Computer Science I

Conditionals

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

1. Introduction

2. Conditionals

3. Logical Operators

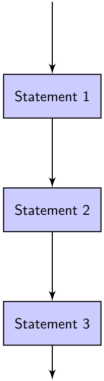
4. Pitfalls

5. Exercises

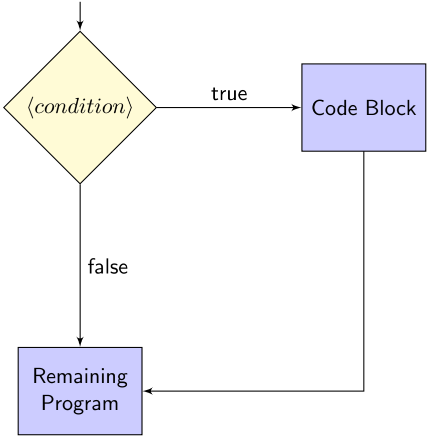
Part I: Introduction

Control Flow & Logical Operators

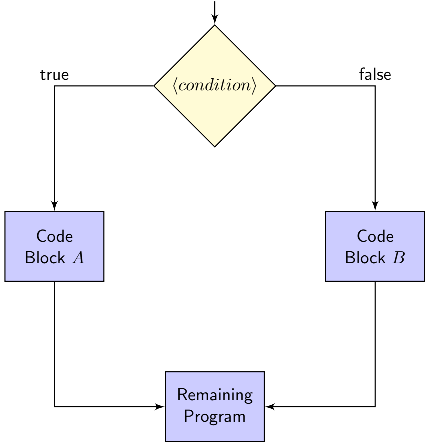
Sequential Control Flow



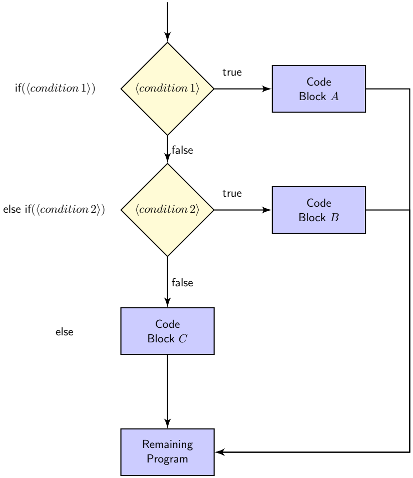
If Statement Flow

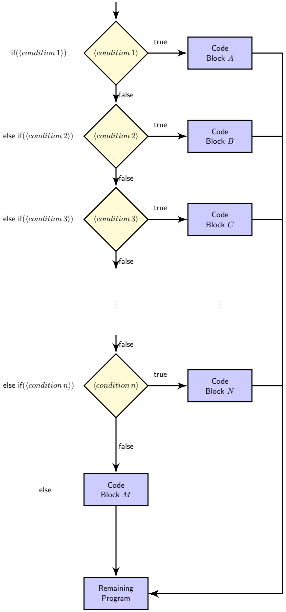


If-Else Statement Flow



If-Else-If Statement Flow





if($\langle\mathit{condition}\,1\rangle$)

$\langle\mathit{condition}\,1\rangle$

true

Code Block A

false

else if($\langle\mathit{condition}\,2\rangle$)

$\langle\mathit{condition}\,2\rangle$

true

Code Block B

false

else if($\langle\mathit{condition}\,3\rangle$)

$\langle\mathit{condition}\,3\rangle$

true

Code Block C

false

$\vdots$

$\vdots$

false

else if($\langle\mathit{condition}\,n\rangle$)

$\langle\mathit{condition}\,n\rangle$

true

Code Block N

false

else

Code Block M

Remaining Program

Boolean Statements

- A *boolean* condition or expression is a logical expression that evaluates to either *true* or *false*

Boolean Statements

- A *boolean* condition or expression is a logical expression that evaluates to either *true* or *false*
- May involve numerical comparisons $a \geq 0$

Boolean Statements

- A *boolean* condition or expression is a logical expression that evaluates to either *true* or *false*
- May involve numerical comparisons $a \geq 0$
- A condition can be *simple* or *complex*

Boolean Statements

- A *boolean* condition or expression is a logical expression that evaluates to either *true* or *false*
- May involve numerical comparisons $a \geq 0$
- A condition can be *simple* or *complex*
- May connect one or more expressions using a logical *and* or a logical *or*

Numeric Comparisons

- We need a way to compare the value stored in variables

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:
    - Are two values equal or not equal?

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:
  - Are two values equal or not equal?
  - Is one value greater than or equal to/lesser than or equal to another?

