CS 215 - Fundamentals of Programming II
Differences in C++, C, and Java

This handout briefly describes the differences between C++, C, and Java with respect to topics covered in CS 210 and the differences between C++ and Java with respect to topics covered in CS 215. The purpose of this handout is to help students transitioning from C or Java to C++ and assumes that readers already know how to do basic programming in C or Java. As such, this handout only will explain the C++ code in detail. The C and Java code will not be explained in detail. This handout is not in any way comprehensive or complete. Comments and suggestions are welcomed.

CS 210 topics

Just about anything that can be done in C is available in C++. However, generally C++ provides superior functionality for the topics covered in this handout. Therefore, **general use of the C equivalents shown here will not be allowed in CS 215**, since one of the objectives of the course is to master the C++ programming language.

C++ syntax differences from C

The following lists the most common C++ syntax differences from C. Some C implementations incorporate these differences.

- A comment may be indicated by `//` and continue to the end of a line, and generally should be used inside function definitions. The older comment style (`/* */`) should be used for large comment blocks outside of function definitions.
- The return type of `main` must be `int`. There are only two possible function headers for `main`:
  - `int main ()`
  - `int main (int argc, char *argv[])`
- A local variable declaration may appear anywhere in a block, and generally should be declared just before first use.
- Loop control variables may be declared in the for-statement header. E.g., `for (int i = 0; i < n; i++)`. This causes the scope of the loop control variable to be only the loop body. Generally, this is preferred unless the value of the loop control variable needs to be used outside of the loop body.
- C++ has strict type-checking, so when no parameters are listed in a function prototype, the function can be called only with no arguments. Thus `void` is not needed (and should not be used) in a parameter list to indicate this.
- Function names may be overloaded based on the number and/or types of the parameters. A type of a parameter includes whether or not it is `const`.
- Function **prototypes** may declare default values for parameters that are substituted when a call is missing a corresponding argument. All default values must be on the right-hand end, and when arguments are omitted they must be on the right-hand end.
- The keyword `typedef` is allowed in C++, but is not required when defining structs. The struct name in a struct definition becomes a type name.
- Functions may be members of a struct. (In fact, a struct is just a class where all of the members are public.)

Including header files

The syntax for including header files is the same in C and C++: **#include<libname>** for system libraries and **#include "header.h"** for user-defined header files. In C++, system libraries **DO NOT** have the "**.h**" extension. Standard C libraries may be used in C++. Their names are prefixed with '**c**' to indicate they are the standard C library of that name, e.g. **<cstdlib>** for **<stdlib.h>** or **<cmath>** for **<math.h>**. Although most of the C++ system libraries have different names than the C libraries providing equivalent utility, note that <string> is the C++ string library (**string** type, etc.) while **<cstring>** is the C string library (**strlen()**, **strcmp()**, **strcpy()**, etc.). All C++ system library constant, variables, functions, and classes are in the **std** namespace. A using directive can be used to put all the system library constant, variables, functions, and classes into the global namespace as is the case in C. (Namespaces are explained later in the namespace section.)

C++

```
#include <cstdlib>   // C utilities
#include <cmath>     // C math funcs
#include <cstring>   // C strings
#include <iostream>  // C++ console
#include <fstream>   // C++ files
#include <string>    // C++ strings
using namespace std;
   // access system library names
   // without qualification
```

C

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
             // console and file I/O
```

Java

```
// import package contents
import java.util.*;  // utilities
import java.io.*;    // file I/O
// import static methods/constants
import static java.lang.Math.*;
                        // math funcs
```

Constant declarations

In C++, the **const** keyword defines an actual constant value, rather than just a variable that cannot be changed. As such, it is used to define named constant values. This is superior to the use of **#define** in C because the constant name is type checked when used in expressions and retains its identity, whereas in C the defined text is substituted directly and the name is lost. Using **#define** for named constant values is not allowed in CS 215.

C++

