

C++ 기본

C++ 기본 구조

```
#include <iostream> //헤더 파일에 확장자를 붙이지 않음
```

```
int main(){
```

```
    std::cout << "Hello"<<std::endl;           //std -> namespace, endl -> 조작자, 줄바꿈
```

```
    std::cout << "첫 번째 맛보기입니다.\n";
```

```
    return 0; //main() 함수에서만 생략 가능
```

```
}
```

주석문과 main() 함수

- 주석문
 - 개발자가 자유롭게 붙인 특이 사항의 메모, 프로그램에 대한 설명
 - 프로그램의 실행에 영향을 미치지 않음
 - 여러 줄 주석문 - `/* ... */`
 - 한 줄 주석문 - `//`를 만나면 이 줄의 끝까지 주석으로 처리
- main() 함수
 - C++ 프로그램의 실행을 시작하는 함수
 - main() 함수가 종료하면 C++ 프로그램 종료
 - main() 함수의 C++ 표준 모양

```
int main() { // main()의 리턴 타입 int
    .....
    return 0; // 0이 아닌 다른 값으로 리턴 가능
}
```

- main()에서 return문 생략 가능

#include <iostream>

- #include <iostream>
 - 전처리기(C++ Preprocessor)에게 내리는 지시
 - <iostream> 헤더 파일을 컴파일 전에 소스에 확장하도록 지시
 - <iostream> 헤더 파일
 - 표준 입출력을 위한 클래스와 객체, 변수 등이 선언됨
 - ios, istream, ostream, iostream 클래스 선언
 - cout, cin, <<, >> 등 연산자 선언

```
#include <iostream>
```

```
....
```

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

```
std::cout << "첫 번째 맛보기입니다.";
```

namespace 개념

- 이름(identifier) 충돌이 발생하는 경우
 - 여러 명이 서로 나누어 프로젝트를 개발하는 경우
 - 오픈 소스 혹은 다른 사람이 작성한 소스나 목적 파일을 가져와서 컴파일 하거나 링크하는 경우
- namespace 키워드
 - 개발자가 자신만의 이름 공간을 생성할 수 있도록 함
 - 이름 공간 안에 선언된 이름은 다른 이름공간과 별도 구분 – 이름 충돌 해결

- 이름 공간 생성

```
namespace hallym { // hallym 이라는 이름 공간 생성
.....           // 이 곳에 선언된 모든 이름은 hallym 이름 공간에 생성된 이름
}
```

- 이름 공간 사용
 - **이름 공간 :: 이름**

- nested namespace

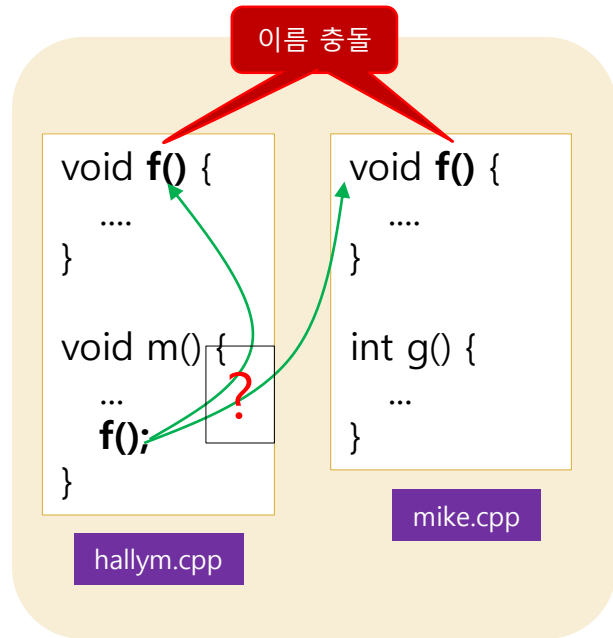
```
namespace A{
  namespace B{
    .....
  }
}
```

