About File Formats

DO NOT copy and paste from PDF files into your code or input files! PDF files contain hidden characters that will make your output incorrect. All of our projects provide sample output in text files, copy and paste from there.

Unfortunately, since DOS was created by Microsoft (long before Windows), they have formatted their text files differently from everyone else, adding an extra character to mark the end of a line. What this means for you is that you must use files that are formatted correctly for the operating system you're on if you use `getline()`.

If you're on a Mac, everything just works for you.

If you're using Windows and Visual Studio, make sure that you have Visual Studio 2017 or newer, and that it's fully patched. These versions seem to be able to read input files in any format (Windows or Unix newline characters), and you shouldn't have any trouble.

About Data Types

We will refer to simple types such as `int`, `double`, `bool`, `size_t`, etc. as primitive, or built-in types. A pointer to any type is itself a primitive type.

If you find yourself having to `static_cast` over and over in your code, that gets really unreadable. Use a different data type instead! For an index, use a `size_t`, or better yet `uint32_t` (4 bytes instead of 8). You might say "but wait, now I can't use -1 for uninitialized". Use a better value. If you `#include <limits>` you gain access to things like this:

```
const uint32_t UNINITIALIZED = numeric_limits<uint32_t>::max();
```

Memory Optimization

Classes and Structures

For a single variable, or a few variables inside a function, don't worry about how big it is, use what makes sense. Worry about them when you make an entire data structure of them, and might have to read 100,000 of them from a file.

When creating a class or structure, create member variables in order from largest to smallest. For example: objects (usually bigger than a single variable) should come first, followed in order by `double` (8 bytes), `long long` (8 bytes), `int` and `float` (4 bytes), `short` integers (2 bytes),

`char` (1 byte), `bool` (1 byte). This is because a variable must start at a memory address that is a multiple of its size. Think about what happens if your structure has an `int`, then a `double`, then a `bool`… There ends up being gaps of wasted space between them.

If you have to use dynamic memory, always write `new` and `delete` in pairs. As soon as you write a `new`, write the corresponding `delete` immediately so that you don't forget and leak memory. Use data structures to your advantage: it's better to have a class or structure with 15 variables inside of it, and create a vector of objects, than to have 15 vectors. NEVER use global variables! They seem harmless enough in a 100 line program, but in a real system (which might have over half a million lines of code), how do you find the one line of code that's modifying a global variable incorrectly, when someone forgot to create a local variable? Using global variables is a crutch that can come back and haunt you later, especially during job interviews. Whenever you find yourself thinking "I could just use a global variable", instead think "I could create a class, add member variables, and make my functions member functions".

Although you should definitely put objects first inside of a data structure, don't confuse the size of a vector with the `sizeof(vector<int>)`. The `sizeof()` operator (yes it looks like a function but it's actually an operator) gives you how much memory is occupied by that type, but something like a vector or a string could have more memory allocated on the heap. Using g++ 6.2.0, the `sizeof()` a vector is always 24 bytes, but that's just the member variables (which includes a pointer to a block of memory on the heap, but not the amount of memory being pointed to).

When choosing what type of variable to use, think about the possible range of values that might be stored.

A `long double` (in g++) can store approximately 18 decimal digits of precision, a `double` can store approximately 15 digits, a `float` can store approximately 7. For the extra 3 digits of precision, a long double isn't really worth double the memory cost unless you're doing very exacting scientific calculations.

The `size_t` type can be used whenever you're referring to the size of a container (such as a `vector` or `deque`)., or a `uint32_t` The number of bytes and the range of values in a `size_t` may be different depending on the compiler or compiler settings. To avoid issues of types being different sizes for different compilers or compiler settings, consider using one of the standard sized integer types (introduced with C++11) in `<cstdint>`:

Type	**Bytes**	**Minimum Value**	**Maximum Value**
`uint64_t, size_t`	8	0	18,446,744,073,709,551,615
`int64_t`	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

`uint32_t`	4	0	4,294,967,295
`int32_t`	4	-2,147,483,648	2,147,483,647
`uint16_t`	2	0	65,535
`int16_t`	2	-32,768	32,767
`uint8_t`	1	0	255
`int8_t`	1	-128	127

Return Status and The `exit()` Function

If your program "succeeds", it should have a return status of 0, if an error occurs that your program cannot handle (such as completely invalid input), the return status should be 1.