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:
  - Are two values equal or not equal?
  - Is one value greater than or equal to/lesser than or equal to another?
  - Is one value strictly greater/lesser than another?

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:
  - Are two values equal or not equal?
  - Is one value greater than or equal to/lesser than or equal to another?
  - Is one value strictly greater/lesser than another?
- Standard mathematical notations

$$= \quad \neq \quad \geq \quad \leq \quad > \quad <$$

Numeric Comparisons

- We need a way to compare the value stored in variables
- Compare the relative value of two variables
- Compare the value stored in one variable with a fixed value (literal)
- Comparisons:
  - Are two values equal or not equal?
  - Is one value greater than or equal to/lesser than or equal to another?
  - Is one value strictly greater/lesser than another?
- Standard mathematical notations

$$= \quad \neq \quad \geq \quad \leq \quad > \quad <$$

- Code versions:

```
==    !=    >=    <=    >    <
```

Logical And

A	B	A *and* B
false	false	false
false	true	false
true	false	false
true	true	true

Logical And

A	B	A *and* B
false	false	false
false	true	false
true	false	false
true	true	true

Code version: `&&`

Logical Or

A	B	A or B
false	false	false
false	true	true
true	false	true
true	true	true

Logical Or

A	B	A or B
false	false	false
false	true	true
true	false	true
true	true	true

Code version: `||`

Logical Negation

A	*not* A
false	true
true	false

Logical Negation

A	*not* A
false	true
true	false

Code version: `!`

Part II: Conditionals

If, If-Else, If-Else-If & Numeric Comparisons

If Statement

```
if(<condition>) {
  //conditional body: code inside this code block
  //will only execute if the <condition> evaluates
  //to true, otherwise it will not execute at all
}
```

- Uses the keyword `if`

If Statement

```
if(<condition>) {
  //conditional body: code inside this code block
  //will only execute if the <condition> evaluates
  //to true, otherwise it will not execute at all
}
```

- Uses the keyword `if`
- The condition is enclosed in parentheses

If Statement

```
if(<condition>) {
  //conditional body: code inside this code block
  //will only execute if the <condition> evaluates
  //to true, otherwise it will not execute at all
}
```

- Uses the keyword `if`
- The condition is enclosed in parentheses
- The code block begins and ends with curly brackets

If Statement

```
if(<condition>) {
  //conditional body: code inside this code block
  //will only execute if the <condition> evaluates
  //to true, otherwise it will not execute at all
}
```

- Uses the keyword `if`
- The condition is enclosed in parentheses
- The code block begins and ends with curly brackets
- Behavior

If-Else Statement

```
if(<condition>) {
  //code block A
} else {
  //code block B
}
```

- Uses the keyword `else`

If-Else Statement

```
if(<condition>) {
  //code block A
} else {
  //code block B
}
```

- Uses the keyword `else`
- Behavior: if `<condition>` evaluates to true code block `A` is executed

If-Else Statement

```
if(<condition>) {
  //code block A
} else {
  //code block B
}
```

- Uses the keyword `else`
- Behavior: if `<condition>` evaluates to true code block `A` is executed
- If `<condition>` evaluates to false, code block `B` is executed

If-Else Statement

```
if(<condition>) {
  //code block A
} else {
  //code block B
}
```

- Uses the keyword `else`
- Behavior: if `<condition>` evaluates to true code block `A` is executed
- If `<condition>` evaluates to false, code block `B` is executed
- The two code blocks are *mutually exclusive*

If-Else Statement

```
if(<condition>) {
  //code block A
} else {
  //code block B
}
```

- Uses the keyword `else`
- Behavior: if `<condition>` evaluates to true code block `A` is executed
- If `<condition>` evaluates to false, code block `B` is executed
- The two code blocks are *mutually exclusive*
- A generalization of the `if` statement

If-Else-If Statement

```
if(<condition1>) {
  //code block A
} else if(<condition2>) {
  //code block B
} else {
  //code block C
}
```

- Multiple conditions: may define as many as you want

If-Else-If Statement

```
if(<condition1>) {
  //code block A
} else if(<condition2>) {
  //code block B
} else {
  //code block C
}
```

- Multiple conditions: may define as many as you want
- The *first* condition that evaluates to true is the one (and only one) that is executed

Introduction
Conditionals
Numerical Comparisons
Examples
Logical Operators
Pitfalls
Exercises

If-Else-If Statement

```
if(<condition1>) {
  //code block A
} else if(<condition2>) {
  //code block B
} else {
  //code block C
}
```

- Multiple conditions: may define as many as you want
- The *first* condition that evaluates to true is the one (and only one) that is executed
- Each code block is *mutually exclusive*

