“sams teach yourself “c++” in 21 days”

***“Day 1: Getting Started”***

**A Brief History of C++**

C++ is a ***“statically typed, compiled, general-purpose, case-sensitive, and free form programming language”*** that supports procedural, object-oriented, and generic programming. C++ is regarded as a **middle-level language**, as it comprises a combination of both **high-level** and **low level** language features. C++ was developed by **Bjarne Stroustrup** starting in **1979** at **Bell Labs** in **Murray Hill**, **New Jersey**, as an enhancement to the C language and originally named **”C with Classes”** but later it was renamed **C++** in **1983**. C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

**Source Code** – Human-Readable Language

**Object Code (Machine Language)** - Computer instructions were represented by long strings of ones and zeros.

**Interpreter (Script Engine)** - translates and executes a program as it reads it, turning the program instructions, or source code, directly into actions.

**Compiler** – translates source code into an intermediary form. This step is called **“compiling”**, and it produces an object file. The compiler then invokes a linker, which combines the object file into an executable program.

“Some languages, such as Visual Basic 6, call the interpreter the ***“Runtime Library”.*** Other languages, such as the Visual Basic .NET and Java have another component, referred to as a *“****Virtual Machine” (VM)*** or *a* ***“Runtime”.*** The VM or Runtime is also an interpreter. However, it is not a source code interpreter that translates human-readable language into computer-dependent machine code. Rather, it interprets and executes a compiled computer-independent ***“Virtual Machine Language”*** or intermediary language.”

**Procedural, Structured, and Object-Oriented Programming**

**Procedure (Function)** *-* also called a function or a ***“method”***, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was changed. To make sense of this potentially confusing situation, **“*structured programming”*** was created.

**Structured programming** - is often called **“*procedural programming”***because of its

Focus on procedures (rather than on “objects”).

**Object-Oriented Programming (OOP)**

**Object-Oriented Programming** - essence of object-oriented programming is to model ***“objects”*** (that is, things or concepts) rather than ***“data”.*** The objects you model might be onscreen widgets, such as buttons and list boxes, or they might be real-world objects, such as customers, bicycles, airplanes, cats, and water.

Objects have characteristics, also called ***“properties”*** or ***“attributes”***, such as age, fast, spacious, black, or wet. They also have capabilities, also called ***”operations”***  or ***”functions”***, such as purchase, accelerate, fly, purr, or bubble. It is the job of object-oriented programming to represent these objects in the programming language.

**C++ and Object-Oriented Programming**

**The Three Pillars of object-oriented development:**

**Encapsulation** - The property of being a self-contained unit is called **“*encapsulation”***. With encapsulation, you can accomplish data hiding. ***“Data hiding”*** is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. You can use a well-designed object without knowing about its internal workings. Changes can be made

to those workings without affecting the operation of the program, as long as the specifications

are met; The actual inner workings of the class can be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

**Inheritance** - With inheritance, you can declare a new type that is an extension of an existing type. This new subclass is said to derive from the existing type and is sometimes called a ***“derived type”***.

**Polymorphism** - C++ supports the idea that different objects do “the right thing” through what is called ***“function polymorphism”*** and ***“class polymorphism”***. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms.

**How C++ Evolved**

As object-oriented analysis, design, and programming began to catch on, Bjarne

Stroustrup took the most popular language for commercial software development, C, and extended it to provide the features needed to facilitate object-oriented programming.

Although it is true that C++ is a superset of C and that virtually any legal C program is a legal C++ program, the leap from C to C++ is very significant. C++ benefited from its relationship to C for many years because C programmers could ease into their use of C++. To really get the full benefit of C++, however, many programmers found they had to ***“unlearn”*** much of what they knew and learn a ***“new way”*** of conceptualizing and solving programming problems.

**Should I Learn C First?**

The question inevitably arises: “***Because C++ is a superset of C, should you learn C first?”*** Stroustrup and most other C++ programmers agree that not only is it unnecessary to learn C first, it might be advantageous not to do so. C programming is based on structured programming concepts; C++ is based on object-oriented programming. If you learn C first, you’ll have to ***“unlearn”*** the bad habits fostered by C.

