프로그래밍 언어론

<https://modoocode.com>

목차

[프로그래밍 언어론 1](#_Toc53507998)

[1. C 2](#_Toc53507999)

[1.1. Summary 2](#_Toc53508000)

[1.1.1. 증가, 감소 연산자 2](#_Toc53508001)

[1.1.2. 조건 만족 시 1, 만족하지 않으면 0 반환 2](#_Toc53508002)

[1.1.3. scanf 사용시 주의점 3](#_Toc53508003)

[1.1.4. 8진수, 16진수 3](#_Toc53508004)

[1.1.5. 부동 소수점 오차 3](#_Toc53508005)

[1.1.6. 비트연산 4](#_Toc53508006)

[1.1.7. 자료형 6](#_Toc53508007)

[1.1.8. 삼항 연산자 8](#_Toc53508008)

[1.1.9. goto 문 8](#_Toc53508009)

[1.1.10. static 변수 9](#_Toc53508010)

[1.1.11. Null 문자와 배열크기 (\0) 9](#_Toc53508011)

[1.1.12. 배열을 함수의 인자로 전달 받을 때 길이도 같이 받는 이유 9](#_Toc53508012)

[1.1.13. 재귀함수 (탈출조건 필요) 9](#_Toc53508013)

[1.2. ★★ 포인터 10](#_Toc53508014)

[1.2.1. 포인터 10](#_Toc53508015)

[1.2.2. 1차원 배열 11](#_Toc53508016)

[1.2.3. 1차원 포인터 배열 (포인터 변수로 이루어진 배열) 12](#_Toc53508017)

[1.2.4. 2차원 배열 13](#_Toc53508018)

[1.2.5. ★ 1차원 배열 vs 1차원 포인터 배열 vs 2차원 배열 vs 2차원 포인터 배열 14](#_Toc53508019)

[1.2.6. 포인터 연산 (++) 15](#_Toc53508020)

[1.2.7. Main 함수로의 인자 전달 (int argc, char\* argv[]) 18](#_Toc53508021)

[1.3. 입출력 19](#_Toc53508022)

[1.3.1. stdin stdout stderr 19](#_Toc53508023)

[1.3.2. EOF (End Of File) 19](#_Toc53508024)

[1.3.3. 입출력 함수 (stdio.h) 19](#_Toc53508025)

[1.3.4. 입출력 이외 문자열 함수 (string.h) 22](#_Toc53508026)

[1.3.5. 변환 함수 (stdlib.h) 23](#_Toc53508027)

[1.4. 파일 입출력 24](#_Toc53508028)

[1.4.1. 결제원 기출 2021 24](#_Toc53508029)

[1.4.2. 파일 입출력 함수 25](#_Toc53508030)

[1.5. 구조체 29](#_Toc53508031)

[1.5.1. 선언, 초기화, 접근 29](#_Toc53508032)

[1.5.2. 구조체 배열 30](#_Toc53508033)

[1.5.3. 구조체와 포인터 31](#_Toc53508034)

[1.5.4. 구조체의 주소값 = 구조체 첫 번째 멤버의 주소값 32](#_Toc53508035)

[1.5.5. typedef struct 33](#_Toc53508036)

[1.5.6. 구조체 연산 (복사, 크기 반환) 33](#_Toc53508037)

[1.5.7. 구조체 안의 구조체 (구조체 중첩) 34](#_Toc53508038)

[1.6. 공용체 (union) 35](#_Toc53508039)

[1.7. 열겨형 (enum) 36](#_Toc53508040)

[1.8. 메모리 구조와 동적할당 37](#_Toc53508041)

[1.8.1. 메모리 구조 37](#_Toc53508042)

[1.8.2. 동적할당 (malloc, calloc, realloc) 37](#_Toc53508043)

[1.9. 매크로와 선행처리기 38](#_Toc53508044)

[1.9.1. 매크로 상수 38](#_Toc53508045)

[1.9.2. 매크로 함수 38](#_Toc53508046)

[1.9.3. 매크로 장단점 38](#_Toc53508047)

[1.9.4. 조건부 코드 삽입을 위한 매크로 39](#_Toc53508048)

[1.9.5. 매개변수 치환 및 연결 (#, ##) 40](#_Toc53508049)

[1.10. 다중 소스 파일 40](#_Toc53508050)

# C

## Summary

### 증가, 감소 연산자

|  |
| --- |
| int num1 = 10;  int num2 = (num1--) + 2; // 연산 후 행이 넘어간 다음에 감소  printf("%d, %d", num1, num2); |
| num1 = 9  num2 = 12  소괄호의 영향 받지 않는다는 점에 주의 |

### 조건 만족 시 1, 만족하지 않으면 0 반환

cf. 0 이 아닌 모든 값을 True로 판정

|  |
| --- |
| int num1 = 10;  int num2 = 12;  int result1, result2, result3;  result1 = (num1 == num2);  result2 = (num1 > num2);  result3 = (num1 < num3);  printf("%d, %d, %d", result1, result2, result3); |
| 0, 1, 0 |

### scanf 사용시 주의점

<https://modoocode.com/32>

|  |  |
| --- | --- |
| 요약 | |
| %s, %d | %c 제외 데이터를 입력 받는 형식은 버퍼에 신경 쓸 필요 X |
| %c | 버퍼에 남아 있는 것 주의 |
| 내용 | |
| char c;  scanf("%c", &c); | - scanf 는 stdin 버퍼에서 한 문자만 읽어오므로, 버퍼에는 여전히 다른 문자들이 남아있음 (개행 문자 포함)  - 만약 해당 버퍼가 fflush(stdin) 되지 않는다면 다음 scanf 문장에서는 이미 버퍼에 문자들이 있으므로 따로 입력을 받지 않고 넘어가게됨. |
| int i;  scanf("%d", &i);  char arr[20];  scanf("%s", arr); | stdin 버퍼에서 숫자가 아닌 문자나 공백, 개행 문자를 만날 때까지 읽음  stdin 버퍼에서 처음에 있는 공백, 개행 문자들을 버리고 문자열을 읽음 |
| 해결방법 | |
| getchar(); | stdin에서 한 문자를 읽어옴(빼옴). 그러나, stdin 버퍼에 다른 문자 남아있을 경우 해결 불가 |
| fflush(stdin); | stdin 자체를 비움. 초기화. 가장 좋은 해결방법  윈도우 컴파일러 지원, But 다른 계열 컴파일러는 아님. |

