C++ 치트시트

1. 기본 구조 및 컴파일

C++ 프로그램은 일반적으로 `#include` 지시문, `main` 함수, 그리고 다양한 구문으로 구성됩니다.

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

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

- 컴파일 및 실행:

```
g++ -o hello hello.cpp
./hello
```

2. 변수, 타입, 연산자

- 기본 타입: `int`, `double`, `char`, `bool`, `float`.
- `std::string`: 문자열 타입.
- `const`: 상수 변수를 선언. `const double PI = 3.14;`
- `auto`: 컴파일러가 타입을 자동으로 추론. `auto x = 5;`
- 연산자: 산술(`+`, `-`, `*`, `/`, `%`), 관계(`==`, `!=`, `<`, `>`), 논리(`&&`, `||`, `!`).

3. 제어 흐름

- `if-else`:

```
if (condition) {
    // ...
} else if (condition) {
    // ...
} else {
    // ...
}
```

- `switch`:

```
switch (variable) {
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    default:
        // ...
}
```

- `for` 루프:

```
for (int i = 0; i < 5; ++i) {
    // ...
}
```

- 범위 기반 `for` 루프 (Range-based for):

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

- `while` 루프: `while (condition) { ... }`

4. 함수

- 함수 정의:

```
return_type function_name(param_type param_name) {
    // 함수 본문
    return value;
}
```

- 함수 오버로딩: 동일한 이름이지만 매개변수 타입이나 개수가 다른 여러 함수를 정의할 수 있습니다.
- 기본 매개변수: `void print(int value, int base=10);`

5. 포인터와 참조

- 포인터 (`*`): 변수의 메모리 주소를 저장.

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

- 참조 (`&`): 변수의 별칭. 선언 시 반드시 초기화해야 함.

```
int var = 10;
int& ref = var; // ref는 var의 또 다른 이름
ref = 20; // var도 20으로 변경됨
```

6. 클래스와 객체

- 클래스 정의:

```
class MyClass {
public: // 접근 지정자
    // 생성자
    MyClass(int val) : my_field(val) {}

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

private:
    // 멤버 변수 (필드)
    int my_field;
};
```

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

7. 표준 템플릿 라이브러리 (STL)

std::vector

동적 배열.

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

std::map

키-값 쌍을 저장하는 연관 컨테이너.

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

std::string

문자열을 다루는 클래스.

```
#include <string>
std::string s = "Hello";
s += ", World!";
s.substr(0, 5); // "Hello"
s.find("World"); // 7
```

알고리즘

`#include <algorithm>` 헤더에 포함.

- `std::sort(vec.begin(), vec.end());`
- `std::find(vec.begin(), vec.end(), value_to_find);`
- `std::for_each(vec.begin(), vec.end(), my_function);`

8. 입출력

콘솔 입출력 (iostream)

```
#include <iostream>
int x;
std::cout << "Enter a number: ";
std::cin >> x;
std::cerr << "This is an error message." << std::endl;
```

파일 입출력 (fstream)

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

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

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

9. STL 컨테이너 심화

시퀀스 컨테이너

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

// std::vector - 동적 배열
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.reserve(100);  // 용량 미리 할당
vec.shrink_to_fit();  // 불필요한 용량 해제
vec.emplace_back(6);  // 직접 생성 (push_back보다 효율적)

// std::deque - 양방향 큐
std::deque<int> dq = {1, 2, 3};
dq.push_front(0);  // 앞쪽에 추가
dq.push_back(4);   // 뒤쪽에 추가
```

```
// std::list - 이중 연결 리스트
std::list<int> lst = {1, 2, 3, 4, 5};
lst.splice(lst.begin(), other_list);  // 다른 리스트와 병합
lst.sort();  // 정렬 (std::sort와 달리 멤버 함수)

// std::array - 고정 크기 배열
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.fill(0);  // 모든 요소를 0으로 채움
```

연관 컨테이너

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

// std::map - 정렬된 키-값 쌍
std::map<std::string, int> scores;
scores["Alice"] = 95;
scores.emplace("Bob", 87);  // 직접 생성
auto it = scores.find("Alice");
if (it != scores.end()) {
    std::cout << it->second << std::endl;
}

// std::set - 정렬된 유니크 요소
std::set<int> numbers = {3, 1, 4, 1, 5};
numbers.insert(2);
auto [iter, inserted] = numbers.insert(3);  // C++17 구조화 바인딩

// std::unordered_map - 해시 테이블
std::unordered_map<std::string, int> cache;
cache.reserve(1000);  // 해시 테이블 크기 미리 할당
cache.max_load_factor(0.75);  // 로드 팩터 설정
```

컨테이너 어댑터

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

// std::stack - 스택
std::stack<int> stk;
stk.push(1);
stk.push(2);
int top = stk.top();
stk.pop();

// std::queue - 큐
std::queue<int> q;
q.push(1);
q.push(2);
int front = q.front();
q.pop();

// std::priority_queue - 우선순위 큐
std::priority_queue<int> pq;  // 최대 힙
pq.push(3);
pq.push(1);
pq.push(4);
int max_val = pq.top();  // 4