**C++, Java, and C#**

C++ is one of the predominant languages for the development of **commercial software**.

In recent years, **Java** has challenged that dominance; however, many of the programmers who left C++ for Java have recently begun to return. In any case, **the two languages are so similar that to learn one is to learn 90 percent of the other.**

**C#** is a newer language developed by Microsoft for the ***”.NET platform”***. C# uses essentially the same syntax as C++, and although the languages are different in a few important ways, **learning C++ provides a majority of what you need to know about C#**. Should you later decide to learn C#, the work you do on C++ will be an excellent investment.

**Microsoft’s Managed Extensions to C++**

With **.NET**, Microsoft introduced Managed Extensions to C++ ***(“Managed C++”)***. This is an extension of the C++ language to allow it to use Microsoft’s new platform and libraries. More importantly, **Managed C++** allows a C++ programmer to take advantage of the advanced features of the .NET environment. Should you decide to develop specifically for the .NET platform, you will need to extend your knowledge of standard C++ to include these extensions to the language.

***“Day 2: The Anatomy of a C++ Program”***

**A simple Program**

* **# (Preprocessor) -** The preprocessor reads through your source code, looking for lines that begin with the pound symbol (#) and acts on those lines before the compiler runs.
* **#include (command) –** Preprocessor instructions that says **“What follows is a filename. Find that file, read it, and place it right here”.**
* ***<iostream> (Input-Output-Stream)*** - is used by ***“cout”*** and ***“cin”***, which assists with writing to the console.
* ***main () (function) –*** A function is a block of code that performs one or more actions.

Usually, functions are invoked or called by other functions, but main() is special. When your program starts, main() is called automatically. main(), like all functions, must state what kind of value it returns. The return value type for main() is int, which means that this function returns an integer to the operating system when it completes. In this case, it returns the integer value 0. Returning a value to the operating system is a relatively unimportant and little used feature, but the C++ standard does require that main() be declared as shown.

***“Some compilers let you declare main() to return void. This is no longer legal***

***C++, and you should not get into bad habits. Have main() return int, and***

***simply return 0 as the last line in main().”***

* **Namespace -** A namespace is a way to say **“when I say cout, I mean the cout that is part of the standard namespace, not some other namespace.”** You say that to the compiler by putting the characters ***“std”*** followed by two colons before the ***“cout”***.
* **Endl -** The purpose of endl is to write a new line to the console.

**Library -** is a collection of classes.

**Standard library** - is the standard collection that comes with every ANSI-compliant compiler.

**Types of Comments**

**Comments -** are text that is ***“ignored”*** by the compiler, but that can inform the reader of what you are doing at any particular point in your program.

**Single-line comments (C++ Programming-Style)** - are accomplished using a double slash **(//)**. The double slash tells the compiler to ignore everything that follows, until the end of the line.

**Multiline comments** **(C Programming-Style)** - are started by using a forward slash followed by an asterisk **(/\*)**. This ***“slash-star”*** comment mark tells the compiler to ignore everything that follows until it finds a star-slash **(\*/)** comment mark.

**Document comments** - and are indicated using three forward slashes **(///)**. The compilers that support this style of comment allow you to generate documentation about the program from these comments.

**A Final Word of Caution About Comments**

***“Some people recommend writing comments at the top of each function, explaining what the function does and what values it returns.”***

Comments that state the obvious are less than useful. In fact, they can be counterproductive because the code might change and the programmer might neglect to update the comment. What is obvious to one person might be obscure to another, however, so judgment is required when adding comments.

***“The bottom line is that comments should not say what is happening, they should say why it is happening.”***

**Using Functions/Methods**

**Function syntax:**

**The Header -** (Return type) (Function name) (Parameter/s or Arguments)

**The Body -** { (return) (statements) }

**Int SampleFunction (1, 2) { return (statements or conditions); }**

***“Day 3: Working with Variables and Constants”***

**What Is a Variable?**

**Variable** - is a location inyour computer’s memory in which you can store a value and from which youcan later retrieve that value. A **variable**is a place to store information.

