CS201, Mock
printf, scanf, Boolean Conditions, If-Then, Test Plans

For historical reasons, let's briefly discuss two commonly used expressions in C: printf and scanf. Since these expressions are used quite often in C, there is a good chance that you will encounter them if you are looking at legacy code or code that contains mixed C and C++.

To use printf or scanf, one must include the file "stdio.h":

```
#include <stdio.h>
```

The printf command is the analog to cout. It is used to print output. The format for this command is:

printf(output_stream, format_string, output_variables);

output_stream refers to the "place" output should be directed. If this parameter is left off, then by default output will go to *stdout*, or the user's terminal Other options include *stderr* (where errors are printed) or descriptors.

format_string is a text string with the data you wish to output. It can include then \ escape characters along with the following special flags for printing variables:

- %d - decimal integer
- %ld - long integer
- %f - floating point number
- %s - string
- %c - char
- %x - hex integer

There are many more format options not listed here to control spacing, formatting, decimal places, etc.. See the man pages on scanf or printf to see them all.

Output_variables would be blank if you have no variables to print, or the list of variables that you wish to print. Any variables listed here must correspond with the variables listed in the format_string.

Some examples will help illustrate the use of printf:

```
#include <stdio.h>
main()
{
  int i = 2;
  float f = 1.2;
  char c = 'A';

  printf("hello, world\n");
  printf("%d multiplied by %f is %f\n", i, f, i*f);
  printf("16 in hex is %x\n", 16);
  printf("The letter %c and once more %c\n", c, 65);
}
```

This program will give the output:

```
hello, world
2 multiplied by 1.200000 is 2.400000
16 in hex is 10
The letter A and once more A
```

The important thing to remember with printf is that variables will be output and treated as if they are of the type listed in the format string. This can sometimes cause errors if the wrong type is listed.

The other function that we will briefly discuss is scanf. This is the analog to cin and is used to input data. The format for scanf is very similar to printf:

scanf(input_stream, format_string, input_variables);

input_stream refers to the "place" input comes from. If this parameter is left off, then by default output will go to *stdin*, or the user's terminal Other options include descriptors.

format_string is a text string with the data you wish to input. It is identical to the printf format string options.

input_variables contains *addresses* where the input data should go. We will talk more about addresses later, but for now we use the & operator to obtain the address of a variable. Any variables listed here must correspond with the variables listed in the format_string.

Some examples will help illustrate the use of scanf:

```
#include <stdio.h>
main()
{
  int i = 2;

  printf("Enter a integer:\n");
  scanf("%d",&i);
  printf("%d multiplied by 1.5 is %f\n", i, i*1.5);
}
```

When run, the output will look like:

```
Enter a integer:
5
5 multiplied by 1.5 is 7.50000
```

Use the man pages!

If you would more information, I encourage you to use the "man" pages on the Unix systems. Within the man pages you can find help and information on almost all C and C++ library functions. For example:

- man printf
- man scanf

Will give you useful information on how to use these functions. If these commands are in different sections of the man pages, sometimes you have to provide a section number:

- man 1 printf

You can also use the apropos command to search for related topics;

- apropos print

Will list some commands that have to do with printing.

If you are using the Visual Studio environment, you can highlight a function name and then hit F1 to get help. Or you can browse the help topics from H)elp, I)ndex from the menu.

Boolean Expressions and Conditions

The physical order of a program is the order in which the statements are *listed.* The logical order of a program is the order in which the statements are *executed*. With conditional structures and control structures that we will examine soon, it is possible to change the order in which statements are executed.

Boolean Data Type

To ask a question in a program, you make a statement. If your statement is true, the answer to the question is yes. If your statement is not true, the answer to the question is no. You make these statements in the form of *Boolean expressions*. A Boolean expression asserts (states) that something is true. The assertion is evaluated and if it is true, the Boolean expression is true. If the assertion is not true, the Boolean expression is false.