```
const int MAX_SIZE = 80;
```

C

```
#define MAX_SIZE 80
```

Java

```
final int MAX_SIZE = 80;
```

Statically-allocated arrays

Statically-allocated arrays are the same in C++ and C. Java does not have statically-allocated arrays.

C++
```
int values[MAX_SIZE];
```

C
```
int values[MAX_SIZE];
```

Java
```
// No statically allocated arrays
```

Boolean type

C++ has a boolean type called `bool` with literal values `true` and `false`. (Note: `true` and `false` are reserved keywords and **not** the strings `"true"` and `"false"`.) As in C, zero values (`0`, `'\0'`, `NULL`) are interpreted as false and non-zero values are interpreted as true. However, such usage generally is considered poor coding practice and will be not be allowed in CS 215.

C++
```
bool done = false;
while (!done) {
   if (...)
      done = true;
}  // while not done
```

C
```
int done = 0;
while (!done) {
   if (...)
      done = 1;
}  // while not done
```

Java
```
boolean done = false;
while (!done) {
   if (...)
      done = true;
}  // while not done
```

String type

C++ has a string type called **`string`** that is defined in the **`<string>`** system library. This type is an object type, so strings have many member functions for accessing and manipulating them. (See a reference document for more information.). In particular, the assignment operator (=) is defined, as are the comparison operators (**`==`**, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**), and a concatenation operator (**`+`**). Occasionally, a C-string (i.e., a null-terminated char array used as a string) is needed, but the general use of C-strings is not allowed in CS 215. The C-string equivalent of a (C++) string may be obtained by using the **`c_str()`** member function. Although string literals are defined to be constant C-strings, they convert to (C++) strings automatically.

C++

```
#include <string>
#include <iostream>
using namespace std;

string word1;  // empty string
string word2 = "dog";
string word3;
int len;
bool cmp;

// Length
len = word2.length(); // 3

// Assignment
word1 = "hot";

// Concatenation
word3 = word1+word2;  // "hotdog"

// Comparison
cmp = word1 < word2;   // false
cmp = word1 == word2;  // false
```

C

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

char word1[MAX_SIZE];
                 /* uninitialized*/
char word2[MAX_SIZE] = "dog";
char word3[MAX_SIZE];
word1[0] = '\0';
   /* initialize to empty string */
int len, cmp;

/* Length */
len = strlen(word2);

/* Assignment is copy */
strcpy (word1, "hot");

/* Concatenation */
strcpy(word3, word1);  // copy first
strcat(word3, word2);  //

/* Comparison */
cmp = strcmp (word1, word2); /* >0 */
```

Java

```
String word1;  // uninitialized
String word2 = "dog";
String word3 = null;  // no reference
int len, cmp;
boolean eq;

// Length
len = word2.length();

// Assignment
word1 = "hot";

// Concatenation
word3 = word1 + word2;

// Comparison
cmp = word1.compareTo(word2); // >0
eq = word1.equals(word2);    // false

// word1 == word2 means same object
```

Console I/O

In C++, I/O is done using an I/O operator with a stream as the left-hand operand and a variable, constant, or literal as the right-hand operand. Data is read from an input stream using the **extraction** operator (**>>**), while data is written to an output stream using the **insertion** operator (**<<**). Both operators are left-associative and defined to return the left-hand stream operand, allowing multiple uses to be chained together. Console I/O is defined in the **<iostream>** system library. This library defines an input stream **cin** (pronounced "see-in") that is connected to the keyboard corresponding to standard input. The extraction operator skips whitespace by default. To read in a line containing whitespace as one string, there is a free function **getline()** with an input stream and a string as arguments. However, since a previous extraction operation does not remove the trailing whitespace from an input stream, the **ignore()** member function may be needed to get rid of it as shown in the example. Two output streams **cout** ("see-out") and **cerr** ("see-err") are defined with **cout** corresponding to standard output and **cerr** corresponding to standard error. By default both are connected to the display. The library also defines **endl** ("end-ell"), a constant value that represents a newline character as well as indicates that an output stream's buffer should be flushed. The examples below show reading in an integer, a real number (double), a string without spaces, and a string with spaces.

C++

```
#include <iostream>
#include <string>
using namespace std;

int anInt;
double aDouble;
string aString, aLine;

// Read in data
cout << "Enter an int: ";
cin >> anInt;
cout << "Enter a real: ";
cin >> aDouble;
cout << "Enter a string "
     << "(without spaces): ";
cin >> aString;
cout << "Enter a line: ";
cin.ignore();  // remove newline
getline(cin, aLine);

// Echo read in data
cout << "Input: " << anInt << ' '
     << aDouble << ' ' << aString
     << endl << aLine << endl
```

C

```
#include <stdio.h>

int anInt;
double aDouble;
char aString[MAX_SIZE];

// Read in data
printf ("Enter an int: ");
scanf ("%d", &anInt);
printf ("Enter a real: ");
scanf ("%lf", &aDouble);
printf ("Enter a string ");
printf ("(without spaces): ");
scanf ("%s", aString);
printf ("Enter a line: ");
getchar();  /* skip newline */
fgets(aLine, MAX_SIZE, stdin);

// Echo read in data
printf ("Input: %d %lf %s\n%s\n",
        anInt, aDouble, aString,
        aLine);
```

Java

```
import java.util.Scanner;

// Wrap System.in into a Scanner
Scanner in = new Scanner (System.in);

int anInt;
double aDouble;
String aString, aLine;

// Read in data
System.out.print ("Enter an int: ");
anInt = in.nextInt();
System.out.print ("Enter a real: ");
aDouble = in.nextDouble();
System.out.print ("Enter a string " +
   "(without spaces): ");
aString = in.next();
System.out.print ("Enter a line: ");
in.nextLine();  // skip newline
aLine = in.nextLine();

// Echo read in data
System.out.println ("Input: " + anInt
   + ' ' + aDouble + ' ' + aString
   + '\n' + aLine);
```

