Table of Contents

Introduction

The pseudocode in these assignments is a simplified subset and mix of high level programming languages such as C++, Java, and Python. As the programming languages are very different, it is not possible for pseudocode completely to resemble any given language. Some pseudocode features are similar to C++, some to Java, and some to Python.

The pseudocode syntax is mostly self-evident. So, here you can find an explanation of pseudocode parts that may be not obvious.

Source Code

Here are some recommendations about writing an actual program based on the pseudocode.

You may implement additional classes, methods, and functions, if so convenient. Each class shall be placed in a separate file unless it is an internal (nested) class. In the case of C++, place non-trivial function and methods declarations in a header file and their definitions in a separate source (or implementation header) file.

The specific method/function names, their parameters, and returned values shall be per the language convention. If the language supports **const, final**, or similar qualificators, use them to express that the parameters and/or this/self object are read only.

You can change functions signatures a little. You shall not change algorithms unless there is a good reason for it. Be sure that your program resembles the pseudocode in some way. For example, use the similar function names and variable names when possible.

The actual test driver (main program) has to read and write files in the same format as the pseudocode driver. Also, it has to test classes in the same way. So, it is better to keep the driver as close to the pseudocode as possible. When you are graded, an output of your program is often compared with the expected output. If there are even small differences, you may be asked to fix them.

Pseudocode Details

General Syntax

Unless multiple statements are written in one line, semicolons "**;**" are not used to separate statements. The statements usually are written one line after another.

Unless multiple function parameters are specified on one line, commas "**,**" are not used to separate parameters.

In the case of blocks, curly brackets **{ }** are not used. The statements inside a block have to be written starting from the same column.

For example,

```
if ( a>b )
   x = a
   g(b)
   b = x
else
   x = a
```

In the case of complex statements, **end** keyword can be used to denote the statements, for example

```
while ( i>0 )
   i = i-1
   …
endwhile
```

Variables (and parameters)

All variables (and parameters) are declared. Most declarations include type specification. For example,

```
int i = 0
boolean message( string x, int length = 0 )
```

Sometimes, the type is not specified. In such a case, keyword **variable** is used. In the actual program, the keyword shall be replaced with a proper language specific keyword and/or appropriate type name.

For example,

```
variable p = set.begin()
```

Input/Output

Output

Less-less "**<<**" sign is used to indicate output statements. An output statement consists of a variable holding the output stream and expressions to be printed, separated by less-less signs. For example

```
outputStream << "Name is " << name << ", weight is " << w << "pounds\n"
```

Keyword **stdout** is used to specify "standard output" stream, usually some kind of console.

Input

Greater-greater "**>>**" sign is used to indicate input statements similar to output. For example,

```
while ( inputStream >> name ) process( name )
```

It is assumed that input statements return true if there is still data to read, and false if there is no more data.

Stream Variables

INPUT_STREAM and OUTPUT_STREAM keywords declare language appropriate stream variables and, usually, have a filename specified as a parameter. For example,

```
INPUT_STREAM inputStream ( "in.txt" )
```

For loop

In the pseudocode, there are forms of for statement

```
for ( first; second; third )
for ( var : collection )
```

For (first; second; third) loop

Statement

```
for ( first; second; third )
        body
```

is the same as

```
first
while (second)
        body
        third
```

For example, the following blocks are the same

```
for ( i=0; i<10; i+=1 )
    stdout << i
```

and

```
int i=0
while (i<10)
        stdout << i
        i+=1
```

For (var : collection) loop

The var variable iterates through all values in the collection. For example,

```
int a[3] = { 1, 2, 3 }
for ( variable x : a ) stdout << x << "\n"
```

will print

1
2
3

Functions

Default parameters

If default parameters are not supported by the actual language, use function overloading (define multiple functions with different number of parameters). For example, if function func is written as

```
void func( int I, int j = 1, int k = 2 )
```

The following calls are possible

```
func(a,b,c)     -> i=a, j=b, k=c
func(a,b)       -> i=a, j=b, k=2
func(a)         -> i=a, j=1, k=2
```

Return types

In the pseudocode, the function return type is specified in front of the function name. Keyword **void** denotes function that does not return value. For example,

```
int weight()                    // function that returns int value
        return 10
void printWeight( int w )       // function that does not return value
        stdout << w
```

Classes

An example:

```
class A
  private
        int height

  public
    constructor( int theHeight = 0 )
       height = theHeight
        int getHeight()
                return height
    static boolean areYouOK()
       return true
```

class keyword starts a class definition. **private** and **public** keywords are hints for the follow-up class variables and methods, use appropriate actual language keywords, if any.

Functions as methods

Methods can static and non-static.

static keyword is used to denote a method that does not have implicit access to the object variables (fields, attributes, etc.). If **static** keyword is not used, the method is non-static and can access object variables using just their names. Non-static methods can also use **this** variable to refer to the object itself.

Constructors

constructor keyword is used to define a constructor. If default parameters are not supported by the actual language, use function overloading (define multiple constructors with different number of parameters).

