Topic 2 Types

내용

- 타입, 변수, 리터럴, 상수
- 형 변환
- string 클래스
- 지역 변수와 전역 변수
- 포인터
- 포인터와 상수
- 배열
- vector 클래스
- Iterator와 알고리즘 함수
- 참조
- 나열형(enum)
- 구조체(struct)
- 공용체(union)
- typedef

타입, 변수, 상수, 리터럴

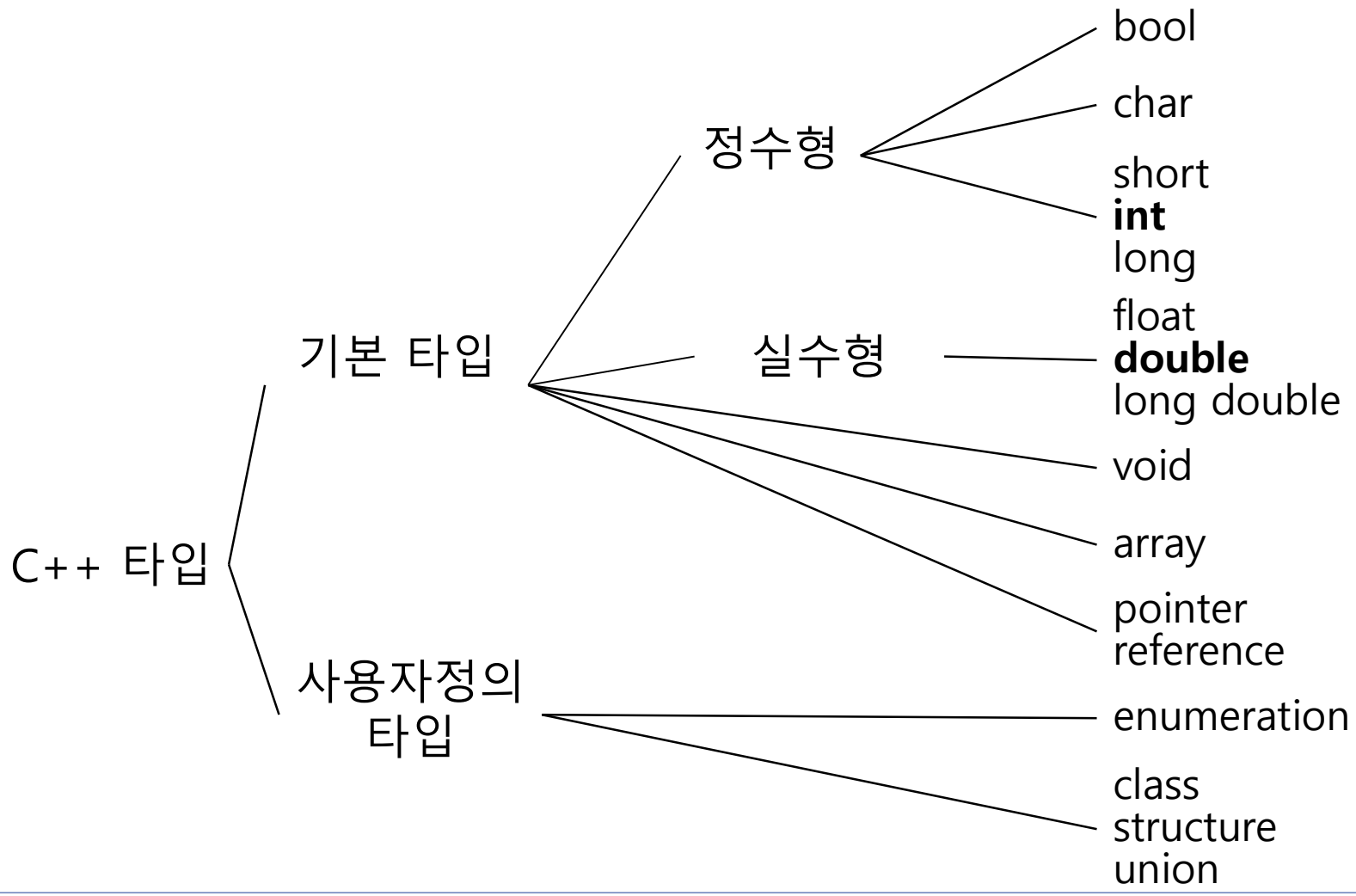
```
# include <iostream>
# include <string>
using namespace std ;
// 상수
const int BASE_SCORE = 50 ;
const int GOOD_SCORE = 80 ;
const string GOOD_MSG = "Good" ;
const string BAD_MSG = "Not Good" ;
int main() {
    cout << "Enter your score: " ;          // 문자열 리터럴
    int score ;                             // 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 저장형태를 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)도 유도 데이터 타입의 일종임

C++의 타입 종류


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

```
// 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

- 수식어(조정자)
  - signed, unsinged, short, long은 수식어로 사용되기도 함
  - 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

```
unsigned int numberOfStudent ;

unsigned long pressure ;

signed int temperature ;              // int temperature와 동일
```

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 ;
}
```

(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 >> op ;
  if ( x1 <= 0 || x2 <= 0 ) {
     cout << "Not positive numbers₩n" ;
     return 0 ;
  }
  sum = x1 + x2 ;
  product = x1 * x2 ;
  cout << sum << end ;
  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 >> op ;
  if ( x1 <= 0 || x2 <= 0 ) {
     cout << "Not positive numbers₩n" ;
     return 0 ;
  }
  int sum = x1 + x2 ;
  int product = x1 * x2 ;
  cout << sum << end ;
  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

- 63개의 키워드

- 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 명의 예

	예	설명
적절한 예	int sum	적합한 변수 명
	int _sum	첫문자로 '_'가 허용됨
	int count, Count	대소문자가 구분됨
	string hello_msg	중간에 '_'가 허용됨
부적절한 예	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. 개발자가 대/소문자를 구분하여 변수를 기억하기는 매우 어렵다.
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 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

타입의 값 범위

타입 간의 크기 관계

정수형	1 = sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
실수형	sizeof(float) <= sizeof(double) <= sizeof(long double)
singed / unsigned	sizeof(N) = sizeof(singed N) = sizeof(unsigned N) N = char, short int, int, or long int

최소 크기

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

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

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 되어야하고
  - 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

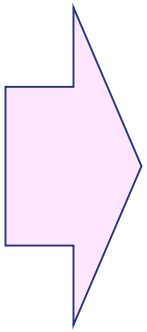
- # define 으로 매크로를 이용하여 상수를 정의하는 것보다는 const 로 상수를 정의함

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 ;

    // 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 ;
}
```