**Storing Data in Memory**

*“Your computer’s memory can be viewed as a series of* ***cubbyholes****. Each cubbyhole is one of many, many such holes all lined up. Each cubbyhole—or* ***memory location****—is numbered sequentially. These numbers are known as****” memory addresses”****. A* ***variable*** *reserves one or more cubbyholes in which you can store a value.*

*Your variable’s name (for example,* ***myVariable****) is a label on one of these cubbyholes so that you can find it easily without knowing its actual memory address. As you can see from the figure, myVariable starts at memory address 103. Depending on the size of myVariable, it can take up one or more memory addresses.”*

**Setting Aside Memory**

When you define a variable in C++, you must tell the compiler what kind of variable it is

***(This is usually referred to as the variable’s “type”)***: an **integer**, a **floating-point number**, a

**character**, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable. It also allows the compiler to warn you or produce an error message if you accidentally attempt to store a value of the wrong type in your variable ***(This characteristic of a programming language is called “strong typing”)***.

Each cubbyhole is one byte in size. If the type of variable you create is four bytes in size, it needs four bytes of memory, or four cubbyholes. The type of the variable (for example, integer) tells the compiler how much memory (how many cubbyholes) to set aside for the variable.

There was a time when it was imperative that programmers understood bits and bytes; after all, these are the fundamental units of storage. Computer programs have gotten better at abstracting away these details, but it is still helpful to understand how data is stored. For a quick review of the underlying concepts in binary math.

**Size of Integers**

**Integer numbers** - On any one computer, each variable type takes up a single, unchanging amount of room.

That is, an **integer** might be **two bytes** on one machine and **four** **bytes** on another, but on either computer it is always the same, day in and day out.

**Single characters** - including **letters**, **numbers**, and **symbols**—are stored in a variable of type ***“char”***. A char variable is most often **one byte long**.

For **smaller integer** numbers, a variable can be created using the **short type**. A **short integer** is **two bytes** on most computers, **a long integer** is usually **four bytes**, and an **integer** (**without** the keyword short or long) is usually **two** or **four** bytes.

You’d think the language would specify the exact size that each of its types should be; however, C++ doesn’t. All it says is that a short must be less than or equal to the size of an int, which, in turn, must be less than or equal to the size of a long.

The size of an integer is determined by the processor (**16 bit**, **32 bit**,or **64 bit**) and the compiler you use. On a 32-bit computer with an Intel Pentium processor, using modern compilers, integers are ***four***bytes.

***“When creating programs, you should never assume the amount of memory that is being used for any particular type.”***

***Whitespace -*** The compiler ignores whitespace **(spaces, tabs, line returns)** and so you can treat these as a **single line**. That’s why you need a ***“;”*** at the end of most lines.

**sizeof** - The ***“sizeof”*** is used like a function. When called, it tells you the **size** of the item you pass to it as a **parameter**.

**signed and unsigned**

***“Any integer without the word “unsigned” is assumed to be signed.”***

**signed integers** - can be **negative** or **positive**. For a **signed short**, half the numbers that can be stored are **negative**; thus, a **signed short** can only represent **positive** numbers **up** to **32,767**. The **signed short** can also, however, represent negative numbers giving it a total range from **–32,768** to **32,767**.

**unsigned integers -** are **always** **positive**. **unsigned short integer** can handle numbers from

**0** to **65,535**.

***“Largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer.”***

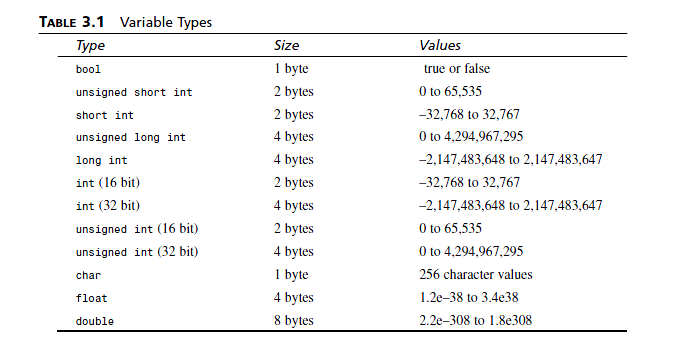
**Fundamental Variable Types**

**Floating-point variables** - have values that can be expressed as fractions—that is, they are real numbers.

**Character variables** - hold a **single** **byte** and are generally used for holding the **256 characters (0-255)** and **symbols** of the ***"ASCII”*** and ***"extended ASCII character”*** sets.

