<2024-2-고급프로그래밍 수업>

C 언어 프로젝트 코딩 가이드

1. 소개

이 가이드는 C 언어 프로젝트에서 일관된 코딩 스타일을 유지하기 위해 작성되었습니다. 이 가이드는 GNU 코딩 표준을 따르며, 변수명 설정과 블럭 스타일에 대한 지침을 포함하고 있습니다.

2. 코딩 스타일

기본 코딩 형식을 정의하고, 들여쓰기와 같은 일반적인 규칙을 설명합니다.

(1) 들여쓰기 규칙

- 기본 들여쓰기는 스페이스 4 칸을 사용합니다.
- 모든 코드 블록(if, for, while 등)과 함수 정의 시 들여쓰기를 일관되게 유지합니다.

(2) 줄 길이 제한

- 한 줄의 최대 길이는 80 자 또는 120 자로 제한하는 것을 권장합니다.
- 이유: 긴 줄은 가독성을 떨어뜨리고, 코드 리뷰나 디버깅 시 불편을 초래할 수 있습니다.
- 예외: 긴 문자열, 주석, 또는 특정 포맷을 유지해야 하는 경우에는 제한을 초과할 수 있습니다.
- 자동 줄 바꿈 규칙: 한 줄이 길어지는 경우, 적절한 위치에서 줄 바꿈을 하여 코드를 여러 줄로 나눕니다. 함수 호출의 긴 매개변수 목록이나 긴 조건문은 다음 줄로 바꾸고 들여쓰기를 추가하여 가독성을 높입니다.

(3) 매직 넘버 사용 금지

- 매직 넘버(Magic Number)란 소스 코드에서 특별한 의미를 가지지만, 그 의미가 코드 자체로는 명확하게 설명되지 않는 하드코딩된 숫자나 값을 말합니다.
- 매직 넘버를 사용하지 않고, 의미 있는 이름을 가진 **상수로 정의**합니다.

- 코드 내에서 숫자 값 대신 #define 이나 const 로 의미 있는 이름을 사용합니다.

(4) 함수 작성 규칙

- 함수는 단일 책임 원칙을 따르며, 하나의 기능만 수행하도록 작성합니다. 두 개 이상의 기능을 포함할 경우, 각 기능을 별도의 하위 함수로 분리하여 구현합니다.
- CQS(Command-Query Separation) 원칙을 적용하여 명령과 조회를 분리합니다.
  - → 명령 함수는 프로그램의 상태를 변경하는 작업을 수행하고, 반환 값이 없도록 작성합니다.
  - → 조회 함수는 프로그램의 상태를 조회하고, 상태를 변경하지 않도록 작성합니다.
- 함수의 매개변수는 함수 내부에서 변경하지 않도록 하며, 필요한 경우에는 반환값을 통해 처리합니다. 이를 통해 함수의 예측 가능성과 코드의 안정성을 높일 수 있습니다.

들여쓰기 규칙	int main() {
	  printf("Hello, world!\n");
	  return 0;
	}

줄 길이 제한	if (student_score > 90 &&
	  student_attendance > 80 &&
	  student_homework_completion == true) {
	  // Do something
	}

매직 넘버 사용 금지	#define MAX_BUFFER_SIZE 1024
	#define MAX_ITERATIONS 60
	
	int buffer[MAX_BUFFER_SIZE];
	for (int i = 0; i < MAX_ITERATIONS; i++) {
	  // Do something
	}

함수 작성 규칙	단일 책임 원칙 적용: add_student_score: 점수 추가만 담당. get_highest_score: 최고 점수를 계산하여 반환. 명령과 조회 분리:

	add_student_score 는 상태를 변경하는 명령 함수. get_highest_score 는 조회 함수로 상태를 변경하지 않음.

3. 블록 스타일

중괄호 "{}"는 같은 줄에 작성합니다. 조건문, 반복문 등의 블록에서 이를 준수합니다.