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

STL string 클래스

```
#include <string>          // <string>을 반드시 포함해야 한다.
#include <iostream>
using namespace std ;
int main() {
    // 초기화
    string strGreeting = "Hello, C++ World !" ;
    // 출력: cout를 이용
    cout << strGreeting << endl ;
    // 길이 조회: length() 멤버 함수
    // 문자 접근: [ ] 연산자
    for ( unsigned int i = 0 ; i < strGreeting.length() ; i ++ )
        cout << strGreeting[i] ;
    // 입력: cin을 이용
    string strVal1, strVal2 ;
    cin >> strVal1 >> strVal2 ;
    // 비교: compare() 멤버 함수
    if ( strVal1.compare(strVal2) == 0 )
        strGreeting += " Same." ; // 문자열 병합: += 연산자
    else
        strGreeting += " Different." ;
}
```

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

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 ;

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

     int localVal1 = 100 ;
     for ( unsigned int i = 0 ; i < 10 ; i ++ ) {    localVal1의 영역
          int localVal2 = i + 1 ;
          localVal1 = localVal2 ;                    localVal2의 영역
          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 회의

- 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 쉽게 발생하지 않을 것**

- 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 << sum(1) ;         // 1
    cout << sum(2) ;         // 3
    cout << sum(3) ;         // 6
}
```

포인터

- 이미 존재하는 다른 변수를 가리키는(point)는 역할을 하는 변수
- 다른 변수에 대한 주소를 값으로서 가짐

항목	예	설명
포인터 변수의 정의	int* pIntVal ;	int 타입 변수를 가리킬 수 있는 포인터 변수 pIntVal의 정의
포인터 변수의 값 대입	int intVal = 100 ; pIntVal = &intVal ;	pIntVal 변수는 intVal 변수를 가리킴. 즉 intVal 변수의 주소를 저장함
원 변수 값의 조회	cout << *pIntVal ;	*pIntVal은 저장된 주소가 가리키는 공간에서 int 값을 구함. 즉 *pIntVal은 intVal과 동일한 값 100이 된다.

포인터 변수의 정의

- T 타입에 대한 포인터: T*

```
int* pIntVal ;                  // int 타입 변수에 대한 포인터
float* pFloatVal ;              // float 타입 변수에 대한 포인터
```

- 포인터 변수 값의 대입

```
int intVal ;
int* pIntVal1 = &intVal ;           // 다른 변수의 주소 조회 후 대입
int* pIntVal2 = pIntVal1 ;          // 다른 포인터 변수의 값 대입
int* pIntArray = new int[100] ;     // new로 할당된 메모리 주소 대입
```

포인터 변수의 값 대입

- 포인터 변수는 주소를 저장함

```
int intVal = 10 ;
int* pIntVal = &intVal ;    // intVal 변수의 주소가 pIntVal에 저장됨
```

	주소	값
	...	...
	...	...
intVal	**0x1000**	10
	...	...
	...	...
pIntVal	...	**0x1000**

포인터 변수의 값 대입

- 포인터 변수는 주소를 대입

```
int intVal ;
int* pIntVal1 = &intVal ;              // 다른 변수의 주소 조회 후 대입
int* pIntVal2 = pIntVal1 ;             // 다른 포인터 변수의 값 대입
int* pIntArray = new int[100] ;        // new로 할당된 메모리 주소 대입
```

원 변수 값의 조회 및 변경

- ***포인터변수: 원 변수 값의 조회 및 변경

```
int intVal = 10 ;
int* pIntVal = &intVal ;   // pIntVal 은 intVal을 가리킴
cout << *pIntVal ;          // 10
intVal ++ ;
cout << *pIntVal ;         // 11
(*pIntVal) ++ ;              // pIntVal이 가리키는 즉 intVal의 값 1증가
cout << intVal ;           // 12
```

포인터 변수의 정의와 사용 예

```
#include <iostream>
using namespace std ;
int main() {
  int intVal = 100 ;
  int* pIntVal = &intVal ;   // 포인터 변수 pIntVal의 정의와 intVal 주소 대입
  cout << intVal << ' ' << *pIntVal << endl ; // 100 100
  intVal += 100 ;
  cout << intVal << ' ' << *pIntVal << endl ;  // 200 200
  *pIntVal = *pIntVal + 10 ;  // *pIntVal을 변경하므로 intVal도 변경됨
  cout << intVal << ' ' << *pIntVal << endl ; // 210 210
}
```

동적 할당: C 언어

```
#include <stdio.h>
#include <stdlib.h>
int main () {
   int length ;
   char * name ;
   printf ("How long do you want the string? ");
   scanf ("%d", &length);
   name = (char*) malloc (length+1);     // malloc() 함수를 이용한 메모리 할당
   scanf( "%s", name) ;
   printf("%s₩n", name) ;
   free (name);                          // free() 함수를 이용한 메모리 해제
}
```

C++ 동적 할당: new와 delete

