1. 메모리 4영역

스택, 힙, 데이터, 코드

스택(Stack) 영역

역할: **함수 호출 시 지역 변수**와 **매개 변수**를 저장합니다.

구조: LIFO(Last In, First Out) 구조로 관리되며 함수 호출 시마다 새로운 스택 프레임이 추가되고 함수가 종료되면 해당 프레임이 제거됩니다.

성질: 자동으로 할당되고 해제되므로 메모리 관리를 직접 할 필요가 없습니다.

※ 스택 영역은 프로그램이 실행될 때 함수 호출과 관련된 정보를 저장하는 메모리 공간

– 호출 시마다 스택 프레임(Stack Frame)이 생성되고, 함수가 종료되면 해당 스택 프레임이 제거

힙(Heap) 영역

역할: **동적 메모리 할당**을 위해 사용됩니다.

구조: 메모리 블록을 동적으로 할당하고 해제하는 방식으로, 관리가 자유롭지만 복잡한 관리가 필요합니다.

성질: 프로그래머가 명시적으로 할당과 해제를 관리해야 하며, 이를 잘못 처리하면 메모리 누수나 파편화가 발생할 수 있습니다.

※ 프로그램 실행 중 동적으로 메모리를 할당하고 해제할 수 있는 메모리 공간이고 크기가 변동될 수 있는 데이터를 관리하는 데 주로 사용

특징 : 동적 메모리 할당, 자유로운 크기, 직접 관리 필요, 메모리누수 위험

데이터(Data) 영역

역할: 전역 변수와 정적 변수를 저장합니다.

구조: **프로그램 시작 시 할당**되며, 프로그램 **종료 시까지 존재**합니다.

성질: 초기화된 데이터와 초기화되지 않은 데이터로 나뉘며, 초기화된 데이터는 .data 섹션에, 초기화되지 않은 데이터는 .bss 섹션에 저장됩니다.

※ 예) int a = 10; int b;와 같이 초기화된 데이터 영역(.data)이나 초기화 되지 않은 영역(.bss)을 저장

코드(Code) 영역

역할: 실행할 프로그램의 코드(명령어)를 저장합니다.

구조: 실행 파일이 로드될 때 메모리에 로드되며 읽기 전용으로 설정되어 있습니다.

성질: 일반적으로 변경되지 않으며, 프로그램의 실행 속도에 영향을 미칠 수 있는 중요한 부분입니다.

※ 초기상태는 코드영역 비어있음, 프로그램이 로드되면 코드가 코드 영역에 저장

예) 초기 상태 : 코드 영역이 비어있는 상태

프로그램 로드 :

```
void foo() { /* some code */ }
void bar() { /* some code */ }
| foo() 코드 |
| bar() 코드 |
```

2. 상수 선언방법

const와 #define

특성	const	#define
타입안전성	있음	없음
스코프	븍록스코프	파일 스코프 전체
디버깅	용이	어려움
컴파일 타임	초기화 시점에 값 결정	전처리기 단계에서 텍스트치환
사용방법	const int Max_USERS = 100;	#define MAX_USERS 100

const 사용법

```
const int MAX_USERS = 100;
const double PI = 3.14159;
```

#define 매크로 사용법

```
#define MAX_USERS 100
#define PI 3.14159
```

const: 타입 안전성을 제공하며, 블록 스코프를 가질 수 있습니다. 디버깅이 용이합니다.
#define: 단순한 텍스트 치환을 수행하며, 타입 안전성이 없고 파일 전체에서 사용됩니다. 디버깅이 어렵습니다.
일반적으로, 타입 안전성과 스코프 제어를 제공하는 const를 사용하는 것이 좋습니다.

3. 포인터 & 참조자

포인터란?(Pointers)

포인터는 메모리 주소를 저장하는 변수 : 포인터는 다른 변수의 메모리 주소를 가리킵니다.

예)

```
int num = 5;
int *ptr = &num; // 포인터 ptr은 변수 num의 주소를 저장
// 포인터를 통해 변수에 접근
cout << *ptr; // 출력: 5
```

포인터의 장점

동적 메모리 할당: new 키워드를 사용하여 메모리 동적으로 할당 가능
배열, 구조체 등 복잡한 자료 구조를 다룰 때 유용

주의할 점

포인터는 NULL 또는 nullptr로 초기화해야 안전하게 사용 가능
잘못된 메모리 주소에 접근하면 프로그램이 충돌할 수 있음

참조자란?(References)

참조자는 변수에 대한 별명(alias) : 다른 변수의 별명으로, 해당 변수를 직접 참조함.