C++ 17

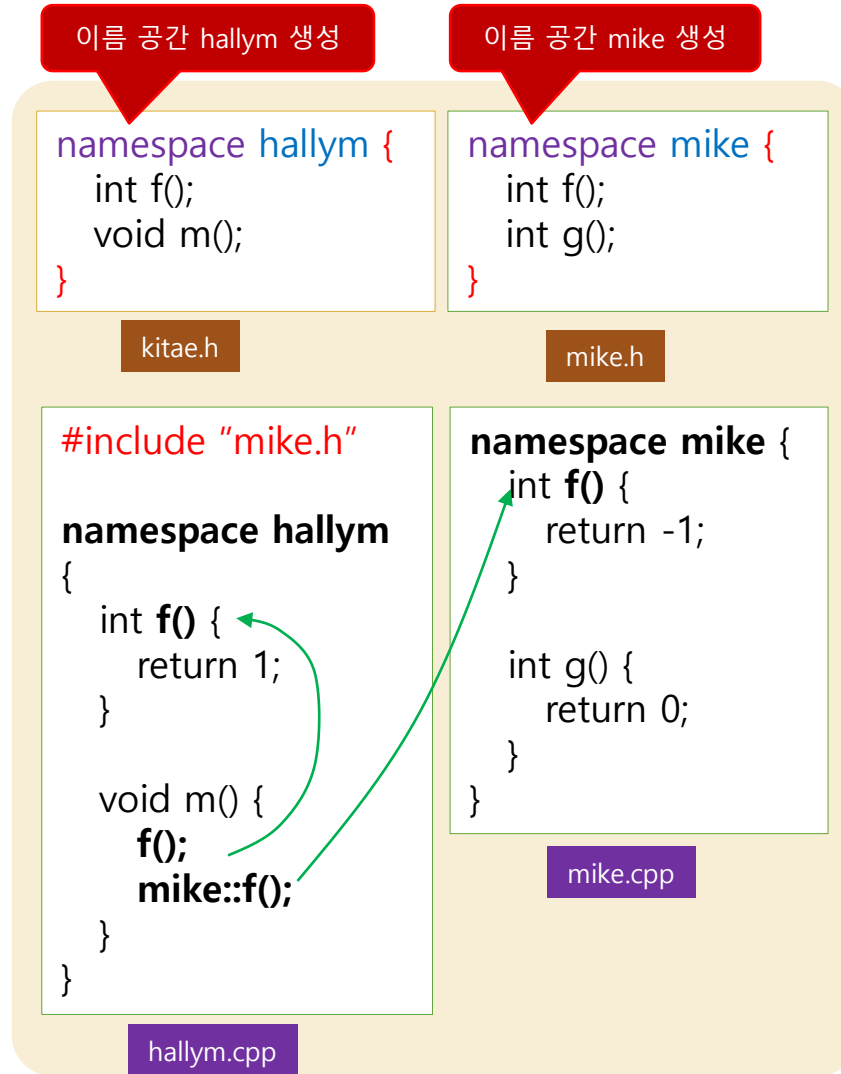


```
namespace A::B{
  ....
}
```

namespace 개념



(a) hallym과 mike에 의해 작성된 소스를 합치면 f() 함수의 이름 충돌. 컴파일 오류 발생



(b) 이름 공간을 사용하여 f() 함수 이름의 충돌 문제 해결

std:: 란?

- std
 - C++ 표준에서 정의한 **이름 공간(namespace)** 중 하나
 - C++ 표준 라이브러리는 std에 작성 됨
 - 표준 라이브러리에 선언된 identifier 사용시 std:: 접두어 사용
 - 사용 예 : std::cout, std::cin, std::endl

std:: 생략

using 지시어 사용

std:: 생략

```
using std::cout; // cout에 대해서만 std:: 생략
```

```
.....
cout << "Hello" << std::endl; // std::cout에서 std:: 생략
```

```
using namespace std; // std 이름 공간에 선언된 모든 이름에 std:: 생략
```

```
.....
cout << "Hello" << endl; // std:: 생략
```

std:: 생략

std:: 생략

화면 출력

- **cout과 << 연산자 이용**
- cout 객체
 - 스크린 출력 장치에 연결된 **표준** C++ 출력 스트림 객체
 - <iostream> 헤더 파일에 선언
 - std 이름 공간에 선언 -> **std::cout**으로 사용
- << 연산자
 - 스트림 삽입 연산자(stream insertion operator)
 - C++ 기본 산술 시프트 연산자(<<)가 스트림 삽입 연산자로 재정의됨
 - ostream 클래스에 구현됨
 - 오른쪽 피 연산자를 왼쪽 스트림 객체에 삽입
 - cout 객체에 연결된 화면으로 출력
 - << 연산자 연속 사용 가능

```
std::cout << "Hello\n" << "첫 번째 맛보기입니다.";
```