If-Else-If Statement

```
if(<condition1>) {
  //code block A
} else if(<condition2>) {
  //code block B
} else {
  //code block C
}
```

- Multiple conditions: may define as many as you want
- The *first* condition that evaluates to true is the one (and only one) that is executed
- Each code block is *mutually exclusive*
- The most specific conditions come *first*, more general *last*

If-Else-If Statement

```
if(<condition1>) {
  //code block A
} else if(<condition2>) {
  //code block B
} else {
  //code block C
}
```

- Multiple conditions: may define as many as you want
- The *first* condition that evaluates to true is the one (and only one) that is executed
- Each code block is *mutually exclusive*
- The most specific conditions come *first*, more general *last*
- You may omit the final `else` block if there is no "final case" to consider

Numerical Comparisons

- Comparison operators:
  `<`, `>`, `<=`, `>=`

Numerical Comparisons

- Comparison operators:
  `<`, `>`, `<=`, `>=`
- Equality operator: `==`

Numerical Comparisons

- Comparison operators:
  `<`, `>`, `<=`, `>=`
- Equality operator: `==`
- Inequality operators `!=`

Numerical Comparisons

- Comparison operators:
  `<`, `>`, `<=`, `>=`
- Equality operator: `==`
- Inequality operators `!=`
- May be used in combinations of *literals* (hardcoded numerical values), variables or expressions

Numerical Comparisons

```c
int a, b, c;

//comparing a variable to a literal
if(a == 0) {
  printf("a is zero!\n");
}

//comparing two variable values:
if(a == b) {
  printf("the two values are equal\n");
}

//you can, but shouldn't do the following
if(10 == a) {
 //...
}

if(b * b - 4 * a * c < 0) {
 printf("looks bad...\n");
}

//you can but shouldn't:
if(10 < 20) {
 printf("duh, that's always true\n");
}
```

Conditional Examples

```
int huskerScore;
int opponentScore;

//a simple if statement:
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
}

//an if-else statement:
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else {
  printf("Huskers do not win.\n");
}

//an if-else-if statement:
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

Coding Style

```
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

- Use of spaces

Coding Style

```
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

- Use of spaces
- Opening curly brackets on the same line as keywords

Coding Style

```
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

- Use of spaces
- Opening curly brackets on the same line as keywords
- Closing curly brackets on the same indentation level

Coding Style

```
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

- Use of spaces
- Opening curly brackets on the same line as keywords
- Closing curly brackets on the same indentation level
- All blocks are indented at the same level

Coding Style

```c
if(huskerScore > opponentScore) {
  printf("Huskers Win!\n");
} else if(huskersScore < opponentScore) {
  printf("Huskers Lose!\n");
} else {
  printf("Tie, let's go to overtime!\n");
}
```

- Use of spaces
- Opening curly brackets on the same line as keywords
- Closing curly brackets on the same indentation level
- All blocks are indented at the same level
- Consistency is the most important thing

Part III: Logical Operators

Negation, Logical And, Logical Or

Negation Operator

- Any logical statement can be negated using `!`

Negation Operator

- Any logical statement can be negated using `!`
- Negation of `(a == b)` can be `!(a == b)`

Negation Operator

- Any logical statement can be negated using `!`
- Negation of `(a == b)` can be `!(a == b)`
- Negation of `(a <= b)` can be `!(a <= b)`

Negation Operator

- Any logical statement can be negated using `!`
- Negation of `(a == b)` can be `!(a == b)`
- Negation of `(a <= b)` can be `!(a <= b)`
- Better to use: `(a != b)` and `(a > b)`

Negation Operator

- Any logical statement can be negated using `!`
- Negation of `(a == b)` can be `!(a == b)`
- Negation of `(a <= b)` can be `!(a <= b)`
- Better to use: `(a != b)` and `(a > b)`
- Usually a negation is used on a "flag" variable: a variable that simply holds a truth value (true or false)

Flag Variables

- C has no “boolean variables”

Flag Variables

- C has no “boolean variables”
- Any numerical value can be treated as a boolean value

Flag Variables

- C has no “boolean variables”
- Any numerical value can be treated as a boolean value
- `0` is false

Flag Variables

- C has no "boolean variables"
- Any numerical value can be treated as a boolean value
- `0` is false
- Any non-zero value is true

Flag Variables

- C has no “boolean variables”
- Any numerical value can be treated as a boolean value
- `0` is false
- Any non-zero value is true
- `3, 3.5, 3.14, -10` are all true

Flag Variables

- C has no “boolean variables”
- Any numerical value can be treated as a boolean value
- `0` is false
- Any non-zero value is true
- `3, 3.5, 3.14, -10` are all true
- Convention: use `1` as true

Flag Variables

- C has no “boolean variables”
- Any numerical value can be treated as a boolean value
- `0` is false
- Any non-zero value is true
- `3, 3.5, 3.14, -10` are all true
- Convention: use `1` as true
- Best practice: only use `int` variables as booleans

Flag Variables

Example

