Primitive Types in C++ and Java and Their Input/Output

Questions:
1. What’s the relationship between tokens, identifier, and variable names?
2. C++ gives you three different character types:

char
signed char
unsigned char

What’s the difference between the three?

Is "char" a "signed char" or an "unsigned char"?
3. For a sequence of characters to be an identifier, what conditions must hold?
4. What character set is typically used for C++ identifiers?
5. What character set is used for Java identifiers?
6. When a Java program reads a text file consisting of ASCII characters, how does it convert the characters into Unicode characters?
7. When is a variable name considered to be of primitive type?
8. The backslash character (\) serves two roles in C++ and Java source code. What are those?
9. C++ Question: Suppose I randomly select a memory location and examine a byte of memory at that location, would it be correct to say that the selected byte could be read into a variable of type char?
10. Suppose I randomly select a memory location and examine two adjacent bytes of memory at that location, would it be correct to say that the two bytes taken together represent a Java char?
11. As the following figure shows, there are two types of escapes for specifying a character value and there are two ways of writing numeric escapes.

What are the constraints on the different ways of specifying an escape?

```
          ---------- numeric in hex ('\x62')
            /
            /
            ------- numeric
            /
            /
            escapes
            ------- numeric in octal ('\142')
            /  
            /   
            \\
            ------ character 'n', 't', 'f', etc.
```
12. The ASCII integer code for the bell (that’s the sound your terminal makes when it wants to draw your attention) is 7. How would you set a character variable equal to the bell character using

a) the octal representation for an escape sequence

b) the hex representation for an escape sequence
13. What’s the difference between a hex escape in Java and a hex escape in C++?
14. What does the ANSI standard for C++ have to say about how much memory is to be allocated for the various integral types? How about Java?
15. What is an integer literal and in how many different ways can you specify it?
16. Suppose I want to use a hex integer literal to initialize the value of an int variable, which of the following statements is correct. I want the decimal value of the variable to equal 98.

```
int x = '0x62';

int x = '\0x62';

int x = '\x62';

int x = 0x62;
```
17. What are the four different contexts in which both C++ and Java perform automatic type conversion?
18. How do you understand the phrases:

   automatic type conversion

   implicit type conversion

   explicit type conversion
19. How does one characterize the loss of information in type conversion for the arithmetic types?
20. What are the two different modes of automatic type conversion in C++? And how do these two modes differ?
21. What is a widening type conversion in Java for the primitive types?
22. How do C++ and Java differ with respect to what is allowed for automatic type conversion?
23. Why is Java considered to be more strongly typed than C++?
24. So if you need a narrowing type conversion in Java, how can you bring it about?
25. Is it acceptable C++ syntax? Will it work anyway?

```cpp
int* p = (int*) malloc( 100 * sizeof( int ) );
```

What is the recommended form for the cast here?

```cpp
int* p = static_cast<int*>( malloc( 100 * sizeof( int ) ) );
```
In C++ and Java, data I/O is carried out with *streams*.

You open a stream to the destination to which you want to write the data.

You open a stream to the source from where you want to bring in the information.
The C++ Stream Hierarchy
In a standard C++ implementation, a stream class name such as `ifstream` would be a `typedef`:

```cpp
typedef basic_ifstream<char> ifstream;
typedef basic_ifstream<wchar_t> wifstream;
```
Input operations are supported by classes of type `istream`.

Output operations by classes of type `ostream`.

The subclass `iostream`, which inherits from both `istream` and `ostream`, provides operations for both input and output.
Input-Output Operations for Character Streams

```cpp
ofstream out("invoice.dat");
if ( !out ) print_error( "cannot open output file" );

int x = 1234;
out << x << " " << 56.78 << " " << "apples" << '\n';
```

would cause the character string

```
1234 56.78 apples
```

to be written out into the file `invoice.dat`.

You must include the header `fstream` for this to work since the class `ofstream` is defined in that header file.
When you construct an `ofstream` object in the manner shown, the file named as the argument will be created by default if it does not exist already.

If the file existed previously, its contents would be overwritten starting at the beginning.