In C++, data type **bool** is used to represent Boolean data. Each **bool** constant or variable can contain one of two values: **true** or **false**. The bool data type is a recent construct. In older compilers, true actually corresponds to any integer value that is nonzero. The false value corresponds to an integer value of zero. For backward compatibility, the bool type of true is set to 1, and false is set to 0. For example:

```
cout << true << " " << false << endl;
```

will output: 1 0

Relational Operators

A Boolean expression can be a simple Boolean variable or constant or a more complex expression involving one or more of the relational operators. Relational operators take two operands and test for a relationship between them. The following table shows the relational operators and the C++ symbols that stand for them.

C++ Symbol	*Relationship*
==	Equal to
!=	Not equal to (no <>)
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

For example, the Boolean expression

number1 < number2

is evaluated to **true** if the value stored in **number1** is less than the value stored in **number2**, and evaluated to **false** otherwise.

When a relational operator is applied between variables of type **char**, the assertion is in terms of where the two operands fall in the collating sequence of a particular character set. For example,

character1 < character2

is evaluated to **true** if the character stored in **character1** comes before the character stored in **character2** in the collating sequence of the machine on which the expression is being evaluated. Although the collating sequence varies among machines, you can think of it as being in alphabetic order. That is, *A* always comes before *B* and *a* always before *b*, but the relationship of *A* to *a* may vary. This is an artifact of the way the alphabet was defined in the ASCII code.

We must be careful when applying the relational operators to floating point operands, particularly equal (==) and not equal (**!**=). Integer values can be represented exactly; floating point values with fractional parts often are not exact in the low-order decimal places. Therefore, you should compare floating point values for near equality. For now, *do not compare floating point numbers for equality*. For example, it may be possible to have:

float f = 2.3 / 4.6;

and then attempt:

f == 0.5

to be evaluated to false, not true. As we will see, it is better to compare floating point numbers to a range, e.g. within 0.499 and 5.001.

Boolean Operators

A simple Boolean expression is either a Boolean variable or constant or an expression involving the relational operators that evaluates to either true or false. These simple Boolean expressions can be combined using the logical operations defined on Boolean values. There are three Boolean operators: AND, OR, and NOT. Here is a table showing the meaning of these operators and the symbols that are used to represent them in C++.

C++ Symbol *Meaning*

&&	AND is a binary Boolean operator. If both operands are true, the result is true. Otherwise, the result is false. **True** True False **False** False False
||	OR is a binary Boolean operator. If at least one of the operands is true, the result is true. Otherwise, the result is false. **True** True True **False** True False
!	NOT is a unary Boolean operator. NOT changes the value of its operand: If the operand is true, the result is false; if the operand is false, the result is true. **Not Value** **True** False **False** True

If relational operators and Boolean operators are combined in the same expression in C++, the Boolean operator NOT (**!**) has the highest precedence, the relational operators have next higher precedence, and the Boolean operators AND (**&&**) and OR (||) come last (in that order). Expressions in parentheses are always evaluated first.

For example, given the following expression (**stop** is a **bool** variable)

```
count <= 10 && sum >= limit || !stop
```

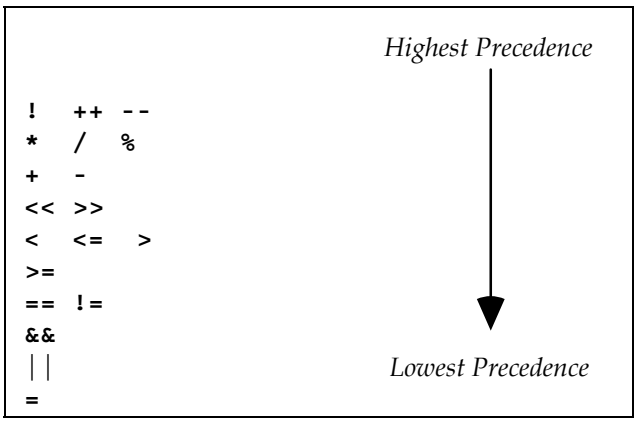
!stop is evaluated first, the expressions involving the relational operators are evaluated next, the **&&** is applied, and finally the || is applied. C++ uses *short-circuit evaluation*. The evaluation is done in left-to-right order and halts as soon as the result is known. For example, in the above expression if both of the arithmetic expressions are true, the evaluation stops because the left operand to the OR operation (|| operator) is true. There is no reason to evaluate the rest of the expression: true OR anything is true. Short-circuit evaluation raises some subtleties that we will examine shortly.

