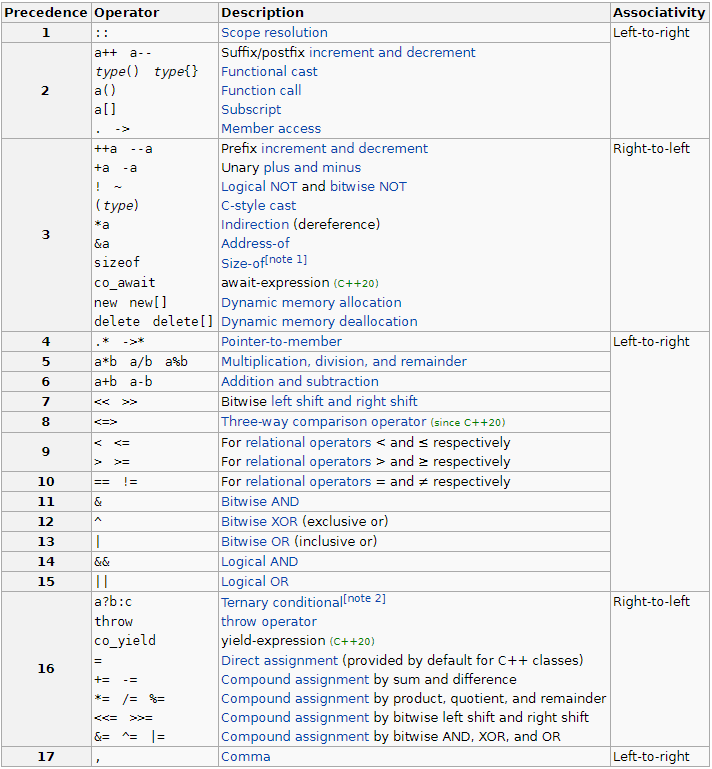
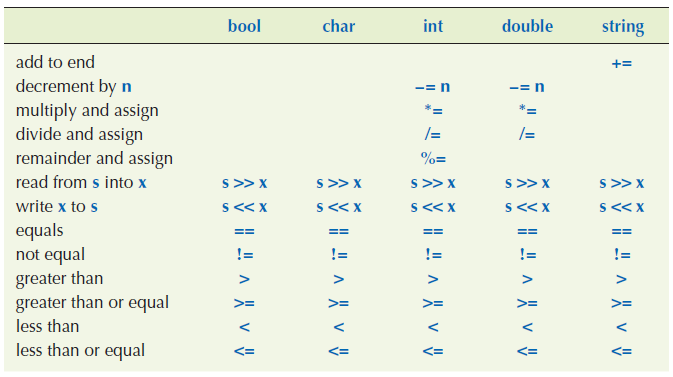
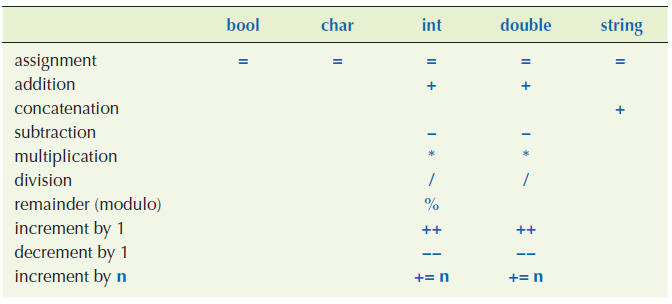
**ONLY ADD THINGS THAT ARE DIFFERENT FROM C AND JAVA.**

**Part 1**

* **Operators**



**-Operation vs operator:** Operation is the english name of the term; “assignment”, addition”, etc. Operator is the programming language symbol of the term; “=”, “+”.



* **Data Types**

-An object is a region of memory with a type that specifies what kind of information can be placed in it. A named object is called a variable.

-You can assign a variable its maximum or minimum value. (#include <limits>)

double smallest = std::numeric\_limits<double>::min();

double largest = std::numeric\_limits<double>::max(); **-Data Type Conversions**

In order to cast using safe or unsafe conversion, you can do things like multiply by 1.0 or 1.  
  
The type{value} notation prevents narrowing, but the type(value) notation does not. We can use both of these notations both for assignment and for explicit casting(without explicit casting, data type change will be done implicitly when necessary).  
  
 double d (5.2); // Assignment  
 double d {5.2}; // Assignment  
 int d {5.2}; // Error. Requires narrowing conversion.  
  
 int i = int(5.2); // Unnecessary since unsafe conversion will be done anyway.  
 int i = int{5.2}; // Error. Requires narrowing conversion.  
  
