This worksheet is for your use during and after lecture. It will not be collected or graded, but I think you will find it a useful tool as you learn C++ and study for the exams. Explain all false answers for the “True or False” questions; in general, show enough work and provide enough explanation so that this sheet is a useful pre-exam review. I will be happy to review your answers with you during office-hours, via Email, or instant messaging.

1. True or False: `Calc%02` is a valid C++ function name. If false, state why.

   Solution: False, function names follow the rules for C++ variable names, also called identifiers. `%` cannot be used in C++ identifiers.

2. True or False: You have not written any user-defined functions in the course assignments thus far. If false, name the function or functions.

   Solution: False, in each lab, at least one `int main()` function was written.

3. Write the function definition for a function that returns the sum of `int` and `double` arguments as a double value. Call the function `silly_sum`.

   Solution:

```
double silly_sum( int a, double b )
{
    return a + b;
}
```

4. True or False: The compiler must encounter every function’s prototype or definition before it is used in a program listing. If false, cite a counter example.

   Solution: True.

5. True or False: Every user-defined function must have a prototype *in the same source file* as its function definition. If false, cite a counter example.

   Solution: False. Prototypes may be provided in files used with `#include` statements. Consider, for instance, that the library function protoype `double sqrt( double x );` is provided this way via `#include <cmath>`.

6. (a) Write a function prototype for your answer to question 3.

   Solution: `double silly_sum( int a, float b );`

   (b) Can you write another, equally correct, function prototype for question 3?

   Solution: `double silly_sum( int, float );`
   Omitting argument names from prototypes is generally considered a **poor programming practice**.

7. Write the function definition for a `void` function that does not accept arguments and does nothing.

   (a) Call the function `Super_Void`.

Solution:

```
void Super_Void( void )
{
    return;
}
```

(b) Now, write a different function definition for the same `void` function that differs by one word.

Solution:

```
void Super_Void( )
{
    return;
}
```

```
void Super_Void( void )
{
}
```

```
void Super_Void( )
{
}
```

Either of the first two will work (since the question specifies only one word to be different). In general, these are the 4 ways to write the function definition described. The answer to part a and these three variations.

8. True or False: There are two different techniques for a programmer to show that a function does not return a value.

Solution: False. The only way to declare a `void` function is to use the `void` keyword before the function name in its prototype or definition.

9. Explain in detail the difference between the two function prototypes below, and what they mean to the programmer *using* these functions.

```
int a( int i, string& s );        int b( int& i, string& s );
```

Solution: In the function `a`, the first argument is provide by value, changes to `i` within the statements of `a` does not change the value of `i` in the caller. The function `b` passes `i` by reference, so changes to `i` within the statements of `b` **will** be seen by the caller.

10. What is a `void` function?

Solution: A `void` function does have a return value or return type. It is useful only for its side-effects.

11. Write the prototype for a function that returns a Boolean value and does not accept any arguments. Call the function `DecisionMaker`.

Solution:
`bool DecisionMaker();` or `bool DecisionMaker(void);`

12. (a) What limitation does the return value have for a (non-`void`) function that must communicate values back to its caller?

Solution: A `return` statement can communicate only one value back to the caller.

(b) What mechanism may be used by a function that must communicate more than one data value back to its caller?

Solution: Declare (non `const`) parameters passed by reference, and place extra "return values" in these.

(c) We have learned another technique that might be used — what is it?

Solution: Information could also be passed through files written by a function and read by the caller after the function has completed.

13. (a) Write a function called `CircleMeasures` that is called with the radius of a circle (as a `double`) and calculates the diameter, circumference, and area for the caller.

Solution:

```
#include <cmath>
using namespace std;
/***
 * returns the area, values diameter and circ
 */
double CircleMeasures( double r, double& diameter, double& circ )
{
    const double PI( acos(-1));
    diameter = 2*r;
    circ = diameter*PI;  // or, 2(pi)r if you like
    return PI*pow(r,2);  // area is pi*r
}
```

(b) In what ways might your classmates' answers differ from yours yet still be correct?

