CS 162 (C++): Common Errors

Guide to Common Errors, Debugging, and Best Practices

To navigate this document, search for keywords or use the outline on the side (left).

Debugging / Programming Best Practices

- Compile often, even if you don't run your program! You do not want to be coding for hours and then face a wall of compilation errors once you try to compile and run your program.

- Fix the **first error** that pops up and then try to compile again (usually reduces errors)

- Look up documentation! You can find code examples and other bonus info/features:
  - https://en.cppreference.com/w/
  - https://devdocs.io/ (documentation for almost all languages!)
  - https://cplusplus.com/reference/ (C++ beginner friendly)

- When debugging, consider any assumptions about your code (e.g., file exists with expected date, array is of some size, object exists and has expected values). Are those assumptions true/verified? Can you verify some code or block is running as expected for a given input?
  - Using debuggers like gdb can make it very simple to check program flow, examine values in memory, and resolve any logical errors.

  - In a quick bind / small logical errors, decisive and carefully labeled print statements can be useful. **Tip**: Label with the variable being printed or specify where you are in the code (make it unique).

  - When testing your program for errors/logic, pick a **consistent** known set of inputs (e.g., when sorting an array don't create an array of random integers). This allows you to have a predictable output that is easy to verify.

- Spending some time planning your design before implementing it (e.g, pseudocode and convert to code) can save you time debugging a flawed plan.

  - "Measure twice, cut once"

- Help! This project is big / I don't know where to start.
  - Break the problem down into pieces. Can you find some part to start with?
  - Read the problem in 2 iterations:
    1. read for nouns/names these become members/variables of your class objects
    2. read for verbs/actions these become your methods/functions

- Keep your code **DRY** (Don't Repeat Yourself). Instead of writing similar to repetitive code, use functions, loops, and conditionals to automate this (avoid errors + maintainability).
- Avoid having multiple returns or exit points in a function, it makes it easier to trace through a function or program flow (*not a strict rule, see *early returns / guard clauses*).

Some of these bullets are expanded on with examples under Good Coding Practices / Styles
NOTE: All code examples can be tested on the server with c++11 standard or higher.

FAQ / Terminology

What's the difference between compilation errors and runtime errors?

Compilation errors are errors found at compile time when you try to compile a program (e.g., cannot convert type X to Y, variable is out of scope).

Runtime errors are errors thrown/caught during runtime when the program is running (e.g., segmentation faults).

What's the difference between an expression and a statement?

Like math, expressions express some value or operation (e.g., `x + 3`) while a statement is an executable instruction like a line of code (e.g., `return "Hello" + username + "!\n";`). This is more of a general definition, often an expression is also a statement or statements are composed of expressions.

What's the difference between a function declaration and a function definition?

A function declaration (or function prototype) just declares a function exists with a name:
```
void func();
```

A function definition defines the code {...} for the function:
```
void func() {
    //code
}
```

What's the difference between a function parameter and an argument?

Parameters are what a function accepts (has a type+name) while arguments are the values or objects being passed to the function when it is invoked with ().

What is scope?

Scope refers to how long a variable is known (visibility/lifetime), generally within {...}. This can also apply to functions, structs, and class objects. By extension, scope also allows for the same variable names to exist in different scopes and they will not have the same value.

- Scope is important to know as it can help explain errors where objects/variables get deleted when they go out of scope. In other words, you cannot access them anymore.

Scope Example

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

// Global scope variable
int num_birds = 5;

int main() {
    int four = 4;    // scope of “four” is within main()

    cout << nothing << endl;  // generates compiler error, not declared

    cout << "birds: " << num_birds << endl;

    for (int i = 0; i < 5; i ++) {
        // ^ i is a local variable to the for loop
        cout << "four: " << four << endl;
        cout << "i: " << i << endl;
    }

    // Can't access i out of for loop,
    // will generate a compiling error
    cout << i << endl;
}
```

Compilation Errors

There are two types of compilation errors:

1. Syntax Error
    - Violation of a programming language rules/structure (syntax)
2. Linker Error
    - Program is compiling but some files are not connected properly (e.g., circular include or dependency, missing include, redefinition errors). No executable is made.

Source: Errors in CPP

How to Read the Error Message | Structure of Error Message

C++ compiler errors can be hard to read at first, you will get better at this with practice! First take some time to read the error and then try googling it (or bing or another search engine)

Error messages can contain a lot of information that you may not need to read or have to search to make sense of it. Below is the general structure of a compilation error. The text in **bold** are the key parts to look at or use when debugging.

<filename>:<line_number>: location of error in a file

<description>: description of error, use this when searching for an explanation/solution!

<context>: (optional) describes where the error occurred / propagated from source files / function calls

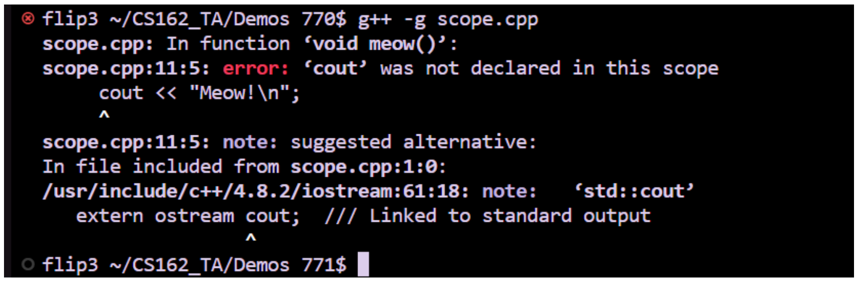
```
General format of an error message        (*can look different)
```

```
<filename>:<line_number>:<col_number>: error: <description>
    <snippet of code associated with error>
    <context>
```

You may see a fix suggestion like this:

```
<filename>:<line_number>:<col_number>: note: suggested alternative
In file included from <filename>:<line_number>:<col_number>:
<path to library headers>:<num>:<num>: note:   <suggestion>
    <more text>
```

