Topic 2 Types
Part I

내용

- 타입, 변수, 리터럴, 상수
- 형 변환
- brace initialization
- 지역 변수와 전역 변수
- 포인터
- new, delete
- 스마트 포인터
- 포인터와 상수
- 포인터와 참조
- nullptr(since C++11)
- auto, decltype(since C++11)

타입, 변수, 상수, 리터럴

```
# include <iostream>
# include <string>
using namespace std ;
// 상수
const int BASE_SCORE = 50 ;
const int GOOD_SCORE = 80 ;
const string GOOD_MSG = "Good" ;
const string BAD_MSG = "Not Good" ;
int main() {
    cout << "Enter your score: " ;          // 문자열 리터럴
    int score ;                             // int 타입의 변수 score
    cin >> score ;
    int result = BASE_SCORE + score ; // 상수 BASE_SCORE의 사용
    string msg ;                            // string 객체 msg
    if ( result >= GOOD_SCORE )
        msg = GOOD_MSG ;
    else
        msg = BAD_MSG ;
    cout << "The result: " << result << " is " << msg << endl ;
}
```

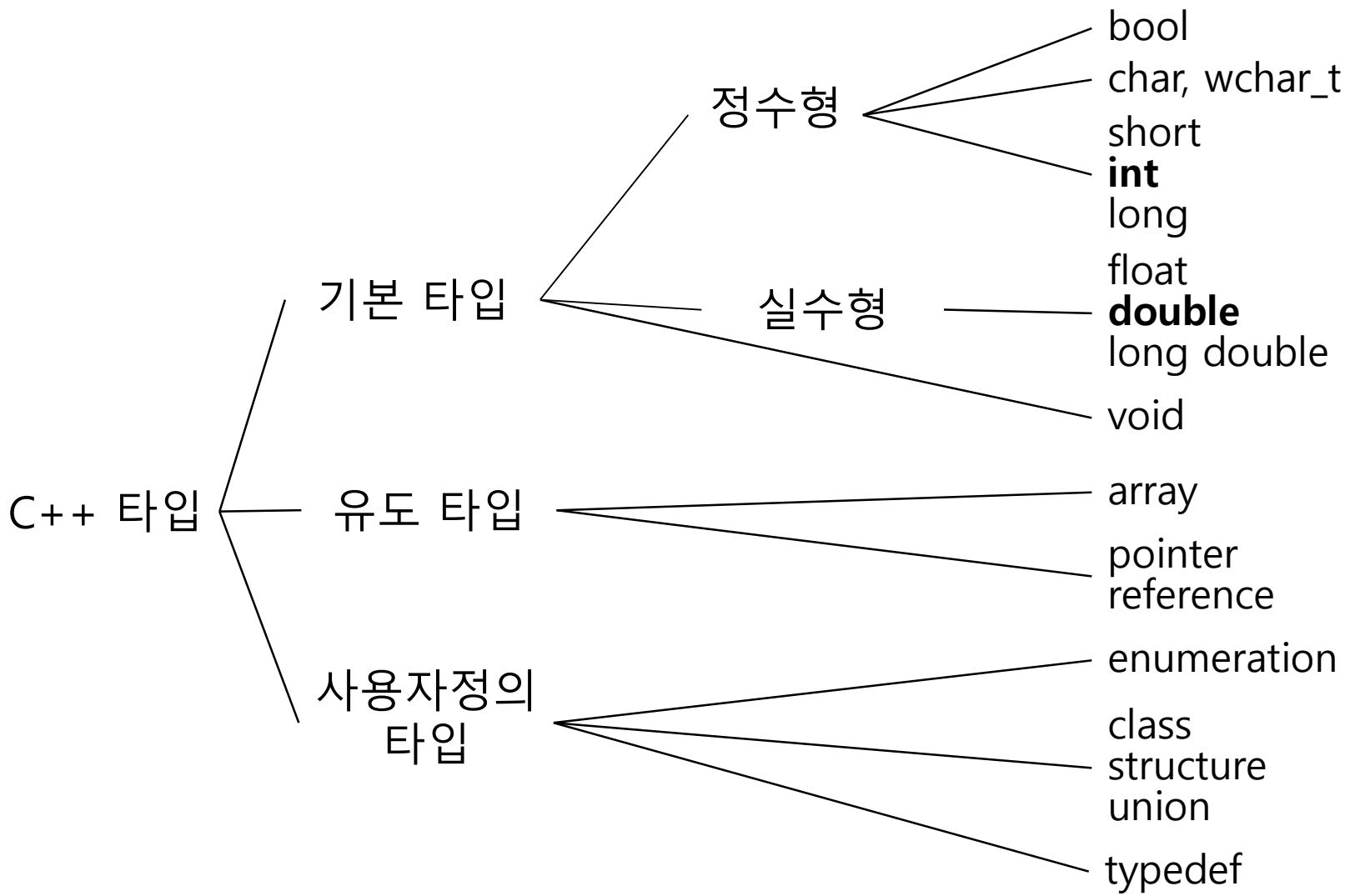
타입	변수	리터럴	상수
int	score	50 80	BASE_SCORE GOOD_SCORE
string	msg	"Enter your score: " "Good" "Not Good" "The result: " " is "	GOOD_MSG BAD_MSG

