C++ 치트시트

기본 구조 및 컴파일 환경

C++ 프로그램은 일반적으로 `#include` 전처리기 지시자, `main` 함수, 그리고 실행 로직을 담은 구문들로 구성됩니다.

```
#include <iostream> // 입출력 스트림 라이브러리 포함

// 프로그램의 시작점인 메인 함수
int main() {
    std::cout << "Hello, World!" << std::endl; // 표준 출력 스트림으로 메시지 전송
    return 0; // 프로그램 정상 종료
}
```

- 컴파일 및 실행 (G++ 기준):

```
g++ -o hello hello.cpp # 컴파일하여 실행 파일 생성
./hello                # 생성된 실행 파일 실행
```

데이터 타입, 변수 및 연산자

- 기본 데이터 타입: `int` (정수), `double` (실수), `char` (문자), `bool` (논리형), `float` (실수).
- `std::string`: 문자열 처리를 위한 표준 클래스 타입.
- `const`: 상수를 선언할 때 사용합니다. `const double PI = 3.14;`
- `auto`: 컴파일러가 변수의 타입을 자동으로 추론합니다. `auto x = 5;`
- 주요 연산자: 산술(`+`, `-`, `*`, `/`, `%`), 관계(`==`, `!=`, `<`, `>`), 논리(`&&`, `||`, `!`).

제어문 (조건문 및 반복문)

- `if-else` 조건문:

```
if (조건식) {
    // 조건이 참일 때 실행
} else if (다른조건) {
    // 다른 조건이 참일 때 실행
} else {
    // 모든 조건이 거짓일 때 실행
}
```

- `switch` 선택문:

```
switch (변수) {
    case 1:
        // 값 1일 때 실행
        break;
    case 2:
        // 값 2일 때 실행
        break;
    default:
        // 일치하는 값이 없을 때 실행
}
```

- 반복문:
  - `for` 루프: `for (int i = 0; i < 5; ++i) { ... }`
  - 범위 기반 `for` 루프 (C++11 이상):

```
std::vector<int> v = {1, 2, 3};
for (int item : v) {
    std::cout << item << std::endl;
}
```

  - `while` 루프: `while (조건식) { ... }`

함수 및 오버로딩

- 함수 정의 방식:

```
반환타입 함수명(매개변수타입 매개변수명) {
    // 함수 내부 로직
    return 반환값;
}
```

- 함수 오버로딩: 이름은 같지만 매개변수의 타입이나 개수가 다른 여러 함수를 정의할 수 있습니다.
- 기본 매개변수 (Default Arguments): `void print(int value, int base=10);`

포인터(Pointer)와 참조(Reference)

- 포인터 (`*`): 변수의 메모리 주소를 저장하는 변수입니다.

```
int var = 10;
int* ptr = &var; // var의 주소를 ptr에 저장
std::cout << *ptr; // ptr이 가리키는 실제 값(10)을 출력 (역참조)
```

- 참조 (`&`): 기존 변수의 또 다른 이름(별칭)입니다. 선언과 동시에 반드시 초기화해야 합니다.

```
int var = 10;
int& ref = var; // ref는 var의 별칭이 됨
ref = 20; // var의 값도 20으로 변경됨
```

객체 지향 프로그래밍 (클래스와 객체)

- 클래스 정의:

```
class MyClass {
public: // 외부에서 접근 가능한 영역
    // 생성자 (Constructor)
    MyClass(int val) : my_field(val)
{}

    // 멤버 함수 (메서드)
    void myMethod() {
        std::cout << "Field is " << my_field << std::endl;
    }

private: // 클래스 내부에서만 접근 가능한 영역
    // 멤버 변수 (필드)
    int my_field;
};
```

- 객체 생성: `MyClass obj(10);`
- 멤버 접근: `obj.myMethod();`
- 상속 (Inheritance): `class Derived : public Base { ... };`

