

객체지향언어(11)

4주차 연습문제 과제

제출일: 2023.10.09

전공: 게임공학과

학번: 2019184020

성명: 윤은지

목차

1. 객체를 캡슐화하는 목적
2. 클래스와 객체에 관한 설명
3. Circle 클래스의 틀린 부분 수정
4. Tower 클래스의 틀린 부분 수정
5. 코드의 틀린 부분 수정
6. Year를 10으로 초기화하는 생성자와 year 값을 리 턴하는 getYear()를 구현
7. 생성자의 설명 중 틀린 것
8. 소멸자에 대한 설명 중 틀린 부분
9. 프로그램에 대한 물음
10. 객체가 생성되고 소멸되는 순서
11. 프로그램의 오류 지적
12. 프로그램의 오류 지적
13. 코드에서 자동 인라인 함수 찾기
14. 인라인 함수의 장단점
15. 인라인 함수에 대한 잘못된 설명
16. 컴파일러가 인라인으로 처리하기에 바람직한 것
17. 구조체에 대해 잘못 설명한 것
18. 예시 구조체와 동일한 의미를 갖는 클래스
19. 예시 클래스를 구조체로 선언
20. **객체를 캡슐화하는 목적은 무엇인가?**
21. 데이터 은닉 : 객체를 캡슐화하면 객체 내부의 데이터를 외부에서 직접 접근하거나 수정하는 것을 제한할 수 있습니다. 이렇게 하면 객체의 내부 상태를 보호하고 무결성을 유지할 수 있습니다. 오직 객체의 멤버 함수를 통해서만 데이터에 접근할 수 있도록 제한하므로, 예기치 않은 데이터 변경을 방지할 수 있습니다.
22. 추상화 : 객체를 캡슐화하면 객체가 특정 기능을 제공하고, 사용자에게 인터페이스를 제공하는 추상화된 개념이 됩니다. 이로써 사용자는 객체 내부의 복잡한 구현 세부 사항을 알 필요 없이 객체를 사용할 수 있습니다. 이는 프로그램의 모듈화와 유지보수를 쉽게 만들어 줍니다.
23. 코드 재사용 : 객체 지향 프로그래밍에서 객체를 캡슐화하면 해당 객체를 다른 부분에서도 재사용할 수 있습니다. 객체의 인터페이스를 정의하고 구현 세부 사항을 숨기므로, 같은 객체를 여러 프로그램이나 모듈에서 사용할 수 있습니다.
24. 보안 향상 : 객체를 캡슐화하면 민감한 정보나 기능을 숨길 수 있으므로 보안 향상에 도움이 됩니다. 다른 부분에서 객체 내부 정보에 직접 접근할 수 없도록 하여 데이터 보호를 강화할 수 있습니다.
25. 코드 유지보수 및 디버깅 용이성 : 객체를 캡슐화하면 객체의 내부 구현을 변경할 때 외부 코드에 영향을 덜 주며, 코드를 보다 쉽게 유지보수하고 디버그할 수 있게 됩니다.
26. **클래스와 객체에 관한 설명 중 틀린 것은?**

3 : 클래스의 멤버들은 private보다 public 접근 지정이 바람직하다. (틀린 설명)

클래스의 멤버들의 접근 지정은 프로그래머가 설계할 때에 따라 다를 수 있으며, 모든 멤버가 public으로 선언되는 것이 바람직하지는 않습니다. 오히려 캡슐화와 정보 은닉을 위해 멤버 변수와 일부 멤버 함수는 private 접근 지정을 사용하여 외부에서 직접 접근하지 못하도록 하는 것이 일반적입니다. 이렇게 하면 객체의 내부 상태를 보호하고 무결성을 유지할 수 있습니다.

1. **다음 코드는 Circle 클래스의 선언부이다. 틀린 부분을 수정하라.**
2. class Circle {
3. public:
4. int radius; // 반지름을 나타내는 멤버 변수
5. double getArea(); // 면적을 계산하여 반환하는 멤버 함수
6. };

위와 같이 ‘public:’ 접근 지정자를 사용하여 멤버 변수와 멤버 함수를 외부에서 접근 가능하도록 지정하고, ‘radius’ 변수와 ‘getArea()’ 함수의 데이터 형식을 명시하여 코드를 수정하였습니다.