Do not call `exit()` unless there is no other way to terminate from an error condition (such as an invalid command line). The reason for this is that when you call `exit()`, normal program shutdown does not happen! Object destructors do not run, dynamic memory does not get deleted, and this can lead to lost points on memory checking. If your program is exiting because the input (either command line or input file) is invalid and cannot be processed, you should `exit(1)` or `return 1` from `main()`. For any input that you can process, `main()` should `return 0`. The autograder uses these return values! For any "invalid" test case the autograder will ignore your program's output, and expect the exit status to be 1. The `valgrind` program will consider a test where you have an exit status of 1 to be invalid, and the autograder will not run leak checking.

Before you call `exit(1)`, make sure to display a meaningful error message to `cerr` (the error output stream). If you don't, you're left wondering why the program ended without saying anything, and the faculty and staff can't help you when we look at the autograder output. The `cerr` stream is never graded by the autograder, but it is kept in a file that we can view.

Using valgrind

Do not *just* use `valgrind` to check for segmentation faults. It can tell you about uninitialized variables, memory leaks, and other issues. Make sure that you build in debugging mode (either using `-g3` when compiling by hand, or `make debug` with our supplied `Makefile`) so that you see line numbers in your code where errors occur. Make sure you do this on CAEN, because any issues that `valgrind` sees might cause you to lose points on the programming projects. Check the autograder FAQ page for reasons why the autograder might report that leak checking could not be run.

Run your program with valgrind, even if you think it's working correctly! Something like this:

```
make debug
valgrind ./program -flags < input.txt
```

Watch for error messages, and the call stack that follows the error message. Look for the highest numbered line of code that is yours, and debug the problem.

Vectors and Deques

When creating a `vector` or `deque`, if possible create it with the correct size. This actually saves both time and memory. When you grow a `vector` via `.push_back()`, it doubles in size each time it gets full, for instance growing from 1024 to 2048. However, if you knew in advance that you were going to have 1200 items to store in it, forgetting to size the `vector` would result in 848 extra locations (2048 - 1200), or 71% more space than necessary. For instance, you can create a `vector` of the correct size:

```
int n;
cin >> n;
vector<int> data(n);
// Or vector<int> data(n, 0) to explicitly show that all items
// are initialized to 0, the two are equivalent.
```

Alternatively you can create the `vector` up front, then resize it later:

```
vector<int> data;
int n;
cin >> n;
data.resize(n);
```

If you use either of these methods, the `vector` is now of size `n`. To put items into the `vector`, use a `for` loop with subscripts, putting them in `data[i]`. If you start using `.push_back()`, you will grow it to `2 * n`, with the first `n` elements all being 0!

```
// Read the data
for (int i = 0; i < n; ++i)
    cin >> data[i];
```

If you want to use `.push_back()`, use `.reserve()` instead of resize. You will also need a temporary variable to read each item into, before pushing it into the vector:

```
vector<int> data;
int n, temp;
cin >> n;
```

```
data.reserve(n);

// Read the data
for (int i = 0; i < n; ++i) {
    cin >> temp;
    data.push_back(temp);
} // for
```

Think of a `vector` as having two sizes: the number of items *currently* in the `vector`, and the *capacity*: the maximum number of elements that it can hold before it has to resize (this is the size of the dynamic memory allocated by `new`, inside the member functions). The `.resize()` member function changes **both** the capacity and the current size numbers, while `.reserve()` changes the capacity. When you use `.resize()` and the container increases in size, new elements **are** initialized: objects get their default constructor run (for example, `string` objects would be set to the empty string), while all primitive types are initialized to 0.