STL (Standard Template Library) 핵심 컨테이너

std::vector

가장 많이 사용되는 동적 배열 컨테이너입니다.

```
#include <vector>
std::vector<int> vec;
vec.push_back(10); // 요소 추가
vec.push_back(20);
int first = vec[0]; // 인덱스를 통한 접근
vec.size(); // 저장된 요소 개수 확인
```

std::map

키-값(Key-Value) 쌍을 정렬된 상태로 유지하는 연관 컨테이너입니다.

```
#include <map>
std::map<std::string, int> ages;
ages["Alice"] = 30;
ages["Bob"] = 25;
```

std::string

문자열 조작을 위한 강력한 기능을 제공하는 클래스입니다.

```
#include <string>
std::string s = "Hello";
s += ", World!";
s.substr(0, 5); // 0번 인덱스부터 5글자 추출 ("Hello")
s.find("World"); // 특정 문자열 위치 검색
```

알고리즘 활용

`<algorithm>` 헤더는 컨테이너 처리를 위한 다양한 함수를 제공합니다.

- `std::sort(vec.begin(), vec.end());`
- `std::find(vec.begin(), vec.end(), value);`
- `std::for_each(vec.begin(), vec.end(), my_func);`

데이터 입출력 (I/O)

콘솔 입출력 (iostream)

```
#include <iostream>
int x;
std::cout << "숫자를 입력하세요: ";
std::cin >> x; // 표준 입력으로부터 값 수신
std::cerr << "에러 메시지 출력" << std::endl; // 표준 에러 스트림
```

파일 입출력 (fstream)

```
#include <fstream>
#include <string>

// 파일에 데이터 쓰기
std::ofstream outFile("data.txt");
outFile << "Hello, file!" << std::endl;
outFile.close();

// 파일로부터 데이터 읽기
std::ifstream inFile("data.txt");
std::string line;
while (std::getline(inFile, line)) {
    std::cout << line << std::endl;
}
inFile.close();
```

STL 컨테이너 상세 분석

시퀀스 컨테이너

```cpp
#include <vector>
#include <deque>
#include <list>
#include <array>

// std::vector: 동적 배열
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.reserve(100);  // 메모리 용량 사전 확보
vec.shrink_to_fit();  // 남는 용량 반환
vec.emplace_back(6);  // 인자로부터 객체를 직접 생성 (성능 우수)

// std::deque: 양방향 큐 (앞/뒤 모두 빠른 삽입/삭제)
std::deque<int> dq = {1, 2, 3};
dq.push_front(0);
dq.push_back(4);

// std::list: 이중 연결 리스트
std::list<int> lst = {1, 2, 3, 4, 5};
lst.sort();  // 리스트 전용 정렬 멤버 함수 사용

// std::array: C 스타일 배열의 객체형 버전 (고정 크기)
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.fill(0);  // 모든 요소를 특정 값으로 채움
```

연관 컨테이너

```cpp
#include <map>
#include <set>
#include <unordered_map>
#include <unordered_set>

// std::map: 정렬된 키-값 저장소
std::map<std::string, int> scores;
scores.emplace("Bob", 87);
auto it = scores.find("Alice");
if (it != scores.end()) std::cout << it->second << std::endl;

// std::set: 중복을 허용하지 않는 정렬된 요소 집합
std::set<int> numbers = {3, 1, 4, 1, 5};
auto [iter, inserted] = numbers.insert(3); // C++17 구조화된 바인딩 사용

// std::unordered_map: 해시 기반의 빠른 검색 제공
std::unordered_map<std::string, int> cache;
cache.reserve(1000); // 해시 테이블 크기 사전 할당
```

컨테이너 어댑터

```cpp
#include <stack>
#include <queue>
#include <priority_queue>

// std::stack: LIFO(후입선출) 구조
std::stack<int> stk;
stk.push(1);
int top = stk.top();
stk.pop();

// std::queue: FIFO(선입선출) 구조
std::queue<int> q;
q.push(1);
int front = q.front();
q.pop();

// std::priority_queue: 우선순위에 따른 추출 (기본값은 최대 힙)
std::priority_queue<int> pq;
pq.push(3);
int max_val = pq.top(); // 가장 큰 값 추출
```