```
#include <iostream>
using namespace std ;
int main() {
   cout << "Enter the length of a name" << endl ;
   int length ;
   cin >> length ;
   char* name = new char[length+1] ;    // new를 이용한 메모리 할당
   cin >> name ;
   cout << name << endl ;
   delete [] name ;                     // delete를 이용한 메모리 해제
}
```

메모리 할당: new

- T* = new T ;
- T* = new T[size] ;

예	설명
char* name = **new char[10]** ;	char 타입 10개 변수의 메모리 할당
int size ; cin >> size ; string* names = **new string[size]** ;	개수를 실행시에 결정할 수도 있음; 즉 size는 프로그램이 실행될 때 사용자가 입력한 값에 의해서 결정됨
char* theChar = **new char** ; string* theName = **new string** ;	한 개의 변수도 할당할 수가 있음

메모리 해제: delete

- delete p ;
- delete [] p ;

메모리 할당	메모리 해제
char* name = new char[10] ;	delete [] name ;
int size ; cin >> size ; string* names = new string[size] ;	delete [] names ;
char* theChar = new char ; string* theName = new string ;	delete theChar ; delete theName ;

C 언어와 C++ 언어의 비교

	C 언어	C++ 언어
할당	int* pIntArray = (int*) malloc (size);	int* pIntArray = new int[size] ; int* pInt = new int ;
해제	free(pIntArray) ;	delete [] pIntArray ; delete pInt ;

상수에 대한 포인터

- 포인터가 가리키는 값을 변경하지 못하는 포인터

	일반적인 포인터	상수에 대한 포인터
정의	int* pIntVal = &intVal ;	const int* pcIntVal = &intVal ;
원 변수 값 조회	cout << *pIntVal ;	cout << *pcIntVal ;
원 변수 값 변경	**(*pIntVal) ++ ; // 허용**	**(*pcIntVal) ++ ; // 불허**

- 상수에 대한 포인터의 활용 예

```
size_t strlen ( const char * str );
char* strcat ( char* destination, const char* source) ;
const char * strstr ( const char * str1, const char * str2 );
```

Good Design: 상수에 대한 포인터의 활용

- 포인터가 가리키는 변수의 값이 변경되지 않아야 한다면 상수에 대한 포인터를 항상 사용하는 것이 권장

```
int countUppercase(const char* str, const int length) {
   int count = 0 ;
   for ( int i = 0 ; i < length ; i ++ )
      if ( str[i] >= 'A' && str[i] <= 'Z' ) count ++ ;
   return count ;
}
int main() {
   char* msg = "Hello World" ;
   cout << countUppercase(msg, strlen(msg)) ; // 2
}
```

상수 포인터

- 일반 포인터는 다른 주소를 대입함으로써 새로운 변수를 가리키는 것이 가능함

```
int intVal1 = 100, intVal2 = 200 ;
int* pIntVal = &intVal1 ;
pIntVal = &intVal2 ; // pIntVal이 다른 변수 intVal2를 가리킨다.
```

- **상수 포인터**는 초기화된 후에 다른 주소를 대입하는 것이 불가능함

```
int intVal1 = 100, intVal2 = 200 ;
int* const pIntVal = &intVal1 ;
pIntVal = &intVal2 ; // ERROR
```

상수 포인터

- 일반 포인터: 다른 주소를 대입하여 다른 변수를 가리키는 것이 가능
- 상수 포인터: 초기화 한 후에 다른 주소를 대입시키는 것을 불가능

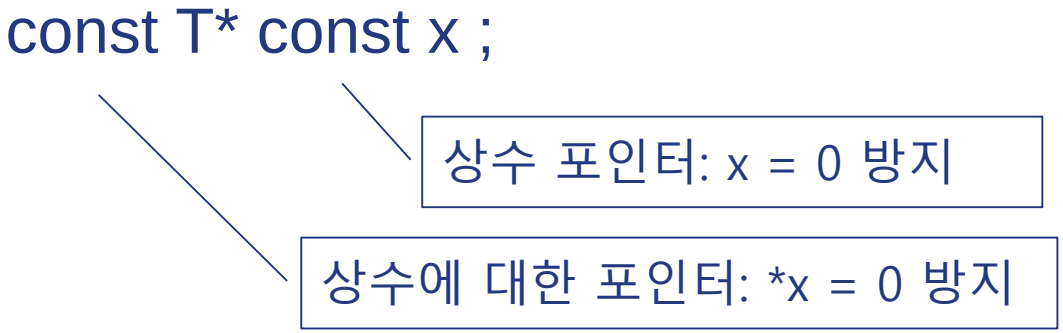
	일반 포인터	상수 포인터
정의	int* pIntVal = &intVal1 ;	int* **const** cpIntVal = &intVal1 ;
원 변수 값 조회	cout << *pIntVal ;	cout << *cpIntVal ;
원 변수 값 변경	(*pIntVal) ++ ;	(*cpIntVal)++ ;
초기화 필수 여부	// 초기화 필수 아님 int* pIntVal1 ; // 허용	**// 초기화 필수** int* const cpIntVal1 ; // 불허
다른 주소의 대입	pIntVal = &intVal2 ; // 허용	cpIntVal = &intVal2 ; **// 불허**

Good Design: new와 상수 포인터

- new 연산자는 상수 포인터와 함께 사용한다

```
int intVal = 200 ;
int size ;
cin >> size ;
int* const pIntArray = new int[size] ;
pIntArray = &intVal ;        // name이 상수 포인터이므로 불허됨
delete [] pIntArray ;
```

상수에 대한 상수 포인터




```
int intVal1 = 100, intVal2 = 200 ;
const int* const pIntVal = &intVal1 ;
cout << *pIntVal ; // 원 변수값 조회: OK
*pIntVal = 500 ; // ERROR: 상수에 대한 포인터 이므로 원 변수 값 변경 불허
pIntVal = &intVal2 ; // ERROR: 상수 포인터 이므로 다른 주소의 대입 불허
```

포인터와 상수 요약

	일반 포인터	상수에 대한 포인터	상수 포인터	상수에 대한 상수 포인터
정의 방법	int* pIntVal	const int* pIntVal	int* const pIntVal=&intVal1	const int*const pIntVal=&intVal1
초기화 필수 여부	필수 아님	필수 아님	필수	필수
원 변수 값 조회 cout << *pIntVal ;	OK	OK	OK	OK
원 변수 값 변경 *pIntVal ++ ;	OK	ERROR	OK	ERROR
다른 주소의 대입 pIntVal = &intVal2	OK	OK	ERROR	ERROR

Good Design: 참조 변수의 사용

- 상수 포인터 대신에 참조를 사용하는 것이 권장

