# 5장 컴퓨터에 계산을 시켜 봅시다. – 연산자

**학습 목표**

C 언어 프로그램의 기본 구조를 알고 있고, 실행되는 순서를 확인할 수 있다.

지금 나의 상태는 \_\_\_\_점이고 이 장을 학습하고 난 뒤 나의 목표는 \_\_\_\_점이다.

1 전혀 모른다

2 이름만 들어봤다

3 어렴풋이 안다

4 설명하기는 어렵지만 사용할 수 있다

5 무엇인지 정확히 알고 설명할 수 있다

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2장에서 구성한 개발환경을 이용하여 간단한 프로그램을 작성할 수 있다. |  |  |  |  |  |
| 프로젝트 파일의 기본 구조, 소스 코드의 기본 골격을 이해합니다. |  |  |  |  |  |
| 소스 코드를 한 줄씩 실행해 볼 수 있다. |  |  |  |  |  |
|  |  |  |  |  |  |

이번 장에서 배울 내용은

* 구문과 표현식의 차이
* 기본 수치 연산자
* 논리 연산자
* 비트 연산자

4장에서 값을 표현하는 방법을 배웠습니다. 이번 장에서는 값을 변경하는 방법에 대해 알아봅니다.

C언어에서 표현식expression 은 실제로 값을 반환하는 식입니다. 이와 달리 구문statement는 주어진 동작을 수행할 뿐, 값을 반환하지는 않습니다.

변수값을 변경하는데에는 연산자 operator가 사용됩니다. 해당 연산(계산)에 관련된 항목의 개수에 따라 1개만 필요한 단항 연산자, 2개가 필요한 이항 연산자, 3개 항목이 필요한 삼항 renary 연산자로 나뉘어집니다.

우리는 이미 이항 연산자중 가장 일반적인 할당 연산자에 대해 살펴본 적이 있습ㄴ디ㅏ.

int i = 3;

i = 5;

int j = i + 7;

### 산술 연산자

C언어는 기본적인 산술연산자를 제공합니다. 이항 연산자로 사칙연산, 즉 +(덧셈), -(뺄셈), \*(곱셈), /(나눗셈), ^(거듭제곱), %(나머지) 연산자를 제공합니다. 단항연산자는 부호를 변경하는 -연산자가 있습니다.

관계 연산자

모든 관계 연산자는 결과값으로 항상 불린 값 즉 참 또는 거짓을 반환합니다.

<

>

<=

>=

==

!=

문자열일 경우 부등호연산의 결과는 어떻게 될까요?

### 논리 연산자

논리 연산자는 참 또는 거짓을 판단하는 데 사용됩니다. 논리합(OR), 논리곱(AND), 부정(NOT)으로 구분하며 각각 ||, &&, ! 기호를 사용합니다. 논리합과 논리곱은 두 조건이 필요합니다. 논리합은 두 조건 중 하나라도 참이면 결과값은 참이 됩니다. 논리곱은 두 조건 모두 참이면 결과값이 참이 됩니다. 부정은 해당 조건의 논리값을 뒤집습니다.

이를 논리표로 정리하면 다음과 같습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | 논리합(OR)  A || B | 논리곱(AND)  A && B | 부정 (NOT)  !A |
| 참 | 참 | 참 | 참 | 거짓 |
| 참 | 거짓 | 참 | 거짓 | 거짓 |
| 거짓 | 참 | 참 | 거짓 | 참 |
| 거짓 | 거짓 | 거짓 | 거짓 | 참 |

논리 연산자의 결과값은 불린 변수에 저장할 수 있습니다.

초기 C언어에는 불 데이터 타입이 없었습니다. 대신 개발자들은 0을 거짓으로, 그 외는 참으로 사용했습니다.

#### 단축 평가(Short-Circuit Evaluation)

논리표를 보면 흥미로운 사실을 알 수 있습니다. 논리합의 경우 첫번째 조건이 참이면 두번째 조건이 어떤 값이든 결과값은 참이 됩니다. 논리곱의 경우 첫번째 조건이 거짓이면 두번째 조건이 어떤 값이든 결과값은 거짓이 됩니다. 이 규칙을 이용하여 C언어는 논리연산자에서 첫번째 조건과 논리연산자에 따라 두번째 조건을 평가하지 않습니다. 답이 이미 정해져 있으니 굳이 한번 더 논리값을 확인하면서 연산에 필요한 자원과 시간을 낭비할 필요가 없습니다.

이렇게 조건식 전체를 평가하지 않고도 결과값을 알 수 있을 때 뒤에 나오는 조건식을 평가하지 않는 동작방식을 단축평가라고 합니다. 단축 평가는 효율성을 높여줄 뿐만 아니라 데이터 접근의 안정성을 제공합니다. 특정 조건에서 발생할 수 있는 오류를 미리 막을 수 있습니다. 아직 배우지는 않았지만 NULL 포인터 참조를 방지할 수 있습니다.

if (ptr != NULL && \*ptr >0) {

printf(“0보다 큰 수를 가리키고 있습니다.”);

}

조각 코드에서 ptr은 포인터를, \*ptr은 포인터가 가리키는 값을 의미합니다. 그런데 포인터가 NULL일 경우 이 포인터를 참조하면 메모리 접근 오류가 발생하면서 프로그램이 중단됩니다. 하지만 위 코드에서는 포인터가 NULL이라면 단축 평가에 의해 포인터 참조 조건이 실행되지 않습니다. 포인터를 사용하게 되면 이 패턴을 자주 사용하게 됩니다.

연산자 우선 순위

연산자는 기본적으로 항상 왼쪽에서 오른쪽으로 진행됩니다.

연산자의 우선순위가 애매할 때에는 항상 괄호를 사용하여 먼저 계산해야 하는 부분을 명시합니다.

효율성도 좋아지지만 단축평가는 주의해서 사용해야 합니다. 그렇지 않으면 의도치 않았던 결과를 얻을 수 있는데, 이를 부수효과side-effect라고 부릅니다. 예를 들어 다음 코드에서 변수 i의 값은 무엇일까요?