**4. 다음 코드는 Tower 클래스를 작성한 사례이다. 틀린 부분을 수정하라.**

주어진 코드에서 ‘Tower’ 클래스의 생성자(‘Tower()’) 내에서 ‘height’ 변수를 초기화하고 있습니다. 그러나 이러한 초기화는 클래스 멤버 변수의 초기화 리스트를 사용하여 해야 합니다. 또한 생성자에서 명시적으로 ‘return;’을 사용할 필요가 없습니다.

class Tower {

int height = 20; // 멤버 변수의 초기값 설정

public:

Tower() : height(10) {} // 생성자에서 초기화 리스트를 사용하여 height를 초기화

};

위 코드에서 ‘height’ 변수는 멤버 변수 초기값으로 ‘20’으로 설정되며, 생성자에서 초기화 리스트를 사용하여 ‘height’ 변수를 ‘10’으로 초기화합니다. 이렇게 하면 올바르게 초기화가 이루어집니다. ‘return;’ 문은 생성자 내에서 필요 없습니다.

**5. 다음 코드에서 틀린 부분을 수정하라.**

주어진 코드에서 문제는 다음과 같습니다.

1. ‘Building’ 클래스의 생성자는 ‘floor’ 멤버 변수를 초기화하지만, 객체를 생성할 때 인수를 제공하지 않으면 ‘floor’ 변수는 초기화되지 않습니다. 그러므로 기본 생성자를 추가해야합니다.

수정된 코드는 다음과 같습니다.

class Building {

private:

int floor;

public:

// 기본 생성자 추가

Building() : floor(0) {}

Building(int s) {

floor = s;

}

};

int main()

{

Building twin, star;

Building BlueHouse(5), JangMi(14);

}

위 코드에서 기본 생성자(‘Building()’)를 추가하여 객체를 생성할 때 인수를 제공하지 않은 경우 ‘floor’ 변수를 0으로 초기화했습니다. 이렇게 하면 객체를 다양한 방법으로 생성할 수 있게 됩니다.

**6. 다음 코드는 Calendar 클래스의 선언부이다. Year를 10으로 초기화하는 생성자와 year 값을 리턴하는 getYear()를 구현하라.**

‘Calendar’ 클래스의 생성자와 ‘getYear()’ 함수를 구현하려면 다음과 같이 코드를 작성할 수 있습니다.

class Calendar {

private:

int year;

public:

// 생성자를 정의하여 year를 10으로 초기화

Calendar() : year(10) {}

// getYear() 함수를 정의하여 year 값을 반환

int getYear() {

return year;

}

};

위 코드에서 ‘Calendar’ 클래스의 생성자에서 ‘year’ 멤버 변수를 10으로 초기화하고, ‘getYear()’ 함수에서 ‘year’ 값을 반환하도록 정의했습니다. 이제 ‘Calendar’ 객체를 생성하면 ‘year’ 변수가 10으로 초기화되고 ‘getYear()’ 함수를 호출하여 해당 값을 가져올 수 있습니다.

**7. 생성자에 대한 설명 중 틀린 것은?**

2 : 생성자는 오직 하나만 작성 가능하다.

생성자는 클래스 내에 여러 개를 정의할 수 있습니다. 이러한 생성자 중에는 서로 다른 매개변수를 가지고 있어야 하며, 이를 함수 오버로딩이라고 합니다. 객체를 다양한 방법으로 초기화하고 생성할 수 있도록 여러 생성자를 정의할 수 있습니다.

1. **소멸자에 대한 설명 중에 틀린 부분을 지적하라.**

소멸자에 대한 설명 중에 틀린 부분은 다음과 같습니다.