It is a good idea to use parenthesis to make your expressions more readable, e.g:

```
(((count <=10) && (sum>=limit))  ||  (!stop))
```

This also helps avoid difficult-to-find errors if the programmer forgets the precedence rules.

The following table summarizes the precedence of some of the C++ operators we have seen so far.



If-Then and If-Then-Else Statements

The If statement allows the programmer to change the logical order of a program; that is, make the order in which the statements are executed differ from the order in which they are listed in the program. The If-Then statement uses a Boolean expression to determine whether to execute a statement or to skip it. The format is as follows:

```
if (boolean_expression)
        statement;
```

The statement will be executed if the Boolean expression is true. If you wish to execute multiple statements, which is called a *block*, use curly braces:

```
if (boolean_expression) {
        statement1;
        statement2;
        …
        statement99;
}
```

Although the curly braces are not needed when only a single statement is executed, some programmers always use curly braces to help avoid errors such as:

```
if (boolean_expression)
        statement1;
        statement2;
```

This is really the same as:

```
if (boolean_expression)
        statement1;
statement2;
```

Such a condition commonly arises when initially only a single statement is desired, and then a programmer goes back and adds additional statements, forgetting to add curly braces.

We can also add an optional **else** or **else if** clause to an if statement. The else statement will be executed if all above statements are false. Use else if to test for multiple conditions:

```
if (boolean_expression1)
        statement1;                     // Expr1 true
else if (boolean_expression2)
        statement2;                     // Expr1 false, Expr2 true
else if (boolean_expression3)
        statement3;                     // Expr1, Expr2 false, Expr3 true
…
else
        statement_all_above_failed;   // Expr1, Expr2, Expr3 false
```

Here are some examples of if statements:

```
if (number < 0)
        number = 0;
sum = sum + number;
```

```
if (number < 0) {
        number = 0;
}
sum = sum + number;
```

```
if (number < 0) {
        number = 0;
        sum = sum+ number;
}
```

In all of these examples, the expression **(number < 0)** is evaluated. If the result is **true**, the statement **number = 0** is executed. If the result is **false**, the statement is skipped. In the first two cases, the next statement to be executed is **sum = sum + number**. In the last case, **sum = sum = number** is only evaluated if **number < 0**.

Here is another example.

```
cout << “Today is a “;
if (temperature <= 32)
{
   cout << "cold day." << endl;
   cout << "Sitting by the fire is appropriate." << endl;
}
else
{
   cout << "nice day." << endl;
   cout << "How about taking a walk?" << endl;
}
```

There is a point of C++ syntax that you should note: There is never a semicolon after the right brace of a block (compound statement).

Finally here is an example using else-if, also referred to as a **nested if statement**:

```
cout << "Today is a ";
if (temperature <= 32)
   cout << "cold ";
else if (temperature <= 85)
   cout << "nice ";
else
   cout << "hot ";
cout << "day." << endl;
```

Notice that the nested If statement does not have to ask if the temperature is greater than 32 because we do not execute the **else** branch of the first If statement if the temperature is less than or equal to 32.

In nested If statements, there may be confusion as to which **if** an **else** belongs. In the absence of braces, the compiler pairs an **else** with the most recent **if** that doesn't have an **else**. You can override this pairing by enclosing the preceding **if** in braces to make the **then** clause of the outer If statement complete.

Compound Statements and Short Circuit

We can use relational operators to make more complicated if-then statements. Consider the following:

```
int i=0,j=2,k=4;

 if (((k/j)<10) && ((k*j)<9)) {
       cout << "foo" << endl;
 }
 else {
      cout << "bar" << endl;
 }
```

This will evaluate 4/2 as being less than 10. Also 4*2 is less than 9. Consequently, "foo" should be printed.