함수	int main() { }

조건문 if	if (condition) {
	  // Do something
	} else {
	  // Do something else
	}

조건문 switch	switch (grade) {
	  case 'A':
	  printf("Excellent!\n");
	  break;
	  case 'B':
	  printf("Good job!\n");
	  break;
	  default:
	  printf("Invalid grade.\n");
	  break;
	}

반복문 for	for (int i = 0; i < 10; i++) {
	  printf("%d\n", i);
	}

반복문 while	while (count < 5) {
	  printf("Count is %d\n", count);
	  count++;
	}

4. 네이밍 규칙

변수명, 함수명, 구조체명, 열거형명, 매크로 상수의 네이밍 규칙을 정의합니다. 이름을 작성할 때는 코드의 의미가 명확하게 전달되도록 의미 있는 이름을 사용해야 하며, 이 규칙을 따릅니다.

(1) 변수명

- 변수명은 스네이크 케이스(snake_case)를 사용합니다.

- 전역 변수는 "g_"를 접두사로 붙입니다.
- 정적 변수는 "s_"를 접두사로 붙입니다.
- 루프 인덱스(i, j 등)는 반복문 내에서만 사용하며, 그 외의 경우에는 의미 있는 이름을 사용합니다.
- 의미가 명확한 이름을 사용하여 변수의 역할을 쉽게 이해할 수 있도록 합니다.
- 전역 변수는 프로그램의 복잡성을 줄이기 위해 가능한 사용을 피하고, 반드시 필요한 경우 static 으로 선언하여 변수의 접근 범위를 제한하는 것이 좋다.

(2) 함수명: 동사로 시작하며, 스네이크 케이스를 사용하여 함수의 목적을 나타냅니다. 함수명은 명확하고 구체적인 동작을 설명할 수 있어야 합니다.

(3) 구조체명: 파스칼 케이스(PascalCase)를 사용하고, typedef 별칭은 "_t" 접미사를 붙입니다.

(4) 열거형명: 파스칼 케이스를 사용하고, "Enum"을 접미사로 붙이며, 값은 대문자로 작성합니다.

(5) 매크로 상수: 모두 대문자로 작성하고, 단어 사이에 언더스코어(_)를 사용합니다.

변수명	스네이크 케이스 (snake_case)	buffer_size, temp_value, student_count
전역 변수명(global)	"g_"를 접두사로 붙임	g_total_count, g_error_flag
정적 변수명(static)	"s_"를 접두사로 붙임	s_cache, s_temp_value
함수명	스네이크 케이스, 동사로 시작함	calculate_sum(), initialize_buffer()
구조체명	파스칼 케이스 (PascalCase)	StudentInfo, CarSpecs_t
열거형명	파스칼 케이스 "Enum"을 접미사로 붙임	ColorEnum, StatusEnum
열거형 값 예시	값은 대문자로 작성	RED, BLUE, PENDING_STATUS
매크로 상수	모두 대문자, 단어 사이 언더스코어(_)	MAX_BUFFER_SIZE, DEFAULT_TIMEOUT, MIN_RETRY_COUNT

5. 주석 작성 규칙

주석은 코드의 가독성과 이해를 돕기 위해 적절한 수준으로 작성해야 합니다. 주석이 너무 많거나 적으면 오히려 혼란을 줄 수 있습니다.

- 주석 작성: 한 줄 주석은 "//"를 사용하고, 여러 줄 주석은 "/* ... */"로 작성합니다.
- 필요한 경우에만 주석을 작성: 코드가 명확히 설명하는 경우, 주석이 불필요할 수 있습니다.
- 코드와 주석의 일관성 유지: 코드가 변경되면 주석도 함께 수정해야 합니다.
- 명확하고 간결한 주석: 주석은 핵심을 간결하게 전달해야 하며, 불필요하게 길게 작성하지 않습니다.

(1) 함수 주석 규칙

- 모든 함수는 함수의 역할을 명확히 설명하는 주석과 함께 선언합니다.
- 각 함수의 위에 주석을 작성하여 함수의 목적, 매개변수, 반환 값 등을 설명합니다.
- 주석에는 함수의 역할, 입력 파라미터의 설명, 반환 값의 의미를 명확히 기술합니다.

(2) 변수 주석 규칙.

- 중요한 변수의 선언 시, 변수의 용도와 의미를 설명하는 주석을 작성합니다.
- 전역 변수나 정적 변수의 경우에는 변수의 범위와 사용 목적에 대한 주석을 포함합니다.

(3) 파일 주석 규칙.