If you wish to append to a file already in existence, you’d need to make the following invocation of the `ofstream` constructor:

```cpp
ofstream out("invoice.dat", ios::app);
```
To read the contents of the file *invoice.dat* back into a program, you’d need to create an input stream object:

```c++
int x;
double y;
string z;

ifstream in( "invoice.dat" );
if ( !in ) error( "cannot open input file" );

in >> x;
in >> y;
in >> z;

cout << x << " " << y << " " << z << '\n';
```

The extraction operator will, by default, skip over the white spaces (blanks, tabs, newlines, etc.) between the different entries in the file.
For a stream that can be used for both reading and writing, you need to construct an `fstream` object, again supplied by the header file `fstream`:

```cpp
fstream inout( "invoice.dat", ios::in | ios::out );
if ( !inout ) print_error( "file could not be opened" );
```

When using such streams, you have to pay careful attention to what is known as a file position or file pointer.

Each open file has two positions associated with it:

1. The current file position for reading a byte.

2. The current file position for writing a byte.

The file positions can be controlled by invoking the `seekp` function for the put position and the `seekg` function for the get position.
/TestFilePosition.cc

#include <fstream>
#include <string>

int main() {

    fstream inout( "invoice.dat", ios::in | ios::out );

    inout << 1234 << " " << 56.78 << " " << "apples" << '\n';

    cout << inout.tellg() << endl;       // 21
    cout << inout.tellp() << endl;       // 21

    inout.seekg( ios::beg );

    cout << inout.tellg() << endl;       // 0
    cout << inout.tellp() << endl;       // 0

    int x;
    double y;
    string z;

    inout >> x >> y;
    inout >> z;

    cout << x << " " << y << " " << z << endl;
        // 1234 56.78 apples
    return 0;
}
Other useful stream functions dealing with file positions are the two-arg version of `seekg` and `seekp` that allows the position of a stream to be set relative to some other position.
//TestFilePosition2.cc

#include <fstream>
#include <string>

int main() {

  fstream inout( "invoice.dat", ios::in | ios::out );

  inout << 1234 << " " << 56.78 << " " << "apples" << '\n';

  cout << inout.tellg() << endl; // 21
  cout << inout.tellp() << endl; // 21

  inout.seekg( 0 );

  cout << inout.tellg() << endl; // 0
  cout << inout.tellp() << endl; // 0

  int x = 0;
  double y = 0.0;
  string z = "";

  inout >> x;

  inout.seekg( 8, ios::cur );

  inout >> z;

  cout << x << " " << y << " " << z << endl;
  // 1234 0 apples

}
return 0;
}

get() and put() functions for character I/O:

If it is desired to read from or write into an ASCII encoded text file one character at a time, it is convenient to use the `istream` member function `get` for reading and the `ostream` function `put` for writing.
The function `get` has been overloaded so that it can be invoked in the following three different ways.

The function is invoked through a call like

```cpp
ifstream from( "dataFile" );
char ch;
while ( from.get( ch ) ) {
    ...
```

The function `get` in this invocation returns the `istream` object on which it is invoked, unless the stream has reached the end of the file, in which case the function evaluates to false.
To make certain that the \texttt{while} loop was terminated by the stream reaching the end-of-file condition as opposed to getting into an error state, we can invoke the following test after the \texttt{while} loop:

```c
if ( !from.eof() ) {
    // print error message and then
    exit( 1 );
}
```

where \texttt{eof()} is one of the four \textit{condition functions} defined for testing the error state of a stream, the other three being \texttt{bad()}, \texttt{fail()}, and \texttt{good()}. 

The `get()` function is invoked through a no-argument call, as in

```cpp
fstream from( "dataFile" );
int ch;
while ( ch = from.get() && ch != EOF ) {
  ...
```
The reverse of the two \texttt{get} functions shown above is the \texttt{put} function defined for the \texttt{ostream} class. Its signature is

\begin{verbatim}
put( char )
\end{verbatim}
The third version of `get` has the following signature:

```c
get( char* buffer, streamsize size, char deliminter='\n' )
```
//GetThirdType.cc

#include <fstream>

int main() {

    const int max_count = 256;
    // will read max of 255 bytes to
    // allow for null at the end

    char line[ max_count ];

    ifstream in( "a.out" );

    while ( in.get( line, 256 ) ) {
        int count = in.gcount();
        cout << "num of bytes read:" << count << '\n';

        if ( count < max_count - 1 ) in.ignore();
    }
    return 0;
}
Functions that can be very useful for conditional retrieval of characters from an input stream:

**putback(char c)** for pushing the argument byte back into the stream.

**unget()** for resetting the file position to where it was at the time of the previous invocation of **get**.

**peek()** for its return value which is the next character that will be read by a call to **get**; this character will not be extracted from the stream.
Input-Output Operations for Byte Streams

Suppose we wanted to read an image or a sound file, or copy the contents of an executable file, such as an `a.out` file output by a C++ compiler, into another file, how would one do that?

