1️⃣

C++(Modern C++) 시작하기

C++ 언어 참조

- **C++ 시작하기**

```
C Style 프로그래밍의 주요 버그 클래스 중 하나는 메모리 누수입니다.
메모리 누수의 흔한 원인은 new를 사용하여 할당된 메모리의 delete 호출
최신 C++은 Resource Acquisition Is Initialization(RAII) 원칙을
리소스(힙 메모리, 파일 핸들, 소켓 등)는 개체가 소유해야 합니다.
이 개체는 해당 생성자에서 새로 할당된 리소스를 만들거나 받아 해당 소멸지
RAII 원칙은 개체가 범위를 벗어나면 모든 리소스가 운영체제에 제대로 반환
```

```
C++의 본래 요구 사항 중 하나는 C 언어와의 역 호환성이었습니다. 따라서
배열, null 종료 문자열, 기타 기능을 통해 항상 C 스타일 프로그래밍이 가
이로 인해 성능이 향상될 수 있는 반면 버그 및 복잡성이 생길 수도 있습니다
C 스타일 관용구를 사용할 필요성을 크게 줄이는 기능이 강조되었습니다. 기존
여전히 존재합니다. 그러나 최신 C++ 코드에서는 더 적은 수의 코드가 필요합
```

- **리소스 및 스마트 포인터**

```
/*
C 스타일 프로그래밍의 주요 버그 클래스 중 하나는 메모리 누수입니다.
메모리 누수의 흔한 원인은 new를 사용하여 할당된 메모리의 delete 호출
RAII 원칙을 쉽게 채택할 수 있도록 C++ 표준 라이브러리는 std::unique
std::shared_ptr, std::weak_ptr의 세가지 스마트 포인터 형식을 제공
스마트 포인터는 소유하고 있는 메모리의 할당 및 삭제를 처리합니다.
new 및 delete에 대한 호출은 unique_ptr 클래스에 의해 캡슐화됩니다.
범위를 벗어나면 unique_ptr의 소멸자가 호출되어 배열을 위해 할당된 메모
다음은 이를 설명하는 예제입니다.
*/

#include <memory>
```

```
class Widget
{
private:
    std::unique_ptr<int[]> data;

public:
    Widget(const int size) { data = std::make_unique<int[]
    void DoSomething();
};

void FunctionUsingWidget()
{
    Widget w(1000000);
    w.DoSomething();
}
// automatic destruction and /*
가능하면 스마트 포인터를 사용하여 힙 메모리를 관리합니다. new 및 delet
하는 경우 RAII 원칙을 따릅니다.
```

- **std::string 및 std::string_view**

```
/*
c 스타일 문자열은 버그의 또 다른 주요 원인입니다. std::string, std:
c 스타일 문자열과 관련된 거의 모든 오류를 제거할 수 있습니다. 또한 검색
앞에 추가 등에서 맴버 함수의 이점을 얻을 수 있습니다. 두 가지 모두 속도
최적화되어 있습니다. 읽기 전용 엑세스 만 필요한 함수에 문자열을 전달하는
C++17에서는 std::string_view를 사용하여 훨씬 큰 성능상 이점을 얻을
*/
```

- **std::vector 및 기타 표준 라이브러리 컨테이너**

```
/*
std::vector 및 기타 표준 라이브러리 컨테이너
표준 라이브러리 컨테이너는 모두 RAII 원칙을 따르며 요소의 안전한 탐색을
반복기를 제공합니다. 또한 성능에 고도로 최적화되어 있으며, 정확성을 철저
이 같은 컨테이너를 사용하면 사용자 지정 데이터 구조에 유입될 수 있는 버
가능성을 없앨 수 있습니다. C++에서 원시 배열 대신 vector를 순차 컨테
*/
```

```
#include <vector>
#include <string>
#include <map>
int main()
{
    std::vector<std::string> apples;
    appels.push_back("Granny Smith");

    std::map<std::string, std::string> appleColor;
    appleColor["Granny Smith"] = "Green";
}
```

- **표준 라이브러리 알고리즘**

```
/*
프로그램을 위한 사용자 지정 알고리즘 작성이 필요하다고 가정하기 전에
먼저 C++ 표준 라이브러리 알고리즘을 검토하세요. 표준 라이브러리에는
검색, 정렬, 필터링, 무작위화 등 여러 일반적 작업을 위해 계속 늘어나는 알
모음이 포함되어 있습니다. 수식 라이브러리는 광범위합니다. C++17 이상에서
많은 알고리즘의 병렬 버전이 제공됩니다.
몇가지 중요한 예는 다음과 같습니다.
    - 기본 탐색 알고리즘인 for_each(범위 기반 for 루프와 함께 사용)
    - 컨테이너 요소의 not-in-place 수정을 위한 transform
    - 기본 검색 알고리즘인 find_if
    - sort, lower_bound, 기타 기본 정렬 및 검색 알고리즘
비교자를 작성하려면 strict<를 사용하고 가능한 경우 명명된 람다를 사용합
*/

#include <algorithm>

void Algorithm()
{
    auto comp = [](const Widget &w1, const Widget &w2)
    {
        return w1.weight() < w2.weight();
    }
```

```
    sort(v.begin(), v.end(), comp);
    auto i = lower_bound(v.begin(), v.end(), widget{0}, co
}
```