### 8진수, 16진수

|  |
| --- |
| int num1 = 10;  int num2 = 0xA; // 16 진수 표현  int num3 = 012; // **0으로 시작하면 8진수 octet** |

### 부동 소수점 오차

|  |
| --- |
| float num = 0.0;  for(int i=0; i<100; i++)  num += 0.1;  printf("%f", num) |
| 10.000002  // 소수 표현시 2의 지수승 형태로 표현하므로 오차 필연적으로 생김 |

### 비트연산

|  |
| --- |
| AND |
| int num1 = 15;  int num2 = 20;  int num3 = num1 & num2; // && 는 조건문 판단 시, 비트 연산은 &  printf("%d, num3"); |
| num1 = 00000000 00000000 00000000 00001111  num2 = 00000000 00000000 00000000 00010100  num3 = 00000000 00000000 00000000 00000100  출력 : 4 |
| OR |
| int num1 = 15;  int num2 = 20;  int num3 = num1 | num2; // || 는 조건문 판단 시, 비트 연산은 |  printf("%d, num3"); |
| num1 = 00000000 00000000 00000000 00001111  num2 = 00000000 00000000 00000000 00010100  num3 = 00000000 00000000 00000000 00011111  출력 : 31 |
| NOT 연산 |
| int num1 = 15;  int num3 = **~** num1; // 1을 0으로 0을 1로 변환  printf("%d, num3"); |
| 출력 : -16  num1 = 00000000 00000000 00000000 00001111  num3 = 11111111 11111111 11111111 11110000 (**맨 앞 1비트는 부호비트**)  ~~-15~~ 가 아닌 -16이 출력되는 것에 주의  num3(음수) 2의 보수법 통해 양수로 변환 후 앞에 -를 붙여준다.  2의 보수법은 비트를 반전시킨 후 1을 더해야함 |

|  |
| --- |
| Shift 연산 (<<, >>) |
| int num = 15;  int result1 = num << 1; // 왼쪽으로 1칸 이동  int result2 = num << 2; // 왼쪽으로 2칸 이동  int result3 = num >> 2; // 오른쪽으로 2칸 이동  printf("%d, result1");  printf("%d, result2");  printf("%d, result3"); |
| num = 00000000 00000000 00000000 00001111  result1 = 00000000 00000000 00000000 00011110  result2 = 00000000 00000000 00000000 00111100  result3 = 00000000 00000000 00000000 00000011  출력 : 30  출력 : 60  출력 : 3  즉, 왼쪽으로 1칸 이동시마다 정수의 값은 2배  오른쪽으로 1칸 이동시마다 정수의 값은 1/2배 |
| 음수의 오른쪽 Shift 연산 |
| int num3 = -16 // 11111111 11111111 11111111 11110000;  int result = num3 >> 2; |
| CPU에 따라 결과값이 달라진다.  - 부호비트 유지 시스템 : 왼쪽을 1로 채움  - 부호비트 유지하지 않는 시스템 : 왼쪽을 0으로 채움  - 11111111 11111111 11111111 11111100 // -4 (합리적 결과)  - 00111111 11111111 11111111 11111100  부호비트를 유지하는 시스템에서는 오른쪽으로 이동 시 1/2배 및 - 음수값이 유지된다. |

### 자료형

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 구분 | 자료형 | 크기  (바이트) | 표현 |  |
| 문자형  (사실 정수형의 일부) | char | 1 | %c  %s | 연산 시 특이사항 아래 참조  문자 A의 아스키 코드 값 = **65** |
| 정수형 | short | 2 |  |  |
| int | 4 | %d | %o : 8진수  %x : 16진수 |
| long | 4 | %ld |  |
| long long | 8 | %lld |  |
| 실수형 | float | 4 | %f |  |
| double | 8 |  | 출력 시 %f, But, 입력 시 %lf |
| long double | 8바이트 이상 |  |  |
| 포인터 주소값 |  |  | %p |  |

|  |
| --- |
| char num1 = 1, num2 = 2;  printf("%d", sizeof(num1+num2));  printf("%d", sizeof('A')); |
| char 연산의 자료형 반환이 4 바이트로 나옴 |

#### unsinged

|  |
| --- |
| 0 이상의 양수에 대해서만 unsigned 지정 가능  0이상에서의 표현 범위 2배 증가 |

#### 형 변환

|  |
| --- |
| int num1 = 3, num2 = 4;  double divResult;  divResult = num1 / num2; // 0.000000  divResult = **(double)**num1 / num2; // 0.75  printf("%f", divResult) |

#### 서식문자 #

|  |
| --- |
| int num1 = 7, num2 =13;  printf("%o, %#o", num1, num1);  printf("%x, %#x", num2, num2); |
| 7 07  d 0xd  출력 시 몇진법인지 같이 출력해준다. |

#### 필드 폭 정렬 (오른쪽 , 왼쪽 - )

|  |
| --- |
| %8d : 필드 폭을 8칸 확보하고, 오른쪽 정렬  %-8d : 필드 폭을 8칸 확복하고, 왼쪽 정렬 |

#### const (컴파일 에러를 통해 실수를 줄이자)

- 일반적인 상수 선언 const float PI = 3.14

- 포인터 const 선언

|  |
| --- |
| void showAllData(const int \* arr, int len){  int i ;  for (i =0; i<len, i++)  printf("%d", arr[i]);  } |
| 위 함수의 매개변수에서 매개변수 arr을 대상으로 const 선언을 한 이유는?  showAllData 함수 내에서 arr 값을 변경하지 않겠다는 의미 🡪 실수 방지용  만약, 프로그래머가 실수로 arr 값을 변경하면 컴파일 에러 발생 |

### 삼항 연산자

|  |
| --- |
| int num3 = (num1 > num2) ? num1 : num2 ; |
| num1>num2 일 경우 num3 에는 num1 이 들어감  num1>num2 아닐 경우 num3 에는 num2가 들어감  (조건) ? data1 : data2 |

### goto 문

|  |
| --- |
| if(num==1)  goto ONE;  else  goto TTTT;  ONE :  printf("~~~");  TTTT :  return 0; |

Switch 문

|  |
| --- |
| Switch(num){  case 1:  ~~~  break;  case 2;  ~~~~~~  break;  default:  ~~~~  } |
| case n : 별로 break;  default 포함 |

### static 변수

지역변수에 static 선언이 붙게 될 경우

1. 선언된 함수 내에서만 접근 (지역변수 특성)

2. 1회 초기화되고 프로그램 종료 시까지 메모리 공간에 적재 (전역변수 특성)

### Null 문자와 배열크기 (\0)

모든 문자열의 끝에는 기본적으로 Null 문자가 자동으로 들어감.

