

# Chapter 02 - 변수와 연산자

● 생성일	@2025년 9월 3일 오후 4:49
☰ 태그	

## 1. C++ 표준 입출력

숫자를 2개 입력 받아 합을 출력하기

using space 이용하기

아스키 코기

부동 소수점 출력하기

## 2. 변수의 유효 범위와 형식 변환

## 3. 키워드와 리터럴

값 그 자체를 나타내는 리터럴

문자열 표현 방식

사용자 정의 리터럴

## 4. 표현식과 연산자

상수 표현식

단항 연산자 표현식

증감 연산자

논리 NOT

비트 연산자

2의 보수를 구하는 법

이항 연산자 표현식

비트 AND 연산 (&)

비트 OR 연산 ( | )

비트 XOR 연산 ( ^ )

논리 시프트 연산 ( >>, << )

산술 시프트 연산 ( >>, << )

삼항 연산자 표현식

연산자 우선 순위

## 1. C++ 표준 입출력

- C와 C++ 입출력 비교

구분	C	C++
헤더 파일	stdio.h	iostream
입력문	scanf	cin
출력문	printf	cout

## 숫자를 2개 입력 받아 합을 출력하기

```
#include <iostream>

int main()
{
    int i, j;
    std::cout << "Enter num_1: "; // 문자열 출력
    std::cin >> i; // 사용자에게 정수를 입력받아 i에 저장
    std::cout << "Enter num_2: "; // 문자열 출력
    std::cin >> j; // 사용자에게 정수를 입력받아 j에 저장

    std::cout << " num_1 + num_2 = " << i + j << std::endl; //
    두 수의 합 출력

    return 0;
}
```

- 네임 스페이스

- 소속을 지정해주는 역할
- std:: → std라는 네임스페이스에 접근할 때 쓰는 표현
- std → C++ 언어에서 흔히 사용하는 여러가지 함수와 클래스, 객체, 유틸리티가 정의된 네임 스페이스
- 내부 식별자에 범위를 부여해 여러 라이브러리를 포함할 때 이름이 충돌하는 것을 방지하려고 사용

- 소스 앞부분에 **using namespace std** 코드를 작성해 생략 가능  
→ “**cin, cout** 등이 사용될 때는 무조건 **std**에 속한 것을 호출한다”

## using space 이용하기

```
#include <iostream>
using namespace std;      // 네임 스페이스 사용 선언

int main()
{
    int i, j;

    cout << "Enter num_1: ";
    cin >> i;

    cout << "Enter num_2: ";
    cin >> j;

    cout << "num_1 + num_2 = " << i + j << endl;

    return 0;
}
```

- 생략 가능하나, **using space std** 대신 **std::**를 매번 표기하는 게 더 좋음  
→ std 전체 네임스페이스를 가져올 때 이를 선언이 충돌할 수 있기 때문
- C++에서는 %d, %f 같은 형식 지정자 대신 **cout**라는 스트림 객체 사용  
→ 형식 지정자를 사용하지 않고도 문자열이나 정수, 부동 소수점 출력 가능
- **cout**
  - << 연산자로 출력 대상을 전달  
→ 연산자가 가리키는 방향 : 정보의 흐름
  - << 을 여러 개 이용하면 연속적 출력 가능

- 콘솔에서 줄을 바꿀 때는 **endl** 사용 (개행문자 출력 + 출력버퍼 비움)
- **cin** 입력
  - C언어의 printf와 같은 용도
  - cin에서는 **>>** 연산자 사용 (cout 와 반대)
  - **>>** 연산자 다음에는 **스트림에서 읽어온 값을 저장할 변수** 지정
  - 형식 지정자 필요 없음
- 데이터 형식
  - 값을 저장하기 전에 정수, 부동 소수점 수, 문자 등 어떤 값을 지정할지 정해주는 것