cout과 <<를 이용한 화면 출력

```
#include <iostream>
```

```
double area(int r); // 함수 원형 선언
```

```
double area(int r) { // 함수 구현  
    return 3.14*r*r; // 반지름 r의 원 면적 리턴  
}
```

```
int main() {  
    int n = 3;  
    char c = '#';  
    std::cout << c << 5.5 << '-' << n << "hello" << true << std::endl; //true는 1로 출력됨  
    std::cout << "n + 5 = " << n + 5 << '\n';  
    std::cout << "면적은 " << area(n); // 함수 area()의 리턴 값 출력  
}
```

키 입력

- cin과 >> 연산자를 이용
- cin 객체
 - 표준 입력 장치인 키보드를 연결하는 C++ 입력 스트림 객체
 - <iostream> 헤더 파일에 선언
 - std 이름 공간에 선언: **std::cin**으로 사용
 - <Enter>키가 입력될 때까지 입력된 키를 입력 버퍼에 저장
 - 도중에 <Backspace> 키를 입력하면 입력된 키 삭제
- >> 연산자
 - 스트림 추출 연산자(stream extraction operator)
 - 시프트 연산자(>>)가 스트림 추출 연산자로 재정의됨
 - istream 클래스에 구현
 - 입력 스트림에서 값을 읽어 변수에 저장
 - 연속된 >> 연산자를 사용하여 여러 값 입력 가능

```
cout << "너비와 높이를 입력하세요>>";  
cin >> width >> height;
```

cin 사용 예

```
#include <iostream>
using namespace std;
int main()
{
    int width;
    cout << "너비와 높이를 입력하세요 : ";

    int height; //C++에서는 변수 선언 위치에 제한을 두지 않는다.

    cin >> height >> width;

    int area = width * height;
    cout << "면적은 " << area << "\n";
}
```

문제	출력	디버그 콘솔	터미널
			PS E:\lecture_src\cpp_src> g++ a.cpp
			PS E:\lecture_src\cpp_src> .\a
	너비와 높이를 입력하세요 : 3 5		
	면적은 15		
			PS E:\lecture_src\cpp_src> .\a
	너비와 높이를 입력하세요 : 4		
	6		
	면적은 24		
			PS E:\lecture_src\cpp_src>

C++ 자료형

- bool 타입(1byte)
 - 0은 거짓을, 0이 아닌 모든 것을 참으로 처리.
 - 참/거짓은 상수 true/false를 이용해도 됨.
 - true/false 상수를 출력하려면 **boolalpha 조작자** 사용.
 - 조작자를 사용하지 않으면 참은 1로 거짓은 0으로 출력.
 - 조작자 : 입력 혹은 출력 방식을 바꿔주는 함수

조작자	용도
endl	스트림 버퍼를 모두 출력하고 다음 줄로 넘어감
oct	정수 필드를 8진수 기반으로 출력
dec	정수 필드를 10진수 기반으로 출력
hex	정수 필드를 16진수 기반으로 출력
fixed	실수 필드를 고정 소수점 방식으로 출력
boolalpha	불린 값이 출력될 때, "true" 혹은 "false" 문자열로 출력
setprecision(int np)	출력되는 수의 유효 숫자 자리수를 np개로 설정. 소수점(.)은 별도로 카운트
setw(int minWidth)	필드의 최소 너비를 minWidth로 지정

조작자를 사용한 입출력 사용 예

```
#include <iostream>
#include <iomanip> //조작자 사용을 위해 필요
using namespace std;
int main() {
    int data;
    bool flag;
    cout << "여러 진법으로 정수 입력 받기" << endl;
    cout << "10진법으로 입력하기 :";
    cin >> data;    cout << "10진법 : " << data << endl;

    cout << "8진법으로 입출력하기 : ";
    cin >> oct >> data;    cout << " 8진법 : " << oct << data << endl;

    cout << " 16진법으로 입출력하기 :";
    cin >> hex >> data;    cout << "16진법 : " << hex << data << endl;

    cout << "불 자료형으로 입출력하기 :";
    cin >> boolalpha >> flag;    cout << "불 리터럴 출력 : " << boolalpha << flag << endl;

    cout << "고정 소수점 출력하기:" << fixed << setprecision(2) << 2345.12432 << endl;
    return 0;
}
```