3 : ‘매개 변수 있는 소멸자를 작성하여’ 소멸시에 의미 있는 값을 전달할 수 있으며

소멸자는 일반적으로 매개 변수를 가지지 않습니다. 소멸자는 객체가 소멸될 때 자원을 해제하거나 정리하는 데 사용되며, 주로 객체의 내부 상태를 처리하는 데에만 관여합니다. 따라서 소멸자는 보통 매개 변수를 가지지 않으며, 객체가 소멸할 때 필요한 모든 정보는 이미 객체 내부에 저장되어 있어야 합니다.

1. **다음 프로그램에 대해 답하여라.**

‘House’ 클래스의 생성자를 구현하여 ‘n’과 ‘s’로 numOfRooms’와 ‘size’를 초기화하고, 이들 값을 출력하려면 다음과 같이 코드를 작성할 수 있습니다.

#include <iostream>

class House {

int numOfRooms;

int size;

public:

House(int n, int s) : numOfRooms(n), size(s) {

std::cout << "House 생성자 호출 - numOfRooms: " << numOfRooms << ", size: " << size << std::endl;

}

};

int main() {

House a(2, 20);

}

‘House’ 클래스의 소멸자를 작성하여 ‘size’와 ‘numOfRooms’ 값을 출력하려면 다음과 같이 코드를 작성할 수 있습니다.

#include <iostream>

class House {

int numOfRooms;

int size;

public:

House(int n, int s) : numOfRooms(n), size(s) {

std::cout << "House 생성자 호출 - numOfRooms: " << numOfRooms << ", size: " << size << std::endl;

}

~House() {

std::cout << "House 소멸자 호출 - numOfRooms: " << numOfRooms << ", size: " << size << std::endl;

}

};

int main() {

House a(2, 20);

}

객체 ‘a’는 ‘main()’ 함수 내에서 함수 ‘f()’를 호출할 때 생성됩니다. 따라서 객체 생성 순서는 다음과 같습니다.

1. ‘b’ (전역 범위에서 생성)
2. ‘c’ (전역 범위에서 생성)
3. ‘a’ (함수 ‘f()’에서 생성)
4. ‘d’ (main 함수 내에서 생성) 객체 소멸 순서는 객체가 생성된 역순으로 이루어집니다. 즉, 소멸 순서는 다음과 같습니다.
5. ‘d’ (main 함순 내에서 소멸)
6. ‘a’ (함수 ‘f()’에서 소멸)
7. ‘c’ (전역 범위에서 소멸)
8. ‘b’ (전역 범위에서 소멸)

따라서 객체 생성 및 소멸 순서는 ‘b’, ‘c’, ‘a’, ‘d’ 순서로 생성되고, ‘d’, ‘a’, ‘c’, ‘b’ 순서로 소멸됩니다.

1. **다음 프로그램에서 객체 a, b, c가 생성되고 소멸되는 순서는 무엇인가?**
2. ‘c’ (전역 범위에서 생성)
3. ‘b’ (함수 ‘f()’ 내에서 생성)
4. ‘a’ (멤버 함수 ‘test()’ 내에서 생성)
5. ‘a’ (멤버 함수 ‘test()’ 내에서 소멸)
6. ‘b’ (함수 ‘f()’ 내에서 소멸)
7. ‘c’ (전역 범위에서 소멸)

따라서 객체 생성 및 소멸 순서는 ‘c’, ‘b’, ‘a’ 순서로 생성되고, ‘a’, ‘b’, ‘c’ 순서로 소멸됩니다.

1. **다음 프로그램의 오류를 지적하고 수정하라.**

주어진 프로그램에는 다음과 같은 오류가 있습니다.

1. ‘TV’ 클래스의 생성자가 private로 선언되어 있으므로, 객체를 생성할 때 접근할 수 없습니다. 생성자가 private일 경우, 클래스 외부에서 객체를 생성할 수 없습니다.
2. ‘TV LG;’와 같이 기본 생성자를 호출하려고 하면 private생성자에 접근할 수 없기 때문에 컴파일 오류가 발생합니다.

오류를 수정하기 위해 ‘TV’ 클래스의 생성자를 public으로 변경하고, 기본 생성자와 매개 변수를 받는 생성자를 모두 정의해야 합니다. 아래는 수정된 코드입니다.