The `deque` data structure does not have a `.reserve()` member function, only `.resize()`.

When you need data, put it together in one `class` (or `struct`) if it needs to be together, otherwise separate it. For example, Project 1 needs information about the data you read from the file, but the deque needs information about the "next" location to be searched. Make two separate data structures (one for inside the `vector`, one for inside the `deque`), rather than one large data structure contained within both.

Speed Optimization

Speeding up I/O

As the very first line of `main()`, before any other code, do the following:

```
std::ios_base::sync_with_stdio(false);
```

This turns off what is called "synchronized I/O", which means that you're not going to use both C-style (such as `printf()` and `scanf()`) and C++ style (`cin`, `cout`) in the same program. If you were doing both, you would need the compiler to keep your output *synchronized*: if a `cout` was executed before a `printf()`, the output of `cout` should appear first. Since you're unlikely to be using both, telling the compiler that it doesn't need to synchronize the I/O makes I/O much faster.

Why are we having you do this? Because if some students knew this and others didn't, the ones who knew it would have an easier time getting higher scores on the autograder. Thus we give this knowledge to everyone, and expect you to use it in every project. If your program is reading a 1KB input file it doesn't make a difference, but a 1MB file it does…

One warning about this code, it can make some memory show up as “still reachable” when you run valgrind (around 126,880 bytes). The autograder will suppress any memory lost due to `sync_with_stdio()`, and not count it against you.

Another thing that speeds up output is to use `'\n'` instead of `endl`. The reason here is that `endl` doesn't just advance to a new line, it tells the OS to flush your buffer, meaning the data must be written to its final location (such as a file on the server) right now, not saved in memory and written later, when more can be written at once (which saves time).

You may have heard that you can use an `ostringstream` buffer to speed up output, or fill a `string` object with your output. As of g++ 6.2.0 and later the `ostringstream` does not help, and the `string` object makes it slower. Both use more memory.

Function Call Arguments

When passing parameters to a function, follow a few rules, in order:

1) If it must be modified, pass it by reference.
2) Else if it is a primitive type, pass it by value.
3) Else if it is an object, pass it by `const` reference (i.e `const string &name`).

When you pass a primitive type by reference, it can be **slower** than passing it by value! When a variable is passed by reference, behind the scenes what may happen is that a pointer to the variable is passed by value. So instead of passing a 4 byte `int`, you're passing an 8 byte pointer to an `int`. Also, instead of accessing the `int` directly, you have to follow the pointer to where the `int` is located. This effectively doubles the amount of time to access the actual value, and uses up an extra 4 bytes on the call stack.

Code Organization

If code “goes together”, put it together. For example, if I had to manipulate `Student` objects, I would put the declaration in `Student.h` and the implementation in `Student.cpp` (more on this below).

When you're writing a class, implement small member functions (especially getters and setters) inside the header file. If it's a larger function that isn't called often, implement it in the .cpp file. This is due to code inlining. What does that mean? When function `foo()` calls function `bar()`, you've learned that there's a compiled version of `foo()` and a compiled version of `bar()`, and that the information needed to call bar() is put on the call stack, etc. However, if you're using code optimization (the `-O3` flag) the compiler tries to inline every function possible. What this means is that instead of there being one compiled version of `bar()`, the code that makes up `bar()` is actually substituted inside of `foo()`! If `foo()` calls `bar()` twice, there ends up being two copies of the `foo()` code inside of `bar()`. This makes the executable bigger but it runs

faster, because there's no need to put anything on the call stack. This is why we suggest writing small functions like getters and setters inside the .h file, because then they can be inlined by any code that calls them. If you put them inside the .cpp file, they can only be inlined within the same .cpp file, other .cpp files would have to use the call stack.

These suggestions apply to standalone functions also. If you have two functions that call each other numerous times, put them in the same .cpp file. If `foo()` only calls `bar()` once, it doesn't matter whether they're in the same .cpp file or not. Don't avoid making multiple .cpp files just to make things fast! Organize the code well, and put functions that call each other often in the same .cpp file.

