###################### Memory Allocation in C ######################

1. **Auto Local:**
   - **Memory Allocation Time:** Created when a function is called and allocated on the stack.
   - **Memory Release Time:** Automatically released when the function execution scope is exited.
   - **Initialized to:** Garbage/undefined values.
   - **Scope of Visibility:** Limited to the function where it is declared.
   - **Memory Segment:** Stack.

2. **Static Local:**
   - **Memory Allocation Time:** Created when a function is called and allocated on the data segment.
   - **Memory Release Time:** Exists throughout the programs lifetime, retains its value between function calls.
   - **Initialized to:** Zero or null if not explicitly initialized.
   - **Scope of Visibility:** Limited to the function where it is declared.
   - **Memory Segment:** Data! yes Data! always.

```
        /*$ nm Test_ws10_build.o  ws10_build.o

        Test_ws10_build.o:
        0000000000000000 T main
        0000000000000000 d x1
        0000000000000008 b x2
        000000000000000c b x3
        0000000000000004 D x4
        0000000000000000 B x5
        0000000000000004 B x6
        static int x1 = 3;
        static int x2 = 0;
        static int x3;
        int x4 = 10;
        int x5 = 0;
        int x6;*/
```

3. **Global:**
   - **Memory Allocation Time:** Created at program start and allocated on the data segment.
   - **Memory Release Time:** Exists throughout the programs lifetime.
   - **Initialized to:** Zero or null if not explicitly initialized.
   - **Scope of Visibility:** Visible across the entire program (extern)
   - **Memory Segment:** Data / BSS.

4. **Static Global:**
   - **Memory Allocation Time:** Created at program start and allocated on the data segment.
   - **Memory Release Time:** Exists throughout the programs lifetime.
   - **Initialized to:** Zero or null if not explicitly initialized.
   - **Scope of Visibility:** Limited to the file where it is defined.
   - **Memory Segment:** Data / BSS.

5. **Dynamic Allocation:**
   - **Memory Allocation Time:** Dynamically allocated at runtime using functions like `malloc` or `calloc`, and allocated on the heap.
   - **Memory Release Time:** Manually released using `free` when no longer needed.
   - **Initialized to:** Uninitialized (garbage values) unless explicitly initialized.

   - **Scope of Visibility:** Visible throughout the scope where the pointer is accessible.
   - **Memory Segment:** Heap.

6. **Function:**
   - **Memory Allocation Time:** Function code is loaded into memory when the program starts.
   - **Memory Release Time:** Released when the program exits.
   - **Initialized to:** Code instructions.
   - **Scope of Visibility:** Global scope, accessible from anywhere in the program.
   - **Memory Segment:** Code.

7. **Static Function:**
   - **Memory Allocation Time:** Function code is loaded into memory when the program starts.
   - **Memory Release Time:** Released when the program exits.
   - **Initialized to:** Code instructions.
   - **Scope of Visibility:** Limited to the file where it is defined (internal linkage).
   - **Memory Segment:** Code.

###

The `-fno-common` flag is a compiler option used in GCC (GNU Compiler Collection) to disable the implicit creation of common variables in C and C++ programs. This flag is used during the compilation process to enforce a stricter interpretation of variable declarations.

In C and C++, a "common variable" is a global variable that is defined multiple times across translation units (source files). By default, GCC allows the creation of common variables if they are not explicitly defined as `static` or `extern`. This can lead to unexpected behavior and linker errors when the same variable is defined in multiple source files.

Using the `-fno-common` flag instructs the compiler to treat tentative definitions (implicit multiple definitions) of global variables as errors, forcing you to use explicit declarations with `extern` and proper definitions in one translation unit.

For example, consider the following code:

```
int x = 42;
```

If this code is included in multiple source files, you might encounter linker errors due to multiple definitions of `x`. To prevent this, you can use `-fno-common` along with proper `extern` declarations in header files and a single definition in one source file.

Keep in mind that using `-fno-common` might require modifications to your code and build process to ensure proper declarations and definitions of global variables. It enforces stricter rules and helps avoid pitfalls related to common variable behavior.

###

"Strong" and "weak" are terms used in the context of linking and symbol resolution in programming languages like C and C++. They refer to how the linker handles multiple definitions of the same symbol (such as functions or variables) from different source files or libraries.

Strong Symbols:
- Strong symbols are symbols that have a single definition that the linker will use.
- When multiple strong symbols with the same name are encountered during linking, it results in a linker error due to symbol duplication.
- Functions and initialized global variables are typically treated as strong symbols.

Weak Symbols:
- Weak symbols are symbols that can have multiple definitions, and the linker will choose one definition without generating an error.
- If a strong symbol and a weak symbol have the same name, the strong symbol takes precedence over the weak symbol.
- If multiple weak symbols with the same name are encountered during linking, the linker will choose one and discard the others.
- Uninitialized global variables, inline functions, and functions marked as weak are often treated as weak symbols.

For example, let's consider the following code:

```c
// File: file1.c
int x = 42;  // Strong symbol

// File: file2.c
int x = 99;  // Strong symbol
```

If you compile and link these two files together, you'll likely get a linker error due to the duplication of the strong symbol `x`.