CSC112 Fundamentals of Computer Science Spring 2016

Exam # 1 Practice Problems

1. Give C++ statements that will declare a boolean variable `b` and integers `i`, `j`, and `k`.

   Answer:

   ```
   bool b ;
   int i,j,k ;
   ```

2. For this problem, assume the input from the keyboard is:

The quick red fox.

   What is the output of the following program ?

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

   int main()
   {
      char ch ;

      do {
         cin >> ch ;
         if ( ! cin.eof() ) cout << ch ;
      } while ( !cin.eof() ) ;
   }
   ```

   Answer:

   ```
   Thequickredfox.
   ```

3. What is the difference between `cin.fail()` and `cin.eof()` ?

   Answer:
   The function `cin.fail()` will return **true** when an error occurred during the most recent operation involving `cin`. For example, `int x ; cin >> x` will result in an error if the input is not a valid integer. The function `cin.eof()` will return **true** if the most recent attempt to get input using the `cin` object encountered the end of file.

4. What is the difference between single quotes `'` and double quotes `"` in C++ ? When are they used ?

   Answer:
   Single quotes are used to write a single character constant, e.g., `'A'`. Double quotes are used to write a character string constant, e.g., `"Hello, it's me."`

5. What is the range of values for a C++ `short int` ? Hint: a `short int` is 16 bits long.

   Answer: -32768 to 32767

6. What is the difference between `=` and `==` ?

   Answer:
   The lexeme `=` is used for assignment. For example, `x = 5 ;`

   The lexeme `==` is used for comparison. For example. `if ( j == n )`

7. When would you use a `do` – `while` loop instead of a `while` loop ?

   Answer:
   When the problem needs a loop that should be entered at least once. This happens often when processing an input stream. An attempt to read from the input stream needs to be done at least once.

8. Give a few lines of C++ code to print the numbers 1 to 10.

   Answer:

```
int i ;

for ( i = 1 ; i <= 10 ; i++ ) {
    cout << i << endl ;
}
```

9. Using a loop, give a few lines of C++ code to print the numbers `1, 2, 4, 8, 16, ... 1024`.

   Answer:

```
int k ;

k = 1 ;
while ( k <= 1024 ) {
    cout << k << endl ;
    k = 2 * k         ;
}
```

10. Write a C++ function named `truncate` that will truncate a double precision number to two decimal places. The function should be usable as follows:

```
double x ;
x = sqrt( 2.0 ) ;   //  At this point, x = 1.414213562
truncate(x) ;       //  At this point, x = 1.41
```

Answer:

```
void truncate( double & x )
{
   int a ;

   a = (int) ( 100 * x ) ;
   x = a / 100.0 ;
}
```

11. Calling functions – *No question for the practice problems.*

12. What are `formal parameters` and what are `actual parameters` ? What is the difference ?

Answer:
Formal parameters are names used in the header of a function definition. For example:

```
int fun( int A, int B )
```

Both `A` and `B` are formal parameters. They serve as "placeholders" for the values which will be supplied later when the function is used.

Actual parameters are values (variables or expressions) used in the parameter list when a function is called. For example:

```
y = fun( x, 3 * u + 5 ) ;
```

The actual parameter `x` is copied to the formal parameter `A`. Subsequently, the expression `3 * u  5+` is computed, and the result is copied to the formal parameter `B`.

13. Why do we use `#include` statements in our C++ programs ?

Answer:

The `g++` compiler has several phases. The first phase is a **pre-processing** step. A `#include` statement is a pre-processor directive which expands into the contents of the indicated file at the point in the program where the `#include` statement appears. An included file gives definitions of data types and functions of interest to our programs.

For example if we look at the file associated with `#include<cmath>` we see some lines of code as follows:

```
# define M_E            2.7182818284590452354   /* e */
# define M_LOG2E        1.4426950408889634074   /* log_2 e */
# define M_LOG10E       0.43429448190325182765  /* log_10 e */
# define M_LN2          0.69314718055994530942  /* log_e 2 */
# define M_LN10         2.30258509299404568402  /* log_e 10 */
```

