**C++ Guide**

**Numbers and Characters**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>signed int</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>short int</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>signed short int</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>long int</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0 to 4294967295</td>
</tr>
<tr>
<td>signed long int</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>double</td>
<td>-1.7E308 to 1.7E308</td>
</tr>
<tr>
<td>long double</td>
<td>-1.7E308 to 1.7E308</td>
</tr>
<tr>
<td>Data Type example:</td>
<td>char initial;</td>
</tr>
<tr>
<td>Data type each has</td>
<td>Each data type that has type of numbers</td>
</tr>
</tbody>
</table>
C++ has a large number of operators to assist in the simplification of code. Operators are read in order of precedence. If two operators are of the same precedence, the file reads from left to right. The following is a full list of all operators in order of precedence:

### Precedence Level

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>C++ scope access / resolution</td>
</tr>
<tr>
<td>()</td>
<td>Function call</td>
</tr>
<tr>
<td>[]</td>
<td>C++ indirect component selector</td>
</tr>
<tr>
<td>-&gt;</td>
<td>C++ direct component selector</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise (1's) complement</td>
</tr>
<tr>
<td>+</td>
<td>Binary plus</td>
</tr>
<tr>
<td>-</td>
<td>Binary minus</td>
</tr>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>%</td>
<td>Remainder (modulo)</td>
</tr>
<tr>
<td>++</td>
<td>Binary increment</td>
</tr>
<tr>
<td>--</td>
<td>Binary decrement</td>
</tr>
<tr>
<td>1</td>
<td>Left shift</td>
</tr>
<tr>
<td>2</td>
<td>Right shift</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>Conditional</td>
</tr>
</tbody>
</table>

#### Unary and Binary Operators

Unary operators define a single value while binary operators operate on two values. For example:

```cpp
int number = -1;
```

In the first line of this code, the unary "-" makes "1" a negative number. In the second line of code, the value of variable number2 uses a binary operator "-" to find the sum of two numbers.

### Assignment Operator

The "=" sign is the assignment operator. As shown in other sections, it assigns a value or expression to a variable. Multiple assignments can be used to shorten code. For example:

```cpp
age1 = 21;
age2 = 21;
age3 = 21;
or
age1 = age2 = age3 = 21;
```

Note: The above example is an interesting way to shorten code, but code is still more understandable when avoiding multiple assignments.

### Compound assignment operators

Compound assignment operators are designed to simplify the adjustment of variables during program execution. A compound assignment operator takes the variable on the left and subjects it to the value on the right, giving a new value for the variable. For example:

```cpp
b = b + 100;
```

Both statements above produce the same answer.

### Adding and Subtracting One

The increment and decrement operators are similar to compound assignment operators. The variable is subject to an increase or decrease of 1. For example:

```cpp
++q;
```

Both statements above produce the same answer.

### Typecast Operators

Typecast operators change a variable from one data type to another. A typecast operator is an existing data type with the keyword `static_cast`. For example:

```cpp
static_cast<int>(q)
```

When placed in C++ code, the operator looks like this:

```cpp
int q = 17;
```

```cpp
int q = 17;
```

or

```cpp
q++;`
```

or

```cpp
q = q + 1;
```

All four examples will produce the same answer, 18.

### Arrays

An array is a defined number of memory slots for a variable’s value. For example:

```cpp
char yourName[4] = “Don”;
```

When the array is placed in memory it looks like this:

```
[0] D
[1] o
[2] n
```

Note: Make sure space is left for the null zero (page 1).

### All programming languages have if statements. In C++, if statements are designed to test operators. The answer from an if statement will determine which part of the program to execute next. Most if statements are based on relational test such as:

```cpp
if (age < 22)
```

The else statement will tell the program what to do when an if statement is false. Add the else statement after the if statement’s closing blocks like this:

```cpp
if (age < 22)
```

The else statement only have one or two options (true or false). Nesting statements make the program choose between three or more potential options. Nesting statements means placing a statement inside of a statement.