**배열크기 계산 시 주의!!!!!!!!**

**(단, strlen 함수는 순수하게 문자 개수만을 반환한다.)**

### 배열을 함수의 인자로 전달 받을 때 길이도 같이 받는 이유

함수 내에서는 인자로 전달된 배열의 길이를 계산할 수 없다.

인자로 전달된 배열 포인터에 sizeof 연산을 취할 경우 배열 크기가 아닌 포인터의 크기가 반환

### 재귀함수 (탈출조건 필요)

금융결제원 필기 2021

재귀함수 🡪 탈출 조건이 있어야 함.

|  |
| --- |
| // n개의 원반을 from 에서 by를 거쳐 to로 옮긴다.  void hanoi(int n, int from, int by, int to){  if (n == 1)  move(from, to);  else{  hanoi(n - 1, from, to, by); // 1번 N-1개의 원반을 기둥3을 거쳐 2로 옮긴다.  move(from, to); // 2번 기둥 1에서 1개의 원반을 기둥 3으롱 롬긴다.  hanoi(n - 1, by, from, to); // 3번 기둥 2에서 N-1개의 원반을 기둥 3으로 옮긴다.  }  } |

## ★★ 포인터

### 포인터

#### 정수형

|  |
| --- |
| int\* ptr;  \*ptr = 20;  printf("%d", \*ptr); |
| **포인터 변수는 0, NULL, &p 주소값을 통해서만 초기화 될 수 있다.**  위와 같이 곧바로 값을 넣게되면, 어떤 주소를 가르키는지 모르는 상태에서 가리키는 곳의 값을 20으로 초기화하라는 말과 같기 때문에 오류가 발생한다. |

#### 문자열 (상수 형태의 문자열, 고정 형태의 문자열)

|  |
| --- |
| **char str1[]** = "variable string"; // **변수 형태**의 문자열  **char\* str2** = "constant string"; // **상수 형태**의 문자열 (고정형)  고정형 문자열은 문자열 일부의 변경이 불가능한 반면, 변수형 문자열은 자유롭게 변경 가능  str1[10] = 'k' |
| char str1[] = "variable string";  const char\* str2 = "constant string"; // Visual Studio 에서는 이렇게 변경된다. |

#### 함수형 포인터

|  |
| --- |
| int SoSimple(int num1, int num2){….}  **int (\*fptr) (int, int)**  fptr = SoSimple;  fptr (3,3) |
| 1. 반환형  2. 매개변수  위 두가지가 일치하면 함수형 포인터 선언 가능 |

### 1차원 배열

|  |
| --- |
| int a[3] = {5, 6, 7};  printf("%d, %d, %d\n", \*(&a[0]+2), \*(a + 2), \*a + 10); |
| 7, 7, 15  \*(&a[0]+2), \*(a + 2) 🡪 a[1] 을 가리킨다.  \*a + 10 🡪 \*a 의 값에 10을 더한다. |

|  |
| --- |
| char b[3] = {'a', 'b', 'c'};  printf("%c, %c, %c\n", \*b, \*(b + 2), \*b + 5);  printf("%c, %c, %c\n", b[1], \*(&b[1] + 1), &b[1] + 1); |
| a, c, f  b, c, [이상한 문자]  \*b + 5 🡪 b[0] 의 값에 5를 더한다. a b c d e f , a로부터 5번째 위치까지 떨어져 있는 값  마지막 행은 마지막 원소값은 b[2] 의 주소값이 저장 |

|  |  |  |  |
| --- | --- | --- | --- |
| 주소값 표현 | **arr**  **&arr[0]** | **arr+1**  **&arr[1]** | **arr+2**  **&arr[2]** |
| 값 표현 | **\*arr** | **\*(arr+1)**  **\*(&arr[0] + 1)** | **\*(arr+2)**  **\*(&arr[1] + 1)**  **\*(&arr[0] + 2)** |
| int arr[3] | arr[0] | arr[1] | arr[2] |

주소값 표현끼리는 원소이동이 간단함

**단, \*(arr + 2) 과 \*arr + 2이 의미하는 바는 다르다.**

a[3] 출력과 a[0]의 값에 2을 더하느냐의 차이

\* 포인터가 일반 연산자에 비해 앞선다.

### 1차원 포인터 배열 (포인터 변수로 이루어진 배열)

#### 금융결제원 기출 2021 (필기)

|  |
| --- |
| const char\* arr[3] = {"apple","banana","grape"};  printf("%s\n", arr); // 1차원 배열과 달리 아무값도 나오지 않음, 이중 포인터이기 때문  printf("%s\n", arr[0]);  printf("%s\n", arr[1]);  printf("%c\n", \*\*arr ) // apple의 1번째 문자  printf("%c\n", \*(\*arr + 2)); // apple의 3번째 문자  printf("%c\n", \*(arr[1]+2)); // banana의 3번째 문자  printf("%c\n", \*(\*(arr + 2) + 1)); // grape의 2번째 문자 |
| arr 은 여기서 이중 포인터  arr[0], arr[1], arr[2] 에는 각각 문자열의 주소값이 저장된다.  출력 값 : p, n, r  주의! \* 적용이 일반 +,- 연산보다 우선해서 적용된다. |

#### 포인터 배열의 포인터 선언 (더블 포인터, 이중 포인터)

|  |
| --- |
| int num1=10, num2=20, num3=30;  int \*ptr1 = &num1;  int \*ptr2 = &num2;  int \*ptr3 = &num3;  **int \* ptrArr[]** = {ptr1, ptr2, ptr3};  **int \*\* dptr = ptrArr**; |
| dptr[0] = ptrArr[0]  dptr[1] = ptrArr[1]  dptr[2] = ptrArr[2] |