```
int intVal = 100 ;
int * const cpIntVal = &intVal ;      // C/C++ 언어에서의 상수 포인터
int& rIntVAl = intVal;                // C++ 언어에서의 참조 변수
```

```
int main() {
  int intVal1 = 10, intVal2 = 20 ;

  // 일반 포인터
  int* pIntVal ;
  pIntVal = &intVal1 ;                  // ok
  *pIntVal += 10 ;                      // ok

  // 상수에 대한 포인터
  const int* pcIntVal = &intVal1 ;
  *pcIntVal += 10 ;                     // error; pcIntVal은 상수에 대한 포인터이므로
  pcIntVal = &intVal2 ;                 // ok

  // 상수 포인터
  int *const cpIntVal = &intVal1 ;
  *cpIntVal += 10 ;                     // ok
  cpIntVal = &intVal2 ;                 // error; cpIntVal은 상수 포인터이므로

  // 상수에 대한 상수 포인터
  const int *const ccIntVal = &intVal1 ;
  *ccIntVal += 10 ;                     // error; ccIntVal은 상수에 대한 포인터이므로
  ccIntVal = &intVal2 ;                 // error; ccIntVal은 상수 포인터이므로
}
```

배열

- 동일 타입의 변수 집합
- 정의 방법: T 변수명[n]

```
int values[5] = {0, 10, 20, 30, 40} ;
```

values

values[0]	values[1]	values[2]	values[3]	values[4]
0	10	20	30	40

배열의 기본 개념

```
#include <iostream>
using namespace std ;
const int SIZE = 3 ;
int main() {
  // 배열 정의와 초기화
  int intArray[SIZE] = {10, 20, 30} ;
  int sum = 0 ;
  for ( int i = 0 ; i < SIZE ; i ++ ) {
    sum += intArray[i] ; // 원소의 접근
  }
  cout << sum << endl ;
}
```

1차원 배열의 원소 접근 방법

- 배열의 변수 이름은 첫번째 원소에 대한 포인터로 간주

인덱스를 이용한 접근	포인터를 이용한 접근
values[0]	*values
values[1]	*(values+1)
values[2]	*(values+2)
…	…
values[i]	*(values+i)

배열의 초기화

- 배열의 원소 값은 배열을 정의할 때 초기화될 수 있다

```
int intArray1[3] = {1, 2, 3} ;
```

- 배열의 크기는 초기화 목록의 크기로 결정된다

```
int intArray2[] = {1, 2, 3, 4} ;
// int intArray2[4] = {1, 2, 3, 4}와 동일
```

- 지정되지 않은 원소의 초기값은 0으로 결정

```
int intArray3[3] ={1, 2} ;
// int intArray3[3] = {1, 2, 0}과 동일
```

배열의 초기화

- 문자의 배열은 문자열 리터럴을 이용하여 초기화

```
char strArray1[] = “a string“ ;
// char strArray1[] = {'a', ' ', 's', 't', 'r', 'i', 'n', 'g', 0} ; 과 동일
```

- 클래스 객체의 배열도 가능하다

```
string names[5] = {string(“Kim”), string(“Park”)} ;
```

포인터의 배열

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

const int SIZE = 3 ;
int main() {
  int intArrayA[SIZE] ;
  for ( int i = 0 ; i < SIZE ; i ++ )
  {
    cin >> intArrayA[i] ;
    intArrayA[i] += 10 ;
  }
```

```
  // 포인터의 배열 정의
  int* pIntArrayB[SIZE] ;
  int intVal1, intVal2, intVal3 ;
  cin >> intVal1 >> intVal2 >> intVal3 ;

  pIntArrayB[0] = &intVal1 ;
  pIntArrayB[1] = &intVal2 ;
  pIntArrayB[2] = &intVal3 ;
  for ( int i = 0 ; i < SIZE ; i ++ ) {
    int a = intArrayA[i] ;
    int b = *pIntArrayB[i] ;
    int sum = a + b ;
    cout << a << " + " << b <<
      " = " << sum << endl ;
  }
}
```

Good Design: 올바른 인덱스의 사용

- 크기가 N인 배열의 인덱스: 0 .. N-1

올바른 인덱스 사용 예	부정확한 인덱스 사용 예
for (int i = 0 ; i < N ; i ++)	for (int i = 0 ; i <= N ; i ++)

2차원 배열

- 2차원 배열: 다른 배열을 원소로 가짐
- 정의 방법: T 변수명[행수][열수]
- 예) int values[3][5]

	1열	2열	3열	4열	5열
1행	values[0,0]	values[0,1]	values[0,2]	values[0,3]	values[0,4]
2행	values[1,0]	values[1,1]	**values[1,2]**	values[1,3]	values[1,4]
3행	values[2,0]	values[2,1]	values[2,2]	values[2,3]	values[2,4]

```
#include <iostream>
using namespace std ;
const int ROW_SIZE = 3 ;
const int COLUMN_SIZE = 5 ;
int main() {
  int intArrayA[ROW_SIZE][COLUMN_SIZE] ;
  for ( int i = 0 ; i < ROW_SIZE ; i ++ )
    for ( int j = 0 ; j < COLUMN_SIZE ; j ++ )
      intArrayA[i][j] = i + j ;

  int intArrayB[ROW_SIZE][COLUMN_SIZE] ;
  for ( int i = 0 ; i < ROW_SIZE ; i ++ )
    for ( int j = 0 ; j < COLUMN_SIZE ; j ++ )
      intArrayB[i][j] = i * j ;

  cout << "행렬 A + B는" << endl ;
  for ( int i = 0 ; i < ROW_SIZE ; i ++ ) {
    for ( int j = 0 ; j < COLUMN_SIZE ; j ++ ) {
      int a = intArrayA[i][j] ;
      int b = intArrayB[i][j] ;
      int sum = a + b ;
      cout << a << " + " << b << " = " << sum << '₩t' ;
    }
    cout << endl ;
  }
}
```

```
행렬 A + B는
0 + 0 = 0    1 + 0 = 1    2 + 0 = 2    3 + 0 = 3    4 + 0 = 4
1 + 0 = 1    2 + 1 = 3    3 + 2 = 5    4 + 3 = 7    5 + 4 = 9
2 + 0 = 2    3 + 2 = 5    4 + 4 = 8    5 + 6 = 11    6 + 8 = 14
```