In the pseudocode, keyword **new** is used to denote an object creation. This keyword can be safely omitted when actual language does not require it. For example,

```
Node n = new Node("A")
```

Destructors and delete in C++

In the case of C++, **destructor** keyword is used to define a destructor. There is no need to write actual destructor code in the case of other languages.

Also, keyword **delete** is used to release object memory previously allocated by **new** operator. There is no need to write actual delete statement in the case of other languages.

Virtual Keyword

virtual keyword denotes a method that can be reimplemented in subclasses. In Java and Python, all object methods are virtual, so in the actual code **virtual** keyword is not written. On another hand, in C++ it is required.

Nested classes

In the pseudocode, nested (inner) classes do not have implicit access to outer classes. In the actual code, you can implement them using actual language nested classes, or as top-level classes, whichever is easier. For example

```
class Tree
    class Node
```

Note that nested classes are referred to by their inner names (just Node in the example). In the actual code, use appropriate nested class naming convention.

Generic and template types

In the pseudocode **<*Type*>** after a class name denotes a generic (or template) class. For example,

```
class Set<Key>
```

The idea is that the same code can be used with various types. For example,

```
Set<string>     // set of strings
Set<Address>    // set of Addresses
```

If the actual language supports generics (or templates), try to use appropriate language syntax. If the language does not have or does not need generics, just disregard **<*Type*>** part (do not write it in the actual code). Also, if you cannot implement generics, assume specific *Type* depending on the assignment (for example, string).

Constants

In the pseudocode, **constant** keyword is used to indicate constant variables and constant methods. You can replace this keyword with an appropriate actual language syntax or remove it completely. In any case, use it as a hint.

Constants are variables that are assigned once and that values are not changed after it. Constant methods are functions that do not change class and object variable.

References (and pointers where appropriate)

Null

Keyword **null** denotes null “reference” (or pointer, in the languages that have pointers). Use appropriate actual language keyword in place of **null**.

Reference hints

In some languages, an object variable always holds a reference to the object value, not the value itself. In other languages, depending on how a variable is declared, it can hold the actual value, a reference to the value, or even so-called pointer to the value.

In the pseudocode, it is assumed that an object variable holds a reference to the object value. So, it is easy if the actual language has the same interpretation. To compensate it for other languages, star “*****” and ampersand “**&**” signs are used as described below.

If the actual language does not distinguish between objects by value, by reference, and by pointer – just disregard "*****" and "**&**" signs (do not write them in your actual code).

"**&**" after a type means "reference". For example,

```
int size( string& text )
```

"*****" after a type means "pointer". For example,

```
Node* nextNode( Node* currentNode )
```

Comparison

When there is no difference between reference (or pointer) comparison and value comparison, traditional comparison operators are used, such as **< > == != <= >=**

Otherwise, keywords **EQUAL**, **LESS**, **GREATER**, etc. are used to denote a value comparison. In the actual code, replace them with proper language expression.

For example, in some languages, the following comparison will be false

```
string a = "xy"
string b = "x"+"y"

a == b    // false as objects values are not compared,
          // only object references are
```

So, in this case, the pseudocode comparing values will be

```
a EQUAL b
```

Assignment

When there is no difference between reference (or pointer) assignment and value assignment, traditional assignment operator = is used.

Otherwise, keyword **COPY** is used to denote a value assignment. In the actual code, replace them with proper language expression.

For example, in some languages, the following assignment will create a second reference to the same object instead of copying the object

```
A a = new A("x")          // create A object with identifier "x"
B b = a                   // b now refers to the same object as a
a.setName("y")            // change the object name
stdout << b.getName()     // prints "y"
```

So, in this case, the pseudocode assigning values will be

```
A a = new A("x")          // create A object with identifier "x"
B b = COPY a              // b now a copy of a
a.setName("y")            // change the a's name, but not b's
stdout << b.getName()     // prints "x"
```

Iterators

Conceptually, an iterator is reference to an ordered collection element. The iterator can be used to access the element. The iterator also can be advanced to the next element in the collection.

Unfortunately, iterators are implemented very differently in C++, Java, and Python. So, in the pseudocode, an "iterator" does not resemble actual languages. Instead an iterator is a class that has the following methods

```
class Iterator

   boolean hasValue()
   void advance()
   ElementType getElement()
```

The iterable collection class has method

```
class Collection

   Iterator begin()
```

Iterators are used in the pseudocode like this

```
for ( variable p = collection.begin(); p.hasNext(); p.advance() )
   stdout << p.getElement() << "\n"
```

Note that the iterator class may have multiple **get*Element*()** methods, if so convenient (for example, getNode(), getText(), etc.)

In the actual code, you may use the same **begin()**, **hasValue()**, **advance()**, and **get*Element*()** methods, or you can implement iterators in a way that is native to the actual language. In the first case, you can use the pseudocode as a blueprint. In the second case, you will have to write iterator class code yourself, but as a plus, you will be able to use native for each loop to iterate.