Handling invalid data entry

Since C++ is strictly type-checked, the extraction operator (`>>`) fails if the input is not in the correct format for the type being read in, causing the input stream to be put into a failed state. For bullet-proof data entry, every extraction operation of a non-character or string type should be followed by a test of the stream state. When used in a boolean context , the C++ streams are defined to return true if the stream is valid or false if the stream has failed (the return value of the `fail()` member function). Thus the expressions `(cin)` or `(!cin)` can be used to test if the previous extraction operation was valid. In addition, since the extraction operation is defined to return its stream argument, the input expression `(cin >> aVar)` also can be used to test the validity of a stream after the extraction operation. Once a stream has failed, it stays in the failed state until it is cleared using the `clear()` member function. Since the improperly formatted input remains in the stream, it must be removed from the stream before the next attempt to read input. This is usually done by calling `getline()` as shown.

C++

```
#include <iostream>
#include <string>
using namespace std;

bool valid;
int value;
string errorInput;

do {
   cout << "Enter an integer: ";
   if (cin >> value) {
      valid = true;
   } else {
      valid = false;
      cin.clear();
      getline (cin, errorInput);
         // skip bad input
      cout << "Input was invalid. "
           << " Try again." << endl;
   }
} while (!valid);
cout << "Entered integer is: "
     << value << endl;
```

C

```
#include <stdio.h>

int valid, value;
char errorInput[MAX_SIZE];

do {
   printf ("Enter an integer: ");
   /* scanf returns # of
      read entries, 0 when error
   */
   if (valid = scanf("%d", &value) {
      gets (errorInput);
         // skip bad input
      printf ("Input was invalid. ");
      printf (" Try again.\n");
   }
} while (!valid);
printf ("Entered integer is: %d\n",
   value);
```

Java

```
import java.util.Scanner;
import
   java.util.InputMismatchException;

Scanner in = new Scanner (System.in);
boolean valid;
int value = 0;
String input;

do {
   System.out.print ("Enter an "
      + "integer: ");
   try {
      value = in.nextInt();
      valid = true;
  } catch (InputMismatchException ex){
      valid = false;
      in.nextLine(); // skip bad input
      System.out.println
         ("Input was not a number. "
          + " Try again.");
   }
} while (!valid);
System.out.println
   ("Entered integer is: " + value);
```

Formatting real number output

In C++, real number formatting generally is set using output stream member functions **`setf()`** and **`precision()`** as shown below. Real number formatting is controlled by setting control bits defined in the **`ios`** class (thus are prefixed with **`ios::`**). The default formatting is the same as the **`%g`** specifier in C. (I.e., display up to 6 total digits, the **precision,** without leading or trailing zeros and using scientific notation automatically if the number of digits is larger than the precision.) The example below shows the configuration of a common format of fixed point notation and to always show the decimal point. Under this configuration, setting the precision causes exactly the specified number of digits after the decimal point to be shown. The formatting code only needs to be executed once and is in effect until the next formatting function is executed. In the examples below, the C and Java examples have been written to produce the same formatted output as the C++ example.

C++

```
#include <iostream>
using namespace std;

double fifty = 50.0,
   oneHalf = 0.5,
   approxPI = 22.0/7.0
   largeNum = 1234567890.0987654321;

// Default formatting
cout << fifty << endl     // 50
     << oneHalf << endl   // 0.5
     << approxPI << endl  // 3.14286
     << largeNum << endl;
                      // 1.23457e+09

// Using stream member functions
cout.setf(ios::fixed|ios::showpoint);
cout.precision(2);
cout << fifty << endl     // 50.00
     << oneHalf << endl   // 0.50
     << approxPI << endl  // 3.14
     << largeNum << endl;
                    // 123456790.10

// Set back to default
cout.unsetf(ios::fixed|
            ios::showpoint);
cout.precision(6);
```

C

```
#include <stdio.h>

double fifty = 50.0,
   oneHalf = 0.5,
   approxPI = 22.0/7.0,
   argeNum = 1234567890.0987654321;

/* Default formatting */
/* %g is same as C++ default */
printf ("%g\n%g\n%g\n%g\n",
        fifty,
        oneHalf,
        approxPI,
        largeNum);

/* Formatted like C++ example */
printf ("%.2f\n%.2f\n%.2f\n%.2f\n",
        fifty, oneHalf,
        approxPI, largeNum);
```