타입

- 데이터 타입의 역할
  - 해당 타입에 속하는 자료 값의 집합을 결정함
  - 해당 타입에 속하는 자료에 적용 가능한 연산을 결정함
  - 자료의 저장 형태를 결정함

- C++ 데이터 타입 분류
  - 기본 타입(primitive type): C++ 언어가 직접 지원하는 타입
  - 유도 타입(derived type): 다른 타입을 기반으로 하여 구성되는 타입
  - 사용자 정의 데이터 타입(user-defined data type): 구조체, 클래스 등

C++의 타입 종류


C++ 타입
기본 타입
정수형
bool
char, wchar_t
short
int
long
실수형
float
double
long double
void
유도 타입
array
pointer
reference
사용자정의 타입
enumeration
class
structure
union
typedef

```
// 기본 타입인 int, float, bool, char의 사용 예

#include <iostream>
using namespace std ;

int main() {
    int integerNumber, positiveNumber ;
    cin >> integerNumber >> positiveNumber ;

    bool zeroOrMore = integerNumber >= 0 ;

    float product ;
    char signChar ;
    if ( zeroOrMore ) {
        product = integerNumber * positiveNumber ;
        signChar = '+' ;
    }
    else {
        product = - integerNumber * positiveNumber ;
        signChar = '-' ;
    }
    cout << signChar << product << endl ;
}
```

변수의 정의

- 변수는 타입과 함께 정의됨

int baseScore ; int score ; float average ; float variance ; string helloMsg ; string hiMsg ;	int baseScore, score ; float average, variance ; string helloMsg, hiMsg ;
(a) 변수를 별도의 문장으로	(a) 동일 타입의 여러 변수 선언

- 변수는 정의와 함께 초기 값이 지정될 수 있음

int baseScore = 50 ; string helloMsg = "Hello" ; string hiMsg = "Hi" ;	int baseScore = 50 string helloMsg = "Hello", hiMsg = "Hi" ;
(a) 변수를 별도의 문장으로 초기화	(a) 동일 타입의 여러 변수를 초기화

조정자(Modifier)

- 수식어(조정자)
  - signed, unsinged, short, long은 수식어로 사용됨
  - 예: **short** == short int = **signed** short

Table 2.11 Fundamental Data Types

Basic Type	Modifier	Modifier
bool		
char	signed char	unsigned char
wchar_t		
int	short int	long int
unsigned	unsigned short	unsigned long
double	float	long double

signed, unsigned 예제

```
signed int temperature =-1 ; // int temperature와 동일

unsigned int numberOfStudent  = 1;

unsigned long pressure = 10;
```

- 음수 표현법(two's complement)
  - MSB(Most Significant Bit)가 0이면 양수, 1이면 음수
  - -1의 표현은 MSB를 1로, 나머지 bits를 반전 후 +1

```
signed char ch = 1; //00000001

std::cout << (~ch+1) << std::endl; //-1

//00000001   → 11111110 → 11111111
```

signed, unsigned

```
#include <iostream>
using namespace std ;
int main() {
    cout << "Enter unsigned integer and signed integer: " ;
    unsigned int unsignedInt ;
    signed int signedInt ;      // int signedInt와 동일
    cin >> unsignedInt >> signedInt ;
    cout << unsignedInt << endl ;
    cout << signedInt << endl ;
}
```

```
( 10) : 00000000000000000000000000001010
(-10) : 11111111111111111111111111110110
→ unsigned int 출력: 4294967286
```