2차원 배열의 원소 접근 방법

인덱스를 이용한 접근	포인터를 이용한 접근
values[0][0]	*values
values[0][1]	*(values+1)
values[i][0]	*(values + (5 * i) + 0)
values[i][j]	*(values + (5 * i) + j)
…	…

동적 1차원 배열

- new와 delete를 이용하여 프로그램 실행 중에 크기를 결정할 수 있는 동적 배열을 정의

```
#include <iostream>
using namespace std ;
int main()
    cout << "1차원 배열의 크기를 입력하시오." << endl ;
    int intSize ;
    cin >> intSize ;
    // new로 입력된 크기만큼의 int를 할당
    int *intArray = new int[intSize] ;
    for ( int i = 0 ; i < intSize ; i ++ ) cin >> intArray[i] ;
    delete [] intArray ; // delete로 할당된 메모리 반환
}
```

포인터의 동적 배열

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

int main() {
  int intArray1[2] = {0, 1} ;
  int intArray2[2] = {2, 3} ;
  int intArray3[2] = {4, 5} ;

  int** ppIntArray = new int*[3] ; // 3개의 int* 즉 int 배열을 원소로 하는 배열
  ppIntArray[0] = intArray1 ;      // 배열의 이름은 포인터로서 사용됨
  ppIntArray[1] = intArray2 ;
  ppIntArray[2] = intArray3 ;

  for ( int i = 0 ; i < 3 ; i ++ ) {
    for ( int j = 0 ; j < 2 ; j ++ )
      cout << ppIntArray[i][j] << '₩t' ;
    cout << endl ;
  }
  delete [] ppIntArray ;
}
```

동적 2차원 배열

```
#include <iostream>
using namespace std ;
int main() {
   cout << "행렬의크기를입력하시오." << endl ;
   int rowSize, columnSize ;
   cin >> rowSize >> columnSize ;

   cout << "행렬A [" << rowSize << " X " << columnSize << "] 의 값을 입력하시오.₩n" ;
   int **intArrayA = new int*[rowSize] ;
   for ( int i = 0 ; i < rowSize ; i ++ ) {
      intArrayA[i] = new int[columnSize] ;
      for ( int j = 0 ; j < columnSize ; j ++ ) cin >> intArrayA[i][j] ;
   }
   cout << endl ;
   cout << "행렬B [" << rowSize << " X " << columnSize << "] 의 값을 입력하시오.₩n" ;
   int **intArrayB = new int*[rowSize] ;
   for ( int i = 0 ; i < rowSize ; i ++ ) {
      intArrayB[i] = new int[columnSize] ;
      for ( int j = 0 ; j < columnSize ; j ++ ) cin >> intArrayB[i][j] ;
   }
```

동적 2차원 배열

```
  cout << endl << "행렬A + B는" << endl ;
  for ( int i = 0 ; i < rowSize ; i ++ ) {
    for ( int j = 0 ; j < columnSize ; j ++ ) {
      int a = intArrayA[i][j] ;
      int b = intArrayB[i][j] ;
      cout << a << " + " << b << " = " << a+b<< '₩t' ;
    }
    cout << endl ;
  }

  // intArrayA, intArrayB를 delete해 주어야 함
  for ( int i = 0 ; i < rowSize ; i ++ ) {
    delete [] intArrayA[i] ;
    delete [] intArrayB[i] ;
  }

  delete [] intArrayA ;
  delete [] intArrayB ;
}
```

vector<T>의 사용

```
#include <vector>      // vector<T>를 활용하기 위해서는 <vector>를 포함해야 함
#include <iostream>
using namespace std ;

int main() {
  cout << "크기를 입력하시오." << endl ;
  int intSize ;
  cin >> intSize ;
  // vector 정의
  vector<int> vInt1(intSize) ;              // intSize 크기의 int 타입 vector의 생성
  for ( unsigned int i = 0 ; i < vInt1.size() ; i ++ ) { // size() 함수를 이용한 크기
    vInt1[i] = i ;                          // [ ] 연산자를 이용한 원소의 접근
    cout << vInt1.at(i) << '₩t' ;           // at() 함수를 이용한 원소의 접근
  }
  cout << endl ;
  // 배열을 이용하여 vector의 생성및 초기화
  int intA[3] = {10, 20, 30} ;
  // 크기 3인 int vector를 {10, 20, 30}으로 초기화
  vector<int> vInt2(intA, intA+3) ;
  for ( unsigned int i = 0 ; i < vInt2.size() ; i ++ )
    cout << vInt2.at(i) << '₩t' ;
  cout << endl ;
}
```

vector<T>의 사용

- vector의 생성

```
int intSize = 5 ;
// intSize 크기의 int 타입 vector의 생성
// 각 원소는 int의 기본 값인 0 으로 초기화 됨
vector<int> vInt1(intSize) ;
for ( unsigned int i = 0 ; i < vInt1.size() ; i ++ )
        cout << vInt1[i] << '₩t' ; // 0 0 0 0 0
```

- 배열을 이용한 초기화

```
int intA[3] = {10, 20, 30} ;
vector<int> vInt2(intA, intA+3) ;
//크기 3인 int vector가 {10, 20, 30}으로 초기화
```