자료형의 종류					
구분	자료형	크기(byte)	표현범위		비고
기본형	void		-		
문자형	char	1	-128 ~ 127		$-2^7 \sim 2^7 - 1$ =signed char
	unsigned char	1	0 ~ 255		$0 \sim 2^8 - 1$
정수형	bool	1	0 ~ 1		false or true
	_int8	1	-128 ~ 127		$-2^7 \sim 2^7 - 1$
	_int16	2	-32,768 ~ 32,767		$-2^{15} \sim 2^{15} - 1$
	short	2	-32,768 ~ 32,767		$-2^{15} \sim 2^{15} - 1$ =signed short int
	unsigned short	2	0 ~ 65,535		$0 \sim 2^{16} - 1$ =unsigned short int
	_int32	4	-2,147,483,648 ~ 2,147,483,647		$-2^{31} \sim 2^{31} - 1$
	int	4	-2,147,483,648 ~ 2,147,483,647		$-2^{31} \sim 2^{31} - 1$ =signed int
	unsigned int	4	0 ~ 4,294,967,295		$0 \sim 2^{32} - 1$
	long	4	-2,147,483,648 ~ 2,147,483,647		$-2^{31} \sim 2^{31} - 1$ =signed long int
	unsigned long	4	0 ~ 4,294,967,295		$0 \sim 2^{32} - 1$ =unsigned long int
실수형	_int64	8	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807		$-2^{63} \sim 2^{63} - 1$
	float	4	3.4E-38 ~ 3.4E38		$3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$ 7자리
	double	8	1.7E-308 ~ 1.7E308		$1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$ 15자리
	long double	8	1.2E-4932 ~ 1.2E4932		$1.2 \times 10^{-4932} \sim 1.2 \times 10^{4932}$ 19자리

- void : 형식 없음

- signed char : 부호 비트를 가지도록 명시적 표현
- 형식이 없음을 나타내는 void
  - void 형으로는 변수를 선언할 수 없음
  - void형 사용 예시

### 1. 함수가 값을 반환하지 않을 때

```
void print_func(){
    std::cout << "func" << std::endl;
}
```

### 2. 함수의 매개변수가 없음을 표시할 때

```
int input_func(void)
{
    int input_value;
    std::cin >> input_value;
    return input_value;
}
```

### 3. 모든 자료형을 가리킬 수 있는 제네릭 포인터로 사용할 때

```
int int_value;
float float_value;
void *prt_value;
ptr_value = &int_value;
ptr_value = &float_value;
```

- 참, 거짓만 가지는 bool 형식
  - true - 정수 1, false - 정수 0 의미
  - 키워드가 아닌 정수로 저장됨

```

#include <iostream>
using namespace std;

int main()
{
    bool value;

    value = true;
    cout << value << endl;

    value = false;
    cout << value << endl;

    return 0;
}

```

- 문자 형식

- char : 8bit(1Byte) 정수를 저장하는 역할, 문자 전용 데이터 형식은 아님 (아스키 코드 변환 후 사용)

```

#include <iostream>
using namespace std;

int main()
{
    cout << "아스키 코드 출력하기: [32~126]:\n";
    for (char i = 32; i <= 126; i++) // 32부터 126까지
        1씩 증가하며 반복
    {
        // 아스키 코드를 출력할 때 공백을 넣고 16개마다 줄
        바꾸기
        cout << i << ((i % 16 == 15) ? '\n' : ' ');
    }
}

```

- 문자를 표현하는 데 `char`을 사용하는 이유
  - 아스키 코드가 7bit 형태의 체계를 따르고 있기 때문
  - 나머지 1bit는 통신 확인용 패리티 비트
  - `unsigned` 키워드 사용 → 부호 비트까지 활용하여, 0~255까지 더 많은 양수 저장 가능
  - `wchar_t` : 와이드 문자를 저장하는 자료형
- `wchar_t`와 `char`의 차이