**ASCII -** The ASCII character set is the set of characters standardized for use on computers.

ASCII is an acronym for ***(American Standard Code for Information***

***Interchange)***. Nearly every computer operating system supports ASCII, although many support other **international character** sets as well.

**Defining a Variable**

***“Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program.”***

As a **general programming practic**e, avoid such horrific names as ***J23qrsnf***, and restrict **single-letter** variable names ***(such as x or i)*** to variables that are used only very briefly.

Try to use expressive names such as ***myAge*** or ***howMany***. Such names are easier to understand three weeks later when you are scratching your head trying to figure out what you meant when you wrote that line of code.

**Case Sensitivity**

C++ is **case sensitive**. In other words, **uppercase** and **lowercase** letters are considered to be ***different***. A variable named ***age*** is different from ***Age***, which is different from ***AGE***.

**Naming Conventions**

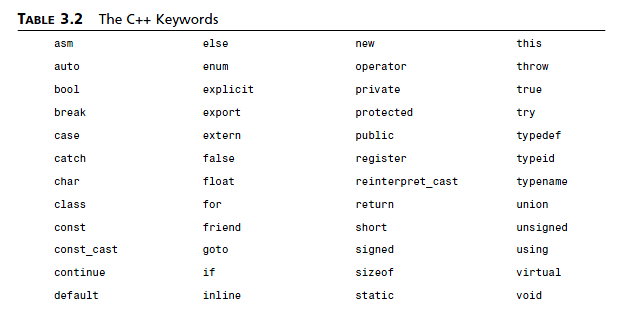
**Various conventions** exist for **how** to name variables, and although it **doesn’t much matter which method you adopt, *“it is important to be consistent throughout your program”.***

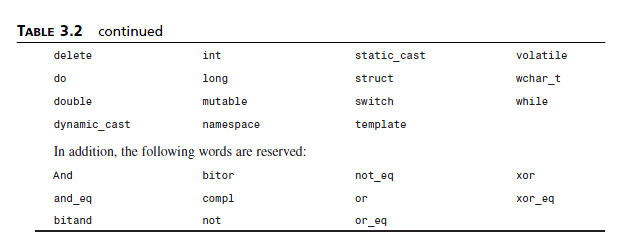
**Programmers prefer for variables**: int ***fred***; or int ***my\_car***

**Camel Notation**: int ***myCar***; or int ***myAge***;

**Hungarian Notation:** if **int** type (***iage***, ***iAge***, ***i\_age***) if **long** type (***lAge***, ***lage***, ***l\_age*** ,***l\_Age***)

**Keywords**

Some words are reserved by **C++**, and you cannot use them as variable names. These keywords have **special** meaning to the C++ compiler. **Keywords** include ***if***, ***while***, ***for***, and ***main***.



**Creating Aliases with typedef**

It can become tedious, repetitious, and, most important, error-prone to keep writing ***unsigned short int***. C++ enables you to create an alias for this phrase by using the keyword ***typedef***, which stands for ***type*** **definition**.

In effect, you are creating a **synonym**, and it is important to distinguish this from creating a new type. ***typedef*** is used by writing the keyword ***typedef***, followed by the ***existing type***, then the **new name**, and ending with a **semicolon**. For example, **“typedef *unsigned short int USHORT;“***

**When to Use short and When to Use long**

If any chance exists that the value you’ll want to put into your variable will be too big for its type, use a larger type.

**unsigned short integers -** assuming that they are two bytes, can hold a value only up to 65,535. **signed short integers -** split their values between positive and negative numbers, and thus their maximum value is only half that of the unsigned.

Although **unsigned long integers** can hold an extremely **large number** ***(4,294,967,295)***, that is still quite finite. If you need a larger number, you’ll have to go to ***float*** or ***double***, and then you lose some precision. **Floats** and **doubles** can hold extremely large numbers, but only the **first seven** or **nine digits** are significant on most computers. This means that the number is **rounded off after that many digits**.