**Code Snippet:**

```cpp
void main()
{
    int age;
    cout << “How old are you?”;
    cin >> age;
    if (age < 10)
    {
        cout << “You are younger than 10, but not 7 years old.”;
    }
    else
    {
        cout << “You are 7 years old.”;
    }
}
```

**Output:**

```
How old are you?
You are 7 years old.
```

### Sequences

A sequence point is a point in the program where the expressions in all threads are guaranteed to be evaluated before the program continues.

**Code Snippet:**

```cpp
int age = 5;
age += (-5+11)/6+2;
cout << “In three years I will be ” << age << endl;
```

**Output:**

```
In three years I will be 8.
```
**switch Statements**

if/else statements are best used with C++ code which must choose between two options. In an earlier section (page 2), nested if/else statements were discussed. The problems with if/else statements are:

1. The more nesting used, the closer the code gets to the right margin.
2. Changing an extensive nested if/else statement is not easy.

This is where switch statements come in handy. A switch statement works the same as an if/else statement, by testing values using relationship operators. The main differences are:

1. Switch statements only search for one matching answer. Once the answer is found, the program stops looking.
2. If the program has no matching answer, the default statement is used. The default statement is added by the user.
3. When editing the statements, it is easier to make changes to switch statements when compared to nested if/else statements.
4. A switch is controlled by just one integer or character value instead of a logical test.

Here is an example of a switch statement:

```cpp
class SwitchStatementsLoops
{
public:
    // Example of a switch statement.
    void main()
    {
        switch (age)
        {
            case 1:
                cout << "You are 1 year old.";
                break;
            case 2:
                cout << "You are 2 years old.";
                break;
            case 3:
                cout << "You are 3 years old.";
                break;
            default:
                cout << "You are older than 3 years old.";
        }
    }
}
```

**The Break Statement**

In the above example the break command appeared at the end of every case. This causes the program to go to the end of the switch statement. If the break was not there, the output would be:

Assuming the user typed a 1:
- You are 1 year old.
- You are 2 years old.
- You are older than 3 years old.
Assuming the user typed a 3:
- You are 3 years old.
- You are older than 3 years old.

A loop is the repeated execution of the same set of programming instructions. To stop the repeat, a variable must be added, otherwise the program will loop indefinitely. The count variable in a control relationship is the standard variable used to stop a loop.

### The while Loop

A while Loop uses a relationship test to stop it from looping. Once the relationship is false, the loop is ended. The following example prints Quick Study 5, times:

```cpp
class ForLoop
{
    int count = 0;
    do
    { for (startExpression; conditional; countExpression)
    {
        cout << "Quick Study" << endl;
        count++;    
    }
    while ((height < 3.5) || (height > 9.5));
    cout << "Tell me the truth...";
    while ((height < 3.5) || (height > 9.5))
    { cout << "You can't be that size!" << endl;
        cout << "Tell me the truth...";
    }
    while ((height < 3.5) || (height > 9.5));
    if (height <= 4.5)
    { cout << "You must be at least 4.5 feet tall to go on this ride." << endl;
        cout << "Sorry!" << endl;
    }
    else
    { cout << "Enjoy the ride.";
        return;
    }
}
```

### The continue Command

The continue command is the opposite of a break command. The continue command only works on loops and switch statements. An example using a break command was shown in the first column of this page. Note: The exit() function will stop the program wherever the exit() is. To use the exit() function, the STDLIB.H file must be included.

**return Command**

The return command will stop a function before it would normally end. An example using a return command was shown in the second column of this page.

### break Command

The break command will stop a loop before it would normally end. A break command only works on loops and switch statements. An example using a break command was shown in the previous page. Note: The exit() function will stop the program wherever the exit() is. To use the exit() function, the STDLIB.H file must be included.

**endl stands for end line. This command, placed at the end of a cout, forces a hard return when the program is executed.**
Functions are a good way to break down a program. C++ was designed to create large programs from little programs. Each function should be a self-contained mini-program. Mini-programs aren’t necessary, but will help the organization of programs.