구분	<code>char</code>	<code>wchar_t</code>
인코딩 방식	멀티바이트(MBCS)	유니코드(UNICODE)
단일 문자 크기	1byte 또는 2byte (영문, 숫자 등의 아스키 코드는 1byte, 한글. 한자 등은 2byte로 표현)	2byte(GCC에서는 기본 4byte)
문자열	유니코드를 제외한 문자열 (ANSI, UTF-8)	와이드 문자, UTF-16으로 인코딩 된 문자열

## 아스키 코딩

```
#include <iostream>
#include <io.h>
#include <fcntl.h>

using namespace std;

int main()
{
    wchar_t message_korean[] = L"반갑다 세계야!"; // 한국어
    wchar_t message_chinese[] = L"你好，世界！"; // 중국어
    wchar_t message_japanese[] = L"ハローワールド！"; // 일본어
    wchar_t message_russian[] = L"Привет мир！"; // 러시아어

    cout << "Hello, World!" << endl;
}
```

```

        _setmode(_fileno(stdout), _O_U16TEXT); // 윈도우 콘솔 창
유니코드 출력 모드 설정

    wcout << message_korean << endl;
    wcout << message_chinese << endl;
    wcout << message_japanese << endl;
    wcout << message_russian << endl;

    return 0;
}

```

## 1. 헤더 파일 포함

- **iostream** : C++의 표준 입출력 스트림(cout, wcout, endl 등)을 사용하기 위해 포함
- **io.h** : \_setmode, \_fileno와 같은 저수준(low-level) 파일 핸들링 함수를 사용하기 위해 포함  
(주로 Microsoft Visual C++ 환경에서 사용)
- **fcntl.h** : \_O\_U16TEXT와 같은 파일 제어 옵션 상수를 사용하기 위해 포함

## 2. 유니코드 문자열 초기화

- **wchar\_t**는 '와이드 문자(wide character)'를 저장하기 위한 데이터 타입
- 2바이트(Windows) 또는 4바이트(Linux) 크기를 가짐
- 이는 ASCII(1바이트)로 표현할 수 없는 다양한 언어의 문자를 저장하는 데 사용됩니다.

## 3. L"..." 구문은 문자열 리터럴을 wchar\_t 타입의 배열로 만듦

- message\_korean, message\_chinese 등 4개의 wchar\_t 배열에 각각 다른 언어의 문자열을 저장

## 4. 첫 번째 출력 (바이트 스트림)

- cout << "Hello, World!" << endl;  
표준 바이트 기반 출력 스트림(cout)을 사용하여 "Hello, World!" 문자열을 화면에 출력  
이 시점에서 콘솔은 기본 모드(일반적으로 텍스트 모드)로 동작합니다.

## 5. 콘솔 모드 변경

- \_setmode(\_fileno(stdout), \_O\_U16TEXT);

**stdout** : 표준 출력 스트림을 가리키는 포인터

**\_fileno(stdout)** : stdout 스트림에 해당하는 파일 디스크립터(정수 번호)를 가져옴

**\_setmode(...)** : 해당 파일 디스크립터의 입출력 변환 모드를 변경

**\_O\_U16TEXT** : 출력 모드를 UTF-16 유니코드로 설정

이 설정 이후, stdout으로 전송되는 데이터는 UTF-16 인코딩으로 해석되어 콘솔에 표시됨

## 6. 두 번째 출력 (와이드 스트림)

- wcout << ... << endl; 라인들은 와이드 문자 기반 출력 스트림(wcout)을 사용  
**\_setmode**로 콘솔이 유니코드 출력을 준비하게 된 상태에서, **wcout을 통해 wchar\_t 배열에 저장된 유니코드 문자열들을 출력함.**  
이로써 한국어, 중국어 등이 깨지지 않고 정상적으로 표시됨