Java

```
double fifty = 50.0,
   oneHalf = 0.5,
   approxPI = 22.0/7.0,
   argeNum = 1234567890.0987654321;

// Default formatting
System.out.println (
   fifty + "\n" +    // 50.0
   oneHalf + "\n" +  // 0.5
   approxPI + "\n" +
              // 3.142857142857143
   largeNum);
              // 1.2345678900987654E9

// Formatted like C++ example
System.out.printf (
   "%.2f\n%.2f\n%.2f\n%.2f\n",
   fifty, oneHalf, approxPI,
   largeNum);
```

Formatting tables

In C++, table formatting generally is done using output stream manipulators. A manipulator is a special object that is used as the right-hand operand of the inserter operator (**<<**) that will change the display behavior. Manipulators are defined in the **<iomanip>** system library. The main manipulator is **setw()** that displays the next operand (only) in a field of the specified width. If a field is not wide enough to contain the output value, the value is displayed as if there is no manipulator. In addition, a value may be displayed within the field with left or right justification using manipulators **left** or **right**, respectively. Justification is in effect until the next justification manipulation. Right justification is the default.

C++

```
#include <iostream>
#include <iomanip>
using namespace std;

string description[] =
   {"Pen", "Pen-box", "Pen-case"};
int quantity[] = {1, 12, 144};
double price[] = {0.59, 2.69, 24.99};
int numItems = 3;

// Format doubles for currency
cout.setf(ios::fixed|ios::showpoint);
cout.precision(2);

// Write heading
cout << "Price list\n\n";
cout << left
     << setw(10) << "Item"
     << right
     << setw(10) << "Qty/Unit"
     << setw (10) << "Price" << endl;

// Write table lines
for (int i = 0; i < numItems; i++)
   cout << left
        << setw(10) << description[i]
        << right
        << setw(10) << quantity[i]
        << setw(10) << price[i]
        << endl;
```

C

```
#include <stdio.h>

char description[][MAX_SIZE] =
   {"Pen", "Pen-box", "Pen-case"};
int quantity[] = {1, 12, 144};
double price[] = {0.59, 2.69, 24.99};
int numItems = 3, i;

/* Write heading */
printf ("Price list\n\n");
printf ("%-10s%10s%10s\n",
   "Item", "Qty/unit", "Price");

/* Write table lines */
for (i = 0; i < numItems; i++)
   printf ("%-10s%10d%10.2f\n",
      description[i], quantity[i],
      price[i]);
```

Output (for all languages)

```
Price List

Item        Qty/unit     Price
Pen                1      0.59
Pen-box           12      2.69
Pen-case         144     24.99
```

Java

```
String[] description =
   {"Pen", "Pen-box", "Pen-case"};
int[] quantity = {1, 12, 144};
double[] price = {0.59, 2.69, 24.99};
int numItems = 3;

// Write heading
System.out.println ("Price list\n");
System.out.printf ("%-10s%10s%10s\n",
   "Item", "Qty/unit", "Price");

// Write table lines
for (int i = 0; i < numItems; i++)
   System.out.printf
      ("%-10s%10d%10.2f\n",
       description[i],
       quantity[i], price[i]);
```

Parameters

C++ has two kinds of function parameters, **value** parameters and **reference** parameters. C only has value parameters, though pointer variables can be used to implement pass by reference. In Java, primitive types like `int` and `char` always are passed by value. A value parameter is a copy of an actual argument in a function call. Thus when inside a function, an assignment to a value parameter or a call to a mutating member function does not modify the corresponding actual argument in the function call. A reference parameter is an alias (i.e., another name) for the corresponding actual argument in a function call. This is indicated in C++, by affixing '`&`' to the type of the parameter. Assignment to a reference parameter will cause the actual argument variable to become a different value/object and a call to a mutating member function will modify the corresponding actual argument. Reference parameters may be declared `const`, which prevents modification of the parameter. This is used commonly as an efficiency technique to avoid copying arguments that are large objects that will not be modified. Although C does not have reference parameters, pointers and pointer variables can be used to explicitly implement pass by reference as shown. In Java, object types like arrays and `String` are passed by value pointers to an object, which means the object can be modified using one of its mutator member functions, but the actual argument variable cannot be changed to refer to a different object. As a result, it is not possible to write an equivalent example swap method using primitive integers in Java.