(a) 실행 예1	Enter unsigned integer and signed integer: 10 10 10 10
(b) 실행 예2	Enter unsigned integer and signed integer: -10 -10 4294967286 -10

Good Design
변수는 실제로 사용되는 시점에 정의

```
int main() {
  int x1, x2, sum, product ;
  cout << "Enter two positive numbers: " ;
  cin >> x1 >> x2 ;
  if ( x1 <= 0 || x2 <= 0 ) {
     cout << "Not positive numbers₩n" ;
     return 0 ;
  }
  sum = x1 + x2 ;
  product = x1 * x2 ;
  cout << sum << endl ;
  cout << product << endl ;
}
```

정의된 위치와 사용되는 위치가 차이가 클 경우에는 프로그램을 이해하기가 어려움

```
int main() {
  cout << "Enter two positive numbers: " ;
  int x1, x2 ;
  cin >> x1 >> x2;
  if ( x1 <= 0 || x2 <= 0 ) {
     cout << "Not positive numbers₩n" ;
     return 0 ;
  }
  int sum = x1 + x2 ;
  int product = x1 * x2 ;
  cout << sum << endl ;
  cout << product << endl ;
}
```

변수는 최초로 사용되는 위치에 정의

식별자 (변수) 명명

- 식별자(identifier): 변수, 상수, 함수, 클래스 등 개발자가 결정한 이름
- C++ 명명 규칙
  - 식별자는 영문자, 숫자, 밑줄로 구성. 단 숫자는 첫문자로는 사용 안됨
  - 영문자는 대문자와 소문자를 구분함(case sensitive)
  - 키워드는 예약어(reserved words)로서 식별자로서 사용 불가

C++ Keywords

- 타입이름
- 데이터 값
- 명령어
- 연산자
- 수식어(modifier)

Table 2.2 Keywords

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

변수 이름의 예

	예	설명
명명 규칙 준수	int sum	적합한 변수 명
	int _sum	첫문자로 '_'가 허용됨
	int count, Count	대소문자가 구분됨
	string hello_msg	중간에 '_'가 허용됨
명명 규칙 위반	int **1**count	첫문자로 숫자를 사용할 수 없음
	int **bool**	bool은 C++의 타입 명 즉 키워드이다.
	int **true**, **false**	true와 false는 C++의 키워드이다.

Good Design: 바람직한 식별자 명명법

- 식별자 명명 규칙
  - 규칙 1: 대/소문자가 구분되기는 하지만, 대/소문자에 의해서 구분되는 식별자를 사용하지 않는다.
  - 규칙 2: 변수의 이름만으로 저장되는 값의 용도/역할을 추축할 수 있도록 해야 한다.
  - 규칙 3: 키워드는 아니지만, STL에서 정의된 표준 식별자는 사용하지 않도록 한다.
  - 규칙 4: '_' 등은 시스템 함수에서 사용하므로 주의해서 사용하도록 한다.
  - 규칙 5: 일반화되지 않은 약어의 사용을 자제한다.

위반 규칙	권장되지 않는 변수 명	설명
규칙 1	int count, Count	대문자와 소문자로 구분하는 변수명을 사용하지 않도록 한다.
규칙 2	int x123	x123만으로는 이 변수에 저장되는 값의 용도를 파악할 수가 없다.
규칙 3	int cin, cout	cin과 cout은 STL에서 표준으로 사용하는 이름이다.
규칙 4	int _sum, hello__msg	많은 시스템 함수가 첫문자로 '_'을 사용하거나 중간에 "__"가 사용되므로 이를 피하도록 한다.
규칙 5	int buf_Size int cName, sName	buf, c 등이 일반화된 약어가 아님; 원래 단어를 사용하는 것이 바람직하다. 대신에 bufferSize, cityName, schoolName를 변수명으로 사용하는 것이 바람직

타입의 크기

데이터 모델

32bit 시스템	ILP32 or 4/4/4 (int, long, and pointer are 32-bit)	Win32 API Linux, macOS
64bit 시스템	LLP64 or 4/4/8 (int and long are 32-bit, pointer is 64-bit)	Win64 API

타입 간의 크기 관계

정수형	1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
실수형	sizeof(float) <= sizeof(double) <= sizeof(long double)