열거형 클래스(enum class)

- 기존 enum의 경우 열거 타입 값이 자동으로 정수로 변환
- enum class
 - 엄격한 타입 적용, 자동으로 열거형 값이 정수로 변환되지 않음
 - 열거 타입 값을 사용할 때 반드시 범위지정연산자(::)를 붙여야 함

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

```
int main() {
    enum Menu {Insert=1, Delete, Update};
    enum class Color {Blue=1, Green, Red, Black};

    Menu choice = Insert;
    Color color = Color::Blue; //열거 타입 값을 사용할 때 반드시 범위지정연산자(::)사용해야 함

    if (choice == 1) //열거 타입 값이 자동으로 정수타입으로 변환
        cout << "Insert" << endl;

    if (static_cast<int>(color) == 1) //정수 타입으로 사용하려면 명시적 형 변환
        cout << "Blue" << endl;
}
```

C++ 캐스팅

- C++ 캐스팅 방법
 - `const_cast`
 - 포인터(pointer) 또는 참조형(reference)의 `const` 속성 추가 또는 제거
 - `static_cast`
 - 언어에서 허용하는 명시적 형 변환, 컴파일 시 타입 검사
 - `dynamic_cast`
 - 동일한 상속 계층에 속한 클래스 타입의 객체 포인터 또는 객체 레퍼런스 사이의 변환, 실행 시간에 타입 검사
 - 부모클래스가 가상 함수를 포함하고 있어야 함
 - `reinterpret_cast`
 - 서로 관련 없는 타입의 포인터 또는 레퍼런스 사이의 변환
 - 함수 포인터 사이의 변환

C++ 캐스팅

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

void method(const int* i) {
    int *d = const_cast<int*>(i); //매개변수 변수 i의 const 속성 해제
}

int main() {
    double d = 34.5;
    int i = (int)d;          //C type
    int si = static_cast<int>(d);

    int digit = 34;
    double* dp = (double*)&digit;          //컴파일 성공
    double* sdp = static_cast<double*>(&digit); //컴파일 오류
    method(&si);
}
```


auto

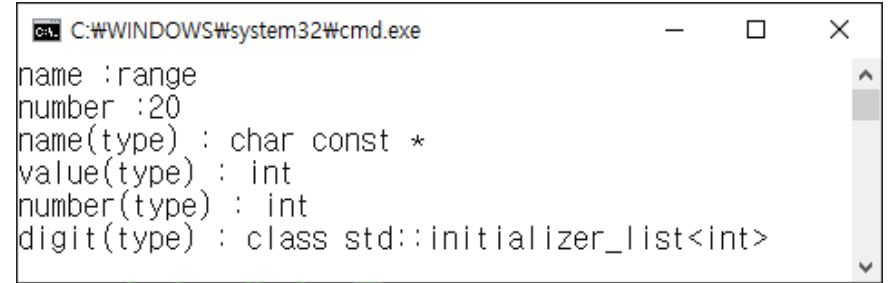
- 변수 정의 때 명시적으로 type을 지정하지 않아도 된다.
- auto로 정의한 변수 타입은 초기화 한 값에 맞춰 컴파일 때 결정.
- 함수 매개변수, 구조체나 클래스의 멤버 변수로 사용 불가

```
#include <iostream>
#include <typeinfo> //typeid 연산자 사용->데이터 타입 정보 반환

using namespace std;

int main() {
    auto name = "range"; // auto 로 선언하면서 반드시 초기화 해야 함
    auto value = 20;
    auto number{ 20 }; // 직접 리스트 초기화(C++17), int
    auto digit = { 20 }; // 복사 리스트 초기화(C++17), initializer_list<int>

    cout << "name :" << name << endl;
    cout << "number :" << number << endl;
    cout << "name(type) : " << typeid(name).name() << endl;
    cout << "value(type) : " << typeid(value).name() << endl;
    cout << "number(type) : " << typeid(number).name() << endl;
    cout << "digit(type) : " << typeid(digit).name() << endl;
}
```