- 정수 형식

- int는 컴퓨터 시스템에 따라 크기가 다름

```
#include <iostream>
using namespace std;

int main()
{
    cout << "short : " << sizeof(short) << " bytes" << endl;
    cout << "unsigned short : " << sizeof(unsigned short) << " bytes" << endl;
    cout << "int : " << sizeof(int) << " bytes" << endl;
    cout << "unsigned int : " << sizeof(unsigned int) << " bytes" << endl;
    cout << "__int8 : " << sizeof(__int8) << " bytes" << endl;
    cout << "__int16 : " << sizeof(__int16) << " bytes" << endl;
    cout << "__int32 : " << sizeof(__int32) << " bytes" << endl;
```

```

        cout << "__int64 : " << sizeof(__int64) << " bytes"
<< endl;
        cout << "long : " << sizeof(long) << " bytes" << endl;
        cout << "unsigned long : " << sizeof(unsigned long)
<< " bytes" << endl;
        cout << "long long : " << sizeof(long long) << " bytes"
<< endl;
        cout << "unsigned long long : " << sizeof(unsigned long long)
<< " bytes" << endl;

    return 0;
}

```

## 결과

```

short : 2 bytes
unsigned short : 2 bytes
int : 4 bytes
unsigned int : 4 bytes
__int8 : 1 bytes
__int16 : 2 bytes
__int32 : 4 bytes
__int64 : 8 bytes
long : 4 bytes
unsigned long : 4 bytes
long long : 8 bytes
unsigned long long : 8 bytes

```

- C++ 언어 표준안

**1 byte == sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)**

- 부호가 있는(signed) 정수는 음수와 양수를 모두 저장 가능

- signed 키워드를 이용해 명시적 선언 가능
  - 부호가 없는(unsigned) 정수는 양수만 가질 수 있음
    - unsigned 키워드를 이용해 명시적 선언 가능

크기별 signed / unsigned	정수 표현 범위
1byte signed	-128 ~ 127
1byte unsigned	0 ~ 255
2byte signed	32,768 ~ 32,767
2byte unsigned	0 ~ 65,535
4byte signed	-2,147,483,648 ~ 2,147,483,648
4byte unsigned	0 ~ 4,294,967,295
8byte signed	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
8byte unsigned	0 ~ 18,446,744,073,709,551,615

### • 부동 소수점 형식

- 자료형의 크기가 정해져 있으므로 부동 소수점의 특정 자리까지만 저장할 수 있고 나머지는 유실됨
- 부동 소수점의 정밀도
  - 데이터 유실 없이 얼마나 많은 유효 자릿수를 나타낼 수 있는지를 말함

## 부동 소수점 출력하기

```
#include <iostream>
using namespace std;

int main()
{
    cout << 9.8765432f << endl; // 값 끝에 f를 붙이면 float
부동 소수점
```

```
    cout << 987654.321f << endl;
    cout << 98765432.1f << endl;
    cout << 0.00000987654321f << endl;
    cout << 0.000000000987654321f << endl;

    return 0;
}
```

## 결과

```
9.87654
987654
9.87654e+07
9.87654e-06
9.87654e-11
```

- 유효한 숫자 표현으로 **6자리만 출력됨.**
- cout은 부동 소수점을 출력할 때 **기본 정밀도가 6**  
→ 6자리까지만 중요하다고 생각하고 나머지는 생략함

## 부동 소수점의 최대 유효 자릿수만큼 출력하기

```
#include <iostream>
#include <iomanip>
#include <limits>

using namespace std;

int main()
{
    float float_value = 9.87654321f;
    double double_value = 9.87654321987654321;
    long double long_double_value = 9.87654321987654321l;
    cout << "float : " << sizeof(float) << " bytes" << endl;
```

```

        cout << "float_value : " << setprecision(numeric_limits
<float>::digits10 + 1) << float_value << endl << endl;

        cout << "double : " << sizeof(double) << " bytes" << en
dl;
        cout << "double_value : " << setprecision(numeric_limit
s<double>::digits10 + 1) << double_value << endl << endl;

        cout << "long double : " << sizeof(long double) << " by
tes" << endl;
        cout << "long_double_value : " << setprecision(numeric_
limits<long double>::digits10 + 1) << long_double_value <<
endl << endl;

        return 0;
}

```

## 결과

```

float : 4 bytes
float_value : 9.876543

double : 8 bytes
double_value : 9.876543219876543

long double : 8 bytes
long_double_value : 9.876543219876543