- 파일의 맨 위에 작성하며, 파일의 목적, 작성자, 작성일, 수정 이력 등을 포함합니다.
- 파일의 주요 기능과 역할, 작성자 정보를 명시하여 코드의 변경 이력을 추적할 수 있도록 합니다.

한 줄 주석 작성	// 한 줄 주석 예제

여러 줄 주석 작성	/* 여러 줄 주석 예제 함수의 목적 및 사용법 설명 */

함수 주석 규칙	/*
	* 함수 설명:
	* 이 함수는 두 정수를 더한 결과를 반환합니다.
	*
	* 매개변수:
	* int a - 첫 번째 정수
	* int b - 두 번째 정수
	*
	* 반환 값:
	* 두 정수의 합
	*/
	int add(int a, int b);
	
	int add(int a, int b) {
	return a + b;
	}

변수 주석 규칙	// 총 학생 수를 저장하는 전역 변수
	int g_total_students;
	
	// 캐시 데이터를 저장하기 위한 정적 변수
	static char s_cache[256];

파일 주석 규칙	/* * 파일 이름: student_manager.c * 설명: 학생 정보를 관리하는 프로그램의 주요 기능을 구현한 소스 파일입니다. * 학생 추가, 삭제, 조회 등의 기능을 포함합니다. * 작성자: 홍길동 * 작성일: 2024-10-22 * 수정 이력: * - 2024-10-22: 파일 생성 (홍길동) * - 2024-11-05: 학생 삭제 기능 추가 (홍길동) * - 2024-11-20: 메모리 누수 수정 (홍길동) */

6. 메모리 관리

동적 메모리 할당 후에는 반드시 메모리를 해제하고, 포인터를 NULL 로 설정하여 댕글링 포인터를 방지합니다.

(1) 메모리 할당과 해제 규칙

- 동적 메모리 할당: 메모리를 동적으로 할당할 때는 malloc, calloc, realloc 등을 사용합니다. 할당된 메모리는 사용이 끝나면 반드시 free 를 사용하여 해제해야 합니다.
- 메모리 해제 후 포인터 초기화: free 를 호출한 후, 해당 포인터를 NULL 로 설정하여 댕글링 포인터(dangling pointer)를 방지합니다.
- 메모리 할당 실패 처리: 메모리 할당에 실패할 경우, NULL 반환 여부를 확인하고, 에러 처리를 구현합니다.

(2) 메모리 누수 방지

- 모든 동적 할당된 메모리는 해제: 프로그램 종료 전에 모든 동적으로 할당된 메모리를 해제해야 합니다. 특히, 반복문 내에서 메모리를 할당할 경우, 필요에 따라 중간에 메모리를 해제합니다.
- 정확한 메모리 크기 할당: 메모리 할당 시 필요한 크기만큼 할당하고, 불필요하게 큰 메모리를 할당하지 않도록 주의합니다.
- 메모리 할당과 해제의 짝 맞추기: 메모리 할당과 해제의 위치와 시점을 명확히 하여, 프로그램의 흐름에 따라 모든 할당된 메모리가 올바르게 해제되도록 합니다.

(3) 포인터 초기화 및 사용 규칙

- 포인터 초기화: 모든 포인터는 선언 시 NULL 로 초기화하고, 할당이 완료되면 유효한 주소로 설정합니다.
- 포인터 사용 전에 NULL 체크: 포인터를 사용하기 전에 반드시 NULL 인지 검사하여, 잘못된 참조를 방지합니다.
- 포인터 연산 시 주의: 포인터 연산을 수행할 때는 자료형의 크기와 메모리 구조를 고려해야 합니다. 잘못된 포인터 연산은 메모리 접근 오류를 유발할 수 있습니다.