**Wrapping Around an unsigned and signed Integer**

***Syntax for unsigned:***

*small number*: ***65535***

*small number*: ***0***

*small number*: ***1***

***Syntax for signed:***

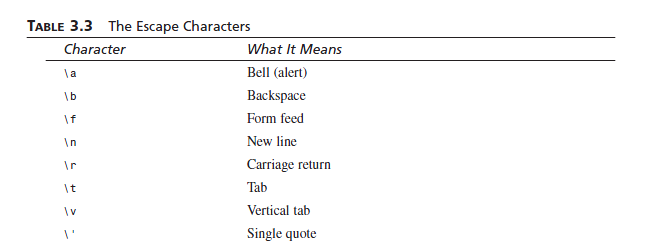
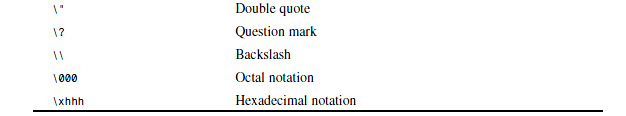
*small number*: ***32767***

*small number*: -***32768***

*small number*: -***32767***

**Special Printing Characters**

**Escape character (\)** - changes the meaning of the character that follows it. For example, normally the character ***n*** means the letter **n**, but when it is preceded by the escape character (/), it means new line.



**Constants**

Like variables, ***constants***are **data** storage locations. Constant variables **don’t change**. You must initialize a constant when you create it, and you **cannot** **assign** a new value later.

Constants cannot be changed **while** the program is **running**. If you need to

**change** the value of your variables, for example, you need to **change** the **code** and

**recompile** again.

***If you later decided to change the number of your variable, you could do so where you define the symbolic constant without having to make a change every place you used that value.***

**Literal Constants**

A ***literal******constant***is a value typed **directly** into your program wherever it is needed. For example:

**int** ***myAge*** = **39**;

***myAge*** is a variable of type **int**; **39** is a ***literal constant***. You **can’t** assign a value to **39**, and its value can’t be **changed**.

**Symbolic Constants**

A ***symbolic******constant***is a constant that is represented by a **name**, just as a variable is represented.

**students** = **classes** \* **studentsPerClass**;

**Defining Constants with #define**

***#define* studentsPerClass *15*** (without **declaring** a variable in the **main** **program**)

Every time the **preprocessor** sees the word ***studentsPerClass***, it puts in the text ***15***.

***“It should be avoided as it has been declared obsolete in the C++ Standard.”***

**Defining Constants with const**

***const unsigned short int* studentsPerClass *= 15;***

This method of declaring constants has several **advantages** in making your code **easier** to maintain and in **preventing** **bugs**. The **biggest** **difference** is that this **constant has a type**, and the compiler can enforce that it is **used** **according** to its ***type***.

***“const* is much better way exists to define constants*”.***

Enumerated Constants

**Enumerated constants** - enable you to **create** **new** **types** and then to **define** **variables** of those types whose values are **restricted** to a **set** of **possible** **values**.

**Syntax for Enumerated Constants:**

**enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };**

**(enum keyword) (new type) { (possible values) };**

**This statement performs two tasks:**

**1.** It makes **COLOR** the name of an **enumeration**; that is a **new** **type**.

**2.** It makes **RED** a **symbolic** **constant** with the **value** **0**, **BLUE** a symbolic constant with the value **1**, **GREEN** a symbolic constant with the value **2**, and so forth.

***enum* Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };**

Then, **RED** has the value **100**; **BLUE**, the value **101**; **GREEN**, the value **500**; **WHITE**, the value **501**; and **BLACK**, the value **700**.

**“Day 4: Creating Expressions and Statements”**

**Starting with Statements**

**Statement** - **controls** the **sequence** of **execution**, evaluates an **expression**. All C++ statements **end** with a **semicolon** and nothing else.

**Null Statement** – Does nothing.

**Blocks and Compound Statements**

**Block (Compound Statement)** - A block begins with an **opening brace ({)** and ends with a **closing brace (})** and does **not** **end** with a ***semicolon***.

**Expressions**

**Expressions** - Anything that evaluates to a **value** is an expression in C++. An expression is said to ***return*** a value. **All** expressions are statements.