 **-Safe Conversions**  
  
bool to char/int/double, char to int, double, int to double.

The rule (when we are only using bool, char, int, double, string) is that if an operator has an operand of the double, we use floating-point arithmetic yielding a double result; otherwise, we use integer arithmetic yielding an int result.  
  
 5 / 2 is 2. Integer arithmetic.  
  
 2.5 / 2 is 1.25. Floating-point arithmetic. 2 is promoted to 2.0.  
  
 ‘A’ + 1 is 66. Integer arithmetic. ‘A’ is promoted to 65. **-Unsafe/Narrowing Conversions**double to int/char/bool, int to char/bool, char to bool.  
  
 int i1 = 2,5 / 2; // Floating-point arithmetic. 2 is promoted to 2.0. 1.25 is // demoted to 1. i is 1.  
  
An example to unsafe int to char conversion.   
  
 int x = 2000;

char c = x;  
  
 2000 % 256 = 208 // Every 256 we add to char will overflow us back to 0.  
 208 – 128 = 80 // We are at 0. We need to add 208. Lets add 128 first and overflow to -128.  
 -128 + 80 = -48 // We are at -128. We add the remaining 80 and get to -48.

-In order to see what really happens, open programmer mode of calculator, enter 2000. Go to inspect bit bit bit mode. Switch to 32 bit(word) mode. As you can see ...0000 0111 1101 0000 is positive 2000. When you take this 32 bit value and assign it to an 8 bit variable, only the 8 least significant bits are copied(switch to 8 bit(byte) mode). Which means the 8 bit variable we have has the bit value 1101 0000 which is negative 48.

**-Default values**Global variables have the default value of 0.  
  
Strings have the default value of empty string “”.  
  
Vector elements get their default values. For example 0 for ints.  
  
Class members are uninitialized unless you provide an initializer (or a default constructor), unlike Java.

**Part 2**

* **Keywords**

**-constexp:** A constexp symbolic constant must be given a value that is known at compile time.

**-const:** A const variable is initialized with a value that is not known at compile time but never changes after initialization.

**Part 3**

* **Input/Output**

-The name cout refers to a standard output stream. Characters “put into cout” using the output operator (<<) will appear on the screen.

std::cout << "Hello, " << first\_name << "!\n";  
  
 std::cout << "Enter an amount of money followed by a currency "

<< "(y, k, p)> ";  
  
 std::cout << "Enter an amount of money followed by a currency "

"(y, k, p)> ";  
  
-The name cin refers to the standart input stream. The second operand of the get from operator (>>) specifies where that input goes.  
  
 std::cin >> first\_name;  
  
-The input operation is sensitive to type; that is, it reads according to the type of variable you read into.  
  
 std::cin >> first\_name;  
 std::cin >> age;  
  
 So if you type in Carlos 22 the >> operator will read Carlos into first\_name, 22 into age.

-Reading of strings is terminated by whitespace.

-We can read several values in a single input statement.  
  
 std::cin >> first\_name >> age;

-In order to continue input as long as there is input we use the following. Also, if you want to stop for ceartin values, you can say while (cin>>n>>v && n!="NoName")

while (std::cin >> n)  
  
 while (std::cin >> operation >> operand1 >> operand2)  
  
-In order to test if the last input operation succeeded,  
  
 double d = 0;  
 std::cin >> d;  
 if (std::cin) or if (!std::cin)

-cin stops reading when it comes across a white space character. If you want to read until the end of the line, use getline which is part of <string> header.

getline(cin, name);

* **Control Flow**

**-Conditional Operator:** We often use conditional operator to write concise code. C++ exercise 4.11 is an example to this.

std::cout << "\nYou have " << pennyNumber << " penn" << ((pennyNumber == 1) ? "y." :   
 "ies.");

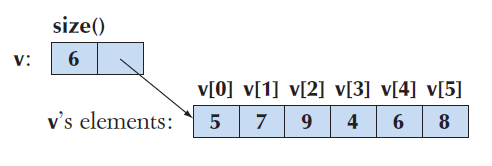
**-switch**The value on which we switch must be of an integer, char or enumeration. In particular, you cannot switch on a string. For strings you can use if statements.