// 최소 힙으로 만들기
std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;
```

10. STL 알고리즘

검색 및 정렬

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

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

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

// 부분 정렬
std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());

// 이진 검색 (정렬된 컨테이너에서)
bool found = std::binary_search(vec.begin(), vec.end(), 5);
auto it = std::lower_bound(vec.begin(), vec.end(), 5);
auto it2 = std::upper_bound(vec.begin(), vec.end(), 5);

// 선형 검색
auto it3 = std::find(vec.begin(), vec.end(), 5);
auto it4 = std::find_if(vec.begin(), vec.end(), [](int x) { return x > 5; });
```

수정 알고리즘

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

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

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

// 복사
std::vector<int> dest(5);
std::copy(vec.begin(), vec.end(), dest.begin());

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

// 채우기
std::fill(vec.begin(), vec.end(), 0);
std::iota(vec.begin(), vec.end(), 1);  // 1부터 순차적으로 채움

// 제거
vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
vec.erase(std::remove_if(vec.begin(), vec.end(), [](int x) { return x > 3; }), vec.end());
```

수치 알고리즘

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

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

// 합계
int sum = std::accumulate(vec.begin(), vec.end(), 0);
int product = std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());

// 내적
std::vector<int> vec2 = {2, 3, 4, 5, 6};
int dot_product = std::inner_product(vec.begin(), vec.end(), vec2.begin(), 0);

// 부분합
std::vector<int> partial_sums(vec.size());
std::partial_sum(vec.begin(), vec.end(), partial_sums.begin());

// 인접 차이
std::vector<int> differences(vec.size() - 1);
std::adjacent_difference(vec.begin(), vec.end(), differences.begin());
```

11. 메모리 관리 심화

스마트 포인터

```
#include <memory>

// std::unique_ptr - 독점 소유권
std::unique_ptr<int> ptr = std::make_unique<int>(42);
std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);

// 소유권 이전
std::unique_ptr<int> ptr2 = std::move(ptr);

// std::shared_ptr - 공유 소유권
std::shared_ptr<int> shared_ptr = std::make_shared<int>(42);
std::shared_ptr<int> shared_ptr2 = shared_ptr;  // 참조 카운트 증가
```

```
// std::weak_ptr - 약한 참조
std::weak_ptr<int> weak_ptr =
shared_ptr;
if (auto locked = weak_ptr.lock()) {
    // weak_ptr이 유효한 동안 사용
    std::cout << *locked << std::endl;
}
```

메모리 풀과 커스텀 할당자

```
#include <memory>
#include <vector>

// 커스텀 할당자 예제
template<typename T>
class PoolAllocator {
private:
    std::vector<T> pool;
    std::vector<bool> used;
    size_t next_free = 0;

public:
    using value_type = T;

    PoolAllocator(size_t size) :
pool(size), used(size, false) {}

    T* allocate(size_t n) {
        if (n != 1) return nullptr;

        for (size_t i = 0; i <
pool.size(); ++i) {
            if (!used[i]) {
                used[i] = true;
                return &pool[i];
            }
        }
        return nullptr;
    }

    void deallocate(T* ptr, size_t n) {
        if (n != 1) return;

        for (size_t i = 0; i <
pool.size(); ++i) {
            if (&pool[i] == ptr) {
                used[i] = false;
                break;
            }
        }
    }
};

// 사용 예제
PoolAllocator<int> allocator(100);
std::vector<int, PoolAllocator<int>>
vec(allocator);
```

이동 의미론 (Move Semantics)

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

class Resource {
private:
    std::string data;
    int* buffer;
    size_t size;

public:
    // 이동 생성자
    Resource(Resource&& other) noexcept
        : data(std::move(other.data)),
          buffer(other.buffer),
          size(other.size) {
        other.buffer = nullptr;
        other.size = 0;
    }

    // 이동 할당 연산자
    Resource& operator=(Resource&&
other) noexcept {
        if (this != &other) {
            delete[] buffer;
            data =
std::move(other.data);
            buffer = other.buffer;
            size = other.size;
            other.buffer = nullptr;
            other.size = 0;
        }
        return *this;
    }

    // 복사 생성자 (delete로 복사 방지)
    Resource(const Resource&) = delete;
    Resource& operator=(const Resource&)
= delete;

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

// std::move 사용
Resource r1;
Resource r2 = std::move(r1);  // 이동 생
성자 호출
```

12. 템플릿 프로그래밍 심화

클래스 템플릿 특수화

```
// 기본 템플릿
template<typename T>
class Container {
private:
    T data;
public:
    Container(T value) : data(value) {}
    void print() { std::cout <<
"Generic: " << data << std::endl; }
};

// 완전 특수화
template<>
class Container<std::string> {
private:
    std::string data;
public:
    Container(std::string value) :
data(value) {}
    void print() { std::cout << "String:
" << data << std::endl; }
};

// 부분 특수화
template<typename T>
class Container<T*> {
private:
    T* data;
public:
    Container(T* value) : data(value) {}
    void print() { std::cout <<
"Pointer: " << *data << std::endl; }
};
```

가변 인수 템플릿 (Variadic Templates)