예)

```
int num = 5;
int &ref = num; // ref는 num 변수의 별명
ref = 10; // num의 값이 변경됨
cout << num; // 출력: 10
```

참조자의 특징

한 번 선언되면 다른 변수를 가리키지 못하고 계속 같은 변수를 참조

포인터처럼 NULL 값이나 다른 변수로 초기화할 필요 없음

주의할 점

참조자는 반드시 초기화되어야 함

함수 인자로 전달할 때 복사본이 아니라 직접 참조하므로 값이 변경될 수 있음

결론 : 포인터와 참조자

포인터는 동적 할당 및 배열 등의 복잡한 데이터 구조에 유용

참조자는 코드를 간결하게 만들고, 의도를 명확히 전달할 수 있음

포인터와 참조의 비교

개념	포인터	참조
선언	`int* ptr;`	`int& ref = var;`
초기화	`ptr = &var;`	`ref = var;`
값 접근	`*ptr`	`ref`
재할당	가능 (`ptr = &otherVar;`)	불가능 (`ref = &otherVar;`는 불가)
nullptr	사용 가능 (`ptr = nullptr;`)	없음
메모리 주소	직접 저장 및 조작 가능	불가능

4. 객체지향 프로그래밍의 3가지 속성

· 상속 (Inheritance), 캡슐화 (Encapsulation), 다형성 (Polymorphism)

상속 (Inheritance)

· 상속은 객체 지향 프로그래밍에서 가장 기본적인 개념 중 하나

· 부모 클래스(또는 상위 클래스)의 특성(속성과 메서드)을 자식 클래스(또는 하위 클래스)가 물려받는 것을 의미

cpp 코드 복사

```cpp
// Animal 클래스
class Animal {
public:
    void eat() {
        cout << "Animal is eating\n";
    }
};

// Dog 클래스가 Animal 클래스를 상속
class Dog : public Animal {
public:
    void bark() {
        cout << "Dog is barking\n";
    }
};
```

markdown 코드 복사

```
    Animal
      |
      |
    /   \
  Dog
```

캡슐화 (Encapsulation)

· 캡슐화는 객체의 상태(데이터)와 행위(메서드)를 하나로 묶고, 외부에서 접근을 제어하는 것
(데이터와 속성이 하나의 객체로 묶는다고 생각하는게 편함)
· 객체 내부의 데이터에 직접 접근하지 않고, 메서드를 통해 간접적으로 접근
(객체는 인터페이스를 통해서만 상호작용)

cpp 코드 복사

```cpp
class Car {
private: // 외부에서 직접 접근 불가
    int speed;

public:
    void setSpeed(int s) {
        speed = s;
    }

    int getSpeed() {
        return speed;
    }
};
```

sql 코드 복사

```sql
------------------
|      Car       |
------------------
| - speed: int   |
------------------
| + setSpeed(s)  |
| + getSpeed()   |
------------------
```

다형성 (Polymorphism)

- 다형성은 같은 이름의 메서드가 서로 다른 클래스에서 다르게 동작하는 것을 의미
- 오버로딩(Overloading)과 오버라이딩(Overriding)이 다형성의 주요 형태

cpp 코드 복사

```cpp
class Animal {
public:
    void makeSound() {
        cout << "Animal makes a sound\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() { // 오버라이딩
        cout << "Dog barks\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() { // 오버라이딩
        cout << "Cat meows\n";
    }
};
```

lua 코드 복사

```lua
      Animal
        |
        |
    -----------
    |   Dog   |
    -----------
    |         |
    | makeSound() |
    ---------------

    -----------
    |   Cat   |
    -----------
    |         |
    | makeSound() |
    ---------------
```

5. 가상 함수 (Virtual Functions)

- 가상 함수는 **기본 클래스에서 선언**되어 **파생 클래스에서 재정의**할 수 있는 함수입니다.
- 런타임에 객체의 실제 타입에 따라 **동적으로 호출**됩니다.
- C++에서는 **virtual 키워드**를 사용하여 선언합니다.