**-for**

-Using multiple counters  
  
 for (int counter = 0, rightCounter = ((vector1.size() - 1) - counter);   
 counter < rightCounter; counter++)  
 {  
 std::swap(vector1[counter], vector1[rightCounter]);  
 }

* **Vector** (#include <vector>)

std::vector<int> v = { 5, 7, 9, 4, 6, 8 };  
   
 std::vector<int> vi(6, 2); // initialized with custom value 2  
 std::vector<std::string> vs(4); // initialized with default value “”  
  
v.size() gives vector size.  
  
 for (int i = 0; i < v.size(); i++)  
 std::cout << v[i] << ‘\n’;



range-for-loop (tag: range for loop)  
  
 for (int x : v)  
 std::cout << x << ‘\n’;

Pass by value, reference, and const reference don’t just apply to function parameters. They apply to range for loop too. You can pass the values from right side to left side by value, reference, or const reference.

char arr[10]{ '2', '4', '6', '8' };

for (char c : arr)

{

c = 't'; // Modifies pass by value copies.

printf("c = %c\n", c);

}

for (char& c : arr) // Prints original values.

{

printf("c = %c\n", c);

}  
  
v.push\_back() adds a new element to a vector. The new element becomes the last element of the vector.  
  
 v.push\_back(2);  
  
std::sort() sorts the vector. (#include <algorithm>)  
  
 std::sort(v.begin(), v.end());

* **Errors** [See error section of "C Notes" file](file:///C:\Computer%20Engineering\6-%20C\3-%20Notes\C%20Notes.docx)

-When we are checking arguments of a function, that check should usually be done in the function, not by the function caller.

-C++ provides a mechanism to help deal with errors: exceptions. The fundamental idea is to seperate detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored; that is, exceptions provide a mechanism that allows us to combine the best of the various approaches to error handling.  
  
 class Bad\_area { };

int area(int length, int width)

{

if (length <= 0 || width <= 0)

throw Bad\_area{};

}

int main()  
 {

try {

int x = -1, y = 2, z = 4;

int area1 = area(x, y);

int area2 = framed\_area(1, z);

int area3 = framed\_area(y, z);

double ratio = area1 / area3;

}

catch (Bad\_area) {

std::cout << "Bad arguments to area()\n";

}  
 }  
  
-cerr is exactly like cout except that it is meant for error output. By default both cerr and cout write to the screen, but cerr isn’t optimized so it is more resilient to errors, and on some operating systems it can be diverted to a different target, such as a file. Using cerr also has the simple effect of documenting that what we write relates to errors.  
 The call e.what() extracts the error message from the runtime\_error.  
  
 try {

throw std::runtime\_error("test");

}

catch (std::runtime\_error& e) {

std::cerr << "runtime error: " << e.what() << ‘\n’;

return 1;

}  
  
-You can add multiple strings to your error message, including user inputted strings.  
  
 std::string s = "t";  
 throw std::runtime\_error("Delta is negative, " + s);  
  
-Just like Java, superclass exception types catch their subclass exception types. In C++ we can use catch(...) to catch exceptions of any type whatsoever. [C++ Exception hierarchy](https://en.cppreference.com/w/cpp/error/exception)

* **Functions**

**-main Function**

- void main() is explicitly prohibited by the C++ standard and shouldn't be used. The valid C++ main signatures are:  
  
 int main()  
  
and  
  
 int main(int argc, char\* argv[])  
  
which is equivalent to  
  
 int main(int argc, char\*\* argv)  
  
In C++, main function can be left without a return value at which point it defaults to returning 0.

**-Header Files:** In principle, #include “file.h” simply copies the declarations from file.h into your file and the point of the #include. [How to call on a function found on another file](https://stackoverflow.com/questions/15891781/how-to-call-on-a-function-found-on-another-file)  
  
A header will typically be included in many source files. That means that a header should only contain declarations that can be duplicated in several files (such as function declarations, class definitions, and definitions of numeric constants). [How to use extern variables](https://stackoverflow.com/questions/10422034/when-to-use-extern-in-c?noredirect=1&lq=1)  
  
**-Scope:** A scope is a region of program text.An identifier is declared in a scope and is valid (is “in scope”) from the point of its declaration until the end of the scope in which it was declared. If it was a variable declared in a while statement, it will be destroyed when we reach the end of the iteration to check the loop continuation condition.