```
#include <iostream>

// 재귀적 템플릿
template<typename T>
void print(T&& t) {
    std::cout << t << std::endl;
}

template<typename T, typename... Args>
void print(T&& t, Args&&... args) {
    std::cout << t << " ";
    print(std::forward<Args>(args)...);
}

// fold 표현식 (C++17)
template<typename... Args>
auto sum(Args... args) {
    return (args + ...);  // fold
expression
}

// 사용 예제
print(1, 2.5, "hello", 'c');
int result = sum(1, 2, 3, 4, 5);
```

SFINAE (Substitution Failure Is Not An Error)

```
#include <type_traits>

// 템플릿 메타프로그래밍을 이용한 타입 검사
template<typename T>
typename
std::enable_if<std::is_integral<T>::value,
void>::type
process_integer(T value) {
    std::cout << "Processing integer: "
<< value << std::endl;
}

template<typename T>
typename
std::enable_if<std::is_floating_point<T>::value,
void>::type
process_float(T value) {
    std::cout << "Processing float: " <<
value << std::endl;
}

// C++17의 if constexpr 사용
template<typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>)
{
```

```
        std::cout << "Integer: " <<
value << std::endl;
    } else if constexpr
(std::is_floating_point_v<T>) {
        std::cout << "Float: " << value
<< std::endl;
    } else {
        std::cout << "Other type" <<
std::endl;
    }
}
```

13. 동시성 프로그래밍

스레드 관리

```
#include <thread>
#include <mutex>
#include <condition_variable>
#include <atomic>

// 기본 스레드 생성
void worker_function(int id) {
    std::cout << "Worker " << id << " is
running" << std::endl;
}

std::thread t1(worker_function, 1);
std::thread t2([]() { std::cout <<
"Lambda thread" << std::endl; });

t1.join();
t2.join();
```

뮤텍스와 동기화

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

class ThreadSafeCounter {
private:
    mutable std::shared_mutex mutex_;
    int value_ = 0;

public:
    int get() const {

std::shared_lock<std::shared_mutex>
lock(mutex_);
        return value_;
    }

    void increment() {

std::unique_lock<std::shared_mutex>
lock(mutex_);
        ++value_;
    }

    void reset() {

std::unique_lock<std::shared_mutex>
lock(mutex_);
        value_ = 0;
    }
};
```

원자적 연산

```
#include <atomic>

std::atomic<int> counter{0};

// 원자적 연산
counter.fetch_add(1);  // 원자적 증가
counter.store(10);     // 원자적 저장
int value = counter.load();  // 원자적 로
드

// Compare-and-Swap (CAS)
int expected = 10;
if
(counter.compare_exchange_strong(expected,
20)) {
    std::cout << "CAS succeeded" <<
std::endl;
}
```

Future와 Promise

```
#include <future>
#include <chrono>

// std::async를 이용한 비동기 실행
auto future_result =
std::async(std::launch::async, []() {

std::this_thread::sleep_for(std::chrono::seconds(1));
    return 42;
});

// 결과 대기
int result = future_result.get();

// std::promise와 std::future
std::promise<int> promise;
std::future<int> future =
promise.get_future();

std::thread worker([&promise]() {

std::this_thread::sleep_for(std::chrono::seconds(1));
    promise.set_value(100);
});

int value = future.get();
worker.join();
```

14. 성능 최적화 기법

컴파일러 최적화

```
// 인라인 함수
inline int square(int x) {
    return x * x;
}

// 컴파일러 힌트
[[likely]] if (condition) {
    // 이 분기가 실행될 가능성이 높음
}

[[unlikely]] if (rare_condition) {
    // 이 분기가 실행될 가능성이 낮음
}

// 루프 최적화 힌트
#pragma unroll
for (int i = 0; i < 4; ++i) {
    // 루프 언롤링
}
```

캐시 친화적 프로그래밍

```
#include <vector>

// 좋은 예: 순차적 메모리 접근
std::vector<int> data(1000000);
for (size_t i = 0; i < data.size(); ++i)
{
    data[i] = i * 2;
}

// 나쁜 예: 랜덤 메모리 접근
std::vector<std::vector<int>>
matrix(1000, std::vector<int>(1000));
// 행 우선 순회 (캐시 친화적)
for (int i = 0; i < 1000; ++i) {
    for (int j = 0; j < 1000; ++j) {
        matrix[i][j] = i + j;
    }
}
```

프로파일링과 벤치마킹

```
#include <chrono>
#include <iostream>

class Timer {
private:

std::chrono::high_resolution_clock::time_point
start_time;

public:
    Timer() :
start_time(std::chrono::high_resolution_clock::now
{}

    ~Timer() {
        auto end_time =
std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::microsecon
            end_time - start_time);
        std::cout << "Execution time: "
<< duration.count() << " microseconds"
<< std::endl;
    }
};

// 사용 예제
{
    Timer timer;
    // 측정할 코드
    std::vector<int> vec(1000000);
    std::sort(vec.begin(), vec.end());
}
```