### 2차원 배열

#### 표현

포인터 배열과 원리는 동일함

|  |
| --- |
| int a[3][2] = {{1, 2}, {3, 4}, {5, 6}};  printf("%d", ( \*(a+2) )[1] );  printf("%d", \*(a[2]+1) );  printf("%d", \*(\*(a+2)+1) ); |
| 셋 다 전부 a[2][1] 을 가리킴  치환 원리 arr[i] = \*(arr+i) 공식을 가지고 치환하면 동일함을 증명할 수 있음 |

#### 2차원 배열 포인터 선언

|  |
| --- |
| int arr[3][2] = {{1, 2}, {3, 4}, {5, 6}};  int arr[][2] = {{1, 2}, {3, 4}, {5, 6}}; // **행 개수는 자동 결정될 수 있으므로 생략 가능**  **int (\*ptr) [2]; //** 2차원 배열에서의 포인터 선언. 2차원 배열 자체를 가리킴  ptr = arr; |

#### ★ 포인터 배열 vs **2차원 배열 포인터** (**괄호 유무**)

|  |
| --- |
| **괄호** 유무의 차이에 따른 구분  int num1 = 10, num2 =20, num3 =30;  int arr2d[2][4] = {1, 2, 3, 4, 5, 6, 7, 8};  int \* ptrA [3]; // 포인터 배열, **포인터 값(주소)를 4개 가지는 배열**  int (\*ptrB) [4]; // **2차원 배열을 가리키는 이중 포인터**  \* ptrA [3] = {&num1, &num2, &num3};  (\*ptrB) [4] = arr2d; |

### ★ 1차원 배열 vs 1차원 포인터 배열 vs 2차원 배열 vs 2차원 포인터 배열

|  |  |  |
| --- | --- | --- |
| 분류 | 배열 선언 | 배열 포인터 선언  (해당 배열에 접근하여 연산하기 위한 포인터) |
| 1차원 배열 | int arr[3]; | int \*ptr = arr; |
| 1차원 포인터 배열 | int\* arr[3]; | int \*\*ptr = arr; |
| 2차원 배열 | int arr[2][4]; | int (\*ptr) [4] = arr; |
| 2차원 포인터 배열 | int\* arr[2][4]; | int \*(\*ptr) [4] = arr; |

### 포인터 연산 (++)

"int 형 포인터를 대상으로 1을 증가시키면 4가 증가하고, double 형 포인터를 대상으로 1을 증가시키면 8이 증가한다."

**즉, 포인터 변수의 연산은 size of (data type) 단위로 이루어진다.**

|  |  |
| --- | --- |
| int arr[3] = {10, 14 ,30};  int \*ptr = arr;  \*(++ptr) = 20; // 선위증가, \*++ptr과 동일  \*(ptr + 1) = 20; // 포인터 값 자체는 변화하지 않음  \*(ptr++) = 20; // 후위증가, \*ptr++과 동일  (\*ptr)++ ; // ptr이 가리키는 값을 증가 | |
| 공통점 | ptr 에서 주소값의 크기가 4만큼 떨어진 공간의 값을 20으로 변경  arr[1] = 20 이 됨 |
| 차이점 | ++ptr = 20의 경우 ptr 주소값 자체가 4 증가하여 가리키는 곳이 arr[1] 임  \*(ptr + 1) = 20 의 경우 연산 종료 후  \*(ptr++) = 20의 경우 a[0] = 20 으로 설정 후, ptr 이 가리키는 곳이 arr[1] 임  연산 종료 후 ptr 이 가리키는 값의 차이 |
| |  |  |  | | --- | --- | --- | | 분류 | 변경되는 값 | 연산 후 포인터가 가리키는 곳 | | \*(++ptr) = 20; | arr[1] = 20 | arr[1] | | \*(ptr + 1) = 20; | arr[2] = 20 | arr[1] | | \*(ptr++) = 20; | arr[1] = 20 | arr[2] | | (\*ptr)++; | arr[2] = arr[2] + 1 | arr[2] | | |

#### 금융결제원 기출 2021 (서술형)

|  |
| --- |
| int a[3] = {5, 6, 7};  printf("%d, %d", a[0], \*(a+1));  \*a++ = 8;  \*++a = 9;  printf("%d", a[0]+a[1]+a[2]); |
| 컴파일 에러가 발생한다.  **배열의 이름도 포인터이긴 하지만, 상수이기 때문에 연산은 불가능하다.**  **다음에서 a 주소값의 변경이 불가능하기 때문이다.**  \*a++  \*++a |

#### 예제

|  |
| --- |
| int arr[5]={1,2,3,4,5};  int \*ptr = &arr[4];  int total = 0, i;  for (i=0; i<5; i++)  total += \*(ptr--);  printf("%d", total); |
| total = total + \*(ptr)  ptr = ptr - 1 // 후위감소  출력 : 15 |

#### 기출 변형

|  |
| --- |
| int a[3] = {5, 10, 20};  printf("%d, %d", a[0], \*(a+1));  pnum = a;  \*pnum++ = 8; // pnum이 가리키는 곳에 8을 넣고 pnum 증가  \*++pnum = 14; // pnum 포인터 증가 후, 14 대입  printf("%d", a[0], \*(a + 1), \*(&a[0]+2) );  (\*pnum)++; // pnum 이 가리키는 곳의 값 ++  printf("%d", a[0], \*(a + 1), \*(&a[0]+2) ); |
| [8, 10, 14]  [8, 10, 15]  사실 pnum 포인터는 int 형 포인터 변수이므로 pnum = pnum + 1을 할 경우 pnum 은 4가 증가한다. int 배열의 경우 주소값이 4씩 차이가 나기 때문에 결과적으로 원소 개수를 1개 증가하여 가리키는 효과가 나는 것.  **++이 pnum(포인터 변수)에 적용되는지, \*pnum(가리키는 값)에 적용되는지 구분**   |  |  | | --- | --- | | **\*pnum++ = 8;** | \*pnum = 8;  pnum = pnum + 1; | | **\*++pnum = 14;** | pnum = pnum + 1;  \*pnum = 14; | | **(\*pnum)++;** | \*pnum = \*pnum + 1 | |

#### void 형 포인터

void 형 포인터로는 어떠한 포인터 연산도 할 수 없다.