```

- std::setprecision : cout에서 출력 되는 기본 정밀도를 조절하는 함수
- std::setprecision(std::numeric\_limits<데이터 형식>::digits10 + 1);

부동 소수점 크기(byte)	유효 자릿수(자리)
4	6~9, 일반적으로 7
8	15~18, 일반적으로 16
16	33~36

## 2. 변수의 유효 범위와 형식 변환

- 지역 범위 : 선언된 함수 내에서만 선언 변수 사용 가능

### 지역 범위가 지정된 변수

```
#include <iostream>
using namespace std;

void print() {
    // 함수 내부의 지역 변수
    int value = 10;
    cout << "print 함수 내에서의 지역 변수 value: " << value <
< endl;
}

int main()
{
    // main 함수 내부의 지역 변수
    int value = 5;
    cout << "main 함수 내에서의 지역 변수 value : " << value <
< endl;
}
```

### 결과

```
main 함수 내에서의 지역 변수 value : 5
```

- 구문 범위 : if, for, while 등의 구문 안에서만 유효함
- 지역 범위 : 매개변수 이름을 포함하여, 함수 안에 선언한 이름은 해당 함수 내에서만 유효함
  - 지역 범위는 블록 범위 (block scope)라고도 함.
- 전역 범위 : 네임 스페이스나 클래스, 함수 등에 속하지 않고 외부에 선언된 이름
  - 선언 지점부터 파일 끝까지 유효함

- 클래스 범위 : 클래스 멤버의 이름은 선언 지점에 관계 없이 클래스 정의 전체에 걸쳐 확장됨

    클래스 멤버에 대한 접근성은 접근 지정자(public, private 등)로 제어할 수 있음

- 네임스페이스 범위 : 네임스페이스 안에 선언한 이름은 네임스페이스 안에서만 유효함.  
    네임스페이스는 서로 다른 파일들의 여러 블록에서 선언될 수 있음

- 지역 범위 내에 전역 변수와 이름이 같은 변수가 있으면 지역 변수의 우선권이 더 높음.  
    → 이때 전역 범위 연산자를 사용하면 전역 변수에 접근할 수 있음

### 3. 키워드와 리터럴

- 키워드 : 특별한 의미로 미리 정의해 둔 식별자

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	nt	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	
delete	goto	reinterpret_cast	try	

### 값 그 자체를 나타내는 리터럴

- 리터럴 : 코드에 직접 표현된 변하지 않는 값 그 자체

```
int value = 5;
double value = 0.5;
char value = 'A';
```

- 코드에서 5, 0.5, 'A'가 모두 리터럴임.
- 리터럴도 데이터 형식을 가짐
- 부동 소수점 형식의 기본 데이터 형식 → double ( float 아님 )

```
float value = 0.5f;
unsigned int value = 5u;
long value = 5L;
```

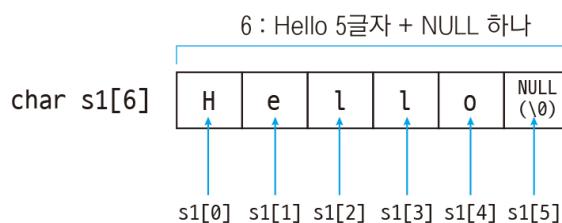
- 기본 리터럴 외에도 별도의 접미사를 붙여 지정할 수 있는 리터럴도 있음

데이터 형식	리터럴 접미사
unsigned int	u, U
long	l, L
unsigned long	ul, uL,
long long	ll, LL
unsigned long long	ull, uLL, UII, ULL, llu, IIU, LLu, LUU
float	f, F
long double	l, L

