CISC 192 Reference

Alexandra Steiner

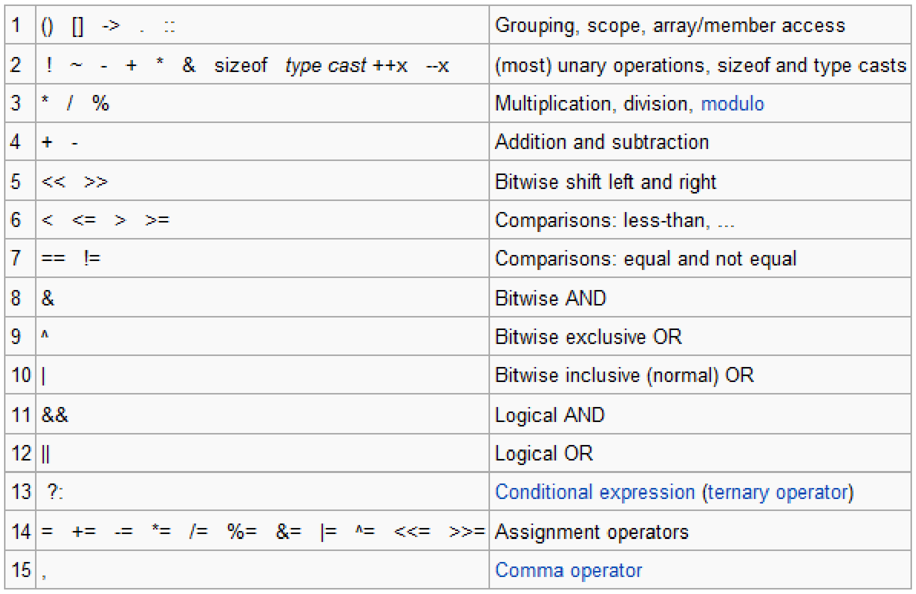
March 2, 2020

Contents

1 General IO

- `#include <iostream>` IOStream package (members are in `namespace std`)
  - `std::cout << output1 << output2 << ...` - Prints outputs to output stream
  - `std::cout.put(someOutput)` - Prints a single, unformatted character to output stream. `someOutput` is of type `char`.
  - `std::cin >> var` - Gets input from the input stream and stores it into var. Input is cast to type of var.
  - `char c = std::cin.get()` - Gets a single, unformatted character from the input stream.
- `#include <stdio.h>` Standard IO Library
  - `printf("myInt:  %i, myString %s", 40, "hello")` - Prints text to the standard output stream (usually the terminal). Text is formatted according to argument 1 (a `const char*` type string) and arguments are specified according to in-text specifiers indicated by '%' symbols. Any arguments specified in the format string must be added as additional arguments to the `printf` function. For more details, see `http://www.cplusplus.com/reference/cstdio/printf/`
  - `sprintf(buffer, "myInt:  %i", anotherInt)` - Formats text using the same syntax as `printf` but writes the completed characters into `buffer` (`buffer` is of type `char*`)
  - `scanf("%i", & myInputInt)` - Reads the data from the standard input stream, formatted according to argument 1, into variable addresses specified in following arguments. For more details, see `http://www.cplusplus.com/reference/cstdio/scanf/`

2 Operators and Expressions



1	() [] -> . ::	Grouping, scope, array/member access
2	! ~ - + * & sizeof *type cast* ++x --x	(most) unary operations, sizeof and type casts
3	* / %	Multiplication, division, modulo
4	+ -	Addition and subtraction
5	<< >>	Bitwise shift left and right
6	< <= > >=	Comparisons: less-than, ...
7	== !=	Comparisons: equal and not equal
8	&	Bitwise AND
9	^	Bitwise exclusive OR
10	|	Bitwise inclusive (normal) OR
11	&&	Logical AND
12	||	Logical OR
13	?:	Conditional expression (ternary operator)
14	= += -= *= /= %= &= |= ^= <<= >>=	Assignment operators
15	,	Comma operator