메모리 할당과 해제 규칙	#include <stdio.h>
	#include <stdlib.h>
	
	int main() {
	  // 메모리 할당
	  int *arr = (int *)malloc(10 * sizeof(int));
	  if (arr == NULL) {
	  // 메모리 할당 실패 처리
	  fprintf(stderr, "메모리 할당 실패\n");
	  return -1;
	  }
	
	  // 메모리 사용
	  for (int i = 0; i < 10; i++) {
	  arr[i] = i * 2;
	  }
	
	  // 할당된 메모리 내용 출력
	  for (int i = 0; i < 10; i++) {
	  printf("%d ", arr[i]);
	  }
	  printf("\n");
	
	  // 메모리 해제
	  free(arr);
	  arr = NULL; // 포인터 초기화
	
	  return 0;
	}

메모리 누수 방지	#include <stdio.h>
	#include <stdlib.h>
	
	int main() {
	  // 10 개의 문자열을 저장할 메모리 할당
	  for (int i = 0; i < 10; i++) {
	  char *str = (char *)malloc(50); // 50 바이트 메모리 할당
	  if (str == NULL) {
	  // 메모리 할당 실패 시 처리
	  fprintf(stderr, "메모리 할당 실패\n");
	  continue;
	  }
	
	  // 문자열 사용 (예: 문자열 초기화)
	  snprintf(str, 50, "String number %d", i);

	printf("%s\n", str);
	
	// 할당된 메모리 해제
	free(str);
	}
	
	return 0;
	}

포인터 초기화 및 사용 규칙	// 포인터 초기화 예시
	int *ptr = NULL;
	
	// 포인터 사용 전에 NULL 체크
	if (ptr != NULL) {
	*ptr = 10;
	}
	
	// 메모리 해제 후 포인터 초기화
	free(ptr);
	ptr = NULL;

7. 파일 구조

파일 구조와 관련된 규칙을 정의하여, 코드의 모듈화를 촉진하고 유지보수성을 높입니다.

(1) 파일명 규칙

- 파일명은 스네이크 케이스(snake_case)를 사용하며, 파일의 역할을 명확히 나타낼 수 있도록 합니다.
- 예: student_manager.h, student_manager.c
- 헤더 파일은 .h 확장자를 사용하고, 소스 파일은 .c 확장자를 사용합니다.
- 파일명이 기능을 반영하도록 작성하여, 해당 파일의 역할을 쉽게 이해할 수 있도록 합니다.

(2) 헤더 파일과 소스 파일 분리

- 헤더 파일(.h): 함수 선언, 매크로 정의, 구조체 정의 등을 포함하며, 다른 파일에서 재사용할 수 있도록 인터페이스를 제공합니다.
- 소스 파일(.c): 함수 정의와 로직 구현을 포함하며, 헤더 파일을 포함하여 실제 기능을 구현합니다.

- 규칙: 헤더 파일은 해당 기능의 인터페이스만 포함하고, 함수의 구현은 소스 파일에 작성합니다.

(3) 헤더 가드 사용

- 헤더 파일에는 헤더 가드를 사용하여, 다중 포함을 방지합니다.
- 헤더 가드는 파일의 고유한 이름을 기반으로 작성하며, 대문자와 언더스코어를 사용합니다.

파일명 규칙	student_manager.h, student_manager.c

파일 분리 중
헤더 파일
student_manager.h
	#ifndef STUDENT_MANAGER_H
	#define STUDENT_MANAGER_H
	
	// 학생 정보를 관리하는 프로그램의 인터페이스
	
	// 학생 추가 함수
	void add_student(const char *name, int age);
	
	// 학생 수를 반환하는 함수
	int get_student_count(void);
	
	#endif // STUDENT_MANAGER_H

파일 분리 중
소스 파일
student_manager.c
	#include "student_manager.h"
	#include <stdio.h>
	#include <string.h>
	
	#define MAX_STUDENTS 100
	
	// 학생 구조체 정의
	typedef struct {
	  char name[50];
	  int age;
	} Student;
	
	// 학생 목록 배열
	static Student students[MAX_STUDENTS];
	static int student_count = 0;
	
	// 학생 추가 함수 구현
	void add_student(const char *name, int age) {
	  if (student_count < MAX_STUDENTS) {
	  strncpy(students[student_count].name, name,
	sizeof(students[student_count].name) - 1);
	  students[student_count].age = age;

	student_count++;
	} else {
	printf("학생 목록이 가득 찼습니다.\n");
	}
	}
	
	// 학생 수를 반환하는 함수 구현
	int get_student_count(void) {
	return student_count;
	}

헤더 가드 사용	#ifndef STUDENT_MANAGER_H
	#define STUDENT_MANAGER_H
	
	// 헤더 파일 내용
	void add_student(const char *name, int age);
	int get_student_count(void);
	
	#endif // STUDENT_MANAGER_H

헤더 가드 사용
(현대적인
코드베이스)
	#pragma once
	
	// 학생 정보를 관리하는 프로그램의 인터페이스
	
	// 학생 추가 함수
	void add_student(const char *name, int age);
	
	// 학생 수를 반환하는 함수
	int get_student_count(void);

8. 테스트 및 디버깅 (선택)

테스트와 디버깅 관련 규칙을 정의하여, 코드의 안정성과 오류 추적의 효율성을 높입니다.