```c
//flag variable to indicate if someone is a
//student (true) or not (false)
int isStudent;

//set the variable to true:
isStudent = 1;

//they are not a student:
isStudent = 0;

if(isStudent) {
  printf("You get a student discount!\n");
}

//using a negation
if(!isStudent) {
  printf("You pay full price!\n");
}
```

Logical And

- Logical And operator: `&&`

Logical And

- Logical And operator: `&&`
- Evaluates to true only if *both* operands evaluate to true

Logical And

- Logical And operator: `&&`
- Evaluates to true only if *both* operands evaluate to true

```
if(subTotal >= 50.0 && isPreferredMember) {
  discount = .20;
  shipping = 0;
} else if(subTotal >= 50.0 && !isPreferredMember) {
  discount = 0.0;
  shipping = 0;
} else {
  discount = 0.0;
  shipping = 10.50;
}
```

Logical Or

- Logical Or operator: `||`

Logical Or

- Logical Or operator: `||`
- Evaluates to true only if *at least one* of its operands evaluate to true

Logical Or

- Logical Or operator: `||`
- Evaluates to true only if *at least one* of its operands evaluate to true

```
if(isStudent || isPreferredMember) {
  discount = .20;
}
```

Examples

```
if(a > 10 && a < 20) {
  //...
}

if(a == b && a < 10) {
  //...
}

if(a > 10 || a < 20) {
  //...
}

if(a == b || a < 10) {
  //...
}
```

Part IV: Pitfalls

Common Errors & Misconceptions

Pitfall

Incorrect Complex Logic

Consider the following code:

```c
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values

Pitfall

Incorrect Complex Logic

Consider the following code:

```
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values
- What happens when `a = 20`?

Pitfall

Incorrect Complex Logic

Consider the following code:

```
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values
- What happens when `a = 20`?
- First comparison: `0 <= 20`

Pitfall

Incorrect Complex Logic

Consider the following code:

```c
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values
- What happens when `a = 20`?
- First comparison: `0 <= 20`
- Evaluates to true (`1`)

Pitfall

Incorrect Complex Logic

Consider the following code:

```c
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values
- What happens when `a = 20`?
- First comparison: `0 <= 20`
- Evaluates to true (`1`)
- Second comparison: `1 <= 10` (true)

Pitfall

Incorrect Complex Logic

Consider the following code:

```
if(0 <= a <= 10) {
  printf("Value is within range!\n");
}
```

- The above code will compile, will execute, but will not work for certain values
- What happens when `a = 20`?
- First comparison: `0 <= 20`
- Evaluates to true (`1`)
- Second comparison: `1 <= 10` (true)
- Incorrect result

Pitfall

Incorrect Complex Logic

Solution: break up your conditions using a `&&`

```c
if(0 <= a && a <= 10) {
  printf("Value is within range!\n");
}
```

Pitfall

Confusing Comparisons & Assignments

Consider the following code:

```
int a = 5;

if(a = 10) {
  printf("a is ten\n");
}
```

- The above code will compile, run, but will give incorrect results

Pitfall

Confusing Comparisons & Assignments

Consider the following code:

```
int a = 5;

if(a = 10) {
  printf("a is ten\n");
}
```

- The above code will compile, run, but will give incorrect results
- `a = 10` results in an assignment of the value 10 to the variable `a`

Pitfall

Confusing Comparisons & Assignments

Consider the following code:

```c
int a = 5;

if(a = 10) {
  printf("a is ten\n");
}
```

- The above code will compile, run, but will give incorrect results
- `a = 10` results in an assignment of the value 10 to the variable `a`
- A value of 10 evaluates to true

Pitfall

Confusing Comparisons & Assignments

Consider the following code:

```
int a = 5;

if(a = 10) {
  printf("a is ten\n");
}
```

- The above code will compile, run, but will give incorrect results
- `a = 10` results in an assignment of the value 10 to the variable `a`
- A value of 10 evaluates to true
- The `if` body gets executed regardless of the original value of `a`

Pitfall

Improper Semicolons

Consider the following code:

```
int a = 5;

if(a == 10); {
  printf("a is ten!\n");
}
```