C++

```
// Use of value parameters
void noswap (int first, int second){
   int tmp = first,
   first = second,
   second = tmp;
}

// Use of reference parameters
void swap (int& first, int& second){
   int tmp = first;
   first = second;
   second = tmp;
}

int a = 3, b = 5, x = 3, y = 5;
swap (a, b);
cout << a << ' ' << b << endl; // 3 5
swap (x, y);
cout << x << ' ' << y << endl; // 5 3
```

C

```
// Use of value parameters
// Same as C++
void noswap (int first, int second){
   int tmp = first,
   first = second,
   second = tmp;
}

// Use of pointers to implement
// reference parameters
void swap (int* first, int* second){
   int tmp = *first;
   *first = *second;
   *second = tmp;
}

int a = 3, b = 5, x = 3, y = 5;
swap (a, b);
printf ("%d %d\n", a, b); // 3 5
swap (&x, &y);
printf ("%d %d\n", x, y); // 5 3
```

Java

```
// Use of value parameters
// Same as C++
static void noswap (int first,
                    int second){
   int tmp = first,
   first = second,
   second = tmp;
}

// No reference parameters for
// primitive types in Java.
// No value parameters for objects
// in Java.  Object references are
// passed by value, which allows
// object mutation, but not changing
// the actual argument to refer to
// another object.
```

Opening files and handling open errors

In C++, files are accessed using streams. File I/O is defined in the **<fstream>** system library. The types **ifstream** and **ofstream** are defined for input file streams and output file streams, respectively. Each has a constructor that takes a C-string (**not** a C++ string, thus the use of the **c_str()** member function) argument that is the name of the file to be opened. (There is also an **open()** member function.) File streams support the same operators and member functions as the console streams. Thus, the use of a file stream in a boolean context is used to determine whether a file opened successfully (e.g. **(!inFile)**). In general, when a file open is unsuccessful, a program should display an error message and exit the program using the **exit()** function defined in the C standard library (**<cstdlib>**) as shown below.

C++

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // exit()
using namespace std;

string inFileName, outFileName;

// Open a file for reading
cout << "Enter input file name: ";
cin >> inFileName;
ifstream inFile (inFileName.c_str());
if (!inFile) {
   cerr << "Failed to open file "
        << inFileName << endl;
   exit(1);
}

// Open a file for writing
cout << "Enter output file name: ";
cin >> outFileName;
ofstream outFile
   (outFileName.c_str());
if (!outFile) {
   cerr << "Failed to open file "
        << outFileName << endl;
   exit(1);
}
```

C

```
#include <stdio.h>
#include <stdlib.h>  /* exit() */

char outFileName[MAX_SIZE],
     inFileName[MAX_SIZE];
FILE *inFile, *outFile;

/* Open a file for reading */
printf("Enter input file name: ");
scanf("%s", inFileName);
inFile = fopen(inFileName, "r");
if (inFile == NULL) {
   printf("Failed to open file %s\n",
          inFileName);
   exit(1);
}

/* Open a file for writing */
printf("Enter output file name: ");
scanf("%s", outFileName);
outFile = fopen(outFileName, "w");
if (outFile == NULL) {
   printf("Failed to open file %s\n",
          outFileName);
   exit(1);
}
```

Java

```
import java.util.Scanner;
import java.io.*;

Scanner in = new Scanner (System.in);
String inputFileName, outFileName;

System.out.print
   ("Enter input file name: ");
inFileName = in.next();
Scanner inFile = null;
try {
   inFile = new Scanner
      (new File(inFileName));
} catch (IOException e) {
   System.out.println
     ("Failed to open file "+inFileName);
   System.exit(1);
}

// Open a file for writing
System.out.print
   ("Enter output file name: ");
outFileName = in.next();
PrintWriter outFile = null;
try {
   outFile = new PrintWriter
      (new FileWriter (outFileName));
} catch (IOException e) {
  System.out.println
    ("Failed to open file "+outFileName);
  System.exit(1);
}
```

Writing to files and closing files

In C++, file streams behave exactly the same way as the console streams, meaning that once the file stream is set up, the operations on a file stream are the same as for a console stream. In particular, for output file streams, the insertion operator is used to write data to a file, and the output formatting methods and manipulators control how the data is written. Output file streams should be closed using the `close()` member function to ensure written data is flushed to the media. The examples below write the same table as in the table formatting example to a file. The contents of the resulting file is the same as the output from that example.

C++

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // exit()
using namespace std;

// Assume previous examples
// variable declarations
// and file opens

// Set up formatting
outFile.setf
   (ios::fixed|ios::showpoint);
outFile.precision(2);

// Write heading
outFile << "Price List\n\n";
outFile << left << setw(10) << "Item"
        << right << setw(10)
        << "Qty/unit" << setw(10)
        << "Price" << endl;

// Write table lines
for (int i = 0; i < numItems; i++)
   outFile << left << setw(10)
           << description[i] << right
           << setw(10) << quantity[i]
           << setw(10) << price[i]
           << endl;

// Close file
outFile.close();
```

C

```
#include <stdio.h>
#include <stdlib.h>  /* exit() */

/* Assume previous examples
   variable declarations
   and file opens
*/

/* Write heading */
fprintf (outFile,
         "Price list\n\n");
fprintf (outFile, "%-10s%10s%10s\n",
         "Item", "Qty/unit",
         "Price");

/* Write table lines */
for (i = 0; i < numItems; i++)
   fprintf (outFile,
            "%-10s%10d%10.2f\n",
            description[i],
            quantity[i],
            price[i]);

// Close file
fclose(outFile);
```