```
C:\WINDOWS\system32\cmd.exe
name :range
number :20
name(type) : char const *
value(type) : int
number(type) : int
digit(type) : class std::initializer_list<int>
```

auto

- 일반 함수의 반환 값 타입으로 사용 가능

```
auto IsMaxLevel(int i) {  
    if (i >= 100) return true;  
    else return false;  
}  
  
int main() {  
    auto result = IsMaxLevel(50);  
  
    cout << std::boolalpha << result << endl;  
    return 0;  
}
```

- 람다 표현식에서 사용

```
int main()  
{  
    auto lambda = [](int a, int b) { return a + b; };  
  
    cout << "result = " << lambda(11, 22) << endl;  
  
    return 0;  
}
```

범위 기반 for

- 'auto'와 더불어 간단하면서 유용한 기능.
- 반복문을 쉽고, 안전하게 사용할 수 있다.
- C++ STL의 컨테이너, 배열, initializer_list 등에 사용

//벡터에서 범위 기반 for문 사용

```
vector<int> v;  
for (int i = 1; i < 11; i++)  
    v.push_back(i);
```

//기존의 for문 사용

```
for (vector<int>::iterator iter = v.begin(); iter != v.end(); ++iter)  
    cout << *iter << " ";
```

//범위 기반 for 문 사용 : iterator를 사용하지 않아도 됨, 복사를 방지하려면 레퍼런스 변수 활용

```
for (int elem : v) //또는 for(auto elem : v)  
    cout << elem << " ";
```

initializer_list

- <initializer_list> 헤더 파일에 정의
- **동일 타입**의 요소를 여러 개 보관하는 템플릿 클래스
- 여러 인수를 받는 함수를 간단히 작성할 수 있음
- Initializer-list 생성자구현에 사용

```
#include <iostream>
```

```
#include <initializer_list>
```

```
using namespace std;
```

```
//initializer_list로 받은 값은 상수, <>안에 반드시 자료형 명시
```

```
void list_exam(initializer_list<int> value) {
```

```
    int hap = 0;
```

```
    for (auto data : value)
```

```
        hap += data;
```

```
    cout << "결과 :" << hap << endl;
```

```
    cout << "평균 :" << hap / value.size() << endl;
```

```
}
```

```
int main() {
```

```
    list_exam({ 3,12,6,37,8,43 });
```

```
    list_exam({ 5,8,21,84,12,31, 27, 7 });
```

```
    for (auto v : { 34,56,87,43,68 })
```

```
        cout << v << " ";
```

```
}
```

uniform initialization

- { }을 사용하여 변수, 배열, 객체 초기화

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

void method(int i) { /*.....*/ }
class Exam {
public:
    Exam() { cout << "Exam 디폴트 생성자" << endl; }
};

int main() { //uniform initialization 사용 예
    Exam e();           //Exam 타입을 반환하는 함수
    Exam ex{};          //디폴트 생성자 호출
    int d{ 3.6 };        //error, 축소 변환 불가
    int f = 3.6;         //컴파일 오류는 없지만 경고, 축소 변환
    int arr[]{ 2,3,4 };  //배열 초기화
    method(4.9);
    method({ 4.9 });     //error, 축소 변환 불가
}
```

참고 : 구조화된 바인딩 - C++ 17

```
#include <iostream>
#include <string>
```

```
using namespace std;
struct Entry {
    string name;
    int value;
};
```