Solution: They might return a different measurement than my solution, and provide the other two through reference parameters. Once we learn about `void` functions, they might return all three measurements through reference parameters.

14. Using a `while` loop, write the function definition for `factorial`, a function that returns an integer, accepts a single integer argument `x` and returns the value of $x!$. Write a simple `main` routine that tests this function.

Solution:

```
int factorial( int x )
{
    int f(1);
    while( x > 1 ) {
        f *= x;
        x--;
    }
    return f;
}
```

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

int factorial(int);
```

```
int main()
{
    cout << "Enter␣x␣for␣x!␣" << flush;
    int x;
    if( cin >> x ) {
        cout << x << "!␣is␣" << factorial(x) << endl;
    }
    return 0;
}
```

15. Recall from Calculus II that the Taylor series expansion of $f(x) = e^x$ is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Use your solution for calculating $x!$ above (question 14) to write the function definition for

`double exp_taylor_term( double x, int n );`

that returns the n^{th} term of the Taylor series for e^x.

Solution:

```
// prototype for factorial()
int factorial( int x );

double exp_taylor_term( double x, int n )
{
    // (alternatively, you could use the pow() function
    // from the math library.
    double xpow(1);
    for( int p=1; p<=n; p++ ) {
        xpow *= x;
    }

    return xpow / factorial( n );
}
```

16. Which of the following will the compiler use to distinguish functions by their signatures.

A. Function return type

B. `const`-ness of function return type

C. The `const`-ness of a user-defined class' member function

D. The user-defined class containing a member function

E. The argument types

F. The order of argument types

G. The argument names

H. The order of argument names

I. The `const`-ness of function arguments

Solution: C, D, E, F, I.

17. Consider the following snippet of code:

```
3  void f( int x, double y );
4  int f( const string& s, int x );
5  double f( string& s, int x );
6  double f( string& s, double y );
7
8  int main()
9  {
10     const string greeting( "hello" );
11     string farewell( "good-bye" );
12     int i( f( greeting, 3 ));
13     f( i, f( farewell, 3.0 ));
14     return 0;
15 }
```

For the following line numbers, determine which functions are called, and in what order (identify a function by its prototype's line number).

(a) Line 12.

Solution: `f` on line 4 (the only `f` that takes a `const string` as first argument).

(b) Line 13.

Solution: First `f` on line 6, then its result is provided as the second argument to `f` on line 3.

18. Consider the following snippet of code:

```
3  void f( int& x );
4  void f( const int& );
5  bool f( int& );
6  bool f( double y );
7  bool f( int );
8  bool f( double x );
9  void f( int x );
10 void f();
```

Find the two pairs of functions prototypes with matching signatures that will cause a compiler error.

Solution: The pair on lines 3 and 5. The pair on lines 9 and 7. The pair on lines 6 and 8 won't cause an error because they are prototypes for the same function signature (recall that argument names are ignored). The compiler does not raise an error when it sees the same prototype twice.

19. Consider the code snippet below to answer the questions at the right.

```
4  int add_two( int a, int& b )
5  {
6      b -= 1;
7      a += 1;
8      return a + b;
9  }
10
11 int add_three( int a, int& b, int c )
12 {
13     return add_two( add_two( a, b ), c );
14 }
15
16 int main()
17 {
18     int a = 10;
19     int b = 20;
20     int c = 0;
21
22     cout << add_three( a, b, c ) << "␣";
23     cout << a << "␣" << b << "␣" << c;
24     cout << endl;
25
26     return 0;
27 }
```

(a) Does this snippet represent function composition? On which lines?

Solution: Yes, at line 13. The argument of the outermost `add_two` comes from the return value of an `add_two` call.

(b) How many times is line `8` traversed by the CPU? What are the values of `a` and `b` each time?

Solution: See below.

(c) What is printed by `main()`?

Solution:

```
30 10 19 0
```

Solution: On the next page, I go through this solution in rather fine detail, attempting to explain every step by the compiler and CPU. Once you become comfortable with functions and the passing of variables by value and reference, this much detailed analysis is rarely needed. By the time the next exam arrives, you should be able to work this problem out in about 5 minutes.

Solution: The key to this question is correctly analyzing the steps the compiler and CPU take during the execution of `main`. One way to do this is to "trace" through the code, rewriting it with the explicit steps the compiler automatically adds to the program. I've done this below by adding the temporary variables `u1, t1, t2, s1`.

```cpp
1  #include <iostream>
2  using namespace std;
3
4  int add_two( int a, int& b )
5  {
6      // ON THE FIRST CALL TO add_two, a=10, b& = 20.
7      // ON THE SECOND CALL TO add_two, a=30, b& = 0.
8
9      b -= 1; // SAME AS b = b - 1; <= THIS IS SEEN IN THE CALLER!
10     a += 1; // SAME AS a = a + 1;
11     int s1 = a + b;
12     return s1;
13 }
14
15 int add_three( int a, int& b, int c )
16 {
17     // CALL add_two WITH a=10, b&=20 ONLY THE 2ND ARG MAY CHANGE!
18     int t1 = add_two( a, b );
19     // RETURNS WITH t1=30, a=10 (unchanged), b&=19
20
21     // CALL add_two with t1=30, c=0, ONLY THE 2ND ARG MAY CHANGE!
22     int t2 = add_two( t1, c );
23     // RETURNS WITH t2=30, t1=30 (unchanged), c=-1
24
25     // RETURN THE VALUE t2=30, THE 2ND ARG (b&) IS NOW 19.
26     return t2;
27 }
28
29 int main()
30 {
31     int a = 10;
32     int b = 20;
33     int c = 0;
34
35     // CALL add_three WITH a=10, b=20, c=0, ONLY THE 2ND ARG MAY CHANGE!
36     int u1 = add_three( a, b, c );
37     // RETURNS u1=30, a, c unchanged, and b is now 19.
38     cout << u1 << " ";
39     cout << a << " " << b << " " << c;
40     cout << endl;
41
42     return 0;
43 }
```

All that remains is to trace through the lines of execution, tracking the changes to variable values. The solution to the second portion of part b are **shown in bold**.

	main vars				add_three vars					add_two vars		
Line(s)	a	b	c	u1	a	b	c	t1	t2	a	b	s1
Before call on line 36	10	20	0									
main calls add_three												
Before call on 18	10	20	0		10	20	0					
add_three calls add_two												
After line 9	10	19	0		10	19	0			10	19	
After line 10	10	19	0		10	19	0			11	19	
After line 11	10	19	0		10	19	0			**11**	**19**	30
add_two returns s1=30												
Before call on 22	10	19	0		10	19	0	30				
add_three calls add_two												
After line 9	10	19	0		10	19	-1	30		30	-1	
After line 10	10	19	0		10	19	-1	30		31	-1	
After line 11	10	19	0		10	19	-1	30		**31**	**-1**	30
add_two returns s1=30												
After call on 22	10	19	0		10	19	-1	30	30			
add_three returns t2=30												
After call on line 36	10	19	0	30								

20. Below is a snippet of C++ source from a file named `oops.cxx`, and below that is an error listing generated during a build attempt. Use the space at the right to explain how the code may be changed to resolve each error message.