Java

```
import java.util.Scanner;
import java.io.*;

// Assume previous examples
// variable declarations
// and file opens

// Write heading
outFile.println ("Price list\n");
outFile.printf ("%-10s%10s%10s\n",
   "Item", "Qty/unit", "Price");

// Write table lines
for (int i = 0; i < numItems; i++)
   outFile.printf
      ("%-10s%10d%10.2f\n",
       description[i],
       quantity[i], price[i]);

// Close file
outFile.close();
```

Reading from files

C++ input file streams behave just like the **`cin`** object, using the extraction operator to read data from a file. Since the extractor operator (**`>>`**) returns the left-hand stream argument and a stream used in a boolean context returns false if the stream is invalid, an input expression may be used in a while-loop condition to read input from a file until the end of file is reached in the same way that **`fscanf()`** is used in C. The examples below output the same table as in the table formatting example, assuming that the input file contains the following data.

```
Pen 1  0.59
Pen-box  12  2.69
Pen-case 144  24.99
```

C++

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// Assume previous file open, output
// formatting, and table heading

string item;
int number;
double cost;

// While not at end of file
while (infile >> item >> number
              >> cost) {
   // Output line of table
   cout << left << setw(10) << item
        << right
        << setw(10) << number
        << setw(10) << cost
        << endl;
}
```

C

```
#include <stdio.h>

/* Assume previous file open and
   table heading */

char item[MAX_SIZE];
int number;
double cost;

/* While not at end of file */
while (fscanf (inFile, "%s %d %lf",
   item, &number, &cost) != EOF) {

   /* Output line of table */
   printf ("%-10s%10d%10.2f\n",
      item, number, cost);
}
```

Java

```
import java.util.Scanner;
import java.io.*;

// Assume previous scanner
// construction and table heading

String item;
int number;
double cost;

// While not at the end of file
while (inFile.hasNext()) {
   // Read line of data
   item = inFile.next();
   number = inFile.nextInt();
   cost = inFile.nextDouble();

   // Output line of table
   System.out.printf
      ("%-10s%10d%10.2f\n",
       item, number, cost);
}
```

Dynamic allocation

C++ pointer variables are the same as in C, as is the syntax for dereferencing a pointer variable. Unlike C, C++ dynamic allocation uses the built-in operators **new** and **delete**, rather than a system library function. Allocation of a single primitive type variable in C++ is done using syntax: **new type**. For object types, the same syntax can be used, which also executes the default constructor, or the syntax: **new type (args)** can be used, which calls the constructor that matches the arguments. For dynamically-allocated arrays, the syntax is **new type[size]**, which allocates an array of **size** elements of **type**. For object types, the default constructor is called for each element; other constructors cannot be called automatically. In all cases, a pointer to the newly allocated memory is returned by the **new** expression. Unlike in C, the returned pointer of C++'s **new** is typed, so there is no type casting. Deallocation of a single object in C++ is done through a pointer variable using syntax: **delete ptrVar**. For arrays, the syntax is **delete [] ptrVar**. Note: the square brackets are **empty**. Using the incorrect delete usually will cause runtime errors. For object types, the destructor is called on all deleted objects. Since Java has garbage collection, it has no explicit deallocation.

C++

```
#include <string>
using namespace std;

int *intPtr, *arrayPtr;
string *strPtr;
int size = 5, len;

intPtr = new int;  // alloc one int
strPtr = new string("word");
         // alloc and initialize
arrayPtr = new int[size];
         // alloc an array of 5 ints

// Dereferencing pointer variables
*intPtr = 10;
len = strPtr->length();
for (int i = 0; i < size; i++)
   arrayPtr[i] = i;

delete intPtr;  // dealloc one item
delete strPtr;
delete [] arrayPtr;
                // dealloc an array
```

C

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

int *intPtr, *arrayPtr;
int size = 5, i;
char **strPtr; /* pointer to array */

/* Dynamic allocation */
intPtr = (int *)malloc(sizeof(int));
arrayPtr = (int *)calloc(size,
                         sizeof(int));
strPtr =
   (char **)malloc(sizeof(char *));
*strPtr = (char *)calloc (size,
                        sizeof(char));
strcpy (*strPtr, "word");

/* Dereferencing pointer variables */
*intPtr = 10;
len = strlen(*strPtr);
for (i = 0; i < size; i++)
   arrayPtr[i] = i;

/* Deallocation */
free (intPtr);
free (arrayPtr);
```

Java

```
import java.lang.Integer;

int size = 5;

/* Dynamic allocation
Integer intWrapper = new Integer(0);
int[] arrayPtr = new int[size];

// Autoboxing
intWrapper = 10;

// Automatic dereferencing
for (int i = 0; i < size; i++)
   arrayPtr[i] = i;

// No explicit deallocation since
// there is garbage collection
```

CS 215 topics

The following topics are covered as new material in CS 215, but have significantly different syntax and/or semantics than the corresponding topics in Java. These sections are intended solely for those coming into CS 215 with Java programming experience.