|  |
| --- |
| int num=20;  void \*ptr = &num;  \*ptr = 20; // 컴파일 에러!!  ptr++; // 컴파일 에러!! |

### Main 함수로의 인자 전달 (int argc, char\* argv[])

금감원 2020 기출

|  |
| --- |
| int main(int argc, char\* argv[]) {  printf("argc : %d\n", argc);  for (int i = 0; i < argc; i++)  printf("argv[%d] : %s\n", i, \*(argv + i));  printf("argv[%d] : %s\n", i, argv[i]);  // **문자열 %s 출력 시 문자열의 주소를 전달**한다.  // **만약, %c 로 한 문자씩 출력할 경우 이중 포인터 사용**해야함 |
|  |

## 입출력

### stdin stdout stderr

콘솔 입출력을 위한 데이터 스트림

File \*Stream 파일 포인터 자리에 대체 가능

|  |  |  |
| --- | --- | --- |
| stdin | 표준 입력 스트림 | 키보드 대상 입력 |
| stdout | 표준 출력 스트림 | 모니터 대상 출력 |
| stderr | 표준 에러 스트림 | 모니터 대상 출력 |

### EOF (End Of File)

|  |  |
| --- | --- |
| 키보드 대상 입출력 | 함수호출의 실패, Windows Ctrl + Z, Linux Ctrl + D |
| 파일 입출력 | 파일의 끝 |
| 값 | -1 |

### 입출력 함수 (stdio.h)

#### 문자 (put char, get char)

|  |  |  |
| --- | --- | --- |
| 종류 | 설명 | 내용 |
| putchar | put char | putchar(char) |
| fputc | file put char | fputchar(char, File\* Stream) |
| getchar | get char | getchar() |
| fgetc | file get char | fgetc(File\* Stream) |

#### 문자열 (put string, get string)

|  |  |  |
| --- | --- | --- |
| puts | put string | puts(char \*string) |
| fputs | file put string | fputs(char \*string, File\* Stream) |
| gets | get string  오버플로우 가능 | gets(char \*string)  char str[7];  **gets(str); // 표준 입력 버퍼로부터 받은 문자열을 str 저장** |
| fgets | file get string  개수 지정  null 문자 자동 포함하여 입력  🡪 배열 크기 계산 시 주의  개행 문자 포함하여 저장 🡪 개행 두번 이루어짐 | fgets(char \*string, **int n**, File\* Stream)  // 스트링은 문자열 주소를 매개로 이루어지며  // n 개의 문자를 입력받겠다고 선언  char str[7];  fgets(str, sizeof(str), sdtin);  puts(str);  입력 : 123456789  출력 : 123456  배열 : 123456 **\0**  Q. size 가 7 인데 6개의 문자만 입력되는 이유?  문자열 입력 시에는 자동으로 Null (\0) 문자가 추가된다.  문자와 문자열의 차이는 Null 문자 핵심  char str[25];  fgets(str, sizeof(str), sdtin);  puts(str); //출력 시 자동 개행  입력 : 123456789 Enter  배열 : 123456789 \n **\0**  출력 : 123456789 \n \n  Q. 개행이 두번 이루어지는 이유?  gets 함수는 \n (개행)을 만날 때까지 입력을 받고,  \n(개행)을 포함하여 저장한다.  따라서, 출력 시에는 출력완료 후 자동추가되는 개행문자를  포함하여 두 번 개행되는 것 |

#### 입력 버퍼 플러시 vs 출력 버퍼 플러시

|  |  |  |
| --- | --- | --- |
| 출력 버퍼  플러시 | fflush(stdout) | 출력 버퍼의 내용을 비우고,  해당 내용을 목적지로 전송한다. |
| 입력 버퍼  플러시 | fflush(stdin) | 단순 소멸 |

#### 입력 버퍼 잔여 데이터 비우기

scanf, fgets, gets, fgetc, getchar 등을 통해 데이터를 입력 받고 난 이후, 입력 버퍼에 데이터가 남아있을 경우 다음 입력 함수 실행 시 데이터 입력을 받지 않고 (입력 함수 실행을 건너뛰고) 진행된다. 🡪 다음 입력 함수 실행 전 입력 버퍼 플러시가 필요함

getchar() // 마지막에 입력된 \n 문자 입력 버퍼에서 빼오기

fflush(stdin) // 입력 버퍼 자체를 소멸

void clearLineReadBuffer(void){

while (getchar() != '\n' )

}

### 입출력 이외 문자열 함수 (string.h)

|  |  |  |
| --- | --- | --- |
| strlen | string length  null 제외 문자열 길이 | char str[] = "1234567";  int length = strlen(str);  **Null 제외**한 문자열의 길이 출력  그러나, \n 개행문자는 포함한다.  char str[100];  fgets(str, sizeof(str), stdin); // 배열크기만큼 입력  printf("길이 : %d, 내용 : %s", strlen(str), str);  // 2번 개행 이루어짐  **RemoveBSN**(str); // 개행 문자 \n 제거  fgets(str, sizeof(str), stdin); // 배열크기만큼 입력  printf("길이 : %d, 내용 : %s", strlen(str), str);  // 1번 만 개행 이루어짐  // printf는 출력 후 기본적으로 개행 진행 |
| strcpy | string copy  목적 🡨 소스  오버플로우 가능 | char str1[30]="Simple String";  char str2[30];  strcpy(str2, str1); |
| strncpy | string n copy  개수 지정하여 입력  단, null 문자가 포함되지 않으면 오류 발생 | strcpy(str2, str1, sizeof(str2));  그러나, 이 경우 오버플로우 가능성은 제거하였지만,  null 문자를 포함하지 않고 복사를 진행할 경우 출력 결과가 이상해질 수 있음  strcpy(str2, str1, sizeof(str2) - 1); // 마지막 빼고 입력  str2[sizeof(str)-1] = 0 // 배열 마지막에 null 문자 입력 |

|  |  |  |
| --- | --- | --- |
| strcat | string concatenate  오버플로우 발생 가능  str1 의 null 문자부터 덧붙임 | strncat(str1, str2, n)  최대 n개까지 덧붙임  char str1[20] = "First~"  char str2[20] = "Second"  char str3[20] = "Simple Num: "  char str4[20] = "1234567890"  strcat(str1, str2)  puts(str1)  strncat(str1, str2, 7)  puts(str3)  출력결과  First~Second  Simple Num: 1234567 |
| strncat | string n concatenate  strncpy와 달리 strncat 함수는 문자열의 끝에 자동적으로 null 문자를 삽입한다. |
| strcmp | string compare | 문자열이 같으면 0,  str1 > str2 이면 양수반환  str1 < str2 이면 음수반환  strcmp("ABCD", "ABCDE")  첫 번째 문자열의 5번째 문자는 Null 문자 (0)  두 번째 문자열의 5번째 문자는 'E'  0 < E 🡪 음수반환  strncmp("ABCD", "ABCDE", 3)  앞의 세글자는 같으므로 0 반환 |
| strncmp | string n compare  문자열의 크고 작음은 아스키코드 값 기준 |

