1장 C에서 C++로 문법 전환하기

<iostream> 인클루드

* 객체지향 - 언어특성 ↔ 절차지향.
  ↳ [Class] ← C언어 함수+구조체 ↳ 대규모 프로젝트 불편.

  개념 → 실체(구현)

* C++ 의 << input redirection.
  모든 장치는 file로 추상화된다. → file 입출력 ⇒ 장치제어 가능.
  ↳ stream << 정보.

C++ → std::(namespace) cout(객체) << "String" << std::endl; ⇒ 구조간결, 디테일 차후 개발可
std: 범위지정 연산. cout: 행위주체 <<: 리다이렉트 이용주체

C → printf()← 사용, 출력 ⇒사용방법 알아야 함.

[C언어 : 사용자가 스스로 "잘" 해야함. (↳유지보수(관리))
 C++ : 사용자 이해도↓ ok, 객체 자체가 중요.) 비용 & 효율 & 성능
 C C++ C

* 인스턴스 : int nData; ⇒ nData는 int 형식에 대한 인스턴스이다.
* namespace : ≒ 소속. 분류, 묶음.

printf() ⇒ 형식문자(ex. %d) 필요.
cout << ⇒ 알아서 형식지정

* cin >> 인스턴스.
  ↳콘솔로부터 입력을 받음.
  유니코드

* C++의 자료형 : long long / char16_t / char32_t
  ↳ auto

* 변수선언
  ① 선언 및 정의 : int a=10; ⇒C언어 st.
  int b(10); ⇒C++ st.
  int b(a); ↳복사.
  int c(10); ↳이름x 가능 / 상수처리.
  ↳이름x 인스턴스 선언&정의

  ② auto b(a); ⇒b는 a의 형식을 따라감.

⇒ 형식선언 한번만 해주면 형식무관 사용.
  ↳ 함수포인터 변수 등에서 유용.

1장 - 이어서

* 메모리 동적 할당 (malloc(), free()) ⇒ C언어
  ↳ 실패시 NULL 반환 (크기 부족) ⇒ 에러처리.

↳ malloc (size_t); → 주소 반환.
  ↓ ↳ 크기 (양의정수)
  memory 확보 / 용도지정 X.

포인터 : 주관적, 자유도 高
  ↳ 실력기복 / 위험도 ↑ / 안전장치 X

int * → 실제 상관없이 접근자가 해석가능.
다른 형식이 들어가있어도 int로 해석.

malloc 실패가능성 ↓ ⇒ ∴ 연산자화: new 자료형
  ① 메모리 동적 할당
  ② class의 생성자 호출.
  해제 : delete 인스턴스명

↳ 크기 자료형에 맞춰 자동할당.
⇒ 개발자가 할당시 크기문제 신경 X

배열)
int *변수명 = new int[5];
delete [] 변수명;

* 참조자 형식 (≈ alias (리눅스))
↳ 참조자는 일종의 '별명'이다. / 사용이유 ⇒ 포인터의 문제점 == 변수이므로 바뀔 수 있다.
⇒ int &rData(nData); ⇒ 수명은 코드가 끝날때까지. 변하지 X.
↳ rData의 값을 바꾸면 nData의 값이 바뀜.
↳ rData와 nData의 주소포인터는 같다.

⇒ 포인터의 상수화.
const int *pnData = &a; a가 상수화.
int &pnData = a; == int * const pnData = &a; pnData 상수화.
  ↳ a의 주소임.

* r-value 참조.
↳ 이름없는 임시객체(성능↓)처리.
↳ r-value : 단순대입의 오른쪽 (상수)
ex) int &&rData = a+5;

a = 10; → 변수, 상수 → 임시객체 (연산의 중간결과, 함수의 반환값 등)
l-value r-value

* 범위기반 for문.
↳ 인덱스 계산실수 예방, 수정용이 ↳ 요소개수 자동
for (auto n : 배열명) { }
  ↳ 배열 형식무관
  ↳ 배열크기 무관.
  ↳ 전체요소에 대하여 반복 / read only

* C++ 엄격한 형식검사 → 범위기반 for의 read only.
for(auto &n : 배열명) { }
  ↳ 참조자
  ↳ 참조자로 read only 해제 可