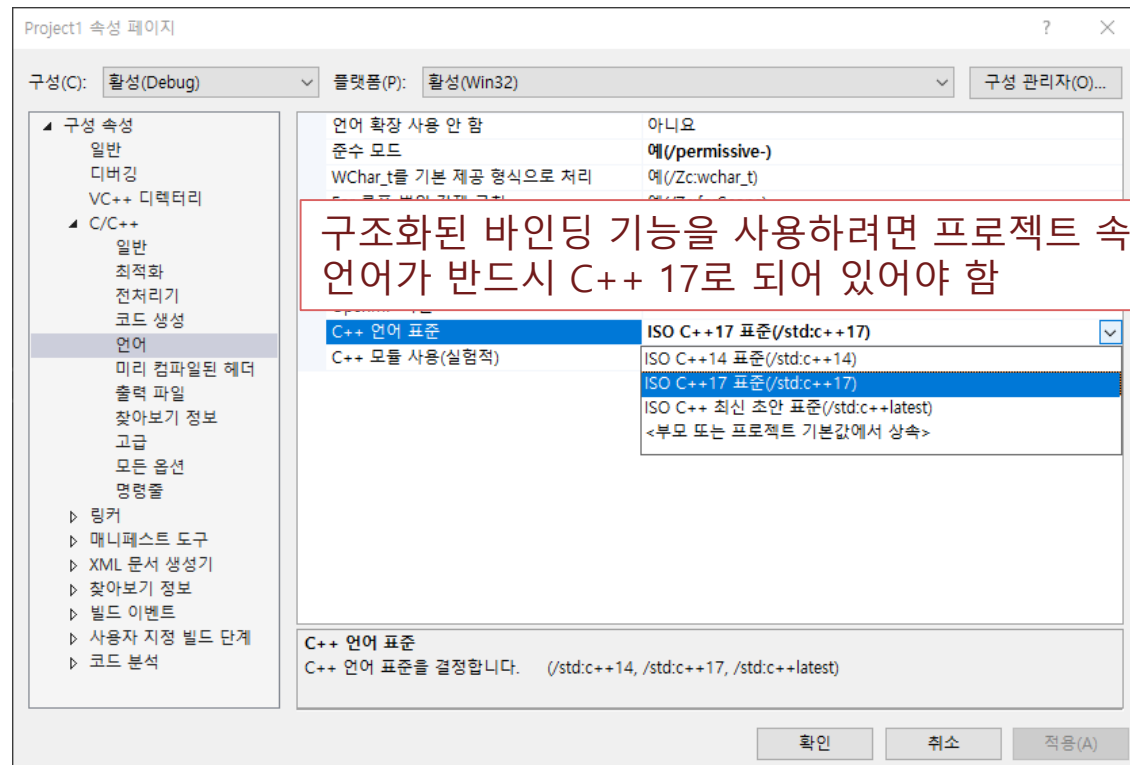
```
Entry read_entry() {
    string s;
    int i;
    cout << "in_string in_int : ";
    cin >> s >> i;
    return { s, i };
}
```

```
int main() {
    auto e = read_entry();
    cout << e.name << e.value << endl;
```

`auto [n, v] = read_entry();` //구조화된 바인딩 : 데이터 멤버에 사용 단, 접근 지정자 **public**
//tuple 분리, map 사용 시 유용함

```
cout << n << v << endl;
```

```
}
```



```
PS E:\lecture_src\cpp_src> g++ cpptest.cpp -std=c++17
PS E:\lecture_src\cpp_src> .\a
```

참고 : 구조화된 바인딩 - C++ 17

```
#include <iostream>
#include <string>
#include <tuple> //tuple 클래스
```

```
using namespace std;
```

```
int main()
```

```
{
    tuple t{ "binding", 23, 45.23 }; //C++ 17 부터 생성자에서 템플릿 인수 추론 기능 추가
```

```
    auto[str, i, d] = t; //구조화된 바인딩 : tuple 분리
```

```
    cout << "str : " << str << endl;
    cout << "i : " << i << endl;
    cout << "d : " << d << endl;
```

```
    return 0;
```

```
}
```

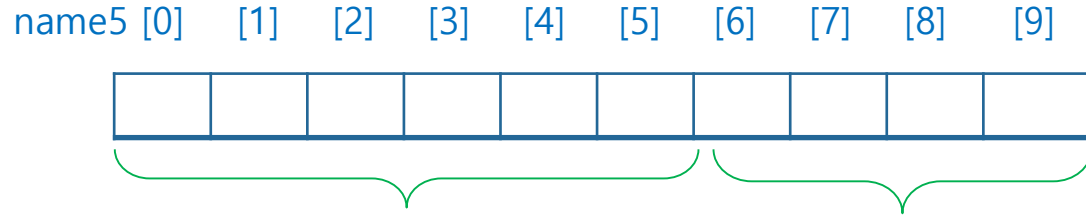
```
int main() {
    tuple<string, int, double> t{ "binding", 23, 45.23 }; //C++17 이전
    string str;
    int i;
    double d;
    tie(str, i, d) = t; //C++ 17 이전 : tie()함수를 사용 tuple 분리
    cout << "str : " << str << endl;
    cout << "i : " << i << endl;
    cout << "d : " << d << endl;
    return 0;
}
```