각 함수에 대한 유닛 테스트를 작성하며, 테스트 파일은 별도의 디렉토리에 구성합니다.

(1) 테스트 코드 작성 규칙

- 유닛 테스트 작성: 모든 함수와 모듈에 대해 유닛 테스트를 작성하여, 개별 기능의 올바른 동작을 검증합니다.
- 테스트 코드의 파일 분리: 테스트 코드는 메인 코드와 분리하여 별도의 테스트 파일에 작성합니다. 파일명은 *_test.c 형식을 사용하여 테스트 파일임을 명확히 합니다.

- 예: student_manager_test.c
- 테스트 프레임워크 사용: CUnit, Check, Unity 와 같은 C 언어용 테스트 프레임워크를 사용하여 테스트 코드를 체계적으로 작성합니다.
- 자동화된 테스트 스크립트 작성: 빌드 프로세스에 테스트 실행을 통합하고, 모든 테스트가 통과할 때까지 코드를 병합하지 않습니다.
- 테스트 커버리지 목표 설정: 코드의 테스트 커버리지를 측정하고, 주요 기능에 대한 높은 커버리지를 유지합니다.

(2) 로그 및 디버그 출력 규칙

- 로그 레벨 설정: 로그 메시지의 중요도에 따라 로그 레벨을 설정합니다. 일반적으로 다음과 같은 레벨을 사용합니다.

→ DEBUG: 개발 시 디버그 정보를 출력하는 로그 레벨

→ INFO: 프로그램의 주요 상태 변경을 나타내는 로그 레벨

→ WARN: 경고 메시지, 예상치 않은 상황을 기록

→ ERROR: 프로그램 오류나 예외 상황을 기록

- 로그 출력 형식: 로그 메시지에는 타임스탬프, 로그 레벨, 메시지 내용을 포함하여 출력 형식을 일관되게 유지합니다.
- 예: [2024-10-23 14:55:12] INFO: 학생 목록이 갱신되었습니다.
- 표준 출력과 표준 오류 분리: 일반 로그 메시지는 표준 출력(stdout)에, 에러 로그는 표준 오류(stderr)에 출력합니다.
- 디버그 코드 관리: 디버그 코드는 #ifdef DEBUG 와 같은 조건부 컴파일을 통해 관리하여, 릴리즈 빌드에서는 제외할 수 있도록 합니다.

테스트 코드 작성 규칙	#include "student_manager.h"
	#include <assert.h>
	#include <stdio.h>
	
	void test_add_student() {
	  add_student("Alice", 20);
	  add_student("Bob", 22);
	
	  assert(get_student_count() == 2);
	  printf("test_add_student passed.\n");
	}
	
	int main() {

	test_add_student();
	printf("All tests passed.\n");
	return 0;
	}

로그 및 디버그 출력 규칙	#include <stdio.h>
	#include <time.h>
	
	void log_message(const char *level, const char *message) {
	time_t now = time(NULL);
	char timestamp[20];
	strftime(timestamp, sizeof(timestamp),
	"%Y-%m-%d %H:%M:%S", localtime(&now));
	printf("[%s] %s: %s\n", timestamp, level, message);
	}
	
	#define LOG_INFO(message) log_message("INFO", message)
	#define LOG_ERROR(message) log_message("ERROR", message)
	
	int main() {
	LOG_INFO("프로그램이 시작되었습니다.");
	LOG_ERROR("파일을 열 수 없습니다.");
	return 0;
	}

9. 에러 처리 (선택)

에러 처리 관련 규칙을 정의하여, 프로그램의 안정성을 높이고 예외 상황에 신속히 대응할 수 있도록 합니다.

함수가 에러를 반환할 경우, 반환값을 항상 검사하여 에러 처리를 합니다.