Note: A structured program is set up with a single function for every task.

The main() function, in a perfect program, should only be a starting or controlling function for all other functions.

All new functions must be defined during the function in which it will be used. When a function is used in another it is referred to as “calling”. When C++ calls a function, the new function gains control until the code has been read, then the original function regains control until the next function is called.

Note: Calling functions could create the same problem as an infinite loop. If one function calls itself or if two functions call each other, that is called recursion. Recursion may cause the program to never end.

Some pre-made functions need special #include lines at the top of the program. For example, strcpy() needs this statement:

```
#include <string.h>
```

Local and Global Variables
Any variable can be placed inside any function in any program. When a variable is defined inside a function, it is considered a local variable. A local variable only exists while its defining function’s block exists. A function block is the code between a function’s brackets. This is an example of a local variable:

```
void main()
{
    int money = 5;
}
```

Integer money is a local variable to function main(). After the second bracket, function main ends, and integer money stops existing.

Global variables are defined after a function ends and before the next function begins. Usually, a global variable is defined before the main() function. These functions will exist from their defining point to the end of the program. They can be used by any function during the program. This is an example of a global variable:

```
int money = 5;
```

void main()
{
    
}

Global variables are very visible to all functions, whether the functions need the variable or not. A local variable places functions on a need-to-know basis.

Code Example:
```
#include <iostream.h>
int drinkAge = 21;

void main()
{
    int yourAge = 15;
cout << "You are" << yourAge << " and not old enough to drink," << drinkAge << "," << endl;
    int momAge = 38;
cout << "Your mom is" << momAge << " She is old enough to drink" << drinkAge << "," << endl;
}
```

Output:
```
You are 15 and not old enough to drink, 21.
Your mom is 38. She is old enough to drink, 21.
```

Data or values can be shared (passed) between two functions. The value being passed is called an argument. The receiving variable is called the parameter. To define the passed value in a new function, the parameters must be placed, in parentheses, on the new function’s definition line. The definition line is the first line of the function.

This is an example of value passing:
```
void FindMe (p4, p5)
{
    the first line of the above example calls for the FindMe function and passes two values. Line two separates two functions. Lines three and four are the first two lines of the FindMe function.
}
```

All functions in C++ must have a prototype. A prototype is what declares a function. The prototype is usually placed at the top of the code, before main().

The #include line at the beginning of all programs is a header file. These header files are prototypes for library functions, such as, strcpy().

There are three different ways to pass values and expressions from one function to the next:
1. by value
2. by address
3. by reference

Passing by Value
In this form of passing, the value is passed on to the next function, but the variable is not. If any changes occur to the value once it has been passed, the changes will not affect the original variable. The receiving function looks at this value as if it were a local value.

Passing by Address
Passing by address means that the entire variable is moved from one function to the next. An address is a variable’s location in memory. Address passing is most useful when passing an array. The address held by the array will also move.

Passing by Reference
This form of value passing is designed to pass non-arrays. Reference passing works the same as address passing except reference passing doesn’t work with arrays.

Code Example:
```
#include <iostream>

using namespace std;

void GetValue(int height);
void GetAdd(char name[10]);
void main()
{
    int height;
    char name[10];
    int age;
cout << "How old are you?";
    cin >> height;
    cin >> name;
    GetValue(height);
    GetAdd(name);
}
```

```
int RemoveAge(int age)
{
    int ageDif = RemoveAge(age);
    cout << "How old are you?";
    cin >> age;
    GetValue(height);
    return;
}
```

```
void GetRef(int &age)
{
    name[6] = '\0'
}
```

```
void GetAdd(char name[10])
{
    name[4] = 'e';
    name[5] = 'r';
    name[6] = '\0'
}
```

```
void main()
{
    int RemoveAge(int age)
    {
        return (100 - age);
    }
```

```
void GetValue(int height)
{
    height += 3;
cout << "If you grow a half an inch a year, you will be" << height << "in six years." << endl;
    return;
}
```

```
void GetRef(int &age)
{
    age += 6;
    return;
}
```