```
1  #include <string>
2  #include <cmath>
3  using namespace std;
4
5  int f( int a );
6
7  string g( double f );
8
9  void h( int m, string& n )
10 {
11     return;
12 }
13
14 int main()
15 {
16     double product = j();
17     string text( "abc" );
18     int result;
19     double product_string;
20
21     product_string = g( product );
22
23     result = h( f(’A’)+h(0,text), "xyz" );
24
25     return 0;
26 }
27
28 double j( void )
29 {
30     return acos(-1)*exp(1);
31 }
32
33 double f( int a )
34 {
35     return j() + a;
36 }
```

Solution:

Line 16

A prototype for function `j` is not provided before `j` is used in `main`. Add a protoype for `j` or place its entire definition before the function `main`.

Line 21

`product_string` is declared as a `double`, and its value is set to the return value of `g`. But the prototype of `g` says it returns a `string`, not a `double`. The compiler can automatically convert between different integer types, and integer types to `double`s, but it won’t automatically convert a `double` to a sequence of ASCII characters.

Line 23

The function `h` is a void function — it doesn’t return a value. But we are attempting to store its “return value” in `result`. Remove the assignment of `result` to correct this line of code.

Line 33

The prototype for function `f` does not match its function definition. Change the return types of `f` so that they match.

undefined reference

This is a linking error. A prototype for function `g` is provided, but the function definition, or implementation, has not been provided. Providing the definition for `string g( double )` corrects this.

ERROR LISTING

```
oops.cxx:16:  error:  ’j’ was not declared in this scope
oops.cxx:21:  error:  cannot convert string to double
oops.cxx:23:  error:  void value not ignored as it ought to be
oops.cxx:33:  error:  new declaration ’double f(int)’ ambiguates old declaration
undefined reference:  string ::g( double )
```

Beginning with question 21, different functions for you to write are described, each using arrays in a slightly different and increasing complex way. I have not attempted to provide space for the answers. Use an additional piece of paper, or better yet, a computer where you can test your solution's correctness. The solutions for all of these excercises are provided in one location at the end of the solution PDF. I will also provide a simple testing program for download (see **my** course website) so that you can easily test the correctness of your code.

21. In statistics, the mean or average value of a set of n data points is the defined by:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Write a C++ function, named `average`, that accepts two arguments: a `const` array of doubles, and a constant integer representing the number of elements in the double array. The function should return $\bar{x}$.

22. Write a void function that accepts six arguments: a `const int` array of data, an `const int size` argument for the array, an `int` argument named `value`, and two `int`s passed by reference named `near` and `far`. The function should place into the `near` and `far` the element offset of the array value *nearest* `value`, and the element offset of the value *farthest* from `value`.

For instance: if the argument is passed an array `{ 1, 13, 5, 8, 21, 2, 13, 3 }` with `value` 6, it should place into `near` the element offset of the element value 5, and into `far` the element offset of the element value 21.

23. The Fibonacci sequence is given by:

$$a_1 = a_2 = 1, \quad a_i = a_{i-1} + a_{i-2}$$

This sequence of numbers has a long history of study in both the mathematical[1] and biological world.[2] Now it's time to study them in the wonderful world of C++ arrays. Write a function, aptly named `fibonacci` that accepts an array of integers and a constant argument specifying the size of the array. `fibonacci` should initialize the array with as many terms of the Fibonacci sequence as it will hold.

24. Recall from Calculus II that all our favorite trigonometric functions have associated ∞-series (remember the names MacLaurin and Taylor?), for instance:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

(a) Write a void function named `sine_terms` that accepts a non-`const double` array (call it `terms`), a constant `size` argument for the array, and a `double` variable named `x`. The function should place into `terms` the first `size` terms of the MacLaurin series of $\sin x$.

(b) Calculus II also introduced the nth partial sum S_n of an ∞-series $\sum_{i=1}^{\infty} a_i$ as simply the sum of the first n terms:

$$S_n = \sum_{i=1}^{n} a_i$$

Now write the function `sine_sums` that accepts all the same arguments as in part a as well as a second non-`const double` array named `sums`. `sine_sums` should first call `sine_terms`, and then calculate the partial sums of the terms and place their values into `sums`. `sine_sums` should return the $S_{\texttt{size}}$ partial sum.

Solution: A simple `main()` routine is provided on my course website for testing your code. When I run it against my own solution, I get the following output:

[1] It is connected to the golden ratio in an incredibly elegant way.
[2] The minimal packing pattern of sunflower seeds, the turns and ridges of sea shells, . . .