최소 크기

타입	최소 크기	타입	최소 크기
char	1 바이트	long	4 바이트
short	2 바이트	long long	8 바이트
int	2 바이트		

https://en.cppreference.com/w/cpp/language/types

타입의 값 범위

- 값의 범위는 바이트 수에 따라서 결정됨

타입	크기	값의 범위
char	1 바이트	signed: -128 ~ 127 unsigned: 0 ~ 255
short int (short)	2 바이트	signed: -32768 ~ 32767 unsigned: 0 ~ 65535
int	4 바이트	signed: -2147483648 ~ 2147483647 unsigned: 0 ~ 4294967295
long int (long)	4 바이트	signed: -2147483648 ~ 2147483647 unsigned: 0 ~ 4294967295
bool	1 바이트	true or false
float	4 바이트	+/- 3.4e +/- 38 (~7 digits)
double	8 바이트	+/- 1.7e +/- 308 (~15 digits)
long double	8 바이트	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 바이트	1 wide character

Good Design: <cstdint>의 사용

- 대상 CPU에 따라서 타입의 바이트가 결정
- 예) 4 바이트 CPU에서의 정상 동작 코드가 2 바이트 CPU에서는 overflow될 수 있음

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after its lift-off.
The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.
It turned out that the cause of the failure was a software error in the inertial reference system. Specifically a **64 bit floating point number** relating to the horizontal velocity of the rocket with respect to the platform was converted to a **16 bit signed integer**.

Good Design: <cstdint>의 사용

- 바이트 수를 명시한 표준 타입의 사용
- C 언어: <stdint.h>
- C++ 언어: <cstdint>

signed	unsigned	설명
int8_t	uint8_t	8 비트 정수
int16_t	uint16_t	16 비트 정수
int32_t	uint32_t	32 비트 정수
int64_t	uint64_t	64 비트 정수

타입의 최대/최소값: <limits> in C

- sizeof(type)
  - returns the amount of memory need for the type in **bytes**
  - if you want to write for maximum portability, it is better to use sizeof
  - The single argument to sizeof is either a *type* or a variable
- 정수형 값 범위
  - 헤더파일 <climits>
  - INT_MAX, INT_MIN 등
- 실수형 값 범위
  - 헤더파일 <cfloat>
  - FLT_EPSILON, FLT_MIN, FLT_MAX 등

```
std::cout << "int: "          << sizeof(int)   <<std::endl;
std::cout << "long: "         << sizeof(long) <<std::endl;
std::cout << "int*: "         << sizeof(int*)   <<std::endl;
std::cout << "int max: "      << INT_MAX   <<std::endl;
std::cout << "float max: " << FLT_MAX   <<std::endl;
```

타입의 최대/최소값

❖ numeric_limits<T> 클래스

구분	방법	예
최소값	numeric_limits<T>::min()	numeric_limits<int>::min() numeric_limits<float>::min()
최대값	numeric_limits<T>::max()	numeric_limits<int>::max() numeric_limits<float>::max()

```
#include <limits>
cout << numeric_limits<int>::max() << " is the maximum int₩n";
```

```
#include <limits>          // numeric_limits<T>를 이용하기 위함
#include <iostream>
using namespace std ;

int main() {
 cout << numeric_limits<short int>::min() << endl ;                  -32768
 cout << numeric_limits<short int>::max() << endl << endl ;          32767

 cout << numeric_limits<int>::min() << endl ;                        -2147483648
 cout << numeric_limits<int>::max() << endl << endl ;                2147483647

 cout << numeric_limits<unsigned int>::min() << endl ;               0
 cout << numeric_limits<unsigned int>::max() << endl << endl ;       4294967295

 cout << numeric_limits<long>::min() << endl ;                       -2147483648
 cout << numeric_limits<long>::max() << endl << endl ;               2147483647

 cout << numeric_limits<float>::min() << endl ;                      1.17549e-038
 cout << numeric_limits<float>::max() << endl << endl ;              3.40282e+038

 cout << numeric_limits<double>::min() << endl ;                     2.22507e-308
 cout << numeric_limits<double>::max() << endl << endl ;             1.79769e+308

 cout << static_cast<int> (numeric_limits<char>::min()) << endl ;    -128
 cout << static_cast<int> (numeric_limits<char>::max()) << endl ;    127
}
```

리터럴(literal)

- 프로그램에서 사용되는 값 자체

```
const int GOOD_SCORE = 80 ;
int score ;
cin >> score ;
int final = score * 2 ;
string msg = “Hello” ;
```

