Sure! In C (and Unix-like systems), the `exec` **family of functions** is used to replace the current process image with a new process image. That means the calling process is **overwritten** by a new program; it does not return unless there's an error. These functions are in `<unistd.h>`.

Let's go **step by step** and cover all the major `exec` functions, their variants, and usage.

---

1. The exec family

The `exec` family consists of several functions:

Function	Description
`execl`	Executes a program with a **list of arguments** (arguments passed individually).
`execle`	Like `execl`, but allows you to **pass custom environment variables**.
`execlp`	Like `execl`, but **searches** `PATH` for the program.
`execv`	Executes a program with **arguments passed as an array**.
`execve`	Like `execv`, but allows **custom environment variables**. This is the **system call**, most others wrap it.
`execvp`	Like `execv`, but **searches** `PATH` for the program.
`execvpe`	Like `execvp`, but allows **custom environment** (GNU extension).

All `exec` functions **do not return** if successful. If they fail, they return `-1` and set `errno`.

---

2. Function signatures

a) `execl`

```
int execl(const char *path, const char *arg0, ..., (char *)NULL);
```

- `path` → Path to the program. - `arg0` → Usually the program name. - `...` → Remaining arguments. - Last argument **must be** `NULL`.

Example:

```
#include <unistd.h>

int main() {
```

```
    execl("/bin/ls", "ls", "-l", "-a", NULL);
    return 0; // only reached if execl fails
}
```

---

b) `execle`

```
int execle(const char *path, const char *arg0, ..., (char *)NULL, char * const
envp[]);
```

- Like `execl`, but lets you **pass environment variables** as `envp`.

Example:

```
#include <unistd.h>

int main() {
    char *env[] = {"MYVAR=123", "PATH=/bin", NULL};
    execle("/bin/printenv", "printenv", "MYVAR", NULL, env);
    return 0;
}
```

---

c) `execlp`

```
int execlp(const char *file, const char *arg0, ..., (char *)NULL);
```

- Searches directories in `PATH` to find `file`. - Otherwise like `execl`.

Example:

```
execlp("ls", "ls", "-l", NULL); // No need to give full path
```

---

d) `execv`

```
int execv(const char *path, char *const argv[]);
```

- Arguments are passed as an **array** instead of a list. - `argv[0]` is the program name. - Array **must end with NULL**.

Example:

```
char *args[] = {"ls", "-l", "-a", NULL};
execv("/bin/ls", args);
```

---

e) `execvp`

```
int execvp(const char *file, char *const argv[]);
```

- Like `execv`, but searches `PATH`.

Example:

```
char *args[] = {"ls", "-l", NULL};
execvp("ls", args);
```

---

f) `execve`

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

- The **system call** all others wrap. - Lets you specify **arguments and environment**.

Example:

```
char *args[] = {"ls", "-l", NULL};
char *env[] = {"PATH=/bin", NULL};
execve("/bin/ls", args, env);
```

---

g) `execvpe` (GNU extension)

- Like `execvp`, but allows a **custom environment array** (`envp`).
- Not POSIX standard, only GNU/Linux.

---

3. Important Notes

1. **No return on success:**
   Once `exec` succeeds, the current process image is replaced. Any code after the `exec` call is **not executed**.

2. **Environment handling:**

3. `execle` and `execve` allow **custom environments**.

4. Others inherit the current environment.

5. **Searching `PATH`:**

6. Functions ending in `p` (`execlp`, `execvp`) search `PATH`.

7. Functions without `p` require **full path**.

8. **Error handling:**

```c
if (execl("/bin/ls", "ls", "-l", NULL) == -1) {
    perror("execl failed");
}
```

9. **Combining with `fork()`:**

10. Common pattern: `fork()` creates a child, child calls `exec` to run a new program, parent continues.

```c
pid_t pid = fork();
if (pid == 0) { // child
    execlp("ls", "ls", "-l", NULL);
    perror("execlp failed");
} else { // parent
    wait(NULL);
}
```

If you want, I can **draw a complete diagram showing all exec calls, their arguments, and how they relate**—it makes remembering them **super easy**.