C++ 문자열

- C++의 문자열 표현 방식 : 2가지
 - C 스트링 방식 : '₩0'로 끝나는 문자 배열

```
char name1[6] = {'G', 'r', 'a', 'c', 'e', '₩0'}; // name1은 문자열 "Grace"
char name2[5] = {'G', 'r', 'a', 'c', 'e'};      // name2는 문자열이 아니고 단순 문자 배열
```

```
char name5[10] = "Grace";
```



- string 클래스 이용
 - <string> 헤더 파일에 선언됨
 - 다양한 멤버 함수 제공, 문자열 비교, 복사, 수정 등

C++ 에서 문자열을 다루는 string 클래스

- string 클래스
 - C++ 에서 강력 추천, C++ 표준 클래스
 - 문자열의 크기에 따른 제약 없음
 - string 클래스가 스스로 문자열 크기게 맞게 내부 버퍼 조절
 - 문자열 복사, 비교, 수정 등을 위한 다양한 함수와 연산자 제공
 - 객체 지향적
 - **<string> 헤더 파일에 선언**
 - **#include <string> 필요**
 - **using namespace std; //std안에 string 정의, 이름 공간 지정 반드시 필요**
 - C 스트링보다 다루기 쉬움

string 클래스를 이용한 문자열 입력 예

```
#include <iostream>
#include <string>      //string 클래스를 사용하기 위한 헤더 파일
using namespace std;  //std안에 string 정의, 이름영역지정 반드시 필요

int main() {
    string song("Falling in love with you"); // 문자열 song, 또는 string song="Falling in love with you";
    string elvis("Elvis Presley");           // 문자열 elvis
    string singer;                           // 문자열 singer

    cout << song + "를 부른 가수는";          // + 로 문자열 연결
    cout << "(힌트 : 첫 글자는 " << elvis[0] << ") ? "; // [] 연산자 사용

    // getline()은 string 타입의 문자열을 입력 받기 위해 제공되는 전역 함수
    // 공백이 포함된 문자열 입력 가능
    getline(cin, singer); // 문자열 입력,
    if(singer == elvis)   // 문자열 비교
        cout << "맞았습니다.";
    else
        cout << "틀렸습니다. " + elvis + "입니다." << endl; // +로 문자열 연결
}
```

배열

- fixed size array : array
 - sequence container, limited size, random access, 반복자와 알고리즘 함수 사용
 - `std::array` -> Stack, Compile time, fast allocation
 - `<array>` 헤더 파일에 정의된 array 클래스 사용 권장
 - 예) `std::array<int, 100> arr;`
- flexible size array : vector
 - sequence container, huge size, random access
 - `std::vector` -> Heap, Run time, slow allocation(`reserve()`로 해결)
 - `<vector>` 헤더 파일에 정의된 vector 클래스 사용 권장
 - 예) `std::vector<int> arr(100);`
- array와 vector 클래스는 컨테이너 클래스 - STL에서 설명

배열 예

```
#include <iostream>
#include <array>    //array 클래스
#include <algorithm> //sort() 함수
using namespace std;
int main()
{
```

```
    array<int, 5> farr{3, 6, 11, 5, 7}; //자료형과 크기를 반드시 명시해야 함
    array<int, 6> sarr;
    sarr = {12, 4, 31, 46, 23, 9};
```

```
    cout << "farr size = " << farr.size() << endl;           //size()함수 : 배열 크기
```

```
    cout << "farr.at(2) = " << farr.at(2) << ", farr[2] = " << farr[2] << endl; //at()함수는 유효 범위 검사, []보다 안정적
```

```
    cout << "farr 첫번째 원소 = " << farr.front() << ", farr 마지막 원소 = " << farr.back() << endl;
```

```
    cout << "farr 배열 상태 = " << boolalpha << farr.empty() << endl; //empty() : 빈배열 true, 원소가 있으면 false
```

```
    cout << "sarr sort = ";
```

```
    sort(sarr.begin(), sarr.end()); //반복자와 sort()함수를 사용 배열 원소 정렬
```

```
    for (auto value : sarr)
```

```
        cout << value << " ";
```

```
}
```

```
farr size = 5
farr.at(2) = 11, farr[2] = 11
farr 첫번째 원소 = 3, farr 마지막 원소 = 7
farr 배열 상태 = false
sarr sort = 4 9 12 23 31 46
```