Figure 1: Order of Operations in C++. Graphic pulled from Wikipedia

2.1 Bitwise Operators

```
unsigned char X = 55; // 00110111
unsigned char Y = 15; // 00001111
```

- `unsigned char Z = X & Y` - Logical AND between X and Y. Z is now 00000111
- `unsigned char Z = X | Y` - Logical OR between X and Y. Z is now 00111111
- `unsigned char Z = Y Ŷ` - Logical XOR between X and Y. Z is now 00111000
- `unsigned char Z = X >> 1` - Bit shift in the direction of the LSB. Z is now 00011011
- `unsigned char Z = X << 1` - Bit shift in the direction of the MSB. Z is now 01101110
- `unsigned char Z =  X;` - Logical NOT. Z is now 11001000

3 Control and Loop Structures

3.1 If-Then-Else

```
if (<Some logical expression>) {
        // Do some stuff
}
else if (<Some other expression>) // Do other stuff;
else {
        // more stuff here
}
```

3.2 Switch

```
switch(<some value>) {
        case 0:
                \\ Case 1
                break;
        case 1:
        case 2:
                \\ This executes for cases 1 and 2!
                break;
        default:
                \\ Do this if all else fails
}
```

Note: great for when there are a bunch of known possible values. Essentially shorthand for a long if-then-else statement

3.3 Ternary Statement

```
X = <expression> ? <return if true> : <return if false>
```

Note: great for simple if else statements like when toggling a variable between two values

3.4 While Loop

```
int X = 0;
while (X < 10) {
        // Do some stuffs
        X++;
}
```

3.5 For Loop

```
for (int X = 0; X < 10; X++) {
        // Do some stuffs
}
```

4 Functions

4.1 General Function Structure

```
<return type> functionName(<type> arg1, ...) {
        // Do something
        return <value to be returned>;
}
```

4.2 Arguments by Value

```
int add(int arg1, int arg2) {
        return arg1 + arg2;
}

int X = 10;
int Y = 10;
int Z = add(X, Y);
```

4.3 Arguments by Pointer Value

```
void add(int* arg1, int* arg2, int* sum) {
        *sum = *arg1 + *arg2;
}

int X = 10;
int Y = 10;
int Z = 0;
add(&X, &Y, &Z);
```

Note: passing arguments by pointer value (address) only has to transfer the size of the pointer itself (usually 4-8 bytes depending on the machine). This can be dramatically faster when passing in large arrays of values such as strings or number lists.

4.4 Arguments by Reference

```
void add(int& arg1, int& arg2, int& sum) {
        sum = arg1 + arg2;
}

int X = 10;
int Y = 10;
int Z = 0;
add(X, Y, Z);
```

Note: passing by reference is only shorthand for passing arguments by pointer value (address). However, it can be confusing to work with since the syntax are nearly the same as passing by value so don't use them unless you know what you're doing.

5 Objects

5.1 Structs

Basic implementation of a struct:

```
struct Vector3D {
        double x;
        double y;
        double z;
};

Vector3D vect;
vect.x = 10;
cout << vect.x << endl;
```

Note: struct member properties cannot be initialized to a value *unless* they are a static member property

Implementation of a struct with an overloaded constructor function:

```
struct Vector3D {
        double x;
        double y;
        double z;
        Vector3D() {
                x = 0;
                y = 0;
                z = 0;
        }
        Vector3D(int X, int Y, int Z) {
                x = X;
                y = Y;
                z = Z;
        }
};

Vector3D vect1; // Called with default constructor
Vector3D vect2(1,2,-1);
```

Example of a struct with static member properties:

```
struct Student {
        static s_value;
};

cout << Student::s_value << endl;
```

Note: the static keyword associates a member property or method to the class itself instead of a specific instance of a class.