## 문자열 표현 방식

```
char *str = "Hello";
char str[] = "Hello";
```

- 이 코드는 내부적으로는 char 배열을 만들고, 해당 배열에 문자를 하나씩 차례로 저장
- 배열의 맨 마지막에는 문자열의 끝을 알리는 널 문자( \0 )를 저장



```
#include <iostream>
// #include <string> // iostream 헤더에 string도 포함됨

using namespace std;

int main()
{
    string string_value("Hello");
    cout << string_value << endl;
    string_value = "World!";
    cout << string_value << endl;

    return 0;
}
```

실행 결과

```
Hello
World!
```

문자 리터럴

- 문자 리터럴 → 'a'나 '\'처럼 작은 따옴표로 묶인 단일 문자
- 문자 리터럴은 프로그램에서 특정 문자를 표현하는 데 매우 유용

```
std::cout << "Hello World!\n";
```

- 다음처럼 줄 바꿈 리터럴을 직접 사용할 수도 있음
- 문자 리터럴은 프로그램에서 특정 문자를 표현하는 데 매우 유용
- 개행 문자 콘솔에 출력 → 이스케이프 시퀀스 '\n' 사용

```
std::cout << "Hello World!" << '\n';
```

문자 리터럴	기본 예	std::string 활용 예
일반 문자	'a'	std::string str("Hello");
와이드 문자	L'a'	std::wstring str3(L"Hello");
UTF-8 문자	u8'a'	std::string str2(u8"Hello"); // C++20 이전 std::u8string u8str2(u8"Hello"); // C++20 부터
UTF-16 문자	u'a'	std::u16string str4(u"Hello");
UTF-32 문자	U'a'	std::u32string str5(U"Hello");

## 사용자 정의 리터럴

- 기본으로 제공되는 리터럴 외에 개발자가 직접 리터럴 정의 가능

반환\_타입 **operator""** 리터럴\_접미사(매개변수\_구성)

- 리터럴을 나타내는 접미사를 함수 이름으로 만들면 됨
- 사용자 정의 리터럴 연산자 → **operator"" 사용**

```
#include <iostream>
using namespace std;

const long double km_per_mile = 1.609344L

long double operator"" _km(long double val)    // _km 사용자
리터럴 정의
{
    return val;    // 아무런 작업 없이 그대로 반환
}

long double operator"" _mi(long double val)    // _mi 사용자
```

```

리터럴 정의
{
    return *km_per_mile; // 마일을 킬로미터로 변환하여 반환
}

int main()
{
    long double distance_1 = 1.0_km; // 킬로미터는 그대로 저장
    long double distance_2 = 1.0_mi; // 마일은 킬로미터 단위로 변환해서 저장

    cout << distance_1 + distance_2 << " km" << endl;
    // 킬로미터로 출력

    return 0;
}

```

실행 결과

2.60934 km

- **\_km** → 전달받은 값을 그대로 반환
- **\_mi** → 마일을 킬로미터로 변환한 후에 반환  
→ 두 거리를 더한 값을 출력할 때는 킬로미터 단위로 출력됨

## 4. 표현식과 연산자

- **표현식** : 프로그래밍에서 계산할 때 사용하는 식

### 상수 표현식

- **상수 표현식** : 상수로만 이루어진 단순한 표현식

- 상수 → 1, 12.345, 'A'처럼 수식에서 변하지 않는 값 의미

## 단항 연산자 표현식

- 단항 연산자 표현식** : 연산자와 피연산자가 일대일로 매칭되는 표현식  
→ 연산에 참여하는 피연산자가 하나인 표현식
- 형 변환, 부호 변경, 증감 연산자 모두 표현

이름	형태
부호 연산자	+a, -a
증감 연산자	++a, --a, a++, a--
형식 변환	(type)a
크기	sizeof(a)
논리 NOT	!a
비트 연산자	~a
포인터 연산자	*a
주소 연산자	&a

## 증감 연산자

- 프로그래밍에서는 변수값을 1만큼 증가시키거나 감소하는 연산을 자주 사용함

```
#include <iostream>
using namespace std;

int main()
{
    int a = 0; // a 최초 값 0
    int b = 0; // b 최초 값 0
    int a_prefix;
    int b_postfix;

    a_prefix = ++a; // a값을 1만큼 증가시킨 후에 a_prefix
    에 대입
    b_postfix = b++; // b값을 b_postfix에 대입한 후에 b값
```