리터럴과 타입

타입	리터럴 예	설명
int	100	십진수 100
	0100	8진수 100 즉 십진수 64
	0x100	16진수 100 즉 십진수 256
unsigned int	100**U**, 100**u**	접미어 U, u를 이용해서 unsigned int를 명시
long	100**L**, 100**l**	접미어 L, l을 이용해서 long int를 명시
unsigned long	100**ul**, 100**UL**	접미어 ul, UL을 이용해서 unsigned long을 명시
double	5.0	실수형 리터럴
	500e-2	
float	5.0**F**, 5.0**f**	접미어 F, f를 이용해서 float를 명시
long double	5.0**L**, 5.0**l**	접미어 L, l을 이용해서 long double를 명시
char	'5'	문자 '5'를 나타냄
char*	"5"	'5', '0'로 구성된 문자열

특수 char 문자열

리터럴	설명	예
'₩₩'	백슬래시를 뜻함	cout << "A**₩₩**B" A₩B를 출력함
'₩t'	탭 문자	cout << "이름" << '**₩t**' << "점수";
'₩n'	개행 문자	cout << '**₩n**' ;
'₩"	작은 따옴표(')	cout << "I**₩'**m ..." ;
'₩"'	큰 따옴표(")	cout << "**₩"**Hello **₩"**" ;
'₩0'	널(null) 문자	char* pName = '**₩0**' ;
'₩OOO'	8진수로 표시한 문자	'₩101' // 8진수 101 즉 'A'
'₩x0OO'	16진수로 표시한 문자	'₩x041' // 16진수 41 즉 문자 'A'

Good Design: 8진수 리터럴 사용은 자제

- 8진수 리터럴은 실수를 유발할 수 있으므로 사용이 권장되지 않음

```
int values[4] ;
values[0] = 200 ;   // 십진수 200
values[1] = 150 ;   // 십진수 150
values[2] = 100 ;   // 십진수 100
values[3] = 050 ;  // 십진수 40
```

```
int flag1 = 0, flag2 = 0, flag3 = 0 ;
flag1 |= 256 ;   // 1 0000 0000
flag2 |= 128 ;   //    1000 0000
flag3 |= 064 ;  //   0011 0100, 52
```

상수

- const 키워드로 변수를 상수로 선언함

```
const float PI = 3.14F ; // 상수 변수 PI의 정의
```

- 상수(constant)는
  - 초기화가 되어야 하고
  - 이후에는 값이 변경될 수가 없음

	오류 상황	설명
1)	const float PI ;	상수의 초기값이 주어지지 않았음
2)	PI = 3.141592F ;	상수의 값은 변경될 수 없음
3)	cin >> PI ;	상수의 값은 변경될 수 없음

상수의 사용

```
#include <iostream>
using namespace std ;
int main() {
  // const 변수 정의 방법; 초기화가 됨
  const float PI = 3.14F ;
  // 사용하는 예: 일반 변수처럼 사용할 수 있음; 단 값을 읽기만 할 수 있음
  float radius ;
  cin >> radius ;
  float area = PI * radius * radius ; // 상수 PI의 값을 읽는 것은 허용됨
  cout << "The area of a circle with radius " << radius
    << " is " << area << endl ;
  // 새로운 값을 대입하는 것은 허용되지 않음
  PI = 3.141592F ; // 'PI' : const인 변수에 할당할 수 없습니다.
  // cin >> PI ; // ERROR
}
```

Good Design: 매크로 대신에 상수를 사용

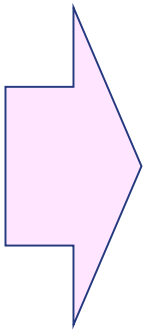
- # define 으로 매크로를 이용하여 상수를 정의하는 것보다는 const 로 상수를 정의함

매크로 사용	상수 사용(권장)
# define PI 3.14F	const float PI = 3.14F ;

Good Design: 리터럴 대신에 상수를

- 가독성과 유지보수성을 위해서 리터럴 대신에 상수를 사용

```
char ch ;
cin >> ch ;
int x = 0 ;
if ( ch == 'R' )
  x = 10 ;
else if ( ch == 'C' )
  x = 20 ;
if ( ch == 'R' ) x ++ ;
```



```
const char RECTANGLE_CODE = 'R' ;
const char CIRCLE_CODE = 'C' ;
char ch ;
cin >> ch ;
int x = 0 ;
if ( ch == RECTANGLE_CODE )
  x = 10 ;
else if ( ch == CIRCLE_CODE )
  x = 20 ;
if ( ch == RECTANGLE_CODE ) x ++ ;
```

