

1. Timey-Wimey

Recall the following documentation for the `time()` function (defined in `<ctime>`):

```
/**  
 * This function takes in a pointer to a time_t variable, and assigns to that  
 * variable the number of seconds since Jan 1, 1970 00:00 UTC, then returns  
 * this value as a time_t variable. Given a nullptr input, it simply returns  
 * the number of seconds since Jan 1st, 1970.  
 * @param timeptr the pointer to time_t  
 * @return the number of seconds elapsed since Jan. 1st, 1970.  
 */  
time_t time(time_t *timeptr);
```

Louis Reasoner wants to write a program that times the number of seconds between user inputs. For instance, the program should behave as follows:

```
Press enter to start the timer!  
<you pressed enter!>  
Press enter again to stop the timer...  
<you pressed enter!>  
Time elapsed: 12 seconds.
```

Does the following program behave correctly? If not, explain why.

```
#include <ctime>  
#include <iostream>  
using namespace std;  
int main() {  
    time_t t;  
    cout << "ENTER to start the timer!" << endl;  
    cin.get(); // Wait for user to hit enter  
    cout << "<you pressed enter!>" << endl;  
    time(&t); // Initialize timer t  
    cout << "Press enter again to stop the timer...\\n";  
    cin.get(); // Wait for user to hit enter again  
    cout << "<you pressed enter!>" << endl;  
    time_t duration = time(nullptr) - t;  
    cout << "Time elapsed: " << duration << " seconds.\\n";  
    return 0;  
}
```

[\[Solution\]](#)

It's correct! Good job Louis.

2. Hole in my Vector

Write a function `vecremove(vector<int>&vec, size_t ind)` that removes the element at index `ind`. Make sure that the elements to the right of "ind" are shifted over. If the input index is out of bounds, then the function should not modify the vector - it must not crash:

```
vector<int> v = {1, 2, 3, 4};  
vecremove(v, 1); // v is now: [1, 3, 4]  
vecremove(v, 0); // v is now: [3, 4]  
vecremove(v, 2); // v is still: [3, 4]
```

You may not use `vector::erase`.

[Solution]

```
void vecremove(vector<int>& vec, size_t ind) {  
    // YOUR CODE HERE  
    if (ind >= vec.size()) {  
        return;  
    }  
    /* Shift all elements past ind to the left */  
    for (size_t i = ind; i < (vec.size()-1); ++i) {  
        vec[i] = vec[i+1];  
    }  
    /* Pop off last element to reduce size */  
    vec.pop_back();  
}
```

3. There Is No Compiler...

For the following, write down the expected output. If an error occurs, describe why. Assume all headers have been included, and that we are using the standard namespace:

int d = 42; int* dp = &d; *dp = 43; cout << *dp << " " << d;	43 43
double x = 42.4; (*x) = (*x) + 1; cout << x;	Error: x is not a pointer, can't dereference!
double x = 13.42; double &xr = x; xr = xr + 1; cout << xr << " " << x;	14.42 14.42
double x = 1.1; double y = 3.2; double &xr = x; xr = &y; xr += 1;	Error: Can't change what a reference points to! ie "xr = &y;" is an error.

cout << xr << " " << y;	
int x = 1; int y = 3; int *xr = &x; xr = &y; xr += 1; cout << xr << " " << y;	002DFDC4 3 Note: When we do "xr = &y", we assign the address of y into xr. Thus, "002DFDC4" is displayed (or whatever the address of y happens to be).
int x = 1; int y = 3; int *xr = &x; xr = &y; (*xr) += 1; cout << *xr << " " << y;	4 4
string s("Spoon"); string *sp = &s; cout << sp.size();	Error: sp is a pointer, not an object! Can't use dot-notation on pointer. Should have done: cout << sp->size(); Recall: sp->size() is equivalent to: (*sp).size()

4. Iterators

Consider the following for-loop that squares all of the even numbers within a vector:

```
vector<int> v = {1, 2, 3, 4, 5, 6};  
for( size_t i = 0; i < v.size(); ++i ) {  
    if ((v[i] % 2) == 0) {  
        v[i] = v[i]*v[i];  
    }  
}
```