- Semicolon (in general) only go after *executable* statements

Pitfall

Improper Semicolons

Consider the following code:

```
int a = 5;

if(a == 10); {
  printf("a is ten!\n");
}
```

- Semicolon (in general) only go after *executable* statements
- Above code will compile, will run, but will not give the correct results

Pitfall

Improper Semicolons

Consider the following code:

```c
int a = 5;

if(a == 10); {
  printf("a is ten!\n");
}
```

- Semicolon (in general) only go after *executable* statements
- Above code will compile, will run, but will not give the correct results
- Conditional statement is *bound* to an empty statement

Non-Numerical Comparisons

- You can compare single `char` values with character literals:

```
char initial = 'C';

if(initial == 'c' || initial == 'C') {
  //...
}
```

Non-Numerical Comparisons

- You can compare single `char` values with character literals:

```c
char initial = 'C';

if(initial == 'c' || initial == 'C') {
  //...
}
```

- You *cannot* use equality and inequality operators on strings (sequences of characters)

```c
if(name == "Chris") {
  printf("Greetings, Professor.\n");
}
```

Non-Numerical Comparisons

- You can compare single `char` values with character literals:

```
char initial = 'C';

if(initial == 'c' || initial == 'C') {
  //...
}
```

- You *cannot* use equality and inequality operators on strings (sequences of characters)

```
if(name == "Chris") {
  printf("Greetings, Professor.\n");
}
```

- The above will *never* give correct results

Precedence Rules

- The logical *and* `&&` is evaluated before the logical *or* `||`

Precedence Rules

- The logical *and* `&&` is evaluated before the logical *or* `||`
- The following are *not* equivalent:

```
a && (b || c)
```

```
a && b || c
```

Precedence Rules

- The logical *and* `&&` is evaluated before the logical *or* `||`
- The following are *not* equivalent:

  ```
  a && (b || c)
  ```

  ```
  a && b || c
  ```
- Use parentheses when necessary

Precedence Rules

- The logical *and* `&&` is evaluated before the logical *or* `||`
- The following are *not* equivalent:

```
a && (b || c)
a && b || c
```

- Use parentheses when necessary
- Best practice: use them even when not necessary to express *intent*

Short-Circuiting

Consider a logical and: `a && b`

- If `a` evaluates to false, it does not matter what `b` evaluates to

Short-Circuiting

Consider a logical and: `a && b`

- If `a` evaluates to false, it does not matter what `b` evaluates to
- Since `a` is false, the entire expression is false

Short-Circuiting

Consider a logical and: `a && b`

- If `a` evaluates to false, it does not matter what `b` evaluates to
- Since `a` is false, the entire expression is false
- Consequently: `b` is not evaluated/executed

Short-Circuiting

Consider a logical and: `a || b`

- If `a` evaluates to true, it does not matter what `b` evaluates to

Short-Circuiting

Consider a logical and: `a || b`

- If `a` evaluates to true, it does not matter what `b` evaluates to
- Since `a` is true, the entire expression is true

Short-Circuiting

Consider a logical and: `a || b`

- If `a` evaluates to true, it does not matter what `b` evaluates to
- Since `a` is true, the entire expression is true
- Consequently: `b` is not evaluated/executed

Short-Circuiting

- Short circuiting is common to the vast majority of programming language

Short-Circuiting

- Short circuiting is common to the vast majority of programming language
- Historic reasons

Short-Circuiting

- Short circuiting is common to the vast majority of programming language
- Historic reasons
- Common *idiom* in many programming languages:

```
if(a != NULL && a[0] == 10) {
  //...
}
```

Part V: Exercises

Exercise

Write a code snippet that determines the maximum of three integer values.

Exercise

Write a program that reads a decibel level from the user and gives them a description of the sound level based on the following categories.

- 0 - 60 Quiet
- 61 - 70 Conversational
- 71 - 90 Loud
- 91 - 110 Very Loud
- 111 - 129 Dangerous
- 130 - 194 Very Dangerous

Exercise

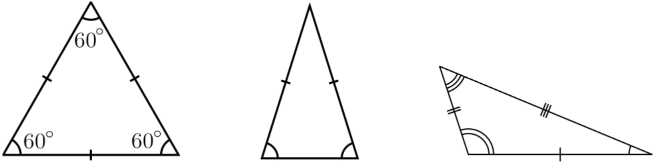


Figure: Examples of Equilateral, Isosceles, and Scalene triangles

3 sides are a valid triangle only if the sum of the length of any two sides is strictly greater than the length of the third side.

Write a program to determine if 3 inputs form a valid triangle and if so, what type.