C++ Primer 북 스터디 1~2장 정리

물론입니다. 아래는 이모티콘을 제거하고, 사람이 자연스럽게 작성한 것처럼 정리한 내용입니다:

1장 - Getting Started

1. C++ 프로그램의 기본 구조

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

중요 개념 요약

- `#include <iostream>` : 전처리기 지시어로, 컴파일 전에 해당 헤더의 내용을 삽입
- `std::` : 네임스페이스를 명시하여 표준 라이브러리 이름 충돌을 방지
- `main()` 함수: 프로그램의 진입점, 반드시 `int` 타입을 반환해야 함

---

2. 입출력 시스템

```
int value;
std::cin >> value;
std::cout << "값: " << value << std::endl;
```

- `<<` 연산자는 왼쪽 피연산자를 반환하므로 연쇄 호출이 가능
- `std::endl`은 줄바꿈과 동시에 출력 버퍼를 플러시(비움)함
    - 성능상 주의가 필요하나, 디버깅에는 유리함

---

3. 제어 흐름

```
// while 루프
int sum = 0, val = 1;
while (val <= 10) {
    sum += val;
    ++val;
}

// for 루프
for (int i = 1; i <= 10; ++i) {
    sum += i;
}
```

- 반복문 구조는 C와 유사
- `++val`은 `val++`보다 약간 더 효율적

---

4. 클래스 기초 (Sales_item 사용 예시)

```
Sales_item item1, item2;
std::cin >> item1 >> item2;
if (item1.isbn() == item2.isbn()) {
    std::cout << item1 + item2 << std::endl;
}
```

디버깅과 버퍼 플러시의 중요성

```
std::cout << "프로그램 시작"; // 버퍼 미출력
// 프로그램이 크래시나면 출력 내용이 사라짐
```

```
std::cout << "프로그램 시작" << std::endl; // 버퍼 플러시됨
```

출력 버퍼가 플러시되지 않으면, 디버깅 시 직전 로그가 보이지 않아 원인을 찾기 어려움

2장 - Variables and Basic Types

1. 기본 타입

```
int i = 42;
long long ll = 42LL;
unsigned int ui = 42U;

float f = 3.14f;
double d = 3.14;
long double ld = 3.14L;

char c = 'A';
wchar_t wc = L'A';
```

- `char`의 signed/unsigned 여부는 컴파일러에 따라 다름
- 부동소수점 정밀도: float(7자리), double(16자리)

---

2. 변수 초기화 방식

```
int a = 0;      // 직접 초기화
int b{0};       // 리스트 초기화 (정보 손실 방지)
int c(0);       // 괄호 초기화
int d = {0};    // 리스트 초기화
```

```
long double ld = 3.14159;
int a{ld};      // 오류: 정보 손실 가능성
int b = ld;     // 경고: 하지만 컴파일 가능
```

---

3. 참조 (Reference)

```
int val = 1024;
int &refVal = val;

refVal = 2;       // val도 2로 바뀜
int ii = refVal;  // ii = 2
```

- 참조는 반드시 초기화해야 함
- 한 번 바인딩된 참조는 다른 객체로 변경 불가

- 참조는 객체가 아니므로 자체 주소가 없음

---

4. 포인터 (Pointer)

```
int ival = 42;
int *p = &ival;

std::cout << *p; // 42 출력
*p = 0;          // ival의 값이 0으로 변경됨
```

포인터의 4가지 상태

1. 객체를 정상적으로 가리킴
2. 객체 끝 다음 위치를 가리킴 (접근 불가)
3. 널 포인터 (`nullptr`)
4. 잘못된 주소를 가리킴 (정의되지 않은 동작)

const 조합 예시

```
const int ci = 42;
const int *p1 = &ci;        // const 객체를 가리키는 포인터
int *const p2 = &ival;      // 포인터 자체가 const
const int *const p3 = &ci;  // const 포인터가 const 객체 가리킴

*p1 = 0;    // 오류
p2 = &ci;   // 오류
```

포인터 vs 참조 비교

특성	포인터	참조
재할당	가능	불가능
초기화	선택적	필수
널 값 가능	가능	불가능
산술 연산	가능	불가능

---

5. auto와 decltype

```
auto i = 42;                // i: int
auto &r = i;                // r: int&
const auto &cr = i;         // cr: const int&

decltype(i) j = i;          // j: int
decltype((i)) k = i;        // k: int& (괄호 주의)
```

- `auto`는 top-level const를 무시함
- `decltype((var))`는 항상 참조 타입
- `decltype(var)`는 변수 타입 그대로 추론

포인터 디버깅 습관

```
int *p;     // 초기화되지 않음
*p = 42;    // 미정의 동작 (런타임 에러 가능)
```

안전한 사용 예시

```
int *p = nullptr;
if (p) {
    *p = 42;
}
```

디버깅을 위한 방법

```
#include <cassert>

void safeAccess(int *p) {
    assert(p != nullptr);  // 디버그 모드에서만 체크
    *p = 42;
}
```

포인터 관련 문제는 런타임 에러로 이어지기 쉽기 때문에, 항상 초기화하고 사용 전 검사를 습관화하는 것이 중요함.