을 1만큼 증가

```
    cout << "a= " << a << ", " << "a_prefix = " << a_prefix
    << endl;
    cout << "b= " << b << ", " << "b_postfix = " << b_postfix
    << endl;

    return 0;
}
```

실행 결과

```
a = 1, a_prefix = 1
b = 1, b_postfix = 0
```

- 같은 증가 연산자임에도 전위 / 후위에 따라 결과가 다르게 나옴

## 논리 NOT

- 논리 NOT 연산자 : 값이나 식별자 앞에 느낌표 !를 붙여서 사용
- true는 false로, false는 true로 반전
- C / C++에서 0은 false, 0 외에는 모두 true로 취급  
→ 5는 true, !5는 0 ( false )

## 비트 연산자

- ~ : 비트열을 반전하라는 뜻, 각 자릿수의 비트값을 반대로 바꿔 1의 보수로 변환

```
#include <iostream>
using namespace std;

int main()
{
    unsigned int value = 0x00000000;      // 0을 16진수(hex)로
```

표현한 값

```
    value = ~value;
    cout << hex << value << endl;

    return 0;
}
```

실행 결과

```
fffffff
```

## 2의 보수를 구하는 법

- 1의 보수에 1을 더하면 2의 보수를 구할 수 있음
- 2의 보수 (음수)

10진수	2진수	1의 보수	2의 보수 (음수)	10진수 (음수)
0	0000 0000	1111 1111	0000 0000	0
1	0000 0001	1111 1110	1111 1111	-1
2	0000 0010	1111 1101	1111 1110	2

## 이항 연산자 표현식

- '피연산자 연산자 피연산자'처럼 연산에 참여하는 피연산자가 2개인 표현식

## 비트 AND 연산 (&)

- 첫 번째 피연산자의 각 비트를 두 번째 피연산자의 비트와 비교해, 양쪽 비트가 모두 1일 때만 결과 비트를 1로 설정함. 그 외에는 해당 비트를 0으로 설정

```
#include <iostream>
#include <bitset>

using namespace std;
```

```

int main()
{
    int a = 13;
    int b = 27;
    int c = a & b; // 비트 AND 연산

    cout << "a = " << bitset<8>(a) << " : " << a << endl;
    cout << "b = " << bitset<8>(b) << " : " << b << endl;
    cout << "c = " << bitset<8>(c) << " : " << c << endl;

    return 0;
}

```

## 실행 결과

```

a = 00001101 : 13
b = 00011011 : 27
c = 00001001 : 9

```

## 비트 OR 연산 (|)

```

#include <iostream>
#include <bitset>

using namespace std;

int main()
{
    int a = 13;
    int b = 27;
    int c = a | b; // 비트 OR 연산

    cout << "a = " << bitset<8>(a) << " : " << a << endl;
    cout << "b = " << bitset<8>(b) << " : " << b << endl;

```

```
    cout << "c = " << bitset<8>(c) << " : " << c << endl;

    return 0;
}
```

실행 결과

```
a = 00001101 : 13
b = 00011011 : 27
c = 00001111 : 31
```

## 비트 XOR 연산 (^)

```
#include <iostream>
#include <bitset>

using namespace std;

int main()
{
    int a = 13;
    int b = 27;
    int c = a ^ b; // 비트 XOR 연산

    cout << "a = " << bitset<8>(a) << " : " << a << endl;
    cout << "b = " << bitset<8>(b) << " : " << b << endl;
    cout << "c = " << bitset<8>(c) << " : " << c << endl;

    return 0;
}
```