Namespaces and classes

A namespace is used to allow identifiers to be used multiple times in different contexts. C does not have namespaces or function overloading, thus every function and global variable identifier must be unique. In C++, all system libraries are in namespace `std`, but namespaces for user-defined free functions and classes are optional. An identifier inside a namespace is used by qualifying it with the namespace name and the scope operator (`::`), e.g., `std::cout`. Identifiers declared outside of any namespace become part of the global namespace and are used unqualified. A `using` directive may be used to make all of the names in a namespace part of the global namepace (e.g., `using namespace std;`) or to make an individual name part of the global namespace (e.g., `using std::cout`). This allows the imported names to be used unqualified. However, since header files may be included by multiple unrelated projects, using directives should **not** be used in header files. I.e., all names in header files should be fully-qualified. Static identifiers inside a class also are accessed via the scope operator. E.g., I/O formatting flags in the ios class such as `std::ios::fixed`, or the iterator type in the collection classes such as `std::list<T>::iterator`. By contrast, in Java, packages are used to group classes and are tied to the underlying file system structure, and classes are used to group static methods.

C++ classes usually are defined in two separate files. A header file contains the class definition that defines the data members and the member function prototypes only. The individual member function definitions are stored in a source file. By including only a header file, source files that use a class can be compiled separately from the class source file. This can speed up compilation when only one source file is modified. The C++ class definition syntax is similar to Java with the following differences:

- There must be a semicolon (`;`) immediately after the closing `}`.
- Top-level classes are public by default.
- Visibility (public, private, protected) is applied in sections rather than to individual items.
- Default arguments allow an explicit-value constructor to serve as the default constructor.
- A main program (`main`) is a free function, not a static method of a class.

Each member function name in the class source file is prefixed by the class name and scope operator (e.g. `point::toString`).

Other C++ syntactic and semantic difference include:

- The default constructor is implicitly called (without any parentheses) when object variables are declared or objects are dynamically allocated.
- There is no Object superclass
- All of the standard collection classes are parameterized and primitive types may be used as type parameters.

C++

```
// Header file: point.h

#include <string>
// No using directive,
// qualify string

namespace example {

class point {
   public:
      // explicit-value constructor
      // w/default args
      point(double initX = 0,
            double initY = 0);
      // return "(x,y)"
      std::string toString ();

   private:  // Coordinates of point
      double x, y;
};  // end point

}  // end example
```

```
// Implementation file: point.cpp

#include <string>
#include <sstream>  // stringstream
#include "point.h"
using std::string;
using std::ostringstream;
// Use string, ostringstream
// unqualified

// Reopen namespace example
namespace example {

point::point(double initX,
             double initY)
   : x(initX), y(initY)
   {}  // end constructor

string point::toString (){
   ostringstream outStr;
   outStr << '(' << x << ','
             << y << ')';
   return outStr.str();
}  // end toString

}  // end example
```

```
// Usage file: main1.cpp

#include <iostream>
#include "point.h"
// No using directives, qualify names

int main (){
   // default construction: (0,0)
   example::point aPoint;
   std::cout << aPoint.toString()
             << std::endl;
   return 0;
}  end main


// Usage file: main2.cpp

#include <iostream>
#include "point.h"
using std::cout;
using std::endl;
using namespace example;
// Use names unqualified

int main (){
   // default construction: (0,0)
   point *aPoint = new point;
   cout << aPoint->toString()
        << endl;
   return 0;
}  // end main
```

Java

```
// Class file: Point.java

package example;

public class Point {

   // Default constructor
   public Point () {
      x = 0.0;
      y = 0.0;
   }

   // Explicit value constructor
   public Point (double initX,
                 double initY) {
      x = initX;
      y = initY;
   }

   // Convert to a String
   public String toString() {
      return "(" + x + "," + y + ")";
   }

   // Coordinates of point
   private double x;
   private double y;
}  // end Point
```

```
// Usage file: Main1.java
// No imports, qualify names

public class Main1 {
   public static void main
      (String[] args) {
      // default construction
      example.Point aPoint =
        new example.Point();
      System.out.println(aPoint);
   }  // end main
}  // end Main1


// Usage file: Main2.java

import example.*;
// Use names unqualified

public class Main2 {
   public static void main
      (String[] args) {
      // default construction
      Point aPoint = new Point();
      System.out.println(aPoint);
   }  // end main
}  // end Main2
```

Exceptions

Exceptions in C++ have similar syntax and semantics to those in Java except there is no **`finally`** clause. Clean-up code either is repeated in both the normal and exceptional cases, or cleanup code is placed after both normal and exceptional case code and an explicit boolean flag that indicates an exception has occurred is used to skip the normal case after exception handling. Generally, exception objects are caught using a **`const`** reference parameter so that they are not copied as they are passed up the exception handling chain. (Alternately, exception objects can be created dynamically and caught using a pointer variable.) The C++ library provides a set of standard exceptions in the **`<stdexcept>`** system library (see a reference document for a list).