### 변환 함수 (stdlib.h)

|  |  |  |
| --- | --- | --- |
| atoi | array to int | ator(str) |
| atol | array to long | atol(str) |
| atof | array to float | atof(str) |

## 파일 입출력

|  |
| --- |
| #include <stdio.h>  int main() {  char str[80];  float f;  FILE\* pFile;  pFile = fopen("myfile.txt", "w+");  fprintf(pFile, "%f %s", 3.1416, "PI");  rewind(pFile);  fscanf(pFile, "%f", &f);  fscanf(pFile, "%s", str);  fclose(pFile);  printf("I have read: %f and %s \n", f, str);  return 0;  } |
| myfile.txt 에 3.1416 과 PI 를 쓴 뒤, 이를 다시 myfile.txt 에서 읽어들인다. 이  때, myfile.txt 를 w+ 형식으로 열었으므로 출력 뒤에 입력을 하기 위해서는 fflush  나 rewind 와 같은 파일 위치 지정자 조정 함수를 호출해야 하는데 이 경우 rewind  함수를 호출하여 위치 표시자를 맨 앞으로 돌렸다. |

### 결제원 기출 2021

|  |
| --- |
| FILE\* pFile;  float f;  char\* str;  ① pFile = fopen("text.txt", "rb");  ② fscanf("%f", &f);  ③ fscanf("%s", str);  ④ printf("%f %s", &f, str);  ⑤ fclose("texst.txt");  위 5개 중 틀린 것은? |
| 정답. 2, 3, 5.  파일 포인터 선언 안함 |

### 파일 입출력 함수

#### open close

|  |  |
| --- | --- |
| 파일 포인터 열고 닫기 | |
| fopen | FILE\* pFILE = fopen("test.txt", "wb");  **+** 는 읽기, 쓰기가 모두 가능한 스트림의 형성을 의미한다.   |  |  | | --- | --- | | r | 읽기 전용. 파일 없을 시 에러 | | w | 쓰기용으로 파일생성, 동일이름이 있다면 덮어씀 | | a | 동일명 파일 존재시 끝에 이어씀 (append)  동일명 파일 없으면 쓰기용 파일 생성 | | r+ | 현재 있는 파일에 read/write 모두 덮어씀 | | w+ | 읽기/쓰기용으로 파일 생성. 같은 파일 존재시 덮어 | | a+ | 파일이 있으면 뒤에 추가, 없다면 새로 생성 | | t | Text Mode (텍스트 모드) | | b | Binary Mode (이진 파일 모드) | |
| fclose | fclose(pFile); |

#### write

|  |  |
| --- | --- |
| 쓰기 작업 | |
| fprintf | 특정한 형식으로 쓰기  fprintf(**pFile**, "%f %s", 3.1416, "PI");  // printf 하고 다른 점은 file pointer를 인자로 전달 |
| fwrite | 데이터 블록 쓰기  char buffer[] = {'x', 'y', 'z'};  **fwrite(buffer, size, count, pFile);**  size \* count 만큼 버퍼에서 읽어서 파일 스트림에 저장 |
| fputc | 스트림에 한 문자를 쓰기  char c;  **fputc(c, pFile);** |

#### read

|  |  |
| --- | --- |
| 읽기 작업 | |
| fscanf | 특정한 형식으로 읽기  fscanf(pFile, "%f", &f);  fscanf(pFile, "%s", str);  **// scanf 하고 다른 점은 file pointer를 인자로 전달** |
| fread | 스트림에서 데이터 블록을 읽기 (버퍼 자리에는 구조체도 대체 가능)  pFile 파일 포인터가 가리키는 스트림에서 size \* count 만큼 읽어서 버퍼에 저장  char\* buffer;  fread(buffer, size, count, pFile);  fread(buffer, 1, 100, pFile); // 1바이트씩 100개 총 100바이트 읽어오기 |
| fgets | 스트림에서 문자열 읽기  file get string  char sentence[256];  fgets(sentence, 255, stdin); // 읽어올 크기 지정  **scanf 는 문자열 읽기에 적합하지 않음. (공백에서 끊어 읽으므로)**  **But, fgets의 경우 읽어올 크기를 지정함으로써 문자열 입력에 적합** |
| fgetc | 스트림에서 문자 하나 읽기  file get char  char c;  c = fgetc(pFile); |

#### file pointer

|  |  |
| --- | --- |
| 파일 포인터 이동 | |
| fseek | 파일 포인터 이동 (fp, offset, 기준점)  int fseek(FILE\* stream, long **int offset**, int origin);  **SEEK\_SET 파일의 시작 (Settle)**  **SEEK\_CUR 현재 파일 포인터의 위치 (Current)**  **SEEK\_END 파일의 끝**  FILE\* pFile;  pFile = fopen("example.txt", "w");  fputs("This is an apple.", pFile);  **fseek(pFile, 9, SEEK\_SET);**  fputs(" sam", pFile);  fclose(pFile);   |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | T | h | i | s | - | i | s | - | a | n | - | a | p | p | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |  |  |  |  |  |  |  |  |  | - | s | a | m |  |   출력결과 : This is a sample. |
| rewind | 스트림의 위치 지정자를 맨 처음으로 설정한다.  **rewind(pFile)**; // 초기화  fseek(stream, 0, SEEK\_SET); 와 동일 |
| ftell | 현재 위치 offset 반환  fseek(pFile, 0, SEEK\_END);  **offset = ftell(pFile);**  fclose(pFile);  printf("Size of myfile.txt: %ld bytes.\n", size); |
| feof | File \*stream = fopen("test.txt", 'r+');  ……  feof(stream) 🡪 0 // 파일의 끝이 아니라면 False  feof(stream) 🡪 0 이 아닌 값 // 파일의 끝이라면 True |