```
Example of a compilation error
```



```
flip3 ~/CS162_TA/Demos 770$ g++ -g scope.cpp
scope.cpp: In function 'void meow()':
scope.cpp:11:5: error: 'cout' was not declared in this scope
     cout << "Meow!\n";
     ^
scope.cpp:11:5: note: suggested alternative:
In file included from scope.cpp:1:0:
/usr/include/c++/4.8.2/iostream:61:18: note:   'std::cout'
   extern ostream cout;  /// Linked to standard output
                  ^
flip3 ~/CS162_TA/Demos 771$
```

Here the error occurred on line 11 in the scope.cpp file. The suggested (and correct) fix is either adding `std::` in front of `cout` or import the std namespace with `using namespace std;`

Missing semicolon ; or {} | expected `;` before

This error can be notoriously hard to find due to it interacting with other lines of code because semicolons define where a statement ends. This means that the error generated may refer to a different line or complain about something seemingly unrelated. Below are some common examples of missing semicolons or curly braces {}.

One way to guard against this error is by having good/consistent formatting, intellisense/syntax warnings, and using syntax color themes in your text editor or IDE that colors paired brackets.

#	Missing ‘;’ from a variable	Error Output

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      string cat = "Meow!"
7
8      return 0;
9  }
```

```
$ g++ missing.cpp -o ex1
missing.cpp: In function ‘int main()’:
missing.cpp:8:5: error: expected ‘,’ or ‘;’ before ‘return’
     return 0;
     ^
```

#	Missing ‘;’ from struct/class	Error Output

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  struct Cat {
6      string name;
7      string breed;
8      float weight;
9  }
10
11 int main() {
12     Cat garfield;
13
14     return 0;
15 }
```

```
$ g++ missing.cpp -o ex1
missing.cpp:9:1: error: expected ‘;’ after struct definition
 }
 ^
```

#	Missing last ‘;’ from header file	Error Output

```
   // in “header.h” file...
1  #ifndef HEADER_EX
2  #define HEADER_EX
3
4
5  struct Generic {
6      int gen_num;
7  };
8
9  void do_stuff();
10
11 void do_stuff2()
12
13 #endif
```

```
   // in “header.cpp” file...
1  #include <iostream>
2  #include "header.h"
3
4  using namespace std;
```

```
$ g++ header.cpp -o ex1
header.cpp:4:1: error: expected initializer before ‘using’
 using namespace std;
 ^
header.cpp: In function ‘void do_stuff()’:
header.cpp:7:5: error: ‘cout’ was not declared in this scope
     cout << "Stuff\n";
     ^
header.cpp:7:5: note: suggested alternative:
In file included from header.cpp:1:0:
/usr/include/c++/4.8.2/iostream:61:18: note:    ‘std::cout’
   extern ostream cout;  /// Linked to standard output
                  ^
header.cpp: In function ‘void do_stuff2()’:
header.cpp:11:5: error: ‘cout’ was not declared in this
scope
     cout << "Stuff2\n";
     ^
header.cpp:11:5: note: suggested alternative:
In file included from header.cpp:1:0:
 5
 6  void do_stuff() {
 7      cout << "Stuff\n";
 8  }
 9
10  void do_stuff2() {
11      cout << "Stuff2\n";
12  }
13
14  int main() {
15      do_stuff();
16      return 0;
17  }
```

```
/usr/include/c++/4.8.2/iostream:61:18: note:    'std::cout'
   extern ostream cout;  /// Linked to standard output
                  ^
```

Here the error “expected initializer” was caused because of the missing semicolon on line 11 in the header.h file. Adding the ; fixes the rest of the errors.

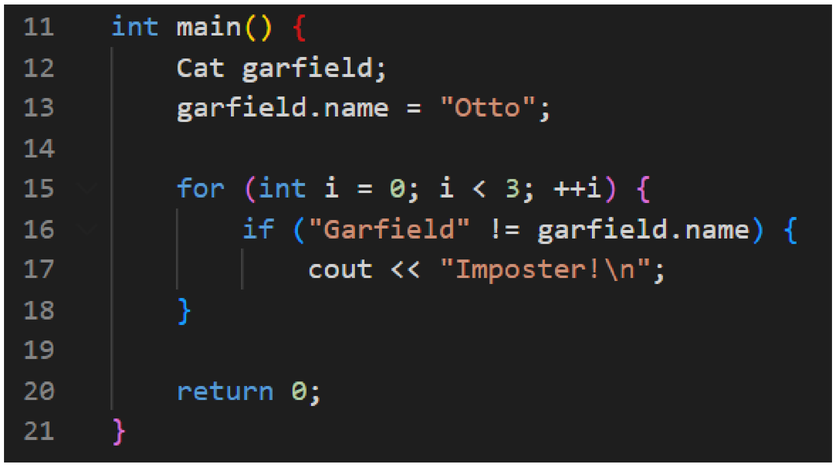
```
 #  Missing '}'
```

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  struct Cat {
 6      string name;
 7      string breed;
 8      float weight;
 9  };
10
11  int main() {
12      Cat garfield;
13      garfield.name = "Otto";
14
15      for (int i = 0; i < 3; ++i) {
16          if ("Garfield" != garfield.name) {
17              cout << "Imposter!\n";
18      }
19
20      return 0;
21  }
```

Error Output

```
$ g++ missing.cpp -o ex1
missing.cpp: In function 'int main()':
missing.cpp:21:1: error: expected '}' at end of input
 }
 ^
```

Example of Syntax highlighting and paired {} to detect issue



```cpp
int main() {
    Cat garfield;
    garfield.name = "Otto";

    for (int i = 0; i < 3; ++i) {
        if ("Garfield" != garfield.name) {
            cout << "Imposter!\n";
    }

    return 0;
}
```

cout/cin, iostream | No match for operator>> | No match for operator<<

If you start seeing a wall of errors for iostream or istream/ostream with includes and/or "`no match for 'operator>>'`", this is likely because the arrow directions for cout or cin are incorrect.

#	Sample Code to Recreate	Error Output Snippet

1
2
3
4
5
6
7
	`#include <iostream>`
	`using namespace std;`
	
	`int main() {`
	`    cout >> "Hello world!";`
	`    return 0;`
	`}`
		`sample.cpp: In function 'int main(int, char**)':`
		`sample.cpp:25:10: error: no match for 'operator>>' (operand types are 'std::ostream {aka std::basic_ostream<char>}' and 'const char [13]')`
		`     cout >> "Hello world!";`
		`          ^`
		`cout_err.cpp:25:10: note: candidates are:`
		`In file included from /usr/include/c++/4.8.2/string:53:0,`
		`                 from /usr/include/c++/4.8.2/bits/locale_classes.h:40,`
		`                 from`
		`. . .`

To fix, check if the direction of the `>>` or `<<` are in the correct direction for `cin` and `cout`. For `cout` (output) the arrow should point toward `cout` (left). For `cin` (input) the arrow should point to the variables where the input is saved (right).

#	Fixed Sample Code

1
2
3
4
5
6
	`#include <iostream>`
	`using namespace std;`
	
	`int main() {`
	`    cout << "Hello world!";`
	`    //   ^  arrow points to console (output is sent there)`

```
7      return 0;
8  }
```

Not Declared in Scope / Not Defined

Scope refers to how long a variable is known (visibility/lifetime), generally within {...} **(see What is scope?)**. This error just means that the compiler could not figure out what “X” means. A general strategy for fixing this error is checking if the variable was defined.

#	Sample Code to Recreate	Error Output

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      cout << cat << endl;
7      return 0;
8  }
```

```
scope.cpp: In function ‘int main()’:
scope.cpp:6:13: error: ‘cat’ was not declared in
this scope
     cout << cat << endl;
             ^
```

#	Fixed Sample Code

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      string cat = "Meow!";
7      cout << cat << endl;
8      return 0;
9  }
```

Undefined Reference

Undefined references means a function or object has not been defined yet (no code). This can also be caused by not including your header file, not including the declaration in the header file, or not writing an implementation for a function yet in your implementation file.

Helpful link: C++ Errors: Undefined Reference, Unresolved External Symbol etc. (softwaretestinghelp.com)

Note: A function declaration (prototype) just declares a function exists while a function definition defines the code {...} for the function.

#	Sample Code to Recreate	Error Output Snippet

```cpp
#include <iostream>

void meow();

int main() {
    meow();
    return 0;
}
```

```
/tmp/ccram8A3.o: In function `main':
<filepath>/scope.cpp:6: undefined reference to
`meow()'
collect2: error: ld returned 1 exit status
```

Here we can fix the above code in two ways depending on when we want to define the function or object. Note that `main()` expects everything to be known/defined before it.

Fixed Sample Code

```cpp
#include <iostream>

// Forward declare function, define later
void meow();

int main() {
    meow();
    return 0;
}

void meow() {
    std::cout << "Meow!\n";
}
```

Alternative Solution

```cpp
#include <iostream>

// Define before main()
// Or include a header file!
void meow() {
    std::cout << "Meow!\n";
}

int main() {
    meow();
    return 0;
}
```

** Insert an example for Vtable **

No match for operator (general)

This error occurs when you have some expression that is incompatible for the operator used since there is no defined behavior for that operation and operands (arguments). For example, what should the compiler do if a program tries to add a number and a string?
Note that operators can be overloaded (can define/overwrite what it does originally).

Sample Code to Recreate

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

int main() {
    string cat = "Meow x ";

    if (cat + 3 == "Meow x 3") {
        cout << "Successful concatenation: " << cat + "3" << endl;
10      }
11      else {
12          cout << "Unexpected result\n";
13      }
14
15      return 0;
16  }
```

Error Output Snippet

```
scope.cpp: In function ‘int main()’:
scope.cpp:8:13: error: no match for ‘operator+’ (operand types are ‘std::string {aka
std::basic_string<char>}’ and ‘int’)
     if (cat + 3 == "Meow x 3") {
             ^
scope.cpp:8:13: note: candidates are:
In file included from /usr/include/c++/4.8.2/bits/stl_algobase.h:67:0,
                 from /usr/include/c++/4.8.2/bits/char_traits.h:39,
                 from /usr/include/c++/4.8.2/ios:40,
                 from /usr/include/c++/4.8.2/ostream:38,
                 from /usr/include/c++/4.8.2/iostream:39,
                 from scope.cpp:1:
/usr/include/c++/4.8.2/bits/stl_iterator.h:333:5: note: template<class _Iterator>
std::reverse_iterator<_Iterator> std::operator+(typename
std::reverse_iterator<_Iterator>::difference_type, const
std::reverse_iterator<_Iterator>&)
     operator+(typename reverse_iterator<_Iterator>::difference_type __n,
...
```

Here the error is reported for line 8 because there is no defined operation for a string + an integer. To fix, we can make the 3 an implicit string value or literal.

```
 #  Fixed Sample Code

 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  int main() {
 6      string cat = "Meow x ";
 7
 8      if (cat + "3" == "Meow x 3") {
 9          cout << "Successful concatenation: " << cat + "3" << endl;
10      }
11      else {
12          cout << "Unexpected result\n";
13      }
14
15      return 0;
16  }
```

Cannot convert from X to Y | No suitable constructor exists to convert | Invalid initialization of …

These error(s) occur because there is some operation (e.g., passing arguments to a function, value assignment, type conversion) where it expects a specific type and the current one is invalid / could not be converted into a valid one.

```
 #  Sample Code to Recreate

 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  class Cat {
 6  public:
 7      string name;
 8      string breed;
 9      float weight;
10  };
11
12  void print_cat(Cat *cat_ptr) {
13      //! Note: cat_ptr->name is the same as (*cat_ptr).name
14      cout << "Name: " << cat_ptr->name << endl;
15      cout << "Breed: " << cat_ptr->breed << endl;
16      cout << "Weight: " << cat_ptr->weight << endl;
17  }
18
19  int main() {
20      Cat garfield {
21            name: "Garfield"
22          , breed: "Orange Tabby"
23          , weight: 5};
24
25      print_cat(garfield);
26
27      return 0;
28  }
```

Error Output

```
$ g++ -std=c++11 conversion_err.cpp -o ex1
conversion_err.cpp: In function ‘int main()’:
conversion_err.cpp:25:23: error: cannot convert ‘Cat’ to ‘Cat*’ for argument
‘1’ to ‘void print_cat(Cat*)’
     print_cat(garfield);

                        ^
```

The error occurred on line 25 where it could not convert Cat to a Cat* which is a pointer, so we use the & operator to get the address of the object `garfield`. This turns the expression into the type Cat*

Fixed Sample Code

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

class Cat {
public:
    string name;
    string breed;
    float weight;
};

void print_cat(Cat *cat_ptr) {
    //! Note: cat_ptr->name is the same as (*cat_ptr).name
    cout << "Name: " << cat_ptr->name << endl;
    cout << "Breed: " << cat_ptr->breed << endl;
    cout << "Weight: " << cat_ptr->weight << endl;
}

int main() {
    Cat garfield {
          name: "Garfield"
        , breed: "Orange Tabby"
        , weight: 5};

    print_cat(garfield);

    return 0;
}
```

Output

```
$ ex1
Name: Garfield
Breed: Orange Tabby
Weight: 5
```

Runtime Errors

Segmentation Faults / segfault / core dump

Segmentation faults, to simplify, are memory errors where the program tries to access or modify memory that it is not allowed to.

Common ways to cause them, not guaranteed (*undefined behavior):

- Accessing or modifying an array at a bad index, out of bounds (e.g., -1 index or using size as index)
- Dereferencing a null pointer

- Returning/accessing an address to a local variable, goes out of scope (return address of local var)
- Trying to store values to uninitialized dynamic arrays or store something too large
- Accessing a value/object/array that got deleted or went out of scope (destructor)

**undefined behavior (UB)*: behavior that is unpredictable / not guaranteed for each program run

In this class, you can use valgrind or gdb to find and fix segfaults.

#	Sample Code to Recreate	Error Output Snippet
1	`#include <iostream>`	`$ g++ -g segfault.cpp -o ex1`
2	`using namespace std;`	`$ ./ex1`
3		`Segmentation fault (core dumped)`
4	`int main() {`	`$`
5	`    int value = 6;`	
6	`    int* ptr = NULL;`	
7		
8	`    cout << "*ptr: " << *ptr << endl;`	
9		
10	`    return 0;`	
11	`}`	

Here we have some pointer that is NULL and it gets dereferenced, while this error may be more apparent not all segfaults are simple to find. The first step is locating where your program segfaults using tools like gdb or valgrind. The next step is fixing/determining ***why*** which can be found with debuggers like gdb or using print statements to investigate.

Using Valgrind to Locate Segfault (and Solve)

This section assumes some familiarity with valgrind (see How To Use Valgrind).

Look for SIGSEGV which is the signal that abruptly ended the program, pay close attention to where the error occurred and the description.
Blue indicates a memory error report by valgrind that was caught during runtime.
Yellow is the signal that interrupted the process/program running.

In the terminal run valgrind with the executable...	Walkthrough / Explanation

`$ valgrind ex1`
`==31197== Memcheck, a memory error detector`
`==31197== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.`
`==31197== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info`
`==31197== Command: ex1`
`==31197==`
	Compile and run your program with valgrind.

`==31197== Invalid read of size 4`
`==31197==    at 0x400839: main` **`(segfault.cpp:8)`**
`==31197==  Address 0x0 is not stack'd, malloc'd or (recently) free'd`
`==31197==`
	Here is a memory error valgrind encountered while the program was running. The invalid read on line 8 of the file.

```
==31197==
==31197== Process terminating with default action of signal 11 (SIGSEGV)
==31197==  Access not within mapped region at address 0x0
==31197==    at 0x400839: main (segfault.cpp:8)
==31197==  If you believe this happened as a result of a stack
==31197==  overflow in your program's main thread (unlikely but
==31197==  possible), you can try to increase the size of the
==31197==  main thread stack using the --main-stacksize= flag.
==31197==  The main thread stack size used in this run was 8388608.
==31197==
==31197== HEAP SUMMARY:
==31197==     in use at exit: 0 bytes in 0 blocks
==31197==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==31197==
==31197== All heap blocks were freed -- no leaks are possible
==31197==
==31197== For lists of detected and suppressed errors, rerun with: -s
==31197== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

Here is where the program ended because of the segfault signal. The segfault occurred on line 8 of the file.

Valgrind reported 1 error, let's reduce this to 0!

Valgrind reports that the error occurred on line 8 in the file segfault.cpp where the program tried to access the address 0x0 which is the NULL address and is inaccessible. So we can fix this by providing the pointer an address by either referencing the value variable or allocating space for an integer(s) on line 6-7.

#	Fixed Sample Code	Alternative Solution
1	`#include <iostream>`	`#include <iostream>`
2	`using namespace std;`	`using namespace std;`
3		
4	`int main() {`	`int main() {`
5	`    int value = 6;`	`    int value = 6;`
6	`    int* ptr = &value;`	`    int* ptr = new int;`
7		`    *ptr = 8;`
8	`    cout << "*ptr: " << *ptr << endl;`	
9		`    cout << "*ptr: " << *ptr`
10	`    return 0;`	`         << endl;`
11	`}`	
12		`    delete ptr;`
13		`    ptr = nullptr;`
14		`    return 0;`
15		`}`

Valgrind Guide

Valgrind is a great tool for pinpointing memory leaks and investigating memory errors (can cause undefined behavior)! Note that valgrind is already installed on the ENGR servers.

Here are some quick guides to understanding valgrind output messages, heap summary, resolving memory leaks that I found helpful:

- https://valgrind.org/docs/manual/quick-start.html

- https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_valgrind.html
- https://codingnest.com/how-to-read-valgrind-output/
- https://developers.redhat.com/blog/2021/04/23/valgrind-memcheck-different-ways-to-lose-your-memory#generating_a_leak_summary

Valgrind full manual:

- https://valgrind.org/docs/manual/manual.html

How to Use (On the ENGR servers)

To get more helpful error messages and see line numbers make sure to compile with the debugging flag -g. If you have a Makefile add it to your CC variable (e.g., CC = g++ -g)

Compile with -g (example)

`g++ -std=c++11 -g <filename> -o <exec_name>`
`// or in a Makefile…`
`CC = g++ -std=c++11 -g`

"-o <exec_name>" specifies output to run, if no "-o ..." default executable is "a.out"

To run valgrind enter (you can also add optional arguments like leak-check):

Run valgrind with executable
`valgrind <valgrind_args> <exec_name>`
Run valgrind with **additional options**
`valgrind --leak-check=full --show-leak-kinds=full a.out`

- --leak-check=full (this shows where memory was allocated but not freed)
- --show-leak-kinds=full (other optional argument ("reports more")

Note: Abruptly ending the program (e.g., ctrl+C), segfaults, or using things like exit() will likely make valgrind report memory leaks when there are none because the deletes/destructors were not called.

⚠️Common issues

- Running valgrind with .cpp files not the executable file
- If your **program abruptly stops** (e.g., ctrl+C, segfault, using `exit()`) your program likely has not called any deletes so you will likely see memory leaks

- Do not use `exit()` to end a program unless you are sure all dynamic memory has been cleared

Case study: Memory Leaks and valgrind

Memory leaks occur when some dynamic memory (user allocated) is allocated but not freed. It is called a leak because generally the address to this block of memory is lost and thus the data stored there is lost. This essentially takes up space and can cause memory fragmentation.

A good rule of thumb is: "For every `new` used, there should be a corresponding `delete`"

- new keyword allocates space on the heap and returns an address to that memory
    - it expects a data type and size/number of that data type to allocate space for
- delete keyword frees/gives up ownership of the memory given an address on the heap
    - it expects an address to an area on the heap
    - when deleting an array, make sure to add [ ] after delete to free the whole block

```
 #  Sample Code to Recreate

 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  #define BASKET_SIZE 4
 6
 7  int main() {
 8      // Using an initialization list to fill dynamic array of strings
 9      string *fruit_basket = new string[BASKET_SIZE]
10          {"orange", "kiwi", "apple", "banana"};
11
12      for (int i = 0; i < BASKET_SIZE; ++i) {
13          cout << "[" << i << "]: " << fruit_basket[i] << endl;
14      }
15
16      cout << "Dump and use new basket...\n" << endl;
17      fruit_basket = new string[BASKET_SIZE+1]
18          {"pineapple", "kiwi", "grape", "banana", "mango"};
19
20      for (int i = 0; i < BASKET_SIZE+1; ++i) {
21          cout << "[" << i << "]: " << fruit_basket[i] << endl;
22      }
23
24      delete [] fruit_basket;
25      fruit_basket = NULL;   // good practice to avoid dangling pointers
26
27      return 0;
28  }
```

Valgrind Output **(Compile with -g to see line numbers!)**

```
$ g++ -std=c++11 -g memleak.cpp -o ex1
$ valgrind ex1
==8013== Memcheck, a memory error detector
==8013== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8013== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8013== Command: ex1
==8013==
[0]: orange
[1]: kiwi
[2]: apple
[3]: banana

Dump and use new basket...

[0]: pineapple
[1]: kiwi
[2]: grape
[3]: banana
[4]: mango
==8013==
==8013== HEAP SUMMARY:
==8013==     in use at exit: 161 bytes in 5 blocks
==8013==   total heap usage: 11 allocs, 6 frees, 363 bytes allocated
==8013==
==8013== LEAK SUMMARY:
==8013==    definitely lost: 40 bytes in 1 blocks
==8013==    indirectly lost: 121 bytes in 4 blocks
==8013==      possibly lost: 0 bytes in 0 blocks
==8013==    still reachable: 0 bytes in 0 blocks
==8013==         suppressed: 0 bytes in 0 blocks
==8013== Rerun with --leak-check=full to see details of leaked memory
==8013==
==8013== For lists of detected and suppressed errors, rerun with: -s
==8013== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Here (yellow) we have some reported leak, we can tell because the HEAP SUMMARY does not say "`All heap blocks were freed -- no leaks are possible`". In order to know what memory hasn't been freed we need to run valgrind with some additional arguments like the one it suggests.

Note: The number of allocs to frees is a good metric to check but do not rely on it to determine if you have no leaks, it can be misleading (e.g., over-freeing or double-frees).

Valgrind Snippet **(Compile with -g to see line numbers!)**

```
$ g++ -std=c++11 -g memleak.cpp -o ex1
$ valgrind --leak-check=full ex1
... <snippet begins>
==17587==
==17587== HEAP SUMMARY:
==17587==     in use at exit: 161 bytes in 5 blocks
==17587==   total heap usage: 11 allocs, 6 frees, 363 bytes allocated
==17587==
==17587== 161 (40 direct, 121 indirect) bytes in 1 blocks are definitely lost in loss record 5 of 5
==17587==    at 0x4C2AC38: operator new[](unsigned long) (vg_replace_malloc.c:433)
==17587==    by 0x400B85: main (memleak.cpp:10)
==17587==
==17587== LEAK SUMMARY:
==17587==    definitely lost: 40 bytes in 1 blocks
==17587==    indirectly lost: 121 bytes in 4 blocks
==17587==      possibly lost: 0 bytes in 0 blocks
==17587==    still reachable: 0 bytes in 0 blocks
==17587==         suppressed: 0 bytes in 0 blocks
==17587==
==17587== For lists of detected and suppressed errors, rerun with: -s
==17587== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

`valgrind --leak-check=full` reports what memory was allocated but never freed, so in the highlighted portion above it says on line 10 is where the issue occurred. The issue was that we did not delete before re-allocating space to the same pointer and thus lost the address to it.

`#` Fixed

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4
 5  #define BASKET_SIZE 4
 6
 7  int main() {
 8      // Using an initialization list to fill dynamic array of strings
 9      string *fruit_basket = new string[BASKET_SIZE]
10          {"orange", "kiwi", "apple", "banana"};
11
12      for (int i = 0; i < BASKET_SIZE; ++i) {
13          cout << "[" << i << "]: " << fruit_basket[i] << endl;
14      }
15
16      delete [] fruit_basket;
17      fruit_basket = NULL;
18
19      cout << "Dump and use new basket...\n" << endl;
20      fruit_basket = new string[BASKET_SIZE+1]
21          {"pineapple", "kiwi", "grape", "banana", "mango"};
22
23      for (int i = 0; i < BASKET_SIZE+1; ++i) {
24          cout << "[" << i << "]: " << fruit_basket[i] << endl;
25      }
26
27      delete [] fruit_basket;
28      fruit_basket = NULL;
29
30      return 0;
31  }
```

Valgrind Snippet **(Compile with -g to see line numbers!)**

```
$ g++ -std=c++11 -g memleak.cpp -o ex1
$ valgrind --leak-check=full ex1
... <snippet begins>
==359==
==359== HEAP SUMMARY:
==359==     in use at exit: 0 bytes in 0 blocks
==359==   total heap usage: 11 allocs, 11 frees, 363 bytes allocated
==359==
==359== All heap blocks were freed -- no leaks are possible
==359==
==359== For lists of detected and suppressed errors, rerun with: -s
==359== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Case Study: Using valgrind to solve a reported error by valgrind

Valgrind will report errors as the program executes and provide a heap summary at the end to check for memory leaks. The errors reported are often memory errors which are good to resolve/eliminate as memory errors can cause undefined behavior or may cause your program to crash/segfault.

uninitialized.cpp

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

int main() {
    int birds_in_park[2];

    // Which park should we visit for bird watching?
    if (birds_in_park[0] > birds_in_park[1]) {
        printf("Watch birds in park 0!\n");
    }
    else {
        printf("Watch birds in park 1!\n");
    }

    for (int i = 0; i < 2; ++i) {
        printf("Birds at park %d: %d\n"
               , i, birds_in_park[i]);
    }

    return 0;
}
```

Output

```
$ g++ -g uninitialized.cpp -o ex1
$ ex1
Watch birds in park 1!
Birds at park 0: -1651794096
Birds at park 1: 32764
$ ex1
Watch birds in park 0!
Birds at park 0: 116510904
Birds at park 1: 32764
```

We got 2 different outputs without changing our code! It also looks like we have garbage values in our array. This signals **undefined behavior** which we want to avoid.

Let's use valgrind! Make sure the program was compiled with -g.

Valgrind

```
$ g++ -g uninitialized.cpp -o ex1
$ valgrind ex1
==3750== Memcheck, a memory error detector
==3750== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3750== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
```

Walkthrough / Explanation

Compile and run your program with valgrind.

```
info
==3750== Command: ex1
==3750==
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x4005AD: main (uninitialized.cpp:8)
==3750==
Watch birds in park 1!
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x56A1C5E: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
==3750== Use of uninitialised value of size 8
==3750==    at 0x569F32B: _itoa_word (_itoa.c:179)
==3750==    by 0x56A35B0: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x569F335: _itoa_word (_itoa.c:179)
==3750==    by 0x56A35B0: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x56A35FF: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x56A1D2B: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
==3750== Conditional jump or move depends on uninitialised value(s)
==3750==    at 0x56A1DAE: vfprintf (vfprintf.c:1634)
==3750==    by 0x56AA4E8: printf (printf.c:34)
==3750==    by 0x4005EA: main (uninitialized.cpp:17)
==3750==
Birds at park 0: -16780144
Birds at park 1: 31
==3750==
==3750== HEAP SUMMARY:
==3750==     in use at exit: 0 bytes in 0 blocks
==3750==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3750==
==3750== All heap blocks were freed -- no leaks are possible
==3750==
==3750== Use --track-origins=yes to see where uninitialised values come
from
==3750== For lists of detected and suppressed errors, rerun with: -s
==3750== ERROR SUMMARY: 29 errors from 7 contexts (suppressed: 0 from 0)
```

As always, start with the first error that pops up! Which is right here (highlighted yellow).
The error says we have an uninitialized value used in a conditional on line 8 in our file. On this line we're accessing our uninitialized array of integers.

Let's fix this by using list initialization!

There are other errors here but we'll ignore them for now as our fix may eliminate or reduce them.

Note: These errors appear longer as valgrind shows a **stack trace** of the function calls that led to the error occurring in a source file up to when it was called by our own program.
So in this block the error first occurred at _itoa.c:179 which then propagated up to uninitialized.cpp:17 in the main() function.

In general when reading the errors pay attention to the first line (top) specifying the error and its location in your **own** code.

We have no memory leak! Good news.

Valgrind reported 29 errors, let's reduce this to 0!

#	uninitialized.cpp (fixed) ✔

```cpp
#include <cstdio>
using namespace std;

int main() {
    int birds_in_park[2] = {20, 30};

    // Which park should we visit for bird watching?
    if (birds_in_park[0] > birds_in_park[1]) {
        printf("Watch birds in park 0!\n");
    }
    else {
        printf("Watch birds in park 1!\n");
    }

    for (int i = 0; i < 2; ++i) {
        printf("Birds at park %d: %d\n"
               , i, birds_in_park[i]);
    }

    return 0;
}
```

Valgrind Output ✔

```
$ g++ -g uninitialized.cpp -o ex1
$ valgrind ex1
==13787== Memcheck, a memory error detector
==13787== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==13787== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==13787== Command: ex1
==13787==
Watch birds in park 1!
Birds at park 0: 20
Birds at park 1: 30
==13787==
==13787== HEAP SUMMARY:
==13787==     in use at exit: 0 bytes in 0 blocks
==13787==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==13787==
==13787== All heap blocks were freed -- no leaks are possible
==13787==
==13787== For lists of detected and suppressed errors, rerun with: -s
==13787== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Good Coding Practices / Styles

While there are some coding styles like consistent naming schemes (e.g., `camelCase`, `PascalCase, or snake_case`) and consistent indentation there are other things you can do! These are just some suggestions to try out in your workflow.

Other resources to explore:

- C++ Core Guidelines (isocpp.github.io)

- Kristories/awesome-guidelines: A curated list of high quality coding style conventions and standards. (github.com)
- CodeAesthetic - YouTube
- Search for advice/recommendations in your specific language or domain

Initialize Variables

Be explicit in your variable values to avoid undefined behavior and have clearer assumptions about your program.

Do ✔️

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

int main() {
    int num_birds = 4;

    if (num_birds < 0) {
        cout << “No bird watching.\n“;
    }
    else {
        cout << “Birds exist\n”;
    }

    return 0;
}
```

Don’t ❌

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

int main() {
    int num_birds;

    if (num_birds < 0) {
        cout << “No bird watching.\n“;
    }
    else {
        cout << “Birds exist\n”;
    }

    return 0;
}
```

Use Labeled, Clear Debugging Print Statements

Sometimes a well-placed print statement is all we need to debug. We may write a random string of letters to check if some conditional block is executing at all. Or print out a variable’s value with no indication on where/what it was. This can make debugging challenging when you have several of these printing statements. When using print statements try adding labels!

Do ✔️

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

int main() {
    int counter = 0;

    // How many times does this loop run?
    for (int i = 0; i < 5; ++i) {
        cout << "i: " << i << endl;
        counter++;
    }

```

Output

```
$ g++ debug.cpp -o ex1
$ ex1
i: 0
i: 1
i: 2
i: 3
i: 4
counter: 5
```

```cpp
    cout << "counter: " << counter << endl;

    return 0;
}
```

Avoid ❌

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

int main() {
    int counter = 0;

    // How many times does this loop run?
    for (int i = 0; i < 5; ++i) {
        cout << i << endl;
        counter++;
    }

    cout << counter << endl;

    return 0;
}
```

Output

```
$ g++ debug.cpp -o ex1
$ ex1
0
1
2
3
4
5
```

Here it can be challenging to know where that “5” printed from.

Replace (some) Comments with Descriptive Variables or Functions

Comments are great, but they can turn into zombies where they do not get updated when your code changes or simply take more space than they should. Replacing some code logic or values with a meaningful variable name or function call makes it reusable and increases readability + maintainability. This also hides details (abstraction!).

Do ✔️

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

// Global constant
#define BIRD_TIME 8

// Function prototypes / function declarations
bool is_bird_watching_time(int, bool);
int get_time();

int main() {
    int num_birds = 4;
    bool garfield_is_indoors = true;

    if (is_bird_watching_time(num_birds, garfield_is_indoors)) {
        cout << "Good time for bird watching!\n";
    }
    else {
        cout << "Not a good time for bird watching.\n";
    }
    return 0;
}

bool is_bird_watching_time(int num_birds, bool cat_is_indoors) {
    return
        num_birds > 0
        && get_time() == BIRD_TIME
        && cat_is_indoors;
}

int get_time() {
    const time_t ct = time(nullptr);  // nullptr is in c++11
    return localtime(&ct)->tm_hour;
}
```

# Avoid ❌

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

#define BIRD_TIME 8
int current_time();


int main() {
    int num_birds = 4;
    bool check_cat_inside = false;

    if (current_time() == BIRD_TIME && num_birds > 0 && check_cat_inside) {
        // bird  watching should happen iff
        // time is 6 am, there is at least 1 bird,
        // and Garfield is inside
        cout << “Good time for bird watching!\n“;
    }
    else {
        cout << “Not a good time for bird watching.\n”;
    }
    return 0;
}

// Get current time (local)
int current_time() {
    const time_t ct = time(nullptr);  // nullptr is in c++11
    return localtime(&ct)->tm_hour;
}
```

DRY Code (Don’t Repeat Yourself)

Making your code modular and avoiding duplicate/repetitive code is a good way to avoid errors, increase readability, and makes it easier to maintain or make changes to your code. The DRY principle becomes especially apparent when the same code or function is made with slight

changes. This can become troublesome when you want to edit it (forced to make edits in several places) or misscount lines. Note that this is a heuristic and not a universal rule.
A couple ways of making your code drier is by using functions, loops, conditionals, and object relationships.

Do ✔️	Output
See code below	See output below

Do ✔️

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

const int ARR_SIZE = 5;

int main() {
    int scores[ARR_SIZE] = {9, 6, 5, 7, 8}; // 35

    int sum = 0;
    for (int i = 0; i < ARR_SIZE; ++i) {
        sum += scores[i];
    }

    cout << "sum: " << sum << endl;

    return 0;
}
```

Output

```
$ g++ dry.cpp -o ex1
$ ex1
sum: 35
```

Avoid ❌

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


int main() {
    int scores[5] = {9, 6, 5, 7, 8};   // 35

    int sum = scores[0] + scores [1] + scores[2]
        + scores[3] + scores[4] + scores[5];

    cout << "sum: " << sum << endl;

    return 0;
}
```

Output

```
$ g++ dry.cpp -o ex1
$ ex1
sum: 32802
```

We got an unexpected result! That's because we accessed an invalid index (5) which can contain garbage values or cause the program to crash (undefined behavior). This can be a simple error to make when manually writing or creating repetitive code.

Let's look at another example with functions!

Do ✔️

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

const int ARR_SIZE = 2;

void player_score(int player_num, int score) {
```

Output

```
$ g++ dry.cpp -o ex1
$ ex1
🎳 Player 1 score: 39
🎳 Player 2 score: 54
```

```
 7      cout << "\U0001F3B3 Player" << player_num
 8           << " score: " << score << endl;
 9  }
10
11  int main() {
12      int scores[ARR_SIZE] = {39, 54};
13
14      for (int i = 0; i < ARR_SIZE; ++i) {
15          int player_number = i + 1;
16          player_score(player_number, scores[i]);
17      }
18
19      return 0;
20  }
```

#	Avoid ❌	Output

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  const int ARR_SIZE = 2;
 5
 6  void player_1(int score) {
 7      cout << "\U0001F3B3 Player 1 score: "
 8           << score << endl;
 9  }
10
11  void player_2(int score) {
12      cout << "\U0001F3B3 Player 2 score: "
13           << score << endl;
14  }
15
16  int main() {
17      int scores[ARR_SIZE] = {39, 54};
18
19      player_1(scores[0]);
20      player_2(scores[1]);
21
22      return 0;
23  }
```

```
$ g++ dry.cpp -o ex1
$ ex1
🎳 Player 1 score: 39
🎳 Player 2 score: 54
```

Notice the lines in yellow (7-8 and 12-13) are pretty much doing the same operation with 1 number different, we can make this more compact especially if we have more than 2 players!

Use Compiler Options / Flags

The compiler we're using for this class (GCC GNU) provides many options to direct how to compile and format/show/repress warnings which can be very useful.

- Warning Options (Using the GNU Compiler Collection (GCC))
- Debugging Options (Using the GNU Compiler Collection (GCC))
- Invoking GCC (Using the GNU Compiler Collection (GCC))

Below are some common ones you may use and all options can be combined together (descriptions are adapted from sources above). Add `-g` when you compile, especially if you want to use gdb or valgrind.

Option	Description
-g	Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information. On most systems that use stabs format, -g enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use -gvms (see below).
-Wall	Enable all warnings for code that may be questionable and are easy to avoid or fix (e.g., unused variable, mismatched-new-delete).
-Wextra	Same as Wall with extra warnings (e.g., uninitialized variables, empty body)
-Werror	Make all warnings into errors (prevent compilation / executable)

You can also add these to your Makefile!

Makefile example with the files main.cpp objectname.cpp and objectname.h

```
# Define Makefile variables
CC = g++ -std=c++11 -g
CFLAGS = -Wall -Werror -Wextra
EXE_FILE = main

# target: <prerequisites/dependencies (optional)>
#     <commands>
#     ...
#  ^  makefiles require tab indents here (throws error)

# run "make" with no target will do the first target in makefile
# ("all" in this case)
all: $(EXE_FILE)

# Compile and link files into executable called main
$(EXE_FILE): main.cpp objectname.o
    $(CC) $(CFLAGS) main.cpp objectname.o -o $(EXE_FILE)

# Compile with "-c" flag to create object files
objectname.o: objectname.cpp objectname.h
    $(CC) $(CFLAGS) -c objectname.cpp

# run "make clean" to use this rule, also an example of how to run
# multiple commands
clean:
    rm -f *.o $(EXE_FILE)
    clear
```

Avoid “using namespace std;”

Why? Because it forces everyone else who might import your code to also use namespace std. This also brings in a *ton* of identifiers that you may not need or use at all and causes naming conflicts (especially bad practice in companies that have their own libraries like in game development). For the scope of this class, the use of namespace is okay.

One way to still have the convenience of “using namespace std;” is by only importing specific identifiers.

Do ✔

```
#include <iostream>
#include <string>
using std::string;

int main() {
    string greeting = “Hello!”;
    std::cout << greeting << std::endl;
}
```

Avoid over-nesting (Keep your code shallow)

Deeply nested code is where a code block has 4 or more indents/levels. Nested code can be challenging to work with a read or debug as it requires keeping track of what block the program is in and what values / assumptions are true in that block. This often occurs because of if-else conditions, especially for “bad / error” case handling, where there is some top-level condition that holds the rest of the code hostage.

Source: Why You Shouldn't Nest Your Code - YouTube (Great explanation + animation!)

One way of reducing or eliminating some levels of nesting is by using `guards` which is a form of an early return in a function if a special case(s) is met.

Source: On Using Guards in C++ - Fluent C++ (fluentcpp.com)

** Code Example in progress **

Substitute Magic Numbers for Constants or Variables (DRY)

Magic numbers refer to those literal values used in some logical expression such as the size of an array, a specific *N*th index, percentage value for randomness, or some range. These can be incredibly difficult to read or modify after some time becoming time sinks (e.g., “Why is there a `length - 2` here?”).

** Code Example in progress **

Got many parameters? Consider using structs!

Have you run into situations where you have a similar group of variables that are passed repeatedly to many functions? Consider grouping them into a struct or class! This provides several advantages:

- Values that are intended to be used together and share a role can be grouped explicitly (can improve readability)
    - ex: Group customer information (name, address, email) into a struct for printing/updating
- Reduces number of parameters a function requires (easier to add a new struct member if you change your mind than having to update implementation and header files)
    - ex: Now store phone numbers for customer information

Source: c - Passing many variables vs. passing struct - Stack Overflow

** Code Example in progress **