General

Errors:

Runtime error:

- An error one will encounter when they run their executable and the code reaches a certain point that attempts to do legal (i.e. access memory at the null pointer) or attempts to access a variable that is not initialized.
- Example
    - int x; printf("%d", x); // will give an error after running

Compile-time error:

- An error one will encounter when they build their project. This will prevent the executable from being built, but some compilers may continue building through errors to check the rest of the files.
- Example
    - `int var = two;` // two is undefined, so the executable will not be generated.

Naming:

- Cannot start with a number
- Should be meaningful. Do not give them some generic names such as x or y because it will make it confusing in the future when you have many names.

What is a keyword?

- It is a reserved word that cannot be used to define a variable. For example, int, void, inline, return, if, switch, default, true, and false are all words that will result in a compile-time error to prevent erratic behavior from the executable.

Where should a space be?

- For a variable:
    - type name = value;
- For a method declaration:
    - return_type myMethod(int something, int something 2) { …
- For a method call:
    - myMethod(1, 3);

Variables

Naming (Basic):

- Start with a lowercase character
- Should use camel case. Ex. anExampleOfCamelCase

```
int              myVar          =      100;
```
^ variable type ^ variable name ^ variable value

Primitive types

Types that are inbuilt to the C compiler so it can be ran on all systems that have the C compiler.

- int
- double
- float
- char
- void

Constants

Naming:

- All uppercase letters with an underscore between words or numbers

Contents:

- No limitations
- Can be a number (1), reference to a variable (2), or a call to a method (3)

```
#define EXAMPLE_1 2
#define EXAMPLE_2 testVar
#define EXAMPLE_3 myMethod3()
```

Include Statements

Use double quotes for when you want to use local header files. Double quotes can technically work for any header file as the compiler will function as if the stdio.h was not found in the project, it will then look in the system headers. For system header files such as stdio.h, angular brackets are required.

```
#include <stdio.h>
#include "myFileName.h"
```

Casting

Casting is when you treat one type as a similar type in a single instance by the convention shown below. This does not change the initial variable's type.

```
float myFloat = 500.239184678671289;

int myInt = myFloat; // Implicit Casting // value = 500
int myInt2 = (int)myFloat; // Explicit Casting // value = 500

myInt = myFloat; // Implicit Casting // value = 500
myInt = (int)myFloat; // Explicit Casting // value = 500
```

Explicit casting meaning forcing the type by the notation: `(*type*)`

Implicit means allowing the compiler to interpret the required type

Pointers

The memory address of a variable. To get the pointer of a variable, simply put a ‘&’ before the variable name (see “Scanf and Printf Methods” for how to use). If you want to use a pointer in a method, use a star post-ceding the type and pre-ceding the variable name. If you are unfamiliar with pointers, do not use this at this point. Pointers are used when you want to use or set a variable somewhere, but you are not in the same block of code where the said variable is defined. In the example below, a ‘void*’ is used as a type, this is valid! Some compilers will catch the difference between the arguments below, but most will see they a pointer and not distinguish the difference. It is good practice to use pointers designed to their specific type to ensure there is not confusing when de-referencing the variable. De-refencing simply refers to calling the variable directly using the pointer. See below of how to de-reference variables.

```
void test(void* generic, int* count, double* decimal);

*generic = 10;
*count   = 100;
*decimal = 20;
```

Scanf and Printf Methods

scanf will read from the user’s keyboard. See syntax and variable types as follows:

```
scanf("%d %lf %f %c", &myInt, &myDouble, &myFloat, &myChar);
```

printf will read from the user’s keyboard. See syntax and variable types as follows:

```
printf("int = %d double = %lf float = %f char = %c", myInt,
myDouble, myFloat, myChar);
```

Please contact me with any further questions regarding printf and scanf statements.

Logic Statements

For a Boolean value (`true` or `false`), you must include <stdbool.h>. This is only required if you use the `bool` keyword.

```
int c1 = 1; // int value of logical true
int c2 = 0; // int value of logical false
bool c3 = true;  // boolean true
bool c4 = false; // boolean false

// Conditional Statements
c1 == c2 // Logical comparison between two numbers, returns true
if equal, false otherwise.
c1 != c2 // Logical not comparison between two numbers, returns
true if not equal, false if equal.
c1 >= c2 // Logical greater than or equal to comparison between
two numbers, returns true if not equal, false if equal.
c1 <= c2 // Logical less than or equal to comparison between two
numbers, returns true if not equal, false if equal.
c1 > c2 // Logical greater than comparison between two numbers,
returns true if not equal, false if equal.
c1 < c2 // Logical less than comparison between two numbers,
returns true if not equal, false if equal.
```

cond*x* should be replaced by a conditional statement or a Boolean variable (i.e., c3, c4)

```
if (cond1) {
     // do something if cond1 is true, otherwise go to the else
if statement
} else if (cond2) {
     // do something if cond2 is true, otherwise go to the next
else if statement
} else if (cond3) {
     // do something if cond3 is true, otherwise go to the next
else if statement
} else if (cond4) {
     // do something if cond4 is true, otherwise go to the else
statement
} else {
     // do something if all the above conditions are false
}
```

How an if statement works:

1. Start at the initial ‘if’ statement.
2. If the condition is false, move to the ‘else if’ following the initial ‘if’ statement
3. If the condition is false again, move to the following ‘else if’ statement

   a. repeat until a true condition is found or the end of the conditions are reached
4. If all of the if/else if statements are true, run the else block.

All If statements require parenthesis around the condition as see above; else should never take a condition. \`else\` and \`else if\` statements are not a required part of if declarations. If a statement contains and \`else\` clause, it is guaranteed to run. If a statement consists of solely \`if\` and \`else if\`s are contained, then there is no guarantee any statement block will run.

```
switch (VAR) {
    case 0:
       break;
    case 1: {
       break;
    }
    case 3:
    {
       break;
    }
    default: break;
}
```

A switch statement is like an \`if\` statement. \`VAR\` must be an integral or enumerated type, or in basic terms, it should be a number. There can an any amount of \`case\` statements, and it should always contain a \`default\` case to ensure there is always an outlet incase none of the cases are matched. While it should work without the \`default\` case, it should always be included to ensure there is no unexpected behavior. For cases with multiple lines, curly braces should be used, such as seen in case 1 and 3. All cases must be terminated with a \`break\` clause since it will prevent code from running from the matched case to the bottom. (i.e., if case 0 did not have a \`break\` statement, case 0 and case 1 would run).

Note: Do not use these statements if you do not fully understand them at this point in the class. You can visit my office hours if you would like more information.

Parts of a Method

```
int myMethod0(char first);

int – return type
myMethod0 – method name
char – parameter
```

The return type is the variable type that will be retuned at the end of the declared method. If \`void\` is not the return type (i.e., int, double, float), then a return statement is required. If the return type is `void`, then a return statement is recommended, but not required.

The method name is a unique value that identifies any specific method. In C, all names must be unique and so-called method overloading is not allowed in C (that will be later in C++). In brief, overloading is where you can have different return types with different parameters using the same name – if this is confusing, do not worry about this now.

The parameter count can be anywhere from zero to many. If it is zero, `void` must be declared as the parameter to tell the compiler that no arguments are allowed. *Note that `void` is only required in the Method stub.* To properly declare an argument, type the variable's type then the name (see above). For more than one argument, see below.

Methods

```
int myMethod1(int first, char second); // Method Stub
```

Naming convention:

- Start with a lowercase letter
- Camel case. example: aVeryLongNameInCamelCase
- Have the name clearly indicate the purpose

Usage

- It should be reusable in nature (at first, this is not necessary)

In this method, int is the return type. This means a return statement is **required** in the implementation of myMethod1(_,_). The return value must be an int, implicit casting is not proper for this scenario. I recommend declaring a variable at the top of the method then returning the value at the end (see below). If you follow that rule, debugging in the future will be easier.

```
int myMethod1(int first, char second) {
    int returnValue;

    returnValue = first + second;

    if (second == 4) { // This has no practical meaning. It is some made up conditional statement.
        returnValue = first + second * 3;
    }

    return returnValue;
}
```

Inline Keyword:

Inline has a somewhat complex usage. It should only be used with smaller methods as it tells the compiler to insert the whole method in the caller's position. If the compiler finds it efficient enough to insert the method, then runtime will improve. If the compiler does not find it more efficient to insert the method, then it has no impact on the method. *If this does not make sense, please do not use this, or come in to my office hours for further information.* See https://en.cppreference.com/w/cpp/language/inline for more information.

```
inline int myMethod2(void);
```

Signed and Unsigned values

Implicitly, all values are declared as signed.

Signed numbers can have a value that is positive or negative.

Unsigned numbers can only be positive.

Size and Ranges of Int, Double, Float, Char

The size of a data type is defined by how many bytes the single variable takes in memory. For those of you taking EE234, you will learn about this in depth.

Please do not worry about how this effects the program at this point in the semester, see examples for relevance.

The range of the data type is defined as the minimum to the maximum some type can be. If a value exceeds the minimum or maximum of the given type, then it will "overflow" into the type's range. *Please look after the table for examples of how overflowing work.* It can pose a security vulnerability to ignore this type of error in real-world applications.

Do not worry about the mechanics behind overflowing but be aware of that this exists and converting or casting between types can be dangerous and yield unwanted or unexpected values.

Primitive Type	Minimum	Maximum	Size (B)
signed int	**-32_768**	**32_767**	**2**
unsigned int	0	65_535	2
signed char	**-128**	**127**	**1**
unsigned char	0	255	1
double	1.7E-308	1.7E+308	8
float	3.4E-38	3.4E+38	4

Note that double and float do not have a signed/unsigned designation. This is due to a reason beyond this class. *Table derived from https://clanguagebasics.com/data-types-in-c/.*

Examples:

Ex 1) Some variable x is an **unsigned int** and assigned the value -500 at initialization.

As we can see above, the min value is 0 for an **unsigned int**. This is an overflow, and the system will (*typically*) flow the number back to the maximum of 65_535 then subtract 500 and assign the value of 65_035 to x. Sometimes the compiler may have a flag where it prevents values to overflow like this and a fatal error will be yielded. A practical example of this would be when we convert a signed int to an unsigned int.

Ex 2) Some variable x is an **unsigned int** and assigned the value -100_000 at initialization.

In the last example, the value was also negative, but it was only 500 out of bounds. Now, we are more than the max value's capacity. The same process applies except the process is repeated. First, we take away 65_535 since it overflows again and get 34_465. This time, x will be assigned a value of 31_070 since it had to go around twice. This may happen when a value starts as a double or float, but then it was casted to an integer, and it is way too large to fit into 65_535.