- **명시적 형식 이름 대신 auto**

```
/*
C++11에서는 변수, 함수, 템플릿 선언에 사용할 auto 키워드가 도입되었습
auto가 개체의 형식을 추론하도록 컴파일러에 지시하므로 명시적으로 입력할
auto는 추론된 형식이 중첩된 템플릿인 경우 특히 유용합니다.
*/

void AutoExample()
{
    std::map<int, list<std::stirng>>::iterator i = m.begin
    auto i = m.begin();
}
```

- **범위 기반 for 루프**

```
/*
배열 및 컨테이너에 대한 C 스타일 반복은 인덱싱 오류가 발생하기 쉬우며 입
이러한 오류를 제거하고 코드를 더 읽기 쉽게 만들려면 표준 라이브러리 컨테
원시 배열과 함께 범위 기반 for 루프를 사용하세요.
*/

#include <iostream>
#include <vector>

void Forloop()
{
    std::vector<int> v{1, 2, 3};

    // c-style
    for (int i = 0; i < v.size(); i++)
    {
        std::cout << v[i];
    }
```

```
    for (auto &num : v)
    {
        std::cout << num;
    }
}
```

- **매크로 대신 constexpr 식**

```
/*
C와 C++의 매크로는 컴파일 전에 전처리기에 의해 처리되는 토큰이빈다. 매크
각 인스턴스는 파일이 컴파일되기 전에 정의된 값 또는 식으로 교체됩니다. 
일반적으로 C 스타일 프로그래밍에서 컴파일 시간 상수값을 정의하는데 사용됩
그러나 매크로는 오류가 발생하기 쉬우며 디버그하기 어렵습니다. 최신 C++에
컴파일 시간 상수에 constexpr 변수를 사용하는 것이 좋습니다.
*/

#define SIZE 10          // C-style
constexpr int size = 10; // modern C++
```

- **균일한 초기화**

```
/*
최신 C++에서는 모든 형식에 중괄호 초기화를 사용할 수 있습니다.
이러한 형태의 초기화는 배열, 벡터 또는 기타 컨테이너를 초기화 할 때 특히
편리합니다. 다음 예제에서는 s 인스턴스 세개를 사용하여 v2가 초기화됩니다
v3는 중괄호를 사용하여 초기화되는 s 인스턴스 세 개를 사용하여 초기화됩니
컴파일러는 v3의 선언된 형식을 기반으로 각 요소의 형식을 추론합니다.
*/

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

void Initialize()
```

```
{
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++
    std::vector<S> v2{s1, s2, s3};


}
```

- **이동 의미 체계**

```
/*
최신 C++는 불필요한 메모리 복사본을 제거할 수 있는 이동 의미 체계를 제
이전 버전의 C++에서는 특정 상황에서 복사본이 불가피했습니다. 이동 작업은
복사본을 만들지 않고 한 개체에서 다음 개체로 리소스의 소유권을 이전합니다
일부 클래스는 힙 메모리, 파일 핸들 등의 리소스를 소유합니다. 리소스 소유
이동 생성자를 정의하고 할당 연산자를 이동할 수 있습니다. 컴파일러는 복사
상황에서 오버로드 확인 중에 이러한 특수 맴버를 선택합니다. 표준 라이브러
형식은 개체에 대해 이동 생성자를 호출합니다.
*/
```

- **람다 식**

```
/*
C 스타일 프로그래밍에서는 함수 포인터를 사용하여 함수를 다른 함수에 전달
함수 포인터는 유지 관리하고 이해하기 불편합니다. 함수 포인터가 참조하는
호출되는 지점과 멀리 떨어진 소스코드 다른곳에서 정의될 수 있습니다.
또한 형식이 안전하지 않습니다. 최신 C++는 함수 개체, 연산자를 재정의 o
하는 클래스를 제공하므로 함수처럼 호출할 수 있습니다. 함수 개체를 만드는
인라인 람다 식을 사용하는 것입니다.
*/
```

```
void LambdaExpression()
{
    std::vector<int> v{1, 2, 3, 4, 5};
    int x = 2;
    int y = 4;
    auto result = find_if(begin(v), end(v), [=](int i)
                          { return i > x && i < y; });
}

/*
람다식 [=](int i) {return i > x && i < y;}은 형식 int의 단일 인
인수가 보다 크고 작은 xy지 여부를 나타내느 부울을 반환하는 함수로 읽을
변수와 y 주변 컨텍스트의 변수를 x 람다에서 사용할 수 있습니다.
해당 [=] 변수가 값으로 캡처되도록 지정합니다. 즉, 람다 식에는 해당 값의
*/
```

- **예외**

```
/*
최신 C++는 오류 조건을 보고하고 처리하는 가장 좋은 방법으로 오류 코드가
*/
```

- **std::atomic**

```
/*
std::atomic
스레드간 통신 메커니즘에 C++ 표준 라이브러리 std::atomic 구조체와 관
*/
```

- **std::variant**

```
std::variant
C 스타일 프로그래밍에서는 일반적으로 공용 구조체를 사용하여
서로 다른 형식의 멤버가 동일한 메모리 위치를 점유할 수 있도록 함으로써
메모리를 보존합니다. 하지만 공용 구조체는 형식이 안전하지 않으며 프로그래
오류가 발생하기 쉽습니다. C++17에서는 공용 구조체보다 강력하고 안전한
```

대안으로 std::variant 클래스가 도입되었습니다. std::visit 함수를 사
variant 형식의 멤버에 형식이 안전한 방식으로 액세스할 수 있습니다.