**Working with Operators**

**Operator** - is a **symbol** that **causes** the compiler to take an **action**. Operators act on **operands**, and in C++ **any** expression **can** **be** an **operand**.

**The first two categories of Operators:**

**Assignment Operators (=)** - This operator causes the operand on the left side of the assignment operator to have its value changed to the value of the expression on the right side of the assignment operator.

**Mathematical Operators (+) (-) (\*) (/) (%)** - A second category of operators is the mathematical operators. Five mathematical operators are addition (+), subtraction (–), multiplication (\*), division (/), and modulus (%).

**l-values and r-values:**

An operand that legally can be on the **left side of an assignment operator** is called an

***l-value (Left-value)***. That which can be on the right side is called an ***r-value (Right-value).***

**Modulus (%)** - The fifth mathematical operator might be new to you. The modulus operator (%) tells you the remainder after an integer division. To get the remainder of 21 divided by 4, you take 21 modulus 4 (21 % 4). In this case, the result is 1.

Finding the modulus can be very useful. For example, you might want to print a statement on every 10th action. Any number whose value is 0 when you modulus 10 with that number is an exact multiple of 10. Thus 1 % 10 is 1, 2 % 10 is 2, and so forth, until 10 % 10, whose result is 0. 11 % 10 is back to 1, and this pattern continues until the next multiple of 10, which is 20. 20 % 10 = 0 again.

**Combining the Assignment and Mathematical Operators**

**Self-assigned:**

Self-assigned addition – ***(+=) : myAge = myAge + 2;*** // read as

Self-assigned subtraction – ***(-=) : myAge = myAge - 2;*** // read as

Self-assigned multiplication – ***(\*=) : myAge = myAge \* 2;*** // read as

Self-assigned division – ***(/=) : myAge = myAge / 2;*** // read as

Self-assigned modulus – ***(%=) : myAge = myAge % 2;*** // read as

**Incrementing and Decrementing**  
**Incrementing (++)**  – **Increasing** by 1.

**Decrementing** (--) – **Decreasing** by 1.

**Prefixing Versus Postfixing**

**Prefix Variety (++var / --var)** -The **prefix operator** is evaluated **before** the **assignment**; Increment the value in the variable **and** **then** fetch or **use** **it**.

**Postfix Variety (var++ / var--) -** The **postfix operator** is evaluated **after** the **assignment**; **Fetch** or **use** the value **and** then **increment** the **original** **variable**.

If ***x*** is an integer whose value is **5** and using a **prefix** **increment** **operator** you write:

**x** = 5; int **a** = ++**x**; you have told the compiler to **increment** ***x*** (making it **6**) and then fetch that value and assign it to ***a***. Thus, ***a*** is **now** **6** and ***x*** is **now** **6**. If, **AFTER** doing this, you **use** the **postfix** **operator** to write: int **b** = ***x***++;

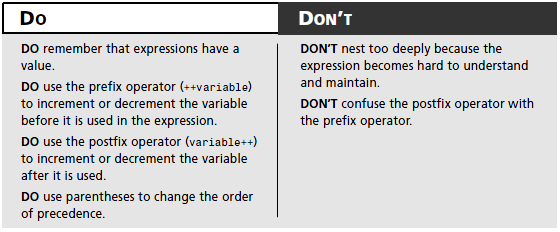
you have now told the compiler to **fetch** the value in ***x*** (**6**) and **assign** it to ***b***, and then **go**

**back** and **increment** ***x***. Thus, ***b*** is **now** **6**, **but** ***x*** is now **7**.

**Understanding Operator Precedence**

**Operator Precedence –** The **condition** ofbeing **more important** operator **than other** operator**.**

**Nesting Parentheses -** For **complex expressions,** youmight **need** to **nest parentheses** one **within another.** This **expression** is **easy** for **a computer** to **understand,** but **very difficult** fora **human** to **read, understand,** or **modify.**

****

The Nature of Truth

**Every** expression can be **evaluated** for its **truth** or **falsity**. **Expressions** that evaluate mathematically to **zero** **return** **false**; all others **return** **true**.