형 변환

- 값은 자신과 일치하는 타입의 변수에 저장되어야 함
- 일치하지 않는 경우 해당 타입에 맞추어 값이 변경됨

```
float average = 100 ;       // int에서 float로의 변환이 필요
short int v = 100 ;         // int에서 short int로의 변환이 필요
```

- 형 변환 방법의 종류
  - 묵시적(implicit) 방법
  - 명시적(explicit) 방법

묵시적 형 변환

- 산술식에서의 자동(implicit) 형 변환: data loss가 없음
  - bool, char, short, enum ➔ int ➔ unsigned
  - int < unsigned < long < unsigned long
    < float < double < long double

예	설명
long sum = 200 ;	int ➔ long 200은 원래 int 타입이지만 sum에 저장하기 위하여 long 타입으로 변환시킨다.
float average = 100 ;	int ➔ float 100은 원래 int 타입이지만 average에 저장하기 위하여 float 타입으로 변환시킨다.
char c = 41 ;	int ➔ char 41은 원래 int 타입이지만 c에 저장하기 위하여 char 타입으로 변환시킨다. 41이 char 타입의 범위에 속하므로 형 변환이 됨

묵시적 형 변환

- 자료 손실 가능성이 있는 경우 컴파일러가 경고함

예	설명
int i1 = 100.5F ;	float ➔ int
int i2 = numeric_limits<float>::min() ;	float ➔ int
float f = numeric_limits<double>::min() ;	double ➔ float
short int short1 = 1000000 ;	int ➔ short int

명시적 형 변환

- 명시적으로 형 변환을 지정함

```
char c = 'A' ;
int i1 = static_cast<int>(c) ;        // char → int로 명시적으로 형변환
```

```
int n1 = 10 , n2 = 20  ;
float divide = static_cast<float>(n1) / n2 ;
```

```
int main() {
    char charVal ;
    cin >> charVal ;

    // 1) promotion: 자료 손실이 없는 묵시적 변환
    // char ==> short int promotion
    short int shortValue = charVal ;
    // short int ==> int promotion
    int intValue = shortValue ;
    // int ==> long promotion
    long  longValue = intValue ;
    cout << longValue << endl ;

    // 2) 묵시적 변환과 명시적 변환
    int integerNumber ;
    float floatNumber ;

    cin >> integerNumber ;
    // 묵시적 변환: 경고; 그러나 위험스럽지는 않음
    floatNumber = integerNumber ;
    // 명시적 변환: 경고 없음; static_cast<T>를 이용하여 명시적 변환을 함
    floatNumber = static_cast<float> (integerNumber) ;
    cout << floatNumber << endl ;

    cin >> floatNumber ;
    // 묵시적 변환: 경고; 위험스러움
    // numeric_limits<int>::max() 이상의 값이 입력된 경우 문제가 발생함
    integerNumber = floatNumber ;
    // 명시적 변환: 경고 없음; 그러나 여전히 위험스러움
    integerNumber = static_cast<int> (floatNumber) ;
    cout << integerNumber << endl ;
}
```

```
a
97
5
5
2.3
2
```

명시적 형 변환 방법

방법	예
C 언어	(float) i
구형 C++	float (i)
표준 C++	static_cast<float>(i)

Good Design: 명시적 형 변환은 자제

- 명시적 형 변환을 함으로써 앞서 묵시적 형 변환시 발생하였던 경고를 회피할 수가 있음

```
int i1 = static_cast<int>(100.5F) ;
```

- 자료 손실이 발생하고 결국은 프로그램의 오동작이 야기될 수가 있음

(추가) Initialization Motivation

- Direct initialization
  - 초기값을 직접 전달
- Copy initialization
  - = 로 초기화, explicit 이면 불가 (C++17)
- Default initialization
  - 일치하는 생성자가 정의됐을 때 객체를 초기화
- Zero initialization
  - 객체가 0으로 초기화, 전역(global), 정적(static) 변수에서 사용
- Value initialization
  - 객체는 항상 값을 가짐 (생성자나 0)
- List initialization
  - 객체가 { } (curly brace)로 초기화, 객체는 항상 값을 가짐
- Aggregate initialization
  - 타입이 aggregate이면, List initialization의 특수한 형태
- **→ 초기화를 일관된 방법으로 해보자! (하위 호환성을 가지고)**

```
#include <iostream>
int main() {
    int i (42);    //direct initialization
    int j {42};    //direct list initialization
    int k=42;    //copy initialization
    int l={42};   //copy list initialization
}
```