#### fflush

|  |
| --- |
| FILE \*fp = fopen("text.txt", "wt");  …..  fflush(fp); // 출력 버퍼에 있는 내용을 목적지로 전송한다. |
| test.txt에 출력 버퍼에 있던 내용이 저장된다.  입력버퍼를 비우는 것과 출력버퍼를 비우는 것은 차이가 있음에 주의할 것  파일 출력버퍼에 대한 fflush는 가능하지만,  파일 입력버퍼에 대한 fflush 호출은 불가능하다.  stdout 출력버퍼에 대한 fflush (목적지 전송 및 저장),  sdtin 입력버퍼에 대한 fflush (소멸), 단, 윈도우 계열의 컴파일러에서만 지원 |

## 구조체

### 선언, 초기화, 접근

|  |
| --- |
| // 선언  struct person{  char name[20];  char phoneNum[20];  int age;  };  // 초기화 시에는 단순히 데이터의 나열로 초기화 가능  // 초기화 되지 않은 멤버는 0 으로 초기화  struct person man = {"이승기", "010-0000-0000", 21};  // 구조체 멤버 접근  printf("%s, %s, %d", man.name, man,phoneNum, man,age);  // 초기화 이후 문자열 데이터 변경 시 strcpy 함수 이용  // 일반 정수형은 바로 접근 가능  struct person man2;  strcpy(man2.name, "홍길동");  strcpy(man2.phoneNum, "010-0000-1111");  man2.age = 30;  // 입력받기 scanf  scanf("%s", man2.name);  scanf("%d", man2.age); |

### 구조체 배열

|  |
| --- |
| // 구조체 배열 선언 및 접근  struct point{  int xpos;  int ypos;  }  int main(){  struct point arr[3]; // 구조체 배열 선언  scanf("%d, %d", &arr[0].xpos, &arr[0].ypos); // arr[0] 입력  scanf("%d, %d", (arr+1).xpos, (arr+1).ypos); // arr[1] 입력 받기  printf("%d, %d", arr[0].xpos, arr[0].ypos); // arr[0] 출력  printf("%d, %d",\*(arr+1).xpos, \*(arr+1).ypos ); // arr[1] 출력  return 0;  } |
| // 구조체 배열 초기화  struct person{  char name[20];  char phoneNum[20];  int age;  };  struct person arr[2] = {  {"이승기", "010-000-0000", 21},  {"홍길동", "010-111-1111", 23}  } |

### 구조체와 포인터

#### 포인터 접근 표기법

|  |
| --- |
| struct point{  int xpos;  int ypos;  }  int main(){  struct point pos1 = {1, 2};  struct point \*pptr = &pos1; // 구조체 포인터 선언 및 초기화  (\*pptr).xpos += 4;  (\*pptr).ypos += 5; // {5, 7}  pptr 🡪 xpos++;  pptr 🡪 ypos++; // {6, 8}  } |
| pprt 🡪 xpos  (\*pprt).xpos  위 두 표현 방식은 동일한 표현 방법이다. |

#### 포인터 변수를 멤버로 가질 경우

|  |
| --- |
| struct point{  int xpos;  int ypos;  }  struct circle{  double radius;  struct point \* center;  }  int main(){  struct point cen = {2, 7};  double rad = 5.5;  struct circle ring = {rad, &cen};  printf("원의 중심의 좌표 : (%d, %d)", ring.center🡪xpos, ring.center🡪ypos);  } |

#### 연결 리스트 구현

|  |
| --- |
| struct node{  int data;  struct node \*ptr;  }  int main(){  struct node node1 = {10}; // 초기화 시 recursive 포인터 제외하고 초기화 가능  struct node node2 = {20};  struct node node3 = {30};  node1.ptr = &node2; // Linking 작업  node2.ptr = &node3;  printf("node2 데이터 출력 : %d", node1.ptr🡪data);  // 만약 node3 데이터 출력을 원하면 ptr를 계속해서 따라가는 작업이 필요  } |

### 구조체의 주소값 = 구조체 첫 번째 멤버의 주소값

|  |
| --- |
| struct point{  int xpos;  int ypos;  }  int main(){  struct point pos = {10, 20};  printf("주소값 확인 **%p, %p", &pos, &pos.ypos); // 결과 동일함**  } |

### typedef struct

|  |
| --- |
| 구조체 변수를 선언할 때마다 struct 선언을 하는 것이 귀찮으므로,  typedef 선언을 하면 대표자 정의자로 편리하게 선언 가능 |
| 아래 표기법 모두 동일   |  |  | | --- | --- | | **typedef** struct point{  int xpos;  int ypos;  } **Point**; | **typedef** struct ~~point~~{  int xpos;  int ypos;  } **Point**; | | struct point{  int xpos;  int ypos;  };  **typedef** struct point **Point**; |  |   int main(){  **Point** pos = {10,20}  } |

### 구조체 연산 (복사, 크기 반환)

|  |
| --- |
| 구조체의 경우 다음과 같이 복사, 크기 반환 2가지 연산이 가능하며,  덧셈, 뺄셈 등의 연산은 사용자가 따로 정의해야 한다.  typedef struct point{  int xpos;  int ypos;  } Point;  int main(){  Point pos1 = {10, 20};  Point pos2;  pos2 = pos1 // 구조체 복사  printf("%d", sizeof(pos1)); // 구조체 크기 반환 (전체 멤버 데이터 크기 합)  } |

### 구조체 안의 구조체 (구조체 중첩)

|  |
| --- |
| typedef struct point{  int xpos;  int ypos;  } Point;  typedef struct circle{  Point center;  double radius;  } Circle;  void ShowCircleInfo(Circle \*cptr){  printf("(%d, %d)", (cptr🡪center).xpos, (cptr🡪center).ypos);  printf("(%d)", cptr🡪 radius);  }  int main(){  Circle c1 = {10, 20, 3.5}; // 구조체 선언 및 초기화  ShowCircleInfo(&c1);  return 0;  } |