```
The average of 1 and 0 zeros is 1
The average of 1 and 1 zeros is 0.5
The average of 1 and 2 zeros is 0.333333
The average of 1 and 3 zeros is 0.25
The average of 1 and 4 zeros is 0.2
The average of 1 and 5 zeros is 0.166667
The average of 1 and 6 zeros is 0.142857
The average of 1 and 7 zeros is 0.125
The average of 1 and 8 zeros is 0.111111
The average of 1 and 9 zeros is 0.1
The average of 1 and 10 zeros is 0.0909091
The average of 1 and 11 zeros is 0.0833333
The average of 1 and 12 zeros is 0.0769231
The average of 1 and 13 zeros is 0.0714286
The average of 1 and 14 zeros is 0.0666667
The average of 1 and 15 zeros is 0.0625
The average of 1 and 16 zeros is 0.0588235
The average of 1 and 17 zeros is 0.0555556
The average of 1 and 18 zeros is 0.0526316
The average of 1 and 19 zeros is 0.05
The first 20 Fibonacci terms are 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
Fib term 1 is closest to -1
Fib term 6765 is farthest from -1
Fib term 6765 is closest to 10000
Fib term 1 is farthest from 10000
Fib term 4181 is closest to 4184
Fib term 1 is farthest from 4184
Fib term 2584 is closest to 2585
Fib term 6765 is farthest from 2585
Fib term 5 is closest to 6
Fib term 6765 is farthest from 6
sin(3/2 * PI) = -1 (should be -1)
sin(1) = 0.841471 (should be 0.84147...)
```

Here are my solutions for the programming problems:

```cpp
/***
 * Calculate an average
 */
double average( const double data[], const int size )
{
    double sum(0);
    for( int i(0); i<size; i++ ) {
        sum += data[i];
    }

    return sum/size;
}

/***
 * Fibonacci
 */
void fibonacci( int fib[], const int size )
{
    // 1, 1, 2, 3, 5, 8, 13, ...
    if( size >= 1 ) {
        fib[0] = 1;
    }
    if( size >= 2 ) {
        fib[1] = 1;
    }
    // fill in the rest
    for( int i(2); i<size; i++ ) {
        fib[i] = fib[i-2] + fib[i-1];
    }
}
```

```
/***
 * returns the greater of two ints
 * (a utility function for locality)
 */
int intabs( int a )
{
    // this is a common use of the ternary operator (see chapter 2
    // lecture slides)
    return a >= 0 ? a : -a;
}

/***
 * Fill nearest and farthest with the offsets of the
 * data elements that meet the named criteria.
 */
void locality( const int data[], const int size, int value,
        int& near, int& far )
{
    // a wise precaution
    if( !size ) return;

    // init with first element
    near=0;
    far=0;
    int d( value-data[0] );
    int near_distance = intabs(d);
    int far_distance = near_distance;

    for( int i(1); i<size; i++ ) {
        d = value-data[i];
        d = intabs(d); // abs value
        // check closest
        if( d < near_distance ) {
            near_distance = d;
            near = i;
        }
        // check farthest
        if( d > far_distance ) {
            far_distance = d;
            far = i;
        }
    }
}
```

```
/***
 * sine_terms
 */
void sine_terms( double terms[], const int size, double x )
{
    // be safe
    if( !size ) return;

    // the factorial term
    int factterm(1);
    // the denominator factorial value
    double fact(1);  // factorial of denominator
    // tracks the sign of each term
    double neg(1);    // first term is positive
    // the current power of x
    double powx(x);
    // powers of x increase by x squared
    double xsquared(x*x);