cpp 코드 복사

```cpp
class Animal {
public:
    virtual void makeSound() {
        cout << "Animal makes a sound\n";
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Dog barks\n";
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Cat meows\n";
    }
};
```

6. 가상 함수 테이블 (Virtual Function Table, VTable)

- 가상 함수 테이블은 클래스의 **가상 함수들에 대한 포인터들을 담고 있는 테이블**입니다.
- 객체의 인스턴스가 어떤 클래스의 인스턴스인지 식별하고, 해당 클래스의 가상 함수들을 실행할 때 사용

동작원리

- 각 클래스는 자신의 가상 함수들에 대한 테이블을 가집니다. 이 **테이블은 컴파일 시 생성**됩니다.
- 객체가 생성될 때, 해당 객체의 가상 함수 테이블이 가리키는 것은 그 객체의 실제 타입에 따라 달라집니다.
- 가상 함수 호출 시 객체의 VTable을 통해 실제 호출할 함수를 결정합니다.

7. 런타임 타입 식별 (Runtime Type Identification, RTTI)

- RTTI는 실행 중에 객체의 실제 타입을 식별하는 메커니즘입니다.
- dynamic_cast나 typeid 연산자를 사용하여 객체의 타입을 확인할 수 있습니다.

```cpp
Animal* ptr = new Dog();
if (dynamic_cast<Dog*>(ptr)) {
    cout << "ptr is pointing to a Dog\n";
} else if (dynamic_cast<Cat*>(ptr)) {
    cout << "ptr is pointing to a Cat\n";
}

// 또는 typeid를 사용할 수 있음
if (typeid(*ptr) == typeid(Dog)) {
    cout << "ptr is pointing to a Dog\n";
} else if (typeid(*ptr) == typeid(Cat)) {
    cout << "ptr is pointing to a Cat\n";
}
```

쉬운예시

```cpp
Animal* ptr = new Dog(); // Dog 객체를 Animal 포인터로 가리킴

// 가상 함수 호출
ptr->makeSound(); // Dog 클래스의 makeSound()가 호출됨

// 런타임 타입 식별
if (typeid(*ptr) == typeid(Dog)) {
    cout << "ptr is pointing to a Dog\n";
} else if (typeid(*ptr) == typeid(Cat)) {
    cout << "ptr is pointing to a Cat\n";
}
```

여기서 ptr이 실제로 Dog를 가리키고 있으므로, typeid(*ptr)은 Dog를 반환할 것
typeid연산자 외 dynamic_cast 연산자는 객체 포인터나 참조를 안전하게 다운캐스팅

8. 동적 할당 (Dynamic Allocation)

· 메모리 사용 효율성: 필요한 만큼의 메모리만 할당하므로 메모리를 효율적으로 사용할 수 있음
· 프로그램 유연성: 실행 중에 동적으로 메모리를 조작할 수 있어서 프로그램의 유연성을 높임

cpp 코드 복사

```cpp
int* ptr = new int; // int형 변수를 동적으로 할당

*ptr = 10; // 동적으로 할당된 메모리에 값 저장

cout << *ptr; // 출력: 10

delete ptr; // 동적으로 할당된 메모리 해제
```

new int는 힙(heap)영역 메모리에 정수형 변수를 동적으로 할당.
delete ptr는 할당된 메모리를 해제 (해제하지 않으면 메모리 누수 발생 : 프로그램의 성능이 저하)

9. 스타일 캐스팅 4총사

- static_cast
  - 상식적인 변환만 허용
  - 컴파일 시간에 캐스팅
- dynamic_cast
  - 상속구조에서 다형성(vftable)이 있어야 사용가능한 cast
  - 캐스팅이 실패하면 nullptr을 반환한다.
  - 런타임에 캐스팅한다.
- const_cast
  - 상수객체를 비상수객체로 캐스팅할 수 있다.
- reinterpert_cast (서버와 클라이언트에서 주로 활용)
  - 모든 캐스팅 통과
  - 위험한 캐스팅

- static_cast
  - 기본적인 타입 변환을 수행
  - 주로 기본 자료형 간의 변환, 포인터 간의 변환 등에 사용
  - 컴파일 타임에 타입 체크를 수행하므로 상대적으로 안전

주로 사용되는 경우:

- 기본 자료형 간의 변환 (예: int를 double로 변환)

```
int i = 42;
double d = static_cast<double>(i);
```