If it is desired to read and write an uninterrupted stream of bytes, the stream object created must be set to the binary mode.

```cpp
ifstream from( "dataFile", ios::binary );
```
You might think that if you are reading only one byte at a time, it shouldn’t matter if you open the file in a text mode or do not set the mode of the stream to binary.

But beware that if you use the text mode for reading a binary file, you may not see all the bytes if you are using one of the \texttt{get} functions to read the bytes.
For example, on Windows platforms, when a binary file is opened in text mode, the pair of characters corresponding to a newline, `<cr><lf>`, is read as a single character by the `get()` method.

So if your binary file contains the following characters

```
hello <cr><lf>
```

A hex dump of the file if it was opened as a text file and read by `get()` would look like:

```
0x68 0x65 0x6c 0x6c 0x6f 0x20 0x0A
```
In general, performing character based I/O on a binary file may cause your operating system to carry out system-specific character translations on some of the bytes.
#include <fstream>

const int N = 1000;

int main(int argc, char** argv) {

    char buffer[ N ];

    if (3 != argc)
        print_error("wrong number of arguments, usage: function input_file output_file");

    ifstream from(*++argv, ios::binary);
    ofstream to(*++argv, ios::binary);

    while( from.read( buffer, N ) ) {
        to.write( buffer, N );
    }

    if ( from.eof() ) {
        int count = from.gcount();
        to.write( buffer, count );
    }

    if ( !from.eof() ) {
        print_error("something strange happened");
        exit( 1 );
    }

    from.close();
}
to.close();
return 0;
}

void print_error(const char* p, const char* p2) {
  cerr << p << ' ' << p2 << '\n';
  exit(1);
}
It is possible to read all the bytes in one fell swoop by opening a file with the \texttt{ios::ate} option that puts the get-position pointer at the end of the file.

Then by invoking the \texttt{tellg()} function, we can figure out the size of the file in bytes.
```cpp
#include <fstream>

int main(int argc, char** argv) {

    if (3 != argc)
        print_error("wrong number of arguments, \n        usage: function input_file output_file");

    ifstream from(*++argv, ios::binary | ios::ate);

    long N = from.tellg();
    char* buffer = new char[N];
    from.seekg( ios::beg );

    ofstream to(*++argv, ios::binary);

    from.read( buffer, N );
    to.write( buffer, N );

    if ( !from.good() ) {
        print_error("something strange happened");
        exit( 1 );
    }

    delete[] buffer;
    from.close();
    to.close();
    return 0;
}
```
void print_error(const char* p, const char* p2) {
    cerr << p << ' ' << p2 << '
';
    exit(1);
}
Let’s now consider the case when we just want to write some data internal to a program into a file, but we wish for the output to be the binary representation of the data.
/WriteBinaryIntsToFile.cc

#include <fstream>

int main() {

    ofstream to( "out.dat", ios::binary);
    if (!to)
        cerr << "cannot open output file" << endl;

    int data[3] = {1, 2, 3};
    to.write( data, sizeof( data ) );
    to.close();
    return 0;
}
The content of out.dat On a Windows machine:

```
01 00 00 00 02 00 00 00 03 00 00 00 00
```

The content of out.dat On a Sun machine:

```
00 00 00 01 00 00 00 02 00 00 00 03
```
```cpp
#include <fstream>

int main() {

    ofstream to( "out.dat", ios::binary);
    if (!to)
        cerr << "cannot open output file" << endl;

    int data[3] = {1, 2, 3};
    to.write( data, sizeof( data ) );
    to.close();

    ifstream from( "out.dat", ios::binary | ios::ate );
    if (!from)
        cerr << "cannot open input file" << endl;

    long size = from.tellg();
    from.seekg( ios::beg );

    int* buffer = new int[ size/4 ];
    from.read( buffer, size );

    from.close();
    delete[] buffer;
}
```
return 0;
}

Controlling the Format

Formatting decisions are controlled by stream functions and manipulators in C++.
To explain the difference between a formatting stream function and a stream manipulator:

Let’s say that we wish for our floating point values to be printed out using the fixed decimal notation with a precision of 10 (the default is 6), C++ gives us two choices for doing this.

We can either invoke the stream function \texttt{precision}:

\begin{verbatim}
    cout.precision( 10 );
    cout << sqrt( 2 );
\end{verbatim}

Or, we can use the stream manipulator \texttt{setprecision}, as in

\begin{verbatim}
    cout << setprecision( 10 ) << sqrt( 2 );
\end{verbatim}
The stream member functions for formatting and the stream manipulators alter the *format state* of a stream.

