This document shows some of the common errors and warnings that you may encounter during this class. Always remember to use the –Wall option if you are using gcc. Also, never neglect warnings!

Note: Messages from the compilers gcc (minGW – codeblocks) and Visual Studio 2013 are shown for various code snippets.

There are generally two broad categories of errors you may encounter.

1. **Syntax Errors**

   These errors are detected at compile time and most often arise due to incorrect syntax. Compiler errors are fatal, which means an executable program will not be generated until all the errors are resolved. Compiler warnings, on the other hand, are not fatal. However, you should still determine the cause of the warning and fix it.

   1. **Misspelled function names**

   Errors may arise if the correct name of variables/functions are not specified. Remember C is a case sensitive language.

   Ex: `Main()` is not the same as `main()`.

   ```c
   #include <stdio.h>
   int Main (void)
   {
       printf("Hello World!\n");
       return 0;
   }
   ```

   `gcc: main.c:(.text.startup+0xa7)||undefined reference to `WinMain@16`

   `VC: MSVCRTD.lib(crtexe.obj) : error LNK2019: unresolved external symbol _main referenced in function __tmainCRTStartup`
2. **Undefined variables**

The compiler usually reports an error for undefined variables or functions. Make sure that variables or functions you declared have appropriate scope.

Ex. Remember to define variables with proper scope.

```c
#include <stdio.h>
int main (void)
{
    int myVar = 10;
    printf("value of x = %d\n", myVar);
}
printf("value of x = %d\n", myVar);  // out of scope for myVar
return 0;
}
gcc: error: 'myVar' undeclared (first use in this function)|
    note: each undeclared identifier is reported only once for each function it
appears in
VC: error C2065: 'myVar' : undeclared identifier
```

3. **Undeclared/Improperly declared function prototypes**

The compiler reports an error if the function being used has no prototype declared. It may report warnings if the definition does not match the prototype.

Ex.1: Absence of Function prototype

```c
#include <stdio.h>  // missing prototype
void myFunc (void);
int main (void)
{
    myFunc();
    return 0;
}
void myFunc (void)
{
    printf("You are in the function\n");
}

gcc: main.c|7|warning: implicit declaration of function 'myFunc' [-Wimplicit-function-declaration]
main.c|12|warning: conflicting types for 'myFunc' [enabled by default]
main.c|7|note: previous implicit declaration of 'myFunc' was here
VC: warning C4013: 'myFunc' undefined; assuming extern returning int
error C2371: 'myFunc' : redefinition; different basic types
```
Ex.2: Remember to use the semicolon correctly. A declaration has a semicolon at the end, while a definition doesn’t.

```c
#include <stdio.h>

void myFunc (void);  // Declaration needs the semicolon

int main (void)
{
    myFunc();
    return 0;
}

void myFunc (void);  // Definition does not need the semicolon – trouble here!
{
    printf("You are in the function\n");
}
gcc: expected identifier or '(' before '{' token
VC: error C2449: found '{' at file scope (missing function header?)
    error C2059: syntax error : '}'
```

4. Parse errors
Parse errors may arise due to a missing semicolon at the end of previous statement, a missing comment delimiter (*/) at the end of last comment, or an undefined macro or type definition (such as using FILE without including stdio.h)

Ex.1: Typical parse error

```c
#include <stdio.h>

int main (void)
{
    int tempVar;
    tempVar = tempVar + 2       // missing ';'  
    return 0;
}

gcc: error: expected ';' before 'return'
VC error C2143: syntax error : missing ';' before 'return'
```
5. Incorrect use of braces

This is a common error in switch and if-else block statements.

Ex: Make sure braces are balanced and properly indented to avoid errors.

```
#include <stdio.h>

int main (void)
{
    int var = 1000;
    printf("Value = %d\n", var);
    if (var == 5)
    {
        printf("Value = %d\n", var);
        // Missing ending brace
    return 0;
}

gcc: error: expected declaration or statement at end of input
    warning: control reaches end of non-void function [-Wreturn-type]
VC: end of file found before the left brace '{' at '..source.c(5)' was matched
```

6. Character constant too long

In C, strings must be surrounded by double quotation marks (" "). If you have used single quotation marks ('), then C expects to find a single character (or an escape character). Also, make sure that you have a balanced number of these quotes to avoid unterminated string or character constant errors.

Ex. Incorrect quotes.

```
#include <stdio.h>

int main (void)
{
    int var = 1000;
    printf("Value = %d",
var);
    -- Used single instead of double quote

    return 0;
}

gcc: warning: missing terminating " character [enabled by default]
    error: missing terminating " character
    error: unexpected expression before 'return'
    error: expected ';' before '}' token
    warning: unused variable 'var' [-Wunused-variable]
    warning: control reaches end of non-void
VC: error C2001: newline in constant
    error C2143: syntax error : missing ')' before 'return'
```
7. **Multiple declarations or definitions**

In the C language any entity can be declared many times but defined only once through the entire program. This can be a problem with a program spanning multiple files. Hence all the .h files are written so that they contain only declarations.

Ex: Multiple Declarations.

```c
#include <stdio.h>

void menu (void);
void menu (void);  // NO error!

int main(int argc, char* argv[])
{
    menu();
    return 0;
}

void menu (void)
{
    printf("Hi\n");
}

void menu (void)  // error
{
    printf("Hello\n");
}
```

`gcc: error: redefinition of ‘menu’`

Previous definition of ‘menu’ was here.

`VC: error C2084: function ‘void menu()’ already has a body`
II. Logical or Runtime Errors

These errors occur during runtime. The program compiles normally. However, it may not run at all or run incorrectly. These errors tend to be difficult to track down and fix.

1. **Uninitialized variables**
   In the C language, not all types of variables are initialized to zero. Remember to explicitly initialize the variables before using them, or else you could see odd program behavior.

   Ex. Uninitialized variables.

   ```c
   int main (void)
   {
      int count;
      while (count < 100)
      {
         printf("Value of count = %d\n", count);
         count++;
      }
   return 0;
   }
   ```

   gcc: watch out there may not be warning at all!
   VC: error C4700: uninitialized local variable 'count' used

2. **Wrong operator error**
   Do not mix-up the following operators:
   a. Assignment operator ( = ) and test for equality operator ( == )
   b. Logical AND operator ( && ) and bit wise AND operator ( & )
   c. Logical OR operator ( || ) and bit wise OR operator ( | )
   d. Shift operator ( >> and << ) and comparison operator ( > and < )
   e. Address operator ( & ) and dereferencing operator ( * )
   f. Incorrect usage of pre and post increment ( ++ ) and decrement ( -- )

3. **Using switch-case constructs with missing “break” statements;**
   This causes all the statements following the chosen case statement to be executed. If unintended, this may result in unknown behavior.

4. **Mismatched “if-else” statements**
   Make sure to deploy if – else statements in a proper block-wise manner using braces. It is very easy to mismatch dangling else statements.
5. **Passing by value and passing by reference**
   Remember that a value passed by value essentially copies the value while a reference modifies the supplied value.

6. **Implicit casting of unsigned to signed variables**
   Implicit typecasting can cause many program errors. If at all needed, carefully consider the consequences and explicitly typecast the expression.

7. **Order of arithmetic, relational, and logical operations**
   The C language has operator precedence and associativity as stated by the standard. Make sure that the intent of your expression matches what the rules of precedence will impose.

8. **Numeric Issues**
   Learn how floating point and integer values are represented in memory and how they can affect a calculation or comparison operation.

   a. **Math over or under flow**
      If it is possible for a numeric exception to arise (e.g., division by zero), make sure your program has code to handle the situation.

   b. **Test for equality**
      Floating point math is not exact. Simple values like 0.2 cannot be precisely represented using binary floating point numbers, and the limited precision of floating point numbers means that slight changes in the order of operations can change the result. Different compilers and CPU architectures store temporary results at different precisions, so results will differ depending on the details of your environment. If you do a calculation and then compare the results against some expected value, it is highly unlikely that you will get exactly the result you intended.

      In other words, if you do a calculation and then do this comparison:

      ```
      if (result == expectedResult)
      ```

      then it is unlikely the comparison will be true. If the comparison is true then it is probably unstable – tiny changes in the input values, compiler, or CPU may change the result and make the comparison be false.

9. **Variable scope**
   Remember the differences between static, global, automatic, and register variables regarding their scope and lifetime.
10. **Arrays**

Arrays and pointers are often used interchangeably in the C language. Make sure that you don’t make following errors.

a. **Array name (address of zero index element) and value of zero indexed element**
   If an array myArray[10] is declared, then the identifier myArray gives you the address of first array element. To access the first element itself, you need to use myArray[0].

b. **Exceeding array bounds**
   This is the classic cause for memory corruption and buffer overflows. No warning or error messages are issued by the compiler. This type of error has historically compromised many computer systems worldwide.

c. **Declaration and index range**
   Declare the size of an array to be large enough to handle both present and anticipated future needs.

11. **Strings**

Strings are characters appended together with a NULL character at the end. Strings and character constants are handled differently. Make sure you know the differences.

a. **No NULL character at end of array**
   Every string has a ‘\0’ at the end. No NULL at the end may cause program failure.

b. **Comparing strings using “==”**
   Strings cannot be compared using “==”! Use standard string library functions.

c. **Insufficient memory for NULL character in string array**
   An array that holds a string should have a declared size with one extra element (size+1) to hold the NULL character.

d. **Differences in string declarations and character declarations**
   Strings are arrays of characters with a NULL at the end. Characters are single values.

e. **Passing strings**
   Strings can be passed to a function as either character arrays or as character pointers.
Most of these runtime errors may be benign for the general PC software industry. However it is dangerous for real-time embedded systems employed in safety critical application like aviation, nuclear, space, automotive, etc.

To avoid this the automotive industry created a set of rules for coding embedded applications in the C language, which has been adopted by many industries. These are known as MISRA C (Motor Industry Software Reliability Association) standards.

(http://en.wikipedia.org/wiki/MISRA_C)

References:

3. http://www.csee.umbc.edu/courses/104/fall05/ratsimor/Compiling-C.htm