- 관련 있는 포인터 타입 간의 변환 (예: 부모클래스 포인터를 자식클래스 포인터로 변환)

```
class Base {};
class Derived : public Base {};
Base* base = new Derived();
Derived* derived = static_cast<Derived*>(base);
```

- 열거형(enum) 타입과 정수형 간의 변환

· dynamic_cast

- 주로 상속 관계에 있는 클래스 사이에서 안전한 다운캐스팅을 수행
- 런타임에 타입 체크를 수행하므로, 실패할 경우 nullptr 또는 예외를 반환
- RTTI (Runtime Type Information)가 필요하므로, 다형성(Polymorphism)이 있는 클래스에서 사용

주로 사용되는 경우:

- 다형성을 사용하는 클래스 계층에서 안전한 다운캐스팅

```
class Base { virtual void foo() {} };
class Derived : public Base {};
Base* base = new Derived();
Derived* derived = dynamic_cast<Derived*>(base);
if (derived) {
    // 다운캐스팅 성공
} else {
    // 다운캐스팅 실패
}
```

- 런타임에 타입 체크가 필요한 경우

· const_cast

- const 또는 volatile 속성을 제거하거나 추가할 때 사용
- 주의해서 사용해야 하며, 잘못 사용하면 정의되지 않은 동작이 발생할 수 있음

주로 사용되는 경우:

- const 속성을 제거해야 할 때 (주의해서 사용해야 함)

```
void func(int* p) {
    // p를 사용
}
const int* p = &i;
func(const_cast<int*>(p));
```

- const 객체의 비 const 메서드 호출 필요 시

```
class Example {
public:
    void nonConstMethod() {}
};
const Example e;
const_cast<Example&>(e).nonConstMethod();
```

· reinterpert_cast

– 거의 모든 타입 간의 변환을 허용

– 포인터 형식이나 정수형 간의 변환에 주로 사용되며, 매우 위험할 수 있음

– 타입에 대한 안전성이 보장되지 않으므로, 꼭 필요할 때만 사용함

주로 사용되는 경우:

– 포인터 타입 간의 변환 (위험할 수 있으므로 주의 필요)

```
int* p = new int(42);
char* c = reinterpret_cast<char*>(p);
```

– 비트 레벨 변환을 해야 하는 경우

```
int i = 65;
char* p = reinterpret_cast<char*>(&i);
// 이제 p는 'A'를 가리킵니다 (65의 아스키 값)
```

– 서버와 클라이언트에서 사용하는 경우

```
enum PacketID
{       NONE = 0,
        LOG_IN,
        LOG_SUCCESS,
        CREATE_ACCOUNT,
        PLAYER_RUN              };
struct Protocol_test_S // 내가 서버한테 보내는 패키지
{       PacketID pkt_Id = PacketID::LOG_IN;
        int id;
        int password;           };
struct Protocol_test_C
{       PacketID pkt_Id = PacketID::LOG_SUCCESS;
        int success;            };

// 클라이언트 로그인 시도
Protocol_test_S pkt;
pkt.id = PacketID::LOG_IN;
pkt.id = 12345;
pkt.password = 12345;

// Send(pkt); 서버에 전송할때
// 서버에서 로그인 체크 후 성공여부 전달
// 12바이트????
Protocol_test_S* s_pkt = reinterpret_cast<Protocol_test_S*>(&pkt);
// 클라이언트에서 로그인 성공여부
```

사용 빈도 및 주의 사항

static_cast는 비교적 자주 사용되며, 안전한 타입 변환을 위한 기본 도구입니다.
dynamic_cast는 다형성을 사용하는 클래스에서 안전한 다운캐스팅을 위해 주로 사용됩니다.
const_cast는 사용 빈도가 낮으며, 주로 const 속성을 제거할 필요가 있을 때 사용합니다. 남용하면 프로그램의 안전성을 해칠 수 있으므로 주의해야 합니다.
reinterpret_cast는 매우 드물게 사용되며, 주로 시스템 프로그래밍이나 특수한 상황에서 사용됩니다. 잘못 사용하면 프로그램이 불안정해질 수 있습니다.
따라서 각 캐스팅 연산자는 특정한 상황에서 사용되어야 하며, 특히 const_cast와 reinterpret_cast는 신중하게 사용해야 합니다.