(1) 에러 코드 정의와 사용

- 에러 코드는 매크로 상수로 정의하여, 명확한 의미를 부여하고 코드의 가독성을 높입니다.
- 에러 코드는 ERROR_ 접두사를 사용하여 구분합니다.
- 에러 코드를 함수의 반환 값으로 사용하여, 함수 호출 결과를 검사하고 에러 처리를 수행합니다.
- 표준 에러 코드를 사용하지 않는 경우, 프로그램 전반에서 일관된 에러 코드를 정의합니다.

(2) 에러 메시지 작성 규칙

- 에러 메시지는 사용자 친화적인 언어로 작성하여, 문제의 원인과 해결 방법을 명확히 전달합니다.
- 에러 메시지에는 함수명, 에러 코드, 에러 설명 등을 포함하여, 문제를 쉽게 추적할 수 있도록 합니다.
- 예: "Error in add_student: Invalid input (ERROR_INVALID_INPUT)"
- 다국어 지원이 필요한 경우, 에러 메시지를 별도의 리소스 파일에 저장하고 로드하는 방식을 사용합니다.

(3) 예외 상황 처리 방식

- 에러가 발생한 경우, 적절한 복구 작업을 수행하거나 프로그램을 안전하게 종료합니다.
- 자원 해제: 예외 상황에서 메모리 할당, 파일 열기 등의 자원을 해제하여, 자원 누수를 방지합니다.
- 조건부 컴파일을 사용하여, 디버그 모드와 릴리즈 모드에서 다른 예외 처리를 수행할 수 있습니다. 디버그 모드에서는 자세한 디버그 정보를 출력하고, 릴리즈 모드에서는 사용자에게 친화적인 메시지를 출력합니다.
- 재시도 로직 구현: 네트워크 연결이나 파일 읽기 같은 일시적 오류의 경우, 재시도 로직을 구현할 수 있습니다.

에러 코드 정의와 사용	#define ERROR_INVALID_INPUT -1 #define ERROR_OUT_OF_MEMORY -2 #define ERROR_FILE_NOT_FOUND -3

에러 메시지 작성 규칙	#include <stdio.h>
	
	void print_error(const char *function_name, int error_code) {
	  printf("Error in %s: ", function_name);
	  switch (error_code) {
	  case ERROR_INVALID_INPUT:
	  printf("Invalid input (ERROR_INVALID_INPUT)\n");
	  break;
	  case ERROR_OUT_OF_MEMORY:
	  printf("Out of memory (ERROR_OUT_OF_MEMORY)\n");
	  break;

	case ERROR_FILE_NOT_FOUND:
	printf("File not found (ERROR_FILE_NOT_FOUND)\n");
	break;
	default:
	printf("Unknown error (code: %d)\n", error_code);
	}
	}

예외 상황 처리 방식	int read_file(const char *filename) {
	FILE *file = fopen(filename, "r");
	if (file == NULL) {
	print_error("read_file", ERROR_FILE_NOT_FOUND);
	return ERROR_FILE_NOT_FOUND;
	}
	
	// 파일 읽기 작업
	// ...
	
	// 파일 닫기
	fclose(file);
	return 0; // 성공 시 0 반환
	}

10. 프로그램 문서화 (선택)

프로그램의 문서화 규칙을 정의하여, 사용자와 개발자 모두가 코드를 더 쉽게 이해하고 사용할 수 있도록 합니다.

(1) 사용법 문서 작성 규칙

- 프로그램 사용법 문서는 README 파일이나 별도의 매뉴얼 파일에 작성하며, 프로그램의 주요 기능, 설치 방법, 실행 방법, 사용 예제 등을 포함합니다.
- 기본적인 프로그램 정보: 프로그램의 이름, 버전, 작성자, 라이선스 정보 등을 명시합니다.
- 설치 방법: 프로그램의 빌드 및 설치 방법을 단계별로 설명합니다.
- 명령줄 옵션 및 설정 파일 설명: 프로그램의 명령줄 옵션이나 설정 파일 형식을 설명하고, 각 옵션의 의미를 명확히 기술합니다.
- 사용 예제: 다양한 사용 사례에 대한 예제를 제공하여, 사용자에게 프로그램의 기능을 쉽게 이해할 수 있도록 합니다.