```
# define M_PI           3.14159265358979323846  /* pi */
# define M_PI_2         1.57079632679489661923  /* pi/2 */
# define M_PI_4         0.78539816339744830962  /* pi/4 */
# define M_1_PI         0.31830988618379067154  /* 1/pi */
# define M_2_PI         0.63661977236758134308  /* 2/pi */
# define M_2_SQRTPI     1.12837916709551257390  /* 2/sqrt(pi) */
# define M_SQRT2        1.41421356237309504880  /* sqrt(2) */
# define M_SQRT1_2      0.70710678118654752440  /* 1/sqrt(2) */
```

The `#define` statements shown above are also compiler directions. They define various names, e.g., `M_PI` to be the appropriate mathematical constants. Folks may notice that `M_PI` represents the transcendental number π.

If we have used `#include <cmath>` in our program, we can write statements such as:

```
double x ;
x = 1.44 * M_PI ;
```

The statement `x = 1.44 * M_PI ;` will be expanded into `x = 1.44 * 3.14159265358979323846 ;` by the pre-processor.

14. What is a C++ namespace ?

    Answer:

    A **namespace** is collection of C++ names (e.g., C++ declarations of types, objects and variables) whose scope is restricted to that namespace. A namespace itself has a name. For example, "**std**" is a widely available C++ namespace. The names of many frequently used C++ objects (e.g., **cin** and **cout**) and data types exist only within the **std** namespace.

    To use a name whose scope is restricted to a namespace, we must either:

    - use the scope resolution operator, e.g. `std::cout`
    - or, bring the names from the namespace into the current scope, e.g., `using namespace std`

15. What does the statement `exit(3) ;` do ? What is the 3 for ?

    Answer:

    A call to the `exit` function terminates a program at the point of the call. The `exit` function accepts a one-byte unsigned integer (in the range 0 to 255), and passes that number back to the shell in a special shell variable with the (somewhat strange) name `#?`. When we run a program, we can type "`echo $?`" to examine the exit status of that program.

16. Give a C++ statement to read an integer `N` using the `scanf` function.

    Answer: `scanf( "%d", &N ) ;`

17. What is the difference between pass-by-value and pass-by-reference ?

    Answer:

    In pass-by-value semantics, the values of the actual parameters are copied to the formal parameters. Changes to the formal parameters made within a function have no effect on variables in the caller.

In pass-by-reference semantics, changes to the formal parameters cause corresponding changes to the actual parameters. When using pass-by-reference semantics, the actual parameter can not be an arithmetic expression; it must be a variable, conforming pointer expression, or array reference.

18. In C++ how does a programmer specify that a parameter is a pass-by-reference parameter ?

    Answer: Using the `&` character in front of a formal parameter. For example

    ```
    void function( int & x )
    ```

19. Consider the following program:

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

    int main()
    {
       int x ;

       x = 42 ;      // The answer is forty two.
       cout << &x << endl ;
    }
    ```

    When run, we get the following output:

    ```
    0x7ffc7519c67c
    ```

    Explain. Why do we get this result ?

    Answer:

    The (rather large) hexadecimal number shown above is the numerical position in memory of the local variable `x`. We received this result because we used the "**address of**" operator on `x` in the output statement.

20. In C++, what is a *pointer* ?

    Answer: A *pointer* is a memory address together with a data type associated with the pointer. I.e., the type of data stored at the indicated memory address.

21. Give an example of pass by reference using pointers.

    Answer:

    ```
    // ----------------------------------------------------------------
    //  Function abso() changes the given value into its absolute value.
    void abso( int * xp )
    {
       if ( (*xp) < 0 ) *xp = - *xp ;
    }
    ```

```
// ----------------------------------------------------------------
int main()
{
   int x ;

   x = 7 ;
   abso( &x ) ;
   cout << "x = " << x << endl ;

   x = -11 ;
   abso( &x ) ;
   cout << "x = " << x << endl ;
}
```

When the program above is run, we see:

```
x = 7
x = 11
```

22. What does the keyword **const** do ? How is is used ? Why would you want to use it ?

    Answer: The keyword **const** declares a variable to be constant. The initial value of the variable must be supplied at the time of its declaration. Once a value of a named constant is established, it can not be changed. For example:

    ```
    const double u = 1.44 ;
    ```

23. What does the keyword **static** do ? How is is used ? Why would you want to use it ?

    Answer: The keyword **static** specifies that a variable is to be stored in an area of memory distinct from the system stack. This area is sometimes called the **heap memory**. The consequence of this storage class is that the memory location associated with the variable is persistent across subsequent function calls. For example, a static integer variable `x` can be declared using:

    ```
    static int x = 1 ;
    ```

    The keyword `static` affects initializations. The assignment `x = 1` will occur only *once* for the entire program run.

    The **static** keyword is usually used for two purposes:

    - It allows a function to retain some information from one function call to the next.
    - The declaration of very large fixed-length arrays (e.g., `int A[10000000] ;` requires more memory than is available on the system stack. Attempting to run such programs will result in a run-time error.

      Such arrays must be declared:

      ```
      static int A[10000000] ;
      ```

      There is far more memory available in the heap area than in the stack area.

Stack Based Memory Organization

24. What is an activation record ?

    Answer:

    An **activation record** is a region of memory on the run-time system stack (also known as the "call stack") allocated for the duration of a function call. The activation record provides storage for the local context. Local (non-static) variables are stored within the activation record. The return address (the address of the instruction following the call instruction) is also stored within the activation record. The return address provides the information needed by the `return` statement to get back to the sequence of instructions following the call.

Algorithms

25. Write a C++ function that finds a target value in an array. Your function should return the position where the target value is found. Return -1 if the target value is not in the array. The following function header gets you started:

```
//  Inputs:
//    A[]  --  an array of integers.
//    n  --  the size of the array
//    target  --  the search target, i.e., find it in A[].
//
int search( int A[], int n, int target )
```

    Answer:

```
//  Inputs:
//    A[]  --  an array of integers.
//    n  --  the size of the array
//    target  --  the search target, i.e., find it in A[].
//
int search( int A[], int n, int target )
{
    bool found ;
    int j      ;

    j = 0 ;
    while ( ( j < n ) && !found ) {
       found = A[j] == target ;
       if ( !found ) j++      ;
    }
    if (found) return j ;
    else return -1      ;
}
```

Problem Solving Skills

26. Write a C++ function to decide if two positive integers `a` and `b` are relatively prime.[1] Use additional functions as needed organize your code. The following function header gets you started:

```
//  Inputs:
//    a  --  a positive integer.
//    b  --  another positive integer
//
//  Output:  Returns true if a and b are relatively prime.
//
bool rel_prime( int a, int b )
```

Answer:

```
#include <cmath>
//  Inputs:
//    a  --  a positive integer.
//    b  --  another positive integer
//
//  Output:  Returns true if a and b are relatively prime.
//
bool rel_prime( int a, int b )
{
   int abmin, sqroot, j ;

   // If a common factor exists, there must be one that is
   // not larger than the square root of the smaller number.
   abmin = min( a, b ) ;  // Built-in standard function: min()
   sqroot = (int) sqrt( abmin ) ;

   // Check to see if 2 divides both numbers 'a' and 'b'.
   if ( ( ( a % 2 ) == 0 ) && ( ( b % 2 ) == 0 ) ) {
       return false;
   }

   // Start with 3, and see if we can find a divisor that
   // divides both 'a' and 'b'.  We consider only odd numbers,
   // since we have already checked divisibility by two.
   for ( j = 3 ; j <= sqroot ; j+=2 ) {
      if ( ( ( a % j ) == 0 ) && ( ( b % j ) == 0 ) ) {
         return false;
      }
   }

   // At this point, no common factor has been found.
   // I.e., no common factor exists.
   return true ;
}
```

[1] Two numbers are relatively prime if they have no common factors.