Now consider the following:

```
int i=0,j=2,k=4;

 if (((k/j)<10) || ((k*j)<9)) {
       cout << "foo" << endl;
 }
 else {
      cout << "bar" << endl;
 }
```

"foo" will still be printed. However, since we have a logical OR this means that we only need one of the top expressions to be evaluated to true. C++ works left to right, so once it finds that k/j < 10 is true, it skips the (k*j)<9 expression. This helps save processing time and also lets us have potentially "invalid" expressions:

```
int i=0,j=2,k=4;

 if (((k/i)<10) || ((k*j)<9)) {
       cout << "foo" << endl;
 }
 else {
      cout << "bar" << endl;
 }
```

This will crash, resulting in a division by 0 error.
However, what about the following:

```
int i=0,j=2,k=4;
 if (((k*j)<9) || ((k/i)<10)) {
        cout << "foo" << endl;
 }
 else {
       cout << "bar" << endl;
 }
```

or

```
int i=0,j=2,k=4;
 if (((k*j)>8) && ((k/i)<10)) {
        cout << "foo" << endl;
 }
 else {
       cout << "bar" << endl;
 }
```

The short-circuit behavior of C++ allows us to avoid the fatal computations. This is useful in some cases when the "illegal" instructions are actually legal in the event that the first condition was true. Nevertheless, such constructs need to be used with care to avoid crashes.

Test Plans

How do you test a specific program to determine its correctness? You design and implement a *test plan*. A test plan for a program is a document that specifies the test cases that should be run, the reason for each test case, and the expected output from each case. The test cases should be chosen carefully. The *code-coverage* approach designs test cases to ensure that each statement in the program is executed. The *data-coverage* approach designs test cases to ensure that the limits of the allowable data are covered. Often testing is a combination of code and data coverage.

Implementing a test plan means that you run each of the test cases described in the test plan and record the results. If the results are not as expected, you must go back to your design and find and correct the error(s). The process stops when each of the test cases gives the expected results. Note that an implemented test plan gives you a measure of confidence that your program is correct; however, all you know for sure is that your program works correctly on your test cases. Therefore, the quality of your test cases is extremely important.

An example of a test plan for the code fragment that tests temperatures is shown on the next page. We assume that the fragment is embedded in a program that reads in a data value that represents a temperature.

Reason for Test Case	*Input Values*	*Expected Output*	*Observed Output*
Test first end point	32	Today is a cold day.	
Test second end point	85	Today is a nice day.	
Test value below first end point	31	Today is a cold day.	
Test value between end points	45	Today is a nice day.	
Test value above second end point	86	Today is a hot day.	
Test negative value	-10	Today is a cold day.	

To implement this test plan, the program is run six times, once for each test case. The results are written in the Observed Output column.

Extremely Common Bug! Confusing = and ==

The assignment operator (=) and the equality test operator (==) can easily be miskeyed one for the other. What happens if this occurs? Unfortunately, the program does not crash; it continues to execute but probably gives incorrect results. Look at the following statements.

```
int i=0;
i == i + 1;
cout << i << endl;
```

This code fragment generates no errors. However, the output it produces is "0" while most likely the programmer is intending the output to be "1". The way that C++ evaluates this code is that it believes we are just making a false statement. The variable i is never equal to i+1. This expression merely returns false, and the computer continues along its way processing the rest of the commands.

Look at the next statement going the other direction:

```
int i=0;
if (i=1) {
        cout << "Value is 1\n";
}
else {
        cout << "Value is 0\n";
}
```

The output for this program is always going to be “Value is 1”. This can be very frustrating for a programmer to debug, because the difference between “=” and “==” easily eludes the eye, while the logic seems to dictate that the value printed should be 0.

The reason for this behavior is because the assignment operator returns the operand. In other words, i=0 returns the value 0. i=1 returns the value 1. i=2 returns the value 2. These values are then compared with true or false. Unless these values are 0, the end result is that the computer will consider the value to be true, and execute the “Value is 1” branch.

For example:

```
cout << (i=3) << endl;
```

Will print the value 3 and at the same time store the value 3 into variable i.

When we execute **if (i=1) { …}** we first store the value 1 into variable i, the value 1 is returned, it is evaluated as being true, and then it executes the statement associated with the true condition.

The moral of this story is to be very careful when you key your program. A simple keying error can cause logic errors that the compiler does not catch and that are very difficult for the user to find. Great care and a good test plan are essential to C++ programming.