Lets say we have three functions primary, term and expression. Primary uses expression in it, term uses primary in it and expression uses term in it. So defining these three functions by themselves will give an error because when we are parsing the primary function, the expression function wont be declared. In order to fix this, we can add a forward declaration for expression on top.  
  
 double expression();  
  
 double primary() {//... expression(); //...}  
 double term() {//... primary(); //...}  
 double expression() {//... term(); //...}  
  
Names in a scope can be seen from within scopes nested within it. The global scope is the scope that’s not nested in any other.  
  
There are several kinds of scopes that we use to control where out identifiers can be used.  
  
 -The global scope: the area of text outside any other scope.  
 -A mamespace scope: a named scope nested in the global scope or in another namespace.  
 -A class scope: the area of text within a class.  
 -A local scope: between {...} braces of a block or in a function parameter list(parameters are in function scope too).  
 -A statement scope: e.g., in a for-statement.

-Nested blocks: You can use braces {} to create a scope in a scope.  
  
 int x;  
 x = 7;  
 {  
 int x = y;  
 ++x;  
 }  
  
**-Function call and return**  
  
In C++, you can do function overloading in addition to method overloading. In Java everything has to be in a class so there are no functions.  
  
**-Pass by value:** You can pass variables or literals when using pass by value.  
  
 void print(std::vector<double> v)

{

std::cout << '{ ';

for (int i = 0; i < v.size(); ++i) {

std::cout << v[i];

if (i != v.size - 1) std::cout << ", ";

}  
 std::cout << " }\n";

}  
  
**-Pass by const reference:** Pass by value is simple, straightforward and efficient when we pass small values, such as an int, a double or an object. But what if a value is large, such as an image, a large table of values or a long string?  
Adresses of these large structures is called reference.  
You can pass variables or literals when using pass by const reference.  
  
 void print(const std::vector<double>& v)

{

std::cout << '{ ';

for (int i = 0; i < v.size(); ++i) {

std::cout << v[i];

if (i != v.size - 1) std::cout << ", ";

}

}  
  
The & means reference and the const is there to stop print() modifying its argument by accident.  
  
**-Pass by reference:** You have to pass a variable when using pass by reference.  
  
 void init(std::vector<double>& v)

{

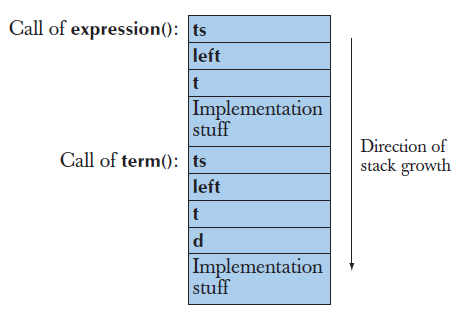
for (int i = 0; i < v.size(); ++i) v[i] = i;

}  
  
int& is a reference to an int, so we can write  
  
 int i = 7;

int& r = i; // r is a reference to i

r = 9; // i becomes 9

i = 10;

std::cout << r << ' ' << i << '\n'; // write: 10 10  
  
**-Rule of thumb**1­- Use pass by value to pass very small objects (1-2 ints, 1-2 double, etc).  
2- Use pass by const reference to pass large objects that you dont need to modify.  
3- Return a result rather than modifying an object through a reference argument.  
4- Use pass by reference only when you have to.  
  
**-Argument checking and conversion:** If you really mean to truncate a value, say so explicitly.  
  
**-Function call implementation:** When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and space for all local variables. Local variables are initialized only if we execute the statement in the function that initializes the variable.  
  
In java we called it method call stack and stack frame. In C++ they are called stack or call stack and activation record respectively. Call stack is composed of activation records.  
  
  
  
**-constexpr funtions:** A constexpr function behaves just like an ordinary function until you use it where a constant is needed. Then it is calculated at compile time provided its arguments are constant expressions.  
  
 constexpr double xscale = 10; // scaling factors  
 constexpr double yscale = 0.8;  
  
 constexpr Point scale(Point P) { return {xscale \* p.x, yscale \* p.y}; };  
  
 void user(Point p1)  
 {  
 constexpr Point p2 {10,10};

Point p3 = scale(p1); // OK: p3 == {100,8}; run-time evaluation is fine

constexpr Point p4 = scale(p2); // p4 == {100,8}

constexpr Point p5 = scale(p1); // error: scale (p1) is not a constant

// expression

constexpr Point p6 = scale(p2); // p6 == {100,8}  
 }  
  
In C++11, a constexpr function must have a body consisting of a single return-statement (like scale()); in C++14, we can also write simple loops. A constexpr function may not have side effects; that is, it may not change the value of variables outside its own body, except those it is assigned to or uses to initialize.  
  
 int gob = 9;

constexpr void bad(int & arg) // error: no return value

{

++arg; // error: modifies caller through argument

glob = 7; // error: modifies nonlocal variable

}  
  
**-Order of evaluation**  
  
 **-Expression evaluation:** If you change the value of a variable in an expression, dont read or write it twice in that same expression.  
  