```
#include <vector>
#include <string>       // string을 사용하기 위함
#include <sstream>      // stringstream을 사용하기 위함
#include <iostream>
using namespace std ;

int main() {
  cout << "크기를 입력하시오." << endl ;
  int intSize ;
  cin >> intSize ;

  vector<string> vString(intSize) ; // string의 vector 생성
  for ( unsigned int i = 0 ; i < vString.size() ; i ++ ) {
    stringstream intStringStream ;
    intStringStream << i ; // 정수 i
    string val ;
    intStringStream >> val ; // 정수 i의 값이 문자열로 변환되어 val에 저장됨

    vString[i] = val ; // vString[i] 는 string 임
    cout << vString[i] << '₩t' ;
  }
  cout << endl ;
}
```

vector<T>의 크기 조정

- 필요하다면 공간을 확장

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> vInt ;
  cout << vInt.capacity() ; // 0
  for ( unsigned i=1; i<10; i++ ) vInt.push_back(i);
        // 1 2 3 4 5 6 7 8 9
  cout << vInt.capacity() ; // 9
  vInt.resize(3);                 // 1 2 3
  vInt.resize(5,100);             // 1 2 3 100 100
  vInt.resize(7);                 // 1 2 3 100 100 0 0
  for ( unsigned i=0; i < vInt.size(); i++ )
    cout << " " << vInt[i]; // 1 2 3 100 100 0 0
  cout << endl;
}
```

Iterator의 사용

- iterator는 vector, set, list, map 등의 컨테이너에 저장된 각 원소를 접근하기 위한 클래스

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
 vector<int> vInt(5) ; // int 타입 크기 5의 vector 정의
 for ( vector<int>::iterator it = vInt.begin() ; it != vInt.end() ; ++ it ) {
    *it = 20 ; // 각 원소의 값을 지정함
    cout << *it << '₩t' ;  // 20
 }
 cout << endl;
}
```

상수 iterator

- const_iterator는 원소의 값을 변경할 수가 없음
- 상수를 접근할 때 사용됨

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
 // 상수 vector의 정의. int의 기본 값인 0으로 초기화 됨
 const vector<int> vInt2(5) ;
 // vInt2가 const vector 이므로 const_iterator를 사용해야 함
 for ( vector<int>::const_iterator it = vInt2.begin() ; it != vInt2.end() ; ++ it ) {
    cout << *it << '₩t' ; // 0 0 0 0 0
    // it는 const_iterator이므로 *it의 값을 변경할 수 없다.
    // *it = 10 ; 또는 cin >> *it는 불허
 }
 cout << endl;
}
```

STL algorithm 함수의 사용

```
#include <iostream>
#include <vector>
#include <algorithm> // sort, max_element, random_shuffle
using namespace std;

int main () {
  int intArray[5] = {1, 2, 3, 4, 5} ;
  vector<int> vInt(intArray, intArray+5) ; // 배열을 이용한 vector의 초기화

  random_shuffle( vInt.begin(), vInt.end() );
  for (vector<int>::const_iterator it = vInt.begin() ; it != vInt.end() ; ++ it )
     cout << *it << '₩t' ; // 예) 5 2 4 1 3
  cout << endl ;

  sort( vInt.begin(), vInt.end() );
  for (vector<int>::const_iterator it = vInt.begin() ; it != vInt.end() ; ++ it )
     cout << *it << '₩t' ; // 1 2 3 4 5
  cout << endl;

  vector<int>::const_iterator largest = max_element( vInt.begin(), vInt.end()-1 );
  cout << "최대값: " << *largest << endl ; // 4
  cout << "위치: " << largest - vInt.begin() << endl ; // 3
}
```

Iterator를 이용한 알고리즘 함수

- 알고리즘 함수는 set, list, map 등에 영향을 받지 않음

```
#include <iostream>
#include <set> // set 정의
#include <algorithm>
using namespace std;

bool isOddNumber (int i) { return ((i%2)==1) ; }
int main () {
  int intArray[5] = {1, 2, 3, 4, 5} ;
  set<int> sInt(intArray, intArray+5) ; // 배열을 바탕으로 set<int> 초기화
  // const_iterator를 이용하여 set의 각 원소 값 출력
  for (set<int>::const_iterator it = sInt.begin() ; it != sInt.end() ; ++ it )
    cout << *it << '₩t' ;
  cout << endl ;
  // count_if() 함수를 이용해서 홀수의 개수를 구함
  int count = count_if (sInt.begin(), sInt.end(), isOddNumber);
  cout << "Set contains " << count << " odd numbers.₩n";
}
```

참조(reference) 변수

- 다른 변수를 가리키는 역할
- T&: T 타입 변수에 대한 참조

```
#include <iostream>
using namespace std ;
int main() {
  // T& x: 변수 x는 타입 T 변수에 대한 참조이다.
  int intVal = 10 ;
  int& rIntVal = intVal ;
  cout << intVal << '₩t' << rIntVal << endl ; // 10 10
  // 원래 변수의 값이 변경되면 참조 변수의 값도 변경됨
  intVal = 20 ;
  cout << intVal << '₩t' << rIntVal << endl ; // 20 20
  // 참조 변수의 값이 변경되면 원래 변수의 값도 변경됨
  rIntVal = 30 ;
  cout << intVal << '₩t' << rIntVal << endl ; // 30 30
}
```

참조 변수의 초기화

- 참조 변수는 기존의 변수를 가리키도록 초기화되며 이후에는 다른 변수를 가리킬 수가 없다

코드	설명
float floatVal = 10.1F ; float& rFloatVal1 = floatVal ;	Ok: 참조 변수가 정의와 동시에 일반 변수로 초기화 함
float floatVal1 = 10.1F ; float floatVal2 = 20.2F ; float& rFloatVal2 = floatVal1 ; rFloatVal2 = floatVal2 ;	OK: 참조 변수가 정의와 동시에 일반 변수로 초기화 함 Error: 초기화 후에 다른 변수를 가리킬 수 없음
float& rFloatVal3 = 10.0F ;	Error: 참조 변수는 리터럴 값으로 초기화 될 수 없음.
const float constVal = 20.0 ; float& rFloatVal4 = constVal	Error: 참조 변수는 상수 변수를 가리킬 수가 없음

