Conditionals

Both Java and C have **conditional statements** (also called **selection statements**)that allow us to write code which can change depending on logical conditions. This is an essential part of game logic, since many things that happen depend on specific states or events in the gameplay.

# If Statements

The simplest conditional is the if statement:

if (expression)  
{  
 statement  
}

This can be read as “if **expression** is true, then execute **statement,**” You could also read it as “if **expression** is false, then do not execute **statement**.” In other words, if the expression evaluates to true, the code inside the curly brackets will be executed. If it evaluates to false, the code will be skipped.

The expression inside the parentheses of the if statement must be a Boolean expression, meaning it must evaluate to either true or false. (Remember that C has no Boolean type, so that 0 is used for false and 1 for true.) This could be a single variable, as in the following example:

|  |  |
| --- | --- |
| **C** | **Java** |
| int someFlag = 1;  if (someFlag)  {  DoSomeLogic();  } | boolean someFlag = true;  if (someFlag)  {  DoSomeLogic();  } |

We can also use comparison and logical operators to achieve a true or false result.

int playerHealth = 10;

int threshold = 5;

if (playerHealth < threshold)

{

ShowWarning();

}

float distance = 4.826f;

float radius1 = 6.1f;

float radius2 = 5.25f;

if (distance < radius1 && distance < radius2)

{

DoCollision();

}

## Short Circuit Evaluation

Something to know about the logical operators **AND** (&&) and **OR** (||) is that they will do what is called **short circuit evaluation**. As soon as they can tell what the result will be they will stop the evaluation. True **OR** anything is always true, so the second statement won’t be evaluated. Similarly, false **AND** anything is always false, so the second statement also won’t be evaluated.

This generally only matters if you are doing something in the statement that makes a change, such as (a > b && ++a <= 6). In this case, if a was not greater than b, then the second expression would not be evaluated and a would not be incremented.

## Else

If statements can have an optional **else** clause.

if (expression)

{

statement1

}

else

{

statement2

}

This reads as “if **expression** is true, then execute **statement1**, otherwise execute **statement2**.” Depending on the expression, either statement1 or statement2 will get executed, but never both.

If we have more than one condition to check, we can use **else if** to nest additional if statements.

if (expression1)

{

statement1

}

else if (expression2)

{

statement2

}

This reads as “if **expression1** is true then execute **statement1**, otherwise if **expression2** is true then execute **statement2**.” You can include any number of else ifs in an if statement, but you can only have one if and one else.

Remember that the expressions get evaluated in order starting at the top, so if an expression further down the chain is executed then you can assume the earlier conditions were false. In the example below, to check if the grade is a B we don’t have to check if the average is between 80 and 90 because we already know it is not 90 or above.

if (average >= 90)

{

grade = 'A';

}

else if (average >= 80)

{

grade = 'B';

}

else if (average >= 70)

{

grade = 'C';

}

else if (average >= 60)

{

grade = 'D';

}

else

{

grade = 'F';

}

# Switch Statements

The switch statement is similar to nested if statements. It looks like this:

switch (expression)

{

case value1:

statement1;

break;

case value2:

statement2;

break;

default:

statement3;

}

The **break** keyword is necessary to stop evaluating the switch statement. If it was not included, after running the statements inside a case it would continue to run the rest of the code inside the switch, and if another case also matched the value it would run those statements as well.

The expression in the parentheses of the switch statement must either be or evaluate to an integer. This integer value will be checked against the cases inside the switch statement to see if it matches one of their values. If none of them match, the default case will be used, if it exists. The value of each case must be constant and known at compile time: it can’t be a variable that could be changed.

Both of the code examples below would produce the exact same output (assuming a function named Print that will print the string that is passed into it to the console):

|  |  |
| --- | --- |
| if (year == 1)  {  Print("Freshman");  }  else if (year == 2)  {  Print("Sophomore");  }  else if (year == 3)  {  Print("Junior");  }  else if (year == 4)  {  Print("Senior");  }  else  {  Print("Invalid year");  } | switch (year)  {  case 1:  Print("Freshman");  break;  case 2:  Print("Sophomore ");  break;  case 3:  Print("Junior ");  break;  case 4:  Print("Senior ");  break;  default:  Print("Invalid year");  break;  } |

# Scope

The **scope** or visibility of a variable is what determines in which parts of the program the variable can be seen and used. Some other scope factors will be discussed in later chapters, but for now we will stick with **block scope**. Blocks are any areas that are delimited by curly braces, such as an if statement. A variable with block scope is visible from its declaration to the end of the block. (Variables with block scope are also called **local** variables, since they are only available within their local area.) This means that variables declared inside an if statement are only usable inside that block, and can’t be used after the closing curly brace.

int a = 5; /\* scope of a starts here \*/

if (a == 5)

{

int x; /\* scope of x starts here \*/

x = a; /\* okay, both in scope \*/

} /\* scope of x ends here \*/

a = x; /\* **error**: x is not in scope \*/

Variable names must be unique, of course, but they only have to be unique within their scope. Once a new scope starts, you are allowed to declare a variable that has the same name as a parent scope. If you do this, however, you can’t access the variable from the parent scope any more, since the name has been overridden.

int a = 1;

int a = 0; /\* **error**: a is already defined in this scope \*/

Print(a); /\* this will print the number 1 \*/

if (a == 1)

{

int a = 5; /\* okay, different scope so different a \*/

int a = 0; /\* **error**: a is already defined in this scope \*/

int x = 6;

Print(a); /\* this will print the number 5 \*/

Print(x); /\* this will print the number 6 \*/

}

if (b == 2)

{

int x = 10; /\* okay, different scope \*/

Print(x); /\* this will print the number 10 \*/

}

Print(a); /\* this will again print the number 1 \*/