Write functions to avoid code duplication. If you find yourself writing the same code twice, take a step back and think about how you can use a function and then write the code once. Otherwise you'll spend too much time trying to fix a bug in one version, make a parallel change in the other version, etc.

You can add command line options that are not required by the project specification! For example, you could add a `--debug` flag, a `--verbose` flag, etc. Just make sure that the corresponding code does not do too much work when these options are not specified, otherwise it might slow down program execution.

Use of STL Functions

When using STL functions, or member functions for STL containers, be aware of exactly what it's doing under the shroud of the function call and what the complexity of doing so is. Sometimes it's not what you wanted!

For example, inserting an element at an arbitrary position in a `std::vector` is a single call to `vector::insert()`, but because `std::vector` is represented "behind the scenes" as an array, to insert into the middle requires an $O(n)$ shift of all the elements, which can end up being very expensive depending on how many times the operation is performed. To find the complexity of any given STL function, you can always check its documentation (for example here or here, search for the word "Complexity").

Data Structures

If I were making a program to manipulate students, I would not create a vector of student names, a vector of student addresses, a vector of student zip codes, etc. I would create a `Student` object, and create a `vector<Student> students`. This not only improves code organization, it improves speed! To understand why this happens, you need to know a little bit about how computers work.

RAM (random access memory) is slow compared to the CPU. Reading one value from a vector might take the same amount of time as performing 20 additions or subtractions. To help alleviate this problem, computers are designed with a cache -- an area of memory that is smaller than RAM but much faster. This cache keeps copies of things from RAM that have been used recently. When I want to read `data[i]` and there's no copy in the cache (called a *cache miss*), the computer spends the time to load data[i] from RAM and keep it in the cache. However, since I'm very likely to be looking at `data[i + 1]` in the near future, and since reading multiple values from slow RAM doesn't take much longer than reading one value, the cache actually makes copies of `data[i]`, `data[i + 1]`, `data[i + 2]` … `data[i + k]`, where `[0...k]` represents the largest number of values that can be read at once (`k` depends on the computer, data type, etc.). Now when my loop increases `i` and reads the next value from `data`, it's already in the cache (called a *cache hit*), and can be accessed very quickly!

If I had two vectors (say student names and student zip codes), I would get a cache miss on `names[i]`, and another cache miss on `zips[i]`. If instead they're in the same vector, I would only get one cache miss, on `students[i]`. This is why having one vector of data can be faster than two.

Red Herrings (i.e., performance optimization *don't*s)

Part of knowing what *is* effective for optimizing programs is knowing what *isn't* effective and your time is better spent looking at other things to improve performance. These are a few techniques that we commonly see that are attractive to students because they *appear* to produce faster code, but in reality have little, if any, impact on performance:

Coding densely
Say you want a function to find the area of a cone. Which version runs faster?

```
double cone_area(double r, double h) {
    double base_area = PI * r * r;

    double slant_length = r + sqrt(h * h + r * r);
    double slant_area = PI * r * slant_length;
```

```
    return base_area + slant_area;
} // cone_area()
```

Or

```
double cone_area2(double r, double h) {
    return PI * r * r + PI * r * (r + sqrt(h * h + r * r));
} // cone_area2()
```

It's easy to think that because `cone_area()` creates all these variables and splits up the computation that it will be slower than `cone_area2()`, where all the computation happens in a single expression. But the strength of compiler optimizations is such that intermediate values, like in `cone_area()`, can be automatically computed without *actually* being stored in variables.

Now, given that both functions will produce the same machine instructions, which version is more readable? If there is a mistake in one of the calculations, which will be easier to debug? Lastly, if you were having trouble with this function in office hours, which version would make it easier for an instructor to figure out what you're trying to do?

Note that while this here is a toy example, you could easily have a much more complex computation sequence where being able to properly name and see the contents of intermediate results makes your life much easier both in writing and debugging the program.