 **-Global initialization:** Global variable and namespace variable initialization takes place before the code in main() is executed.  
Avoid using global variables, giving them short identifiers, using complicated initialization for them(non constant expression) as much as possible.  
  
 const Date default\_date(1970, 1, 1);  
  
Can we know that default\_date is never used before it is initialized? No, so we shouldnt write that definition. Instead call a function that returns a value.  
  
 const Date default\_date()  
 {  
 return Date(1970, 1, 1);  
 }  
  
This constructs the Date every time we call default\_date().  
  
 const Date& default\_date()

{

static const Date dd(1970,1,1); // initialize dd first time we get here

return dd;

}The static local variable is initialized (constructed) only the first time its function is called. Note that we returned a reference to eliminate unnecessary copying and, in particular, we returned a const reference to prevent the calling function from accidentally changing the value. The arguments about how to pass an argument (§8.5.6) also apply to returning values (we can return by value, reference or const reference).  
  
**-Namespaces**In C++namespaces are just about partitioning the available names. Java packages are about modules. The naming hierarchy is just one aspect of it.  
  
For example, we might like to provide a graphics library with classes called Color, Shape, Line, Function and Text.  
  
 namespace Graph\_lib {

struct Color { /\* . . . \*/ };

struct Shape { /\* . . . \*/ };

struct Line : Shape { /\* . . . \*/ };

struct Function : Shape { /\* . . . \*/ };

struct Text : Shape { /\* . . . \*/ };

// . . .

int gui\_main() { /\* . . . \*/ }

}  
  
Most likely somebody else in the world has used those names, but now that doesn’t matter. You might define something called Text, but our Text doesn’t interfere. Graph\_lib::Text is one of our classes and your Text is not. We have a problem only if you have a class or a namespace called Graph\_lib with Text as its member.  
  
A name composed of a namespace name (or a class name) and a member name combined by :: is called a fully qualified name.  
  
 **-using declarations and using directives:** Instead of using the fully qualified name “std::string”, “std::cout”, we can say “by string i mean std::string” and “by cout i mean std::cout”, etc. using a using declaration.  
  
 using std::string;

using std::cout;  
We could also say “if you dont find a declaration for a name in this scope, look in std” using a using directive.  
  
 using namespace std;  
  
It is usually a good idea to avoid “using directives” because you lose track of which names come from where, so that you again start to get name clashes.

namespace X

{

int var;

void print()

{

std::cout << var << '\n';

}

}

namespace Y

{

int var;

void print()

{

std::cout << var << '\n';

}

}

namespace Z

{

int var;

void print()

{

std::cout << var << '\n';

}

}

int main()

{

X::var = 7; // Sets X::var to 7.

X::print(); // Calls X::print

using namespace Y; // Switches to namespace Y

var = 9; // Sets Y::var to 9

print(); // Calls Y::print

{

using Z::var; // Switches to using Z's var in this scope

using Z::print; // Switches to using Z's print in this scope

var = 11; // Sets Z::var to 11

print(); // Calls Z::print

}

print(); // Calls Y::print

X::print(); // Calls X::print

}

**-argument dependent lookup:** When you use the sort function(algorithm.h), you don’t need to say std::, but when you use cout, numeric\_limits (limits.h), vector(vector.h), string you need to use std::. This is because you give a vector as argument to the sort function call and because of argument dependent lookup, you dont need to use std:: for that sort function call.

* **­Classes**

class Token {  
 public:  
 char kind;  
 double value;  
 };  
  
 Token t1{'+'};  
 Token t2{'8', 11.5};

* **String**

-You can use strings as a char array.  
  
 std::string s = "Success!\n";

for (int i = 0; i < 6; ++i)

std::cout << s[i];  
  
-Strings have default values ( “” ) like vectors.