참조 변수와 포인터 변수

	참조 변수	포인터 변수
정의 방법	int intVal=10, otherVal=20 ; int& rIntVal = intVal ;	int intVal=10, otherVal=20 ; int* pIntVal = &intVal ;
초기화 필수 여부	필수 int& rIntVal ; 가 불허됨	필수 아님 int* pIntVal ; 가 허용됨
다른 변수의 지칭	불가능	가능 pIntVal = &otherVal ; 가 허용됨
연산의 적용 대상	참조 변수가 가리키는 원 변수 rInt ++ ; intVal ++ 와 동일	pIntVal ++ ; 다음 주소를 가리킴 원래 변수 값을 접근하려면 (*pIntVal) ++ ; 로 함

Good Design: 포인터 대신 참조 변수가 권장

- 포인터는 +, ++, -- 등의 연산자를 이용해서 임의의 메모리를 가리킬 경우 문제를 유발
- 참조 변수는 기존 변수로 일단 초기화된 후에는 다른 변수를 임의로 가리키도록 변경될 수가 없음

```
#include <iostream>
using namespace std ;
int main() {
    int intVal = 10 ;
    int* pIntVal = &intVal ;
    cout << *pIntVal << endl ;
    pIntVal ++ ;
    cout << *pIntVal << endl ;
}
```

```
// 10
// intVal 다음의 메모리를 가리킴
// 확인되지 않은 값이 출력됨
```

나열형(enum)

- 나열형(enum)은 몇가지 한정된 값을 가지는 타입
- 가질 수 있는 모든 값들을 나열함으로써 정의

```
enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
```

- 나열형 타입 이름은 int, float 등과 동일한 방식으로 변수를 정의할 때 타입 이름으로서 사용

```
Grade curGrade = FRESH ;
```

나열형의 형변환

- **나열형 타입의 리터럴은 내부적으로 int 타입으로 처리**
- 첫번째 주어진 값을 0으로 해서 그 다음 값은 1로 처리
- **나열형 ➔ int 타입으로 묵시적으로 형변환**

```
int nextGrade = curGrade + 1 ;    // OK
Grade curGrade = 1 ;              // Error
```

- **int 값 ➔ 나열형 타입으로 명시적으로 형변환**

```
curGrade = Grade(nextGrade) ;
      또는
curGrade = static_cast<Grade>(nextGrade) ;
```

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
int main() {
  // 현재 학년을 int 값으로 입력 받는다.
  int intGrade ;
  cin >> intGrade ;
  // int 값을 Grade 타입으로 변환한다.
  Grade curGrade = Grade(intGrade) ;
  if ( curGrade < FRESH || curGrade > SENIOR ) {
    cout << "Grade should be between 1 and 4₩n" ;
    return 0 ;
  }
  // 다음 학년을 구한다.
  if ( curGrade != SENIOR ) {
    int nextGrade = curGrade + 1 ;
    // Grade ==> int로의 묵시적 형 변환
    curGrade = Grade(nextGrade) ;
    // curGrade = static_cast<Grade>(nextGrade) 과 동일
  }
  cout << curGrade << endl ; // int로 묵시적 형변환 됨
}
```

Good Design: 나열형의 활용

- 가독성을 위해서

int 타입 사용시	void sort(int data[], int size, int kind) ;
enum 사용시	enum SortKind {ASCEND, DESCEND} ; void sort(int data[], int size, SortKind kind) ;

- 결함 방지를 위해서

int 타입 사용시	int intArray[] = {20, 10, 30, 25} ; sort(intArray, 4, 3) ;

enum 사용시	int intArray[] = {20, 10, 30, 25} ;
	sort(intArray, 4, 0) ; // 컴파일 오류
	sort(intArray, 4, ASCEND) ; // OK

구조체(struct)

- 기존 타입의 변수들의 조합으로서 새로운 타입을 정의

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;

struct Student {
    string name ;
    Grade grade ;
    string phoneNumber ;
} ;
```

구조체 변수/객체의 생성 및 초기화

- 구조체 객체의 생성 및 초기화

```
Student st1 = {"Park", FRESH, "000-0000"} ;

Student* st2 = new Student ;
```

- 동일 구조체 객체에 의한 초기화 및 대입이 가능

초기화	실제 동작
Student st3 = st1 ; 또는 Student st3(st1) ;	Student st3 ; st3.name = st1.name ; st3.grade = st1.grade ; st3.phoneNumber = st1.phoneNumber ;

구조체 멤버 필드의 접근

- 객체.필드
- 객체포인터->필드

```
// 개별 필드의 접근
st2->name = st1.name ;
st2->grade = st1.grade ;
st2->phoneNumber = st1.phoneNumber ;
```

구조체간의 대입

- C++ 언어에서는 구조체 간에 대입문이 가능함
- 구조체를 구성하는 개별 멤버 필드간의 대입과 동일

대입	실제 동작
st3 = st1 ;	st3.name = st1.name ; st3.grade = st1.grade ; st3.phoneNumber = st1.phoneNumber ;

```
enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
// 구조체 타입의 정의
struct Student {
  string name ;
  Grade grade ;
  string phoneNumber ;
} ;
int main() {
  // 개별 필드의 초기화
  Student st1 = {"Park", FRESH, "000-0000"} ;
  // 개별 필드의 접근
  Student* st2 = new Student ;
  st2->name = st1.name ;
  st2->grade = st1.grade ;
  st2->phoneNumber = st1.phoneNumber ;
  delete st2 ;
  // 구조체 초기화
  Student st3 = st1 ; // 또는 Student st3(st1) ;
  // 구조체간의 대입
  st3 = st1 ;
  /* 아래와 같이 개별 필드 별로 대입하는 것과 유사함
  st3.name = st1.name ;
  st3.grade = st1.grade ;
  st3.phoneNumber = st3.phoneNumber ;
  */
  // 개별 필드의 접근
  cout << st3.name << '₩t' << st3.grade << '₩t' << st3.phoneNumber << endl ;
}
```

C,언어와 C++ 언어의 구조체

	C ++ 언어	C 언어
정의 방법	struct Rectangle { ... } ;	
구조체변수 정의	Rectangle r1, r2 ;	struct Rectangle r1, r2 ;
구조체간의 대입 예) r2 = r1 ;	허용	불허
함수의 매개변수 int getArea(Rectangle)	허용	불허 int getArea(struct* Rectangle)는 허용
함수의 반환 값 Rectangle getRect()	허용	불허 Rectangle* getRect()는 허용