In **previous** **versions** of C++, all truth and falsity was represented by integers, but the

**ANSI** standard introduced the **type** ***bool***.

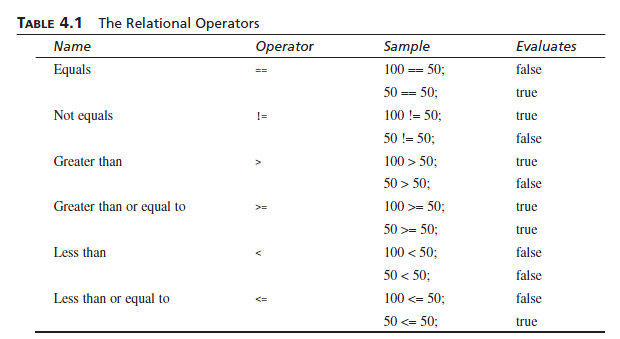
***bool*** - A ***bool*** can **only** have **one** of **two** values: ***false*** or ***true***.

**Evaluating with the Relational Operators**

**Relational** **Operators** - are used to **compare** **two numbers** to **determine** whether they are **equal** or if one is **greater** or **less** **than** the other. Every relational statement evaluates to either **true** or **false**.

The **six** **relational** **operators** are **equals** **(==)** , **less than** **(<)**, **greater** **than** **(>)**, **less** **than** or

**equal** **to** **(<=)**, **greater** **than** or **equal** **to** **(>=)**, and **not** **equals** **(!=)**.



**The *if* Statement**

**if** **Statement** - **enables** you to **test** for a **condition** *(such as whether two variables are equal)* and **branch** to **different** parts of your code, **depending** on the result. If the expression has the value ***false***, the statement is **skipped**. If it evaluates ***true***, the statement is **executed**.

**Syntax of the *if* Statement:**

***if*** (**expression**)

{ **statement**; // or many statements }

**The *else* Statement**

***else* Statement *-*** Often, your program **needs** to take one branch if your **condition** is ***true***, or another if it is ***false***.

**Syntax of the *else* Statement:**  
***if*** (**expression**) {

**statement1**;

***else***

**statement2**;

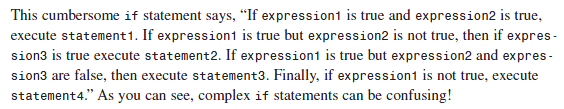
**next**\_**statement**;

}

Advanced if Statements

It is worth noting that any statement can be used in an ***if*** or ***else*** clause, **even** **another** ***if***

or ***else*** statement. Thus, you might see **complex** ***if*** **statements** in the following form:

***if* (expression1)**

**{**

***if* (expression2)**

**statement1;**

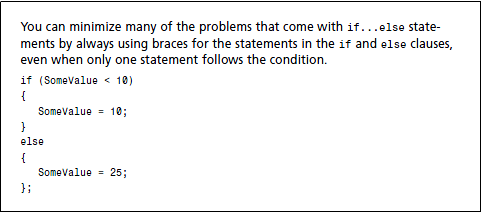
***else***

**{**

***if* (expression3)**

**statement2;**

***else***

**statement3;**

**}**

**}**

***else***

**statement4;**

**Indentation Styles**

**Styles:**

* Putting the initial brace after the condition and aligning the closing brace under the if to close the statement block:

if (expression){

statements

}

* Aligning the braces under the if and indenting the statements:

if (expression)

{ statements

}

* Indenting the braces and statements:

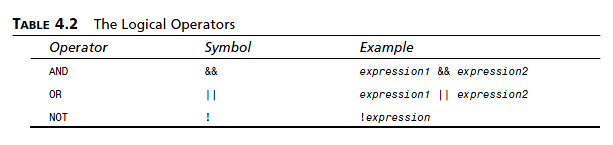
if (expression)

{

statements

}

**Using the Logical Operators**



**Logical *AND* Operator (&&)** - uses the ***AND*** **(&&)** operator **to** **connect** and **evaluates** **two** **expressions**. If **both** expressions **are** **true**, the **logical** ***AND*** **(&&)** statement is **true** as well.

**Logical *OR* Operator (||)** - evaluates two expressions. If **either** **one** is **true**, the expression is **true**.

**Logical *NOT* Operator (!)**- evaluates true **if** the expression being **tested** is **false**. Again, if the expression being tested is **false**, the value of the test is **true**!