The stream manipulators **hex** and **oct** can be used to display an integer in hexadecimal and octal notation, respectively. The default is the decimal notation, represented by the manipulator **dec**.

```cpp
cout << 127 << oct << setw( 7 ) << 127
    << hex << setw( 5 ) << 127 << '\n';
```

will display on a terminal screen the following string

```
127   177   7f
```
I/O Streams for Java

Here are some interesting differences between how I/O is set up in C++ and in Java:

• Whereas in C++ all the needed input/output functionality is packed into a relatively small number of stream classes, Java offers a separate class for practically every situation.

• While the C++ streams are buffered by default, the Java streams are unbuffered by default. However, you can incorporate buffering in a Java stream by invoking classes specified for that purpose.

• In Java, all integer and floating point numbers are output in the big-endian representation regardless of the underlying platform.

• A typical C++ program uses ASCII-encoded bytes for the characters used either individually or in strings. When these characters are
written out to an output device, they remain in their ASCII-encoded form.

On the other hand, a character in a Java program is always represented by its 2-byte Unicode. However, if it is desired to write out these characters to an output device, a typical Java program will do so using the ASCII encodings. This is true of input also.

A typical C++ program will expect the text based input to be ASCII encoded; the characters in such input will then remain ASCII-encoded inside the program.

A typical Java program will also expect its text based input to be ASCII encoded, but will convert the characters into their Unicode representations on the fly as they are read into memory.

We are only describing typical situations here. Java I/O provides the functionality needed for writing out character streams using Unicode and other representations and for reading character streams that are based on Unicode and other representations.
//WriteIntToFile.java

import java.io.*;

class WriteIntToFile {

    public static void main( String[] args ) throws Exception {
        int anInt = 98;

        FileOutputStream fos = new FileOutputStream( "out.fos" );
        fos.write( anInt );
        fos.close();

        FileWriter fw = new FileWriter( "out.fw" );
        fw.write( anInt );
        fw.close();

        DataOutputStream dos = new DataOutputStream( new FileOutputStream( "out.dos" ) );
        dos.writeInt( anInt );
        dos.close();

        DataOutputStream dbos = new DataOutputStream( new BufferedOutputStream (new FileOutputStream( "out.dbos" ) ) );
        dbos.writeInt( anInt );
        dbos.close();

        PrintStream ps = new PrintStream( new FileOutputStream( "out.ps" ) );
        ps.print( anInt );
        ps.close();
    }
}
PrintStream pbs = new PrintStream(
    new BufferedOutputStream( new FileOutputStream( "out.pbs" ) ) );
pbs.print( anInt );
pbs.close();

PrintWriter pw = new PrintWriter( new FileOutputStream( "out.pw" ) );
pw.print( anInt );
pw.close();

PrintWriter pbw = new PrintWriter( new BufferedOutputStream( new FileOutputStream( "out.pbw" ) ) );
pbw.print( anInt );
pbw.close();

PrintWriter pw2 = new PrintWriter( new FileWriter( "out.pw2" ) );
pw2.print( anInt );
pw2.close();
RandomAccessFile ra = new RandomAccessFile( "out.ra", "rw" );
ra.writeInt( anInt );
ra.close();
}
<table>
<thead>
<tr>
<th>filename</th>
<th>file size in bytes</th>
<th>file content in hex</th>
<th>output as displayed by 'cat filename' (cat reads in text mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>out.fos</td>
<td>1</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>out.fw</td>
<td>1</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>out.dos</td>
<td>4</td>
<td>00 00 00 00 62</td>
<td>b</td>
</tr>
<tr>
<td>out.dbos</td>
<td>4</td>
<td>00 00 00 00 62</td>
<td>b</td>
</tr>
<tr>
<td>out.ps</td>
<td>2</td>
<td>39 38</td>
<td>98</td>
</tr>
<tr>
<td>out.pbs</td>
<td>2</td>
<td>39 38</td>
<td>98</td>
</tr>
<tr>
<td>out.pw</td>
<td>2</td>
<td>39 38</td>
<td>98</td>
</tr>
<tr>
<td>out.pbw</td>
<td>2</td>
<td>39 38</td>
<td>98</td>
</tr>
<tr>
<td>out.pw2</td>
<td>2</td>
<td>39 38</td>
<td>98</td>
</tr>
<tr>
<td>out.ra</td>
<td>4</td>
<td>00 00 00 00 62</td>
<td>b</td>
</tr>
</tbody>
</table>

Table 1: