Stack Buffer Overflow:

```
10  void CWE121_Stack_Based_Buffer_Overflow__CWE806_wchar_t_declare_memmove_10_bad()
11  {
12      wchar_t * data;
13      wchar_t dataBuffer[100];
14      data = dataBuffer;
15      if(globalTrue)
16      {
17
18          wmemset(data, L'A', 100-1);
19          data[100-1] = L'\0';
20      }
21      {
22          wchar_t dest[50] = L"";
23
24          memmove(dest, data, wcslen(data)*sizeof(wchar_t));
25          dest[50-1] = L'\0';
26          printWLine(data);
27      }
28  }
```

From the above code, we can see a stack buffer overflow caused by the statement "`memmove(dest, data, wcslen(data)*sizeof(wchar_t))` " at line 24. Specifically, the buffer "dest" has a size of 50 "wchar_t" elements. However, the statement tries to copy the string carried in the buffer named "data" to "dest". As initialized by the statements of "wmemset(data, L'A', 100-1)" at line 18 and "data[100-1] = '\0'" at line 19, the string in "data" will have a size of 99 "wchar_t" elements. Thus, the memmove will overflow "dest" and thus, a stack overflow happens.

Heap Buffer Overflow:

```
16  void CWE122_Heap_Based_Buffer_Overflow__c_CWE805_wchar_t_snprintf_01_bad()
17  {
18      wchar_t * data;
19      data = NULL;
20
21      data = (wchar_t *)malloc(50*sizeof(wchar_t));
22      if (data == NULL) {exit(-1);}
23      data[0] = L'\0';
24      {
25          wchar_t source[100];
26          wmemset(source, L'C', 100-1);
27          source[100-1] = L'\0';
28
29          SNPRINTF(data, 100, L"%s", source);
30          printWLine(data);
31          free(data);
32      }
33  }
```

From the above code, we can see a heap buffer overflow caused by the statement “SNPRINTF(data, 100, L"%s", source)” at line 29. Specifically, “data” is a buffer with 50 “wchar_t” elements, allocated at line 21. However, the statement at line 29 tries to move 100 “wchar_t” elements from “source” to “data”. Thus, the buffer “data” will be overflowed and a heap overflow happens.

Use After Free:

```
10  void CWE416_Use_After_Free__malloc_free_int64_t_11_bad()
11  {
12      int64_t * data;
13
14      data = NULL;
15      if(globalReturnsTrue())
16      {
17          data = (int64_t *)malloc(100*sizeof(int64_t));
18          if (data == NULL) {exit(-1);}
19          {
20              size_t i;
21              for(i = 0; i < 100; i++)
22              {
23                  data[i] = 5LL;
24              }
25          }
26
27          free(data);
28      }
29      if(globalReturnsTrue())
30      {
31
32          printLongLongLine(data[0]);
33
34      }
35  }
```

From the above code, we can see a use-after-free caused by the statement at line 32. Specifically, the code declares a pointer “data” at line 12, and makes “data” point to a buffer allocated on the heap at line 17. The buffer is initialized at line 23 and then freed at line 27. However, the buffer, after being freed, is used again at line 32. Thus, a use-after-free happens.

Integer Overflow:

```
 8  void CWE190_Integer_Overflow__int64_t_max_preinc_12_bad()
 9  {
10      int64_t data;
11      data = 0LL;
12      if(globalReturnsTrueOrFalse())
13      {
14
15          data = LLONG_MAX;
16      }
17      else
18      {
19
20          data = 2;
21      }
22      if(globalReturnsTrueOrFalse())
23      {
24          {
25
26              ++data;
27              int64_t result = data;
28              printLongLongLine(result);
29          }
30      }
31      else
32      {
33
34          if (data < LLONG_MAX)
35          {
36              ++data;
37              int64_t result = data;
38              printLongLongLine(result);
39          }
40          else
41          {
42              printLine("data value is too large to perform arithmetic safely.");
43          }
44      }
45  }
```

From the code at line 10, we can see that the variable “data” has a type of “int64_t” (also known as “long long int”), whose value ranges from -2^31 to 2^31-1. The code at line 15 assigns LLONG_MAX (i.e., 2^31-1) to “data” and then adds 1 to “data” at line 26, which goes beyond the maximal value of a “long long int” and overflows the value in “data”. Thus, an integer overflow happens.

Use of Uninitialized Variable:

```
10  void CWE457_Use_of_Uninitialized_Variable__struct_13_bad()
11  {
12      twoIntsStruct data;
13      if(GLOBAL_CONST_FIVE==5)
14      {
15
16          ;
17      }
18      if(GLOBAL_CONST_FIVE==5)
19      {
20
21          printIntLine(data.intOne);
22          printIntLine(data.intTwo);
23      }
24  }
```

From the above code, we can see the use of uninitialized variables caused by the statements at line 21 and line 22. Specifically, the code first declared a variable “data” at line 12. However, the code never initializes the data in that piece of memory before using the data at line 21 and line 22. Thus, use of uninitialized variables happens.