))
```

Nested if Statements with More Than One Variable

- In most of our examples, we have used nested if statements to test the value of a single variable.
- Consequently, we have been able to write each nested if statement as a multiple-alternative decision.
- If several variables are involved in the decision, we cannot always use a multiple-alternative decision.
- The next example contains a situation in which we can use a nested if statement as a ”filter” to select data that satisfies several different criteria.

Example

- The Department of Defense would like a program that identifies single males between the ages of 18 and 26, inclusive.
- One way to do this is to use a nested if statement whose conditions test the next criterion only if all previous criteria tested were satisfied.
- Another way would be to combine all of the tests into a single logical expression
- In the next nested if statement, the call to `printf` executes only when all conditions are true.

Example

```
/* Print a message if all criteria are met.*/
if ( marital_status == ’S’ )
   if ( gender == ’M’ )
      if ( age >= 18 && age <= 26 )
         printf("All criteria are met.\n");
```

- or we could use an equivalent statement that uses a single if with a **compound condition**:

```
/* Print a message if all criteria are met.*/
if ((maritial_status == ’S’) && (gender == ’M’) &&
  (age >= 18 && age <= 26))
   printf("All criteria are met.\n");
```

Common if statement errors

```
if crsr_or_frgt == ’C’
  printf("Cruiser\n");
```

- This error is that there are no ( ) around the condition, and this is a syntax error.

```
if (crsr_or_frgt == ’C’);
  printf("Cruiser\n");
```

- This error is that there is a semicolon after the condition. C will interpret this as there is nothing to do if the condition is true.

If Statement Style

- All if statement examples in this lecture have the true statements and false statements indented. Indentation helps the reader but conveys no meaning to the compiler.
- The word `else` is typed without indentation on a separate line.
- This formatting of the if statement makes its meaning more apparent and is used solely to improve program readability.

Switch statements

- The switch statement is a better way of writing a program when a series of if-else if occurs.
- The switch statement selects one of several alternatives.
- The switch statement is especially useful when the selection is based on the value of a single variable or of a simple expression
  - This is called the controlling expression
- In C, the value of this expression may be of type `int` or `char`, but not of type `double`.

Example of a switch Statement with Type char Case Labels

```
/* Determines the class of Ship given its class ID */
#include <stdio.h>
int main(void) {
   char classID;

   printf("Enter class id [a, b, c or d]: ");
   scanf("%c", &classID);

   switch (classID) {
    case 'B':
    case 'b':
        printf("Battleship\n");
        break;
    case 'C':
    case 'c':
        printf("Cruiser\n");
        break;
    case 'D':
    case 'd':
        printf("Destroyer\n");
        break;
    case 'F':
    case 'f':
        printf("Frigate\n");
        break;
    default:
        printf("Unknown ship class %c\n", classID);
   }
   return 0;
}
```

Explanation of Example

- This takes the value of the variable `class` and compares it to each of the cases in a top down approach.
- It stops after it finds the first case that is equal to the value of the variable `class`.
- It then starts to execute each line of the code following the matching case till it finds a `break` statement or the end of the switch statement.
- If no case is equal to the value of `class`, then the default case is executed.
    - default case is optional. So if no other case is equal to the value of the controlling expression and there is a default case, then default case is executed. If there is no default case, then the entire switch body is skipped.

Remember !!!

- The statements following a case label may be one or more C statements, so you do not need to make multiple statements into a single compound statement using braces.
- You cannot use a string such as ”Cruiser” or ”Frigate” as a case label.
    - It is important to remember that type `int` and `char` values may be used as case labels, type `double` values cannot be used.
- Another *very* common error is the omission of the `break` statement at the end of one alternative.
    - In such a situation, execution ”falls through” into the next alternative.
- Forgetting the closing brace of the switch statement body is also easy to do.
- In the book it says that forgetting the last closing brace will make all following statements occur in the default case, but actually the code will not compile on most compilers.

Nested if versus switch

- Advantages of if:
  - It is more general than a switch
    - ✓ It can be a range of values such as `x < 100`
  - A switch can not compare doubles

- Advantages of switch:
  - A switch is more readable

- Use the switch whenever there are ten or fewer case labels

Common Programming Errors

- Consider the statement:
  if (0 <= x <= 4)
- This is always true!
  - First it does 0 <= x, which is true or false so it evaluates to 1 for true and 0 for false
  - Then it takes that value, 0 or 1, and does 1 <= 4 or 0 <= 4
  - Both are always true
- In order to check a range use (0 <= x && x <= 4).

- Consider the statement:
  if (x = 10)
- This is always true!
  - The = symbol assigns x the value of 10, so the conditional statement evaluates to 10
  - Since 10 is nonzero this is true.
  - You must use == for comparison

More Common Errors

- Don’t forget to parenthesize the condition.
- Don’t forget the { and } if they are needed
- C matches each else with the closest unmatched if, so be careful so that you get the correct pairings of if and else statements.
- In switch statements, make sure the controlling expression and case labels are of the same permitted type.
- Remember to include the default case for switch statements.
- Don’t forget your { and } for the switch statement.
- **Don’t forget your break statements!!!**