Function calls and code organization
There is sometimes a belief that the use of functions introduces a significant amount of overhead, and that the same code inlined "by hand" (i.e., copied and pasted in place of the function call) will result in better performance. In its worst form, this leads to excessive amounts of code duplication and will cause you massive headaches with debugging and making changes to that segment of code. In reality, compiler optimizations will take care of inlining for you (hooray!), and in the cases where it cannot (the function is large, has weird side effects, etc.), the overhead of calling the function is likely insignificant.

Along the same lines, there is a tendency for students to believe that a data type declared as a `struct` is faster to access or create than a data type declared as a `class`. In C++, structs and classes are nearly identical, with the sole difference being whether their members have public or private access by default. This is limited by the "Code Organization" rules (see above).

Again, these are cases where proper code organization is *encouraged* by C++, which has the design goal of not punishing the use of proper abstractions with runtime costs. Avoiding the above, and in general caring about code organization, will make it easier both for the student to write and debug the projects, and for the instructors to help you.

Reading Input

Read input once, and be efficient about it. Do not use string streams for input! I see students every semester who read an entire line of input with `getline()`, then create a stringstream from that string, then read from the stringstream using >>. This wastes time by processing the same input three times! Just read using >> to begin with.

When using >>, whatever variable follows it will be read if possible. For instance, if your input looks like "`$123.45`" (without the quote marks), use >> to read into a `char`(the dollar sign) then read into a `double`. Remember that >> skips white space (the space character, <Tab>, <Enter>, etc.). If you really want to read a character, even if it is white space, use `cin.get()`.

When switching between >> and `getline()` be careful because >> does not consume the <Enter> that follows, and if you use `getline()` immediately afterward, your string variable might be empty (it reads the `\n` and nothing else). If you know that you're going to be at the end of a line, you can simply call `getline()` twice, and ignore the first result.

Exam Grade Optimization

Students often have trouble with coding questions on the 281 exams, partly because they're used to typing, having an IDE or editor make suggestions about fixing mistakes, and using the compiler and test runs to get it correct. Writing code by hand is a skill, and you should practice it before the exam. Any time you have to write a function, you can treat it as an opportunity to practice. Always start by understanding the problem, writing down examples, diagrams, data structures, etc. This is good advice for labs, projects, or exam questions Then write the code out by hand, on paper. Type in the code and get it tested, revising it until you're sure it's working. Then go back and compare your original handwritten version to the final working version, and see what you did wrong: syntax errors, missing a base case, logic errors on the more complicated cases, etc.

We're encouraging this behavior by having you write code by hand in the labs this semester. Lab 01 starts out easy, Lab 02 will be a little harder, and we'll increase the difficulty to give you more and better practice.

When you're preparing to take an exam, here's a few tips:

1) Treat the entire semester as preparation for the exam. When you have to do multiple choice questions for lab, do them first without your notes, but don't submit them. Then go over them with your notes, and anything that you didn't remember, write it down as possibly needed on your exam cheat-sheet (you get one in 281).
2) After each lecture, think of one or two questions that you think we might ask. Which parts of the material seem the most important? What sort of question might we ask?

Trade these questions with a few classmates, answer theirs and have them answer yours, and explain to each other why you thought that was important.

3) For your final studying push, start several days before the exam, rather than just cramming the night before.
4) Take the practice exam as if it were the real exam, see how you need to manage your time, what you need to study more, what you forgot on your cheat-sheet, etc.
5) Get enough sleep the night before.
6) Find your best mood right before the exam -- do you want to be excited or calm? Do you want to be outside in nature, or looking over your notes one last time?
7) Look over all the problems on the exam, quickly, before doing any of them.
    a) If there's something you just looked at 10 minutes ago, do it first.
    b) If there's a problem that's going to take 20% of the time for 5% of the points, do it last.

I consider 7b to be a terribly written problem and try to avoid it. Some students have reported that they like to do the written coding problems first, because they're better able to budget time (*x* multiple choice at *y* minutes each means I need *x*y* for the multiple choice, leaving *z* = 120 - *x*y* minutes for the written portion).