Rewrite the above loop but using iterators (ie using begin, end, vector<int>::iterator, etc.).

```
// YOUR CODE HERE
```

[Solution]

```
for (vector<int> v::iterator iter = begin(v); iter!=end(v); ++iter) {  
    if ((*iter)%2) == 0)  
        (*iter) = (*iter)*(*iter);  
}
```

5. Best Function Names NA

Consider the following function definitions/declarations. If there are any issues, explain why. Assume all relevant headers are included, and we are using the standard namespace.

double foo(int a, string s) { return s[a + 1]; } Error: foo should return a double, but returns a char.	void bar(int x) { return x + 1; } Error: bar is declared as returning void, but here it returns an int.
char baz(string s) {	string garply(string str, int n) {

<pre> return s[0]; } Correct! s[0] is indeed a char. </pre>	<pre> return s[n]; } Error: s[n] is a char, but garply should return a string! </pre>
<pre> void zzz() { } This is fine, no issue. </pre>	<pre> string hmm() { return 'a'; } Error: 'a' is a char, but hmm() should return a string! </pre>
<pre> double meow(int x) { return x; } This is OK - C++ will simply cast the "int x" into a double. </pre>	<pre> int bark(double d) { return d; } This is OK - however, C++ will cast the "double d" into an integer, ie truncate, so you may lose information, ie 3.14 -> 3. </pre>

6. Swapping Numbers

Part 1: Write a function swap1 that swaps the values of two int variables:

```

int a = 42, b = 16;
swap1(a, b);
cout << a << " " << b; // Outputs: 16 42
// YOUR CODE HERE

```

[Solution]

```

void swap1(int &x, int &y) {
    int tmp = x;
    x = y;
    y = tmp;
}

```

Next, consider the following function:

```

void swap2(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

```

Say I have two integers a and b. How do I call swap2 to swap their values?

```

int a = 2, b = 1;
// YOUR CODE HERE
swap2(&a, &b);
cout << a << " " << b; // Outputs: 1 2

```

7. Not That Kind of Chunks...

Write a function `sort_chunks(vector<int>&v, int k)` that sorts each consecutive group of k elements, ie several "local" sorts, rather than a single global sort:

```
vector<int> v = {4, 1, 2, 3, 0, 5, 9, 18, 1};  
// Sort each group of 3 elements  
sort_chunks(v, 3); // v is now: [1, 2, 4, 0, 3, 5, 1, 9, 18]  
vector<int> v2 = {4, 1, 2, 9, 1};  
sort_chunks(v2, 3); // v2 is now: [1, 2, 4, 1, 9]
```

Note: the length of v may not necessarily be a multiple of k - be sure to handle this case.

You should use the `sort(left_iterator, right_iterator)` function defined in `<algorithm>`:

```
vector<int> v = {0, 3, 2, -1};  
sort(begin(v), end(v)); // v is now: [-1, 0, 2, 3]
```

[Solution]

```
void sort_chunks(vector<int>&v, int k) {  
    // YOUR CODE HERE  
    vector<int>::iterator iter = begin(v); // Initialize iter to v start  
    size_t i = 0;  
    while (iter != end(v)) {  
        /* Determine current chunk size */  
        size_t chunk_size = k;  
        if ((i + k - 1) >= v.size()) {  
            /* Chunk would go past bounds - shrink this chunk */  
            chunk_size = v.size() - i;  
        }  
        /* Sort current chunk */  
        sort(iter, iter + chunk_size);  
        /* Move iter forward to next chunk */  
        iter = iter + chunk_size;  
        i = i + chunk_size  
    }  
}
```

The trickiest part here is correctly handling the case when `v.size()` is not a multiple of k. Personally, I have to do an example scenario on paper to figure out what the correct offsets/indexing is. What's neat here is how flexible iterators are: we can increment them just like integers, yet they are objects - neat!