Brace Initialization (Since C++11)

- The uniform initialization of variables was supported with **{ }**

```
#include <map>
#include <vector>
#include <string>
int main() {
    int i{};                        // i becomes 0
    double d{};                     // d becomes 0.0

    std::string s{};                // s becomes ""
    std::vector<float> v{};         // v becomes an empty vector

    int intArray[] {1,2,3,4,5};
    std::vector<int> intArray1{1,2,3,4,5};

    int v1{10}, v2{20};
    std::vector<int> intArray2{v1, v2};      // can be initialized with variables

    std::map<std::string, int> myMap{{"Kim",1976}, {"Park",1972}};

    const float* pData= new const float[3] {1.1, 2.2, 3.3}; // pData[i] = XXX not allowed
}
```

Brace Initialization (Since C++11)

- Preventing narrowing
  - Narrowing or more precise narrowing conversion is a implicit conversion of arithmetic values including a loss of accuracy.

```
#include <iostream>
int main() {
    char c1(999);                                   // WARNING
    char c2 = 999;                                  // WARNING
    char c3{999};                                   // ERROR
    std::cout << "c1: " << c1 << std::endl;
    std::cout << "c2: " << c2 << std::endl;

    int i1(3.14);                                   // WARNING
    int i2 = 3.14;                                  // WARNING
    int i3{3.14};                                   // ERROR
    std::cout << "i1: " << i1 << std::endl;
    std::cout << "i2: " << i2 << std::endl;
}
```

(추가) Consequences of Uniform Initialization

- Couple of ways to initialize an int

```
int  i1;                    // undefined value
int  i2 = 42;               // note: inits with 42
int  i3(42);                // inits with 42
int  i4 = int();            // inits with 0
int  i5{42};                // inits with 42
int  i7{};                  // inits with 0
int i6 = {42};              // inits with 42
int i8 = {};                // inits with 0
auto  i9 = 42;              // inits int with 42
auto  i10 {42};             // // unless old compiler
auto  i11 = {42};           // inits std::initializer_list<int with 42
auto  i12 = int {42};       // inits int with 42
// don't use ( ) in initializations
int  i13();                 // declares a function
int  i14(7, 9);             // compile-time error
int  i15 = (7, 9);          // OK, inits int with 9 (comma operator)
int  i16 = int(7, 9);       // compile-time error
auto i17 (7, 9);            // compile-time error
auto i18 = (7, 9);          // OK, inits int with 9 (comma operator)
auto i19 = int(7, 9);       // compile-time error
```

is_same (Since C++11)

- Check whether the given two types are the same.

```
#include <iostream>
#include <type_traits>
#include <cstdint>
using std::cout;
using std::is_same;

void print_separator() { cout << "-----₩n"; }
int main() {
    cout << std::boolalpha;

    cout << is_same<int, int32_t>::value << '₩n';   // true
    cout << is_same<int, int64_t>::value << '₩n';   // false
    cout << is_same<float, int32_t>::value << '₩n'; // false
    print_separator();

    cout << is_same<int, int>::value << "₩n";                  // true
    cout << is_same<int, unsigned int>::value << "₩n";      // false
    cout << is_same<int, signed int>::value << "₩n";        // true
    print_separator();

    cout << is_same<char, char>::value << "₩n";                // true
    cout << is_same<char, unsigned char>::value << "₩n"; // false
    cout << is_same<char, signed char>::value << "₩n";     // false
}
```

지역변수와 전역변수의 정의 위치

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

int globalVal ; // 전역(global) 변수

void print(int v) {
    cout << v << '₩t' << globalVal << endl ; // 10 55
}
int main() {
    cout << "Hello, C++ ! " << endl ;