공용체(union)

- 동일한 공간을 공유하는 여러 개의 변수를 정의

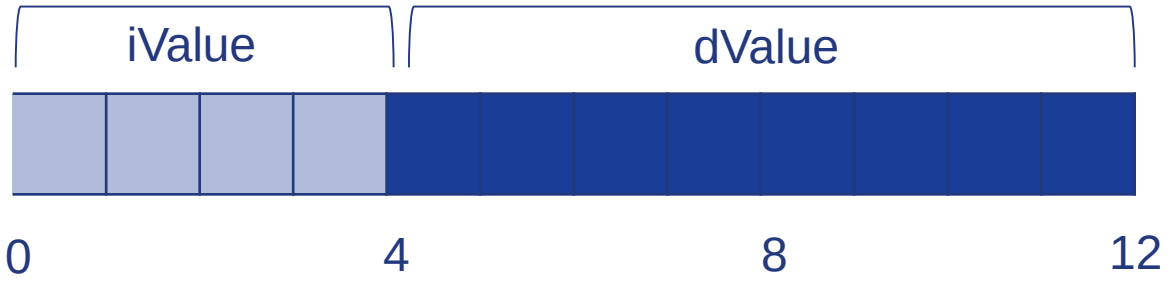
```
union NumericUnion
{
    int          iValue;
    double       dValue;
};
```

예제 프로그램	실행 예

int main() {
  NumericUnion values ;
  values.iValue = 10 ;
  cout << **values.iValue** << endl ;
 	
  values.dValue = 3.14F ;
  cout << **values.dValue** << endl ;
}
	
	// 10
	
	// 3.14

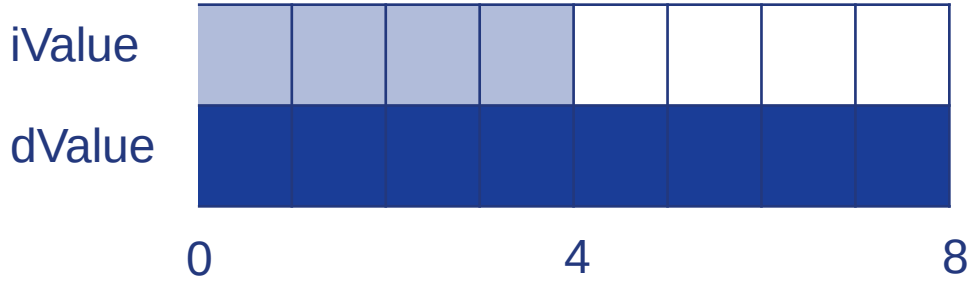
공용체와 구조체

```
struct NumericStruct {
   int      iValue;
   double   dValue;
};
```



(a) 구조체의 메모리 구성

```
union NumericUnion {
   int      iValue;
   double   dValue;
};
```



(b) 공용체의 메모리 구성

Good Design: 공용체의 사용은 자제

- 공용체는 복수 개의 변수가 동일한 메모리를 사용하므로 사용되는 메모리의 양을 줄일 수 있다는 측면에서 잇점
- 동일한 메모리를 사용하고 있으므로 공용체의 각 필드의 값은 서로 영향을 미치게된다

예제 프로그램	실행 예

int main() {
  NumericUnion values ;
  values.iValue = 10 ;
  cout << values.iValue << endl ;
  cout << values.dValue << endl ;
 	
  values.dValue = 3.14F ;
  cout << values.iValue << endl ;
  cout << values.dValue << endl ;
}
	
	// 10
	// 1.4013e-044
	
	// 1078523331
	// 3.14

typedef

- 기존의 타입과 동일한 역할을 하는 새로운 타입 이름을 정의

```
typedef unsigned int Age ;
typedef bool TFFlag ;
```

- 새롭게 정의된 타입 이름은 변수의 정의, 함수의 매개변수 타입 등으로 동일하게 사용

```
Age a1, a2 ;
TFFlag isSame(const Age a1, const Age a2)
```

typedef의 활용 예 1)

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

// typedef를 이용한 Age와 TFFlag의 정의
typedef unsigned int Age ;
typedef bool TFFlag ;
// 반환 타입 및 매개변수 타입으로의 사용
TFFlag isSame(const Age a1, const Age a2) { return a1 == a2 ; }
int main() {
  Age a1, a2 ; // 변수 정의
  cin >> a1 >> a2 ;
  TFFlag same = isSame(a1, a2) ; // 변수 정의
  if ( same ) cout << "Same Age" << endl ;
  else cout << "Different Age" << endl ;
}
```

typedef의 활용 예 2)

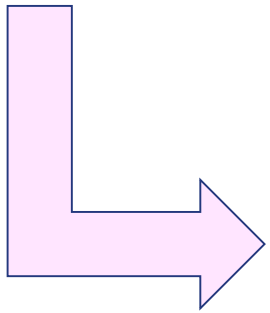
사용 전	for (**vector<Rectangle*>::const_iterator** it = rectangles.begin() ;
	  it != rectangles.end() ; ++ it) {
	  const Rectangle* const rect = *it ;
	  …
	}
	…

사용 후	**typedef vector<Rectangle*>::const_iterator RIT ;**
	for (**RIT** it = rectangles.begin() ;
	  it != rectangles.end() ; ++ it) {
	  const Rectangle* const rect = *it ;
	  …
	}
	…

Good Design: 새로운 타입에 범위를 설정하고 싶다면 클래스를 사용

- typedef는 타입이 취할 수 있는 값에 대한 범위를 제한하지는 못한다

```
typedef unsigned int Age ;
Age a ;
a = -10 ;
```



```
const int INVALID_AGE = -1 ;
class Age {
  int age ;
public:
 Age(const int a) {
     if ( a < 1 ) throw INVALID_AGE ;
     age = a ;
 }
} ;
int main() {
  Age a1(10) ;
  Age a2(-10) ; // INVALID_AGE 예외 발생
}
```