STL 알고리즘 활용

검색 및 정렬

```cpp
#include <algorithm>
#include <vector>

std::vector<int> vec = {5, 2, 8, 1, 9, 3};

// 정렬 라이브러리 활용
std::sort(vec.begin(), vec.end()); // 오름차순
std::sort(vec.begin(), vec.end(), std::greater<int>()); // 내림차순

// 이진 검색 (정렬된 상태에서만 사용 가능)
bool found = std::binary_search(vec.begin(), vec.end(), 5);
auto it = std::lower_bound(vec.begin(), vec.end(), 5); // 하한선 검색

// 특정 조건으로 요소 찾기
auto it_cond = std::find_if(vec.begin(), vec.end(), [](int x) { return x > 5; });
```

데이터 수정 및 조작

```cpp
#include <algorithm>
#include <vector>

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

// 일괄 변환 (Transform)
std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2; });

// 조건부 복사
std::vector<int> evens;
std::copy_if(vec.begin(), vec.end(), std::back_inserter(evens),
             [](int x) { return x % 2 == 0; });

// 특정 요소 제거 (Erase-Remove 관용구)
vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
```

수치 데이터 처리

```cpp
#include <numeric>
#include <vector>

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

// 누적 합계 (Sum)
int sum = std::accumulate(vec.begin(), vec.end(), 0);

// 인접 차이 계산
std::vector<int> diffs(vec.size());
std::adjacent_difference(vec.begin(), vec.end(), diffs.begin());
```

고급 메모리 관리 및 스마트 포인터

스마트 포인터 사용

```cpp
#include <memory>

// std::unique_ptr: 유일한 소유권을 갖는 포인터
std::unique_ptr<int> uptr = std::make_unique<int>(42);
std::unique_ptr<int> uptr2 = std::move(uptr); // 소유권 이전만 가능

// std::shared_ptr: 참조 카운팅 기반의 공유 소유권 포인터
std::shared_ptr<int> sptr = std::make_shared<int>(42);
std::shared_ptr<int> sptr2 = sptr; // 참조 카운트 증가

// std::weak_ptr: shared_ptr의 순환 참조 문제를 해결하는 약한 참조
std::weak_ptr<int> wptr = sptr;
if (auto locked = wptr.lock()) {
    // 유효성 확인 후 사용
    std::cout << *locked << std::endl;
}
```

이동 의미론 (Move Semantics)

```cpp
#include <string>
#include <vector>

class Resource {
    int* buffer;
public:
    // 이동 생성자: 기존 객체의 자원을 빼앗아 효율적으로 생성
    Resource(Resource&& other) noexcept : buffer(other.buffer) {
        other.buffer = nullptr;
    }

    // 이동 할당 연산자
    Resource& operator=(Resource&& other) noexcept {
        if (this != &other) {
            delete[] buffer;
            buffer = other.buffer;
            other.buffer = nullptr;
        }
```

```
        return *this;
    }

    ~Resource() { delete[] buffer; }
};

Resource r1;
Resource r2 = std::move(r1); // 명시적인 자원 이동 수행
```

고급 템플릿 메타프로그래밍 (TMP)

클래스 템플릿 특수화

```
// 일반 템플릿 정의
template<typename T>
class Container {
public:
    void info() { std::cout << "General Type" << std::endl; }
};

// 특정 타입에 대한 완전 특수화
template<>
class Container<std::string> {
public:
    void info() { std::cout << "String Type" << std::endl; }
};

// 포인터 타입에 대한 부분 특수화
template<typename T>
class Container<T*> {
public:
    void info() { std::cout << "Pointer Type" << std::endl; }
};
```

가변 인자 템플릿 (Variadic Templates)

```
#include <iostream>

// C++17 Fold 표현식을 이용한 가변 인자 처리
template<typename... Args>
auto sumAll(Args... args) {
    return (args + ...); // 파라미터 팩 전개
}

// 재귀적 템플릿 처리
template<typename T>
void printAll(T arg) { std::cout << arg << std::endl; }

template<typename T, typename... Args>
void printAll(T first, Args... args) {
    std::cout << first << ", ";
    printAll(args...);
}
```

멀티스레딩 및 동시성 프로그래밍

스레드 생성 및 관리

```
#include <thread>
#include <mutex>
#include <future>

void task(int id) { std::cout << "Thread " << id << " 실행 중" << std::endl; }

int main() {
    std::thread t1(task, 1);
    t1.join(); // 스레드 작업 완료 대기
}
```

동기화 메커니즘

```
#include <mutex>
#include <shared_mutex>

class SafeCounter {
    mutable std::shared_mutex mtx;
    int val = 0;
public:
    int get() const {
        std::shared_lock lock(mtx); // 읽기 전용 공유 잠금
        return val;
    }
    void add() {
        std::unique_lock lock(mtx); // 쓰기 전용 독점 잠금
        val++;
    }
};
```

비동기 결과 처리 (Future/Promise)

```
#include <future>

// async를 통한 비동기 함수 실행
auto fut = std::async(std::launch::async, []() {
    return 42;
});

int result = fut.get(); // 비동기 작업 결과가 준비될 때까지 블로킹
```

코드 최적화 및 성능 향상 기법

컴파일러 최적화 힌트

```
// inline: 함수 호출 오버헤드 제거를 유도
inline int add(int a, int b) { return a + b; }

// C++20 [[likely]] / [[unlikely]]: 조건문 분기 예측 최적화
if (cond) [[likely]] {
    // 참일 확률이 높을 때
} else [[unlikely]] {
    // 거짓일 확률이 높을 때
}
```

메모리 레이아웃 및 캐시 효율

```
// 캐시 적중률(Cache Hit)을 높이기 위해 행 우선 순회 권장
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        process(matrix[i][j]); // 인접 메모리 연속 접근
    }
}
```

성능 벤치마킹 (Chrono 활용)

```
#include <chrono>

auto start = std::chrono::high_resolution_clock::now();
// 측정 대상 코드 실행
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> ms = end - start;
std::cout << "실행 시간: " << ms.count() << "ms" << std::endl;
```

Google Style Guide

명명 규칙 (Naming)

- 파일 이름: `my_useful_class.cc` (소문자, 언더스코어)
- 타입 (클래스/구조체): `MyExcitingClass` (UpperCamelCase)
- 변수 (로컬/데이터 멤버): `my_local_variable`, `my_member_variable_` (소문자, 멤버는 뒤에 언더스코어)
- 상수/열거형: `kDaysInAWeek` (소문자 k로 시작하는 CamelCase)
- 함수: `MyExcitingFunction()` (UpperCamelCase)
- 네임스페이스: `my_namespace` (소문자)
- 매크로: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN`

헤더 파일 (Header Files)

- Self-contained: 헤더는 그 자체로 컴파일 가능해야 함.
- #define Guard: `#ifndef PROJECT_PATH_FILE_H_` 형식 준수.
- Include 순서: 관련 헤더 -> C 시스템 -> C++ 표준 -> 다른 라이브러리 -> 프로젝트 헤더.
- Inline 함수: 10줄 이하의 짧은 경우만 헤더에 정의.

클래스 (Classes)

- Constructors: 생성자에서 가상 함수 호출 금지. 인자가 하나인 생성자는 `explicit` 키워드 사용.
- Struct vs Class: 단순 데이터 전달용은 `struct`, 로직이 포함되면 `class` 사용.
- Inheritance: `public` 상속만 사용. 다중 상속은 지양. `override` 또는 `final` 명시.
- Access Control: 데이터 멤버는 `private`으로 유지.

프로그래밍 관례

- Smart Pointers: 소유권 모델을 명확히 함 (`std::unique_ptr` 선호).
- Namespaces: 코드는 네임스페이스 안에 배치. `using namespace std;` 금지.
- Output Parameters: 결과값은 가급적 반환값으로 전달. 출력 인자는 참조 대신 포인터 사용 고려.
- C++20: 최신 표준(C++20) 사용을 지향하되 비표준 확장 기능 사용 금지.
- Exceptions: Google 내부에서는 지양하나 오픈소스에서는 상황에 맞춰 사용.