실행 결과

```
a = 00001101 : 13
b = 00011011 : 27
```

```
c = 00010110 : 22
```

## 논리 시프트 연산 ( >>, << )

- 비트 연산자인 오른쪽 시프트 ( >> )와 왼쪽 시프트 ( << )는 방향만 차이가 있을 뿐, 동작 방식은 같음.
- 시프트 연산은 '변수 >> 이동 비트 수', '변수 << 이동 비트 수' 형식으로 사용함
- >>는 오른쪽, <<는 왼쪽으로 지정한 숫자만큼 비트를 이동시키며 모자라는 비트는 0으로 채움

## 산술 시프트 연산 ( >>, << )

- 산술 시프트 연산과 논리 시프트 연산은 같은 >>, << 연산자를 사용하지만, 오른쪽 시프트 연산의 동작 방식에 차이가 있음
- signed 자료형에서 최상위 비트는 부호 비트로 사용되므로, 오른쪽 산술 시프트(>>)는 모자라는 최상위 비트를 원래 부호 비트 값으로 채움.
- 만약 산술 시프트에서 원본 데이터의 최상위 비트가 1인 경우, 시프트 후에도 부호 비트인 1을 유지하며 채워짐. 반대로 최상위 비트가 0이면 0으로 채워짐.

```
#include <iostream>
#include <bitset>

using namespace std;

int main()
{
    int a = 13;
    int b = a >> 1; // 1bit 오른쪽으로 시프트
    int c = a << 1; // 1bit 왼쪽으로 시프트
    int d = a >> -1; // 시프트 수행 오류
    int e = a << 32; // 시프트 수행 오류
```

```

    cout << "a = " << bitset<8>(a) << " : " << a << endl;
    cout << "b = " << bitset<8>(b) << " : " << b << endl;
    cout << "c = " << bitset<8>(c) << " : " << c << endl;
    cout << "d = " << bitset<8>(d) << " : " << d << endl;
    cout << "e = " << bitset<8>(e) << " : " << e << endl;

    return 0;
}

```

### 실행 결과

```

a = 00001101 : 13
b = 00000110 : 6
c = 00011010 : 26
d = 00000000 : 0
e = 00001101 : 13

```

- 이동할 비트 수가 음수이거나 너무 크면 시프트 연산은 제대로 수행되지 않음.

## 삼항 연산자 표현식

조건식 ? 참일\_때\_표현식 : 거짓일\_때\_표현식

```

#include <iostream>
using namespace std;

int main()
{
    int a = 7;
    int b = 5;
    int result;

    if (a > b)
        result = a; // a > b가 true이면, result에 a값 저장
    else

```

```
    result = b; // a > b가 false이면, result에 b값 저장  
  
    cout << "result = " << result << endl;  
  
    return 0;  
}
```

실행 결과

```
result = 7
```

같은 예를 삼항 연산자로 바꾸면...

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int a = 7;  
    int b = 5;  
    int result;  
  
    result = a > b ? a : b; // a > b 결과에 따라 result에 a값  
    또는 b값 저장  
  
    cout << "result = " << result << endl;  
  
    return 0;  
}
```

실행 결과

```
result = 7
```

- 삼항 연산자는 if조건문보다 코드를 간략하게 작성할 수 있지만, 무분별하게 사용하면 오히려 가독성이 떨어짐.

## 연산자 우선 순위

- 우선 순위가 헷갈릴 때에는 () 괄호로 우선순위를 저장하면 됨

```
#include <iostream>
using namespace std;

int main()
{
    int a = 5, b = 2, c = 8;

    int result_1 = a + b * c;    // 곱셈 먼저 연산 (오른쪽으로 결합)
    cout << "Result 1: " << result_1 << endl;

    int result_2 = (a + b) * c;  // 괄호로 우선순위 변경
    cout << "Result 2: " << result_2 << endl;

    a += b * c; // 곱셈 먼저 연산
    cout << "Result 3: " << a << endl;

    bool condition = true;
    int result_4 = (condition && a > b) ? a : b; // > 먼저 연산 (왼쪽으로 결합)
    cout << "Result 4: " << result_4 << endl;

    return 0;
}
```

### 실행 결과

```
Result 1: 21
Result 2: 56
Result 3: 21
Result 4: 21
```