C++

```
#include <iostream>
#include <stdexcept>  // invalid_argument, logic_error
using namespace std;

double celsiusToFahrenheit (double c)
{
   const double MIN_CELSIUS = -273.15; // abs. zero
   if (c < MIN_CELSIUS)
      throw domain_error
         ("Celsius argument is too small");
   return (9.0/5.0)*c + 32;
}  // end celsiusToFahrenheit

int main () {
   bool valid = false;
   double cDegrees, fDegrees;

   cout << "Enter a Celsius temperature: ";
   cin >> cDegrees;
   try {
      fDegrees = celsiusToFahrenheit (cDegrees);
      valid = true;
   } catch (const domain_error & e) {
      cerr << e.what() << endl;
      valid = false;
   }  // end try

   if (valid)  // test if there was an error
      cout << "The equivalent Fahrenheit temperature: "
           << fDegrees << endl;
   cout << "Cleanup code goes here." << endl;
}  // end main
```

Java

```
import java.util.Scanner;

public class Main {

   static double celsiusToFahrenheit (double c)
   {
      final double MIN_CELSIUS = -273.15; // abs. zero
      if (c < MIN_CELSIUS)
         throw new IllegalArgumentException
            ("Celsius argument is too small");
      return (9.0/5.0)*c + 32;
   }  // end celsiusToFahrenheit

   public static void main (String[] args) {
      Scanner in = new Scanner (System.in);

      double cDegrees, fDegrees;

      System.out.print ("Enter a Celsius " +
         "temperature: ");
      cDegrees = in.nextDouble();
      try {
         fDegrees = celsiusToFahrenheit (cDegrees);
         System.out.print ("The equivalent Fahrenheit "
            + "temperature: " + fDegrees);
      } catch (RuntimeException e) {
         System.err.println (e.getMessage());
      } finally {
         System.out.println ("Cleanup code goes here");
      }  // end try
   }  // end main
}  // end Main
```

Iterators

Iterators in C++ are abstract pointers semantically and overload the pointer syntax (including `*` for dereference, `++` to move iterator to next item, `--` to move iterator to previous item). Each STL collection class implements its own iterator as an inner class, so to access the iterator types, the collection type is prepended using the scope operator (e.g., `list<int>::iterator`). All STL collections have member functions that return an iterator to the first item in the collection (`begin()`) and return an iterator one past the last item (`end()`). STL collections also have an explicit-value constructor that initializes a collection from another collection by providing a begin iterator and an end iterator. Regular pointers may be used as iterators to arrays. The begin iterator for an array is the address of the first element (i.e., the name of the array) and the end iterator for an array is the address one past the last element of the array (i.e., name of the array plus the size of the array). The example below shows construction of an STL vector of integers initialized using an array, then a list of integers initialized using the vector. C++11 introduced the `auto` type keyword and the range-based for-loop syntax.

C++

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int initValues[] = {1,3,5,7,9,11};
int arrSize = sizeof(initValues)/sizeof(int);

// Construct vector from array
vector<int> intVector (initValues,
                       initValues + arrSize);

vector<int>::iterator vIter = intVector.begin();
while (vIter != intVector.end()) {
   cout << *vIter << endl;  // dereference
   vIter++;                 // advance to next item
}

// Construct list from vector
list <int> intList (intVector.begin(), intVector.end());

// Implicit use of iterators (C++11)
for (auto item : intList)
   cout << item << endl;
```

Java

```
import java.lang.Integer;
import java.util.Vector;
import java.util.List;
import java.util.Iterator;

int [] initValues = {1,3,5,7,9,11};

// Add values from array to vector
Vector<Integer> intVector = new Vector<Integer>();
for (int i = 0; i < initValues.length; i++)
   intVector.add(initValues[i]);


// Explicit use of iterators
Iterator<Integer> vIter = intVector.iterator();
while (vIter.hasNext()) {
   System.out.println(vIter.next());  // auto advances
}

// Get a list from a vector
List<Integer> intList = intVector.subList(0,
   intVector.size());

// Implicit use of iterators
for (int value : intList)
   System.out.println (value);
```