## 공용체 (union)

|  |
| --- |
| typedef struct dbshort{  unsigned short upper;  unsigned short lower;  } DBShort;  typedef union rdbuf{  int iBuf;  char bBut[4];  DBShort sBuf;  } RDBuf;  int main(){  RDBfuf buf;    puts("정수입력 :");  scanf("%d", &(buf.iBuf));    printf("상위 2바이트: %u \n", buf.sBuf.upper);  printf("하위 2바이트: %u \n", buf.sBuf.lower);  printf("상위 1바이트 아스키 코드: %c \n", buf.bBuf[0]);  printf("상위 1바이트 아스키 코드: %c \n", buf.bBuf[3]);  return 0;  } |
| 메모리 공유를 통해 하나의 자원에 대해 다양한 표현이 가능하다.  공용체의 멤버 변수는 모두 다 같은 데이터를 가리키고 있으며,  단지 그 크기에 대한 정의와 표현형 출력만이 다를 뿐이다.  **(공용체 내 가장 큰 멤버 크기가 고용체의 크기)**  Buf 4바이트 (int)   |  |  |  |  | | --- | --- | --- | --- | | 1 Byte | 1 Byte | 1 Byte | 1 Byte | | bBuf[0] | bBuf[1] | bBuf[2] | bBuf[3] | | sBuf.upper | | sBuff.lower | | | iBuf | | | | |

## 열겨형 (enum)

|  |
| --- |
| typedef **enum** week{  Mon=0, Tue=1, Wed=2, Thur=3, Fri=4 // 상수값 지정  } Week;  int main(){  Week week1;  for (week1=0 ; week1 < 5 ; week1++)  printf("%d", week1); // 0 1 2 3 4 출력. 단, 상수형이므로 %s 출력은 안됨  for (week1=Mon ; week1!=Fri ; week1++)  printf("%d", week1); // 0 1 2 3 출력  } |
| 이름있는 상수의 정의를 통한 의미 부여  만약, 상수의 이름이 별도로 지정되지 않을 경우 처음 선언된 상수부터 0,  그 다음부터는 1씩 증가한다. |

## 메모리 구조와 동적할당

### 메모리 구조

|  |  |
| --- | --- |
| code area | 코드 |
| data area | - 전역변수  - static 변수  \* 접근 범위 통제와 프로그램 종료 시까지 메모리 적재를 위함  \* 전역변수는 다른 함수는 물론 다른 소스코드 파일에서도 참조 가능   |  |  |  | | --- | --- | --- | |  | 접근 범위 | 반환 시기 | | 전역 static 변수 | 선언된 소스코드  파일내에서만 접근 가능 | 프로그램 종료 시 | | 지역 static 변수 | 함수 내에서만 접근 | |
| heap | 동적할당 (malloc, calloc, realloc)  서브 함수 호출 시마다 생성되는 변수 필요 |
| stack | - 지역변수  - 매개변수 (함수 호출 시 넘겨주는 인자) |

cf. 541p C언어의 메모리 구조 문제 참고!!!

### 동적할당 (malloc, calloc, realloc)

|  |  |
| --- | --- |
| malloc (사이즈) | int \*ptr = (int \*)malloc(sizeof(int)\*100);  malloc은 (\*void) 타입이기 때문에 형 변환을 해주어야 한다. |
| calloc (개수, 사이즈) | int \*ptr = (int \*)calloc(100, sizeof(int));  - 표현방식의 변화  - 0으로 초기화 |
| realloc (개수, 사이즈) | ptr = (int\*)realloc(200, sizeof(int));  - 이미 할당된 변수에 재할당 |

## 매크로와 선행처리기

소스파일 🡪 (**선행처리기**) 🡪 소스파일 🡪 (**컴파일러**) 🡪 목적코드(0과 1로 이루어짐) 🡪 (**링커**) 🡪 실행파일

링커: 라이브러리에 있는 함수들을 삽입한다.

#include <stdio.h> // 기본 경로

#include "test.h" // 소스코드 파일로부터의 상대 경로

### 매크로 상수

|  |
| --- |
| #define PI 3.1415  지시자 / 매크로 / 매크로 몸체 |

### 매크로 함수

|  |
| --- |
| #define SQUARE(X) X \* X // 올바르지 못한 예  SQUARE(3+2)의 결과가 25가 아닌 11이 나옴.  Why? 문자 있는 그대로의 치환이기 때문에 3+2\*3+2 = 11  #define SQUARE(X) ((X)\*(X)) // 올바른 예 (괄호를 많이 해주어야함) |

### 매크로 장단점

|  |  |
| --- | --- |
| 장점 | 1. 매크로 함수는 일반함수에 비해 실행속도가 빠르다.  2. 자료형에 따라 별도로 함수를 정의하지 않아도 된다. |
| 단점 | - 정의하기 까다롭고, 디버깅이 어렵다. |

### 조건부 코드 삽입을 위한 매크로

#### if, elif, else, end if

|  |
| --- |
| #define A 1  #define B 0  int main(){  #if A // A 값이 1, True 로 정의되었으므로 해당 코드 영역이 실행된다.  ~~ ㅅ  #endif  #if B // B 값이 0, False 로 정의되었으므로 해당 코드 영역이 실행되지 않는다.  ~~~~~  #endif  } |
| #define Num 3  // 응용  #if Num == 2  #elif Num == 3  // 이 부분의 코드 실행  #else  #endif |

#### ifdef, ifndef (정의되었다면, 정의되지 않았다면)

|  |
| --- |
| #define A 1  // #define B 0  int main(){  #ifdef A // 매크로 A 정의되어 있으므로 실행 O  ~~ ㅅ  #endif  #ifdef B // 매크로 B가 정의되지 않았으므로 실행 X  ~~~~~  #endif  }  cf. ifndef 🡪 if not define 의 경우도 유사하게 사용한다. |

### 매개변수 치환 및 연결 (#, ##)

|  |
| --- |
| #define JOB(A, B) **#A** "의 직업은 " **#B** "입니다." // 치환  #define CON(UPP, LOW) UPP **##** 00 **##** LOW // 단순연결 |

## 다중 소스 파일

**extern** 변수는 다른 소스파일에 해당 변수가 선언되어 있음을 알리는 것.

헤더파일과 소스코드 여러 개를 잘 정의하고 나누는 방법에 대해서는 시험 범위에 어긋나서 생략

p580, Chapter 27 파일의 분할과 헤더파일의 디자인 참조