vector 예

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> aarr(5);

    for(int i=0; i<5; i++) aarr[i] = i*2;

    for(int v : aarr) std::cout << v << " ";
    std::cout << std::endl;

    std::vector<int> barr{1,2,3,4,5};

    for(auto v : barr) std::cout << v << " ";
    std::cout << std::endl;

    barr.emplace_back(6);
    std::cout << "barr.size = " << barr.size() << std::endl;

    for(int v : barr) std::cout << v << " ";
    std::cout << std::endl;

    barr.pop_back();
    for(const int &v : barr) std::cout << v << " ";

}
```

문제 출력 디버그 콘솔 터미널

```
PS E:\lecture_src\cpp_src> g++ cpptest.cpp
PS E:\lecture_src\cpp_src> .\a
0 2 4 6 8
1 2 3 4 5
barr.size = 6
1 2 3 4 5 6
1 2 3 4 5
PS E:\lecture_src\cpp_src> 
```

실습

- 기본 입출력 활용
 - 입력한 10진 정수를 다양한 진법으로 출력하기
 - : 제어문 & 조작자 & 문자열 활용
- initializer_list 활용
 - 두번째 인수와 가장 가까운 거리에 있는 문자 출력하기

```

문제   출력   디버그 콘솔   터미널

PS E:\lecture_src\cpp_src> g++ cpptest.cpp
PS E:\lecture_src\cpp_src> ./a
10진수 입력 : 55

여러 진법으로 출력 하기
oct(8), hex(16), digit(10)
해당 진법 입력 : 16
=> 16진법 : 37
PS E:\lecture_src\cpp_src> ./a
10진수 입력 : 55

여러 진법으로 출력 하기
oct(8), hex(16), digit(10)
해당 진법 입력 : hex
=> 16진법 : 37
PS E:\lecture_src\cpp_src>

```

```

{ 'd', 'p', 'r', 'w', 'g', 'f' }문자 중 h와 가까운 문자는 : g
{ 'k', 'q', 'b', 'r', 'a', 'e', 'v', 'z' }문자 중 w와 가까운 문자는 : v

```

```

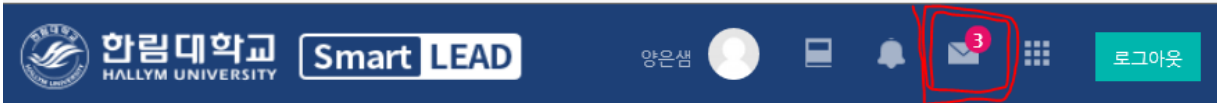
int main() {
    cout << "{ 'd', 'p', 'r', 'w', 'g', 'f' }문자 중 h와 가까운 문자는 : ";
    cout << list_exam({ 'd', 'p', 'r', 'w', 'g', 'f' }, 'h') << endl;

    cout << "{ 'k', 'q', 'b', 'r', 'a', 'e', 'v', 'z' }문자 중 w와 가까운 문자는 : ";
    cout << list_exam({ 'k', 'q', 'b', 'r', 'a', 'e', 'v', 'z' }, 'w') << endl;
}

```

Q & A

- "C++ 기본"에 대한 학습이 모두 끝났습니다.
- 새로운 내용이 많았습니다. 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.



- cpp_02_기본_ex.pdf 에 확인 학습 문제들을 담았습니다.
- 이론 학습을 완료한 후 확인 학습 문제들로 학습 내용을 점검 하시기 바랍니다.
- 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.
- 수고하셨습니다.^^