    int localVal1 = 100 ; // 지역(local) 변수
    for ( unsigned int i = 0 ; i < 10 ; i ++ ) {
        int localVal2 = i + 1 ; // 지역(local) 변수
        localVal1 = localVal2 ;
        globalVal += localVal2 ;
    }
    print(localVal1) ; // 10 55
}
```

블록 내부의 임의의 위치에서 변수의 정의가 가능

영역(scope)

- 변수가 사용될 수 있는 프로그램 상에서의 부분
- 전역 변수의 영역
  - 자신이 선언/정의된 지점부터
  - 파일의 끝까지
- 지역 변수의 영역
  - 자신이 정의된 지점부터
  - 자신이 정의된 블록의 끝까지

지역변수와 전역변수의 영역(scope)

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

int globalVal ;                                          globalVal의 영역

void print(int v) {
    cout << v << '₩t' << globalVal << endl ; // 10 55
}
int main() {
    cout << "Hello, C++ ! " << endl ;

    int localVal1 = 100 ;                                localVal1의 영역
    for ( unsigned int i = 0 ; i < 10 ; i ++ ) {
        int localVal2 = i + 1 ;                          localVal2의 영역
        localVal1 = localVal2 ;
        globalVal += localVal2 ;
    }
    print(localVal1) ; // 10 55
}
```

영역의 가림(hiding)

```
#include <iostream>
using namespace std ;
// 전역 변수 intVal의 정의
int intVal = -100 ;

int main() {
  // 전역 intVal을 가림
  int intVal = 100 ;

  int intResult ;
  if (intVal >= 0 ) { // 지역 intVal(100)을 가리킴. 전역변수 intVal의 가림
    intResult = intVal * 20 ; // 지역변수 intVal ; 즉 100 * 20 임
    intResult += ::intVal ; // 전역변수 intVal ; 즉 intResult = 2000 – 100 임
  }
  else {
    int intVal = intVal ; // 8행의 지역변수와 전역변수 intVal을 모두 가림
    // 결정되지 않은 자신의 값으로 자신을 초기화하므로 경고임
    intResult = intVal * 10 ; // 16행의 지역변수 intVal을 가리킴
  }
  // 8행의 지역변수 intVal을 가리킴
  cout << intVal << '₩t' <<intResult  << endl ; // 100  1900
}
```

내부 블록의 변수에 의해서 외부 블록 또는 전역 변수를 접근할 수가 없는 문제

Good Design
의미 있는 변수명으로 변수 충돌 회피

- C++ 언어에서는 영역 연산자 즉 '::' 을 이용해서 전역 변수를 항상 지칭이 가능함

- 근본적인 것은 실제로 전역 변수 및 지역 변수가 동일한 이름을 가져야 하는 지를 점검

- **변수가 저장하는 정보를 정확하게 뜻하는 용어를 변수 이름으로 사용한다면 위와 같이 지역/전역 변수의 이름 충돌 문제를 회피하는 것이 바람직함**

- 대규모 프로그램에서는 **네임스페이스**를 이용해서 전역 변수/함수 등을 체계적으로 관리함으로써 이름 충돌 문제를 회피

초기화

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

int globalIntVal ;        // int globalIntVal = 0과 동일
bool globalBoolVal ;    // bool globalBoolVal = false와 동일
float globalFloatVal ;    // float globalFloatVal = 0.0F와 동일
int main() {
  cout << globalIntVal << endl ;              // 0
  cout << globalBoolVal << endl ;             // 0(false)
  cout << globalFloatVal << endl ;            // 0.0
  // 초기화되지 않은 지역 변수의 garbage 값이 출력됨
  int localVal ;
  cout << localVal ;
}
```

전역 변수가 차지하는 공간의 값이 프로그램 실행 시 자동으로 초기값으로 사용

지역 변수의 값을 자동으로 초기화되지 않는다

정적 지역 변수

- static local variable
  - 해당 변수의 공간이 함수의 수행이 종료된 후에도 유효하다.
  - 그러므로, 함수가 종료된 후에도 그 값이 유지되는 효과가 있다

```
int print( int val ) {
    static int sum = 0 ;     // static 지역 변수 sum
    sum += val ;
    return sum ;
}
int main() {
    cout << print(1) << endl ;       // 1=(0+1)
    cout << print(2) << endl ;       // 3=(1+2)
    cout << print(3) << endl ;       // 6=(3+3)
}
```

지역변수와 전역변수: 요약

	지역 변수	전역 변수
정의 위치	함수의 내부	함수의 외부
영역(scope)	정의된 위치부터 블록의 끝까지	정의/선언된 위치부터 파일의 끝까지
메모리 확보 시점	함수 호출시	프로그램 시작시
초기화	초기화 안 됨	초기화 됨
메모리 할당 위치	프로그램 스택	프로그램 데이터 공간