CGI (Common Gateway Interface) - CPP98 / Webserv

? What Is CGI?

CGI is a standard interface between web servers and external programs (scripts or binaries) that generate dynamic content.

- Allows Webserv to execute programs and return their output as HTTP responses
- CGI programs can be written in C++, Python, Bash, or any executable language
- Webserv invokes CGI with environment variables and stdin/stdout communication

🔧 Key Concepts

Environment variables: pass request info to CGI

- `REQUEST_METHOD`: GET, POST, etc.
- `QUERY_STRING`: data after `?` in URL
- `CONTENT_LENGTH`: size of POST body
- `CONTENT_TYPE`: MIME type
- `SCRIPT_NAME`, `PATH_INFO`, `SERVER_NAME`, `SERVER_PORT`

Standard I/O:

- stdin: CGI reads POST body
- stdout: CGI writes HTTP response (headers + body)

Security: restrict which scripts can execute

In Webserv: CGI is usually forked per request, output captured via pipe

📦 Headers / Imports

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <cstring>
#include <cerrno>
#include <iostream>
#include <cstdlib>
```

⚙ Declaration / Syntax

```
pid_t pid = fork();
if (pid == 0) {
    // child process: execute CGI
```

```
    execve(script_path, argv, envp);
    perror("execve failed");
    exit(EXIT_FAILURE);
} else if (pid > 0) {
    // parent process: read output via pipe
    waitpid(pid, &status, 0);
} else {
    // fork failed
    perror("fork failed");
}
```

🔧 Parameters / Details

fork():

- Returns 0 in child process, PID of child in parent, -1 on error

execve():

- `script_path`: path to CGI executable
- `argv`: argument array (argv[0] = program name, argv[n] = NULL)
- `envp`: environment variables array (key=value, NULL-terminated)
- Replaces current process image with new program

Environment variables:

- `REQUEST_METHOD`: GET / POST / etc.
- `QUERY_STRING`: for GET data
- `CONTENT_LENGTH`: bytes to read from stdin (POST)
- `CONTENT_TYPE`: MIME type of body
- `SCRIPT_NAME`, `PATH_INFO`, `SERVER_NAME`, `SERVER_PORT`: server info

Stdin / stdout with CGI:

- Parent writes POST body to pipe → child reads stdin
- Child writes headers + body → parent reads stdout for response

🔔 Basic Example

```
int pipe_fd[2];
pipe(pipe_fd);
pid_t pid = fork();
if (pid == 0) {
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    char *argv[] = {"/path/to/cgi_script", NULL};
    char *envp[] = {"REQUEST_METHOD=GET", NULL};
    execve(argv[0], argv, envp);
    perror("execve failed");
    exit(EXIT_FAILURE);
} else {
    close(pipe_fd[1]);
```

```
    char buffer[1024];
    int n = read(pipe_fd[0], buffer, sizeof(buffer));
    std::cout << std::string(buffer, n);
    waitpid(pid, NULL, 0);
}
```

Advanced Considerations

- Must handle POST data: read `CONTENT_LENGTH` bytes from stdin
- Capture both stdout and stderr for debugging CGI
- Handle timeouts to avoid blocking server on slow CGI
- Sanitize input to prevent command injection
- Consider caching frequent CGI results for performance

⚠ Gotchas

- Forgetting to set environment variables → script fails
- Not closing unused pipe ends → deadlock
- Partial reads/writes: always loop until done
- Overwriting stdout/stderr incorrectly → output lost
- Ignoring `fork()/execve()` errors → server crashes

42 Specific Notes

- Must fork for each request
- Keep functions <25 lines (norm compliance)
- Parse stdin for POST requests correctly
- Combine CGI output with HTTP headers properly
- Handle disconnects, large payloads, and errors gracefully

CPP98 Constraints

- No auto keyword or range-based loops
- Use arrays or `std::vector` for arguments/environment
- Manual memory management if needed
- Use only CPP98-compatible system calls