C++ Syntax Notes: Basic Operators, If-Else, and Type Casting

Here are some basic syntax notes about C++.

Assignment vs. Equality

Notice that "=" is the assignment operator in C++ and "==" is the comparison operator that checks equality:

```
#include <iostream>

int main() {
  int x = 2; // This line assigns a value.
  int y = 2;

  if (x == y) {
    // Note that we had to use "==" to check equality!
    // That's two equal signs!
    std::cout << "x and y are equal" << std::endl;
  }
  else {
    std::cout << "x and y are not equal" << std::endl;
  }

  return 0;
}
```

Run

Reset

Be very careful not to mix these up or make a typo! That is a common mistake that even long-time C++ programmers make. It can cause bugs that are hard to find.

If-Else

The above example also shows one of the most fundamental control structures in C++: The **if** and **else** keywords. The basic structure is always like this:

```
if (condition) {
    // true case
}
else {
    // false case
}
```

You can also chain these together with more if-else combinations. In this situation, the first condition that is true will be the section that is executed:

```
if (condition1) {

}
else if (condition2) {

}
else if (condition3) {

}
else {
    // If none of the other conditions were met...
}
```

It's crucial to note that in the above example, the use of "if... else if... else", in a chain, makes the conditional cases mutually exclusive. Only one of them will execute: the first one that is true. Be very careful to use "else" where appropriate to get this behavior. If you simply put several "if" statements in a sequence, then potentially, several of them will execute, as in this example:

```
int x = 100;
if (x > 0) {
    // This will execute
}
if (x > 10) {
    // This will execute
}
if (x > 50) {
    // This will execute
}
else {
    // This is the only section that won't execute!
}
```

Here, all of the conditional blocks will execute except for the last "else" part: the last "else" at the end only affects the preceding "if," and since it's true that x is greater than 50, that "if" block will execute and the final "else" will not. The important lesson is to always be intentional about where you choose to write "else" (or not), depending on whether you intend cases to be mutually exclusive.

You should also be aware of the strange-looking "ternary operator," also called the "conditional operator," which is actually a combination of two operators, "?" and ":" (question mark and colon). The basic format for this syntax is:

[Boolean-valued condition] ? [expression to evaluate if true] : [expression to evaluate if false]

So, you could compare "A ? B : C" to "if (A) {B;} else {C;}" but that the ternary operator is allowed in situations where "if" is not allowed. This is because "?:" is evaluated at the level of an expression, so it can be a sub-expression within a larger expression, whereas the "if" statement is a top-level flow control statement that can't be nested within an expression. This means that you can put ?: to the right of an assignment = operator, which you cannot do with an "if". Here is an example:

```
7
```

This ?: ternary operator is useful in other situations where you can use it creatively to make code more concise and compact, although some people find it confusing to read. We recommend that you do not use it more than necessary, but you will see it occasionally as you explore C++ source code.

Comparison Operators and Arithmetic Operators

There are many other operators in C++ that are compatible with various types. You should get familiar with some of them and be able to tell the difference between a comparison and an assignment.

```
#include <iostream>

int main() {
  if (3 <= 4) {
    std::cout << "3 is less than or equal to 4" << std::endl;
  }

  int x = 3;
  x += 2; // This increases x by 2. It's the same as writing: x = x + 2;
  std::cout << "x should be 5 now: " << x << std::endl;

  return 0;
}
```

Run

Reset

You can read about many more operators here: http://www.cplusplus.com/doc/tutorial/operators/

Type Casting

An important concept to understand in C++ is "casting." This is the phenomenon where a value is automatically converted to a different type in order to allow an operation to continue. For example, if you try to add together a floating point **double** value and an integer **int** value, the int will be converted to double first, and then the result of the addition will have type double as well. But if you save a double value to an int variable, the floating point value will be truncated to an integer! For example:

```
#include <iostream>
int main() {
  int x = 2;
  double y = 3.5;
  std::cout << "This result will be a double with value 5.5: " << (x+y) << std::

  int z = x+y; // This expression is calculated as a double, but then it's cast
  std::cout << "This result will be an int with value 5: " << z << std::endl;
  return 0;
}
```

Run

Reset

It's also the case that various values can be cast to the Boolean **bool** type. So, they can be used as a condition. For example, usually, nonzero numeric values will be considered **true**, and zero will be considered **false**.

```
#include <iostream>
int main() {
  if (0) {
    std::cout << "You won't see this text." << std::endl;
  }
  if (-1) {
    std::cout << "You WILL see this text!" << std::endl;
  }
  return 0;
}
```

Run

Reset

You need to be aware that casts are happening invisibly in your code! Sometimes this can cause hard-to-find bugs. If you want to read more details about the casting operation, especially if you are curious about a debugging situation later, see these references:

A more detailed primer on the casting topic: http://www.cplusplus.com/doc/tutorial/typecasting/

In-depth documentation: https://en.cppreference.com/w/cpp/language/implicit_conversion