    /****
     * Note that neg, powx, and fact are already initialized
     * for the first term
     */

    for( int i(0); i<size; i++ ) {
        // fill in the term
        terms[i] = (neg*powx)/fact;
        /****
         * calculated factors for the next term
         */
        // sign change
        neg = -neg;
        // numerator = numerator * x * x
        powx *= xsquared;
        // factorial increases by two integer values
        fact *= ++factterm;
        fact *= ++factterm;
    }
}
```

```
/***
 * sine_sums
 */
double sine_sums( double terms[], const int size, double x, double sums[] )
{
    // get the terms
    sine_terms( terms, size, x );

    // calculate partial sums
    double s(0);
    for( int i(0); i<size; i++ ) {
        s += terms[i];
        sums[i] = s;
    }

    // return the partial sum of the first size terms
    return sums[size-1];
}
```

25. In mathematics, a *matrix* is a $p \times p$ grid of real ($\Re$) or complex ($\Re + \Im$) numbers. Each element of a matrix A is denoted by a_{ij} where i and j are the row and column *numbers*, respectively. The *transpose* of a matrix A is A^T, it is a matrix with A's elements reflected across the main upper-left to lower-right diagonal. These examples show the case for $p = 3$.

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \mathrm{A}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

More succinctly, $A^T_{ji} = A_{ij}$. Write a C++ void function `transpose` that works on square matrices (the number of rows is equal to the number of columns) with a maximum dimension of `MAXP`. The matrix should be represented in C++ by a `MAXP`$\times$`MAXP` double array. `transpose` should accept the array data, and a single constant integral parameter `p` ($1 \leq$`p`$\leq$`MAXP`) representing the size of the input array. `transpose` should turn the array argument into its transpose, this is called taking the transpose *in place*.

Solution:

```
void transpose( double array[][MAXP], const int p )
{
    double temp;
    // from the first to the last row
    for( int row(0); row<p; row++ ) {
        // from the first column to one column before the
        // main diagonal
        for( int col(0); col<row; col++ ) {
            // swap
            temp = array[row][col];
            array[row][col] = array[col][row];
            array[col][row] = temp;
        }
    }
}
```

26. In mathematics, a vector $\boldsymbol{x}$ is a one dimensional array of numbers, typically visualized "vertically".

Vectors may multiply arrays (on the array's right-hand side) and the result is another vector. We write the following for the $p = 3$ case in mathematics:

$$\mathrm{A}\boldsymbol{x} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 \end{bmatrix}$$

Note that the product $\mathrm{A}\boldsymbol{x}$ is another vector of size p, each element of which is the sum of p products. The size, or dimension, of $\boldsymbol{x}$ must agree with the number of columns in A.[3]

Vectors in C++ are typically represented by 1*d* arrays. Complete the C++ void function below. `Ax` should calculate the product $\mathrm{A}\boldsymbol{x}$ for any arbitrary $1 \le p \le$`MAXP`.

```
void Ax( const double A[MAXP][MAXP], const double x[],
        double product[], const int p )
```

Solution:

```
void Ax( const double A[MAXP][MAXP], const double x[],
        double product[], const int p )
{
    // for each row of the matrix A
    for( int arow(0); arow<p; arow++ ) {
        double sum(0);
        // for each component of the vector
        for( int xrow(0); xrow<p; xrow++ ) {
            // multiply it by the column of the matrix row
            sum += A[arow][xrow]*x[xrow];
        }
        // store in the vector product
        product[arow] = sum;
    }
}
```

[3]For purposes of this worksheet, we are only interested in square matrices. So the dimension of $\boldsymbol{x}$ will also equal the number of rows in A. But this is not the general mathematical case! In general, a vector $\boldsymbol{x}$ of dimension n may multiply any matrix with dimensions $m \times n$, resulting in a new vector $\boldsymbol{y}$ that is m dimensional (the number of rows of A), and each component of $\boldsymbol{y}$ is the sum of n products.