#include <iostream>

class TV {

public:

int channels;

// 기본 생성자

TV() {

channels = 256;

}

// 매개 변수를 받는 생성자

TV(int a) {

channels = a;

}

};

int main() {

TV LG;

LG.channels = 200;

TV Samsung(100);

}

위 코드에서 ‘TV’ 클래스의 생성자를 public으로 변경하고, 기본 생성자와 매개 변수를 받는 생성자를 모두 정의하여 객체를 생성하고 초기화할 수 있도록 수정되었습니다.

1. **다음 프로그램의 오류를 지적하고 수정하라.**
2. ‘channels’ 멤버 변수는 ‘private’로 선언되어 있으므로 클래스 외부에서 직접 접근할 수 없습니다. 따라서 멤버 변수에 접근하기 위한 ‘public’ getter와 setter 함수를 추가하는 것이 좋습니다.
3. 생성자에서 ‘channels’와 ‘colors’를 초기화할 수 있지만, 초기화 리스트를 사용하여 더 효율적으로 초기화할 수 있습니다.

아래는 개선된 코드 예제입니다.

#include <iostream>

class TV {

private:

int channels;

int colors;

public:

TV() : channels(256), colors(0) {}

TV(int a, int b) : channels(a), colors(b) {}

// Getter 함수

int getChannels() const {

return channels;

}

int getColors() const {

return colors;

}

// Setter 함수

void setChannels(int a) {

channels = a;

}

void setColors(int b) {

colors = b;

}

};

int main() {

TV LG;

LG.setChannels(200);

LG.setColors(60000);

TV Samsung(100, 50000);

std::cout << "LG: Channels=" << LG.getChannels() << ", Colors=" << LG.getColors() << std::endl;

std::cout << "Samsung: Channels=" << Samsung.getChannels() << ", Colors=" << Samsung.getColors() << std::endl;

}

이 코드에서는 ‘public’ getter와 setter 함수를 사용하여 ‘channels’ 와 ‘colors’ 에 접근하고 수정할 수 있습니다.

1. **다음 코드에서 자동 인라인 함수를 찾아라.**

주어진 코드에서 ‘TV’ 클래스의 멤버 함수 중에서 자동으로 인라인으로 처리되는 함수는 ‘getChannels()’ 함수입니다.

inline int TV::getChannels() { return channels; }

‘inline’ 키워드가 함수 정의 앞에 사용되어 있으므로 이 함수는 자동으로 인라인 함수로 처리됩니다. 인라인 함수는 함수 호출 오버헤드를 줄이고 실행 속도를 향상시키는 데 도움이 되며, 간단한 작업을 수행하는 함수에 자주 사용됩니다.

1. **인라인 함수의 장단점을 설명한 것 중 옳은 것은?**

옳은 설명은 다음과 같습니다.

1 : 인라인 함수를 사용하면 컴파일 속도가 향상된다.

인라인 함수는 함수 호출 오버헤드를 줄여 프로그램의 실행 속도를 향상시키는 데 도움이 됩니다.

1. **인라인 함수에 대해 잘못 설명한 것은?**

1 : 인라인 선언은 크기가 큰 함수의 경우 효과적이다.(틀린 설명)

틀린 이유는 다음과 같습니다.

1. 인라인 함수는 함수 호출을 줄이기 위해 사용됩니다. 함수 호출은 호출 스택에 매개 변수와 지역 변수를 저장하고 복구하는 과정 등 오버헤드를 발생시킵니다. 작은 함수를 인라인으로 선언하면 이러한 함수 호출 오버헤드를 줄일 수 있습니다.
2. 그러나 크기가 큰 함수의 경우 함수의 코드가 길어지며, 이를 인라인화 하면 컴파일된 코드가 더욱 커질 수 있습니다. 이로 인해 컴파일 시간이 길어질 수 있고, 컴파일된 코드의 크기가 늘어나면 메모리 사용량도 증가할 수 있습니다.
3. 또한, 크기가 큰 함수를 인라인화하면 코드의 가독성과 유지보수성이 저하될 수 있습니다. 코드가 길어지면 이를 이해하고 수정하는 것이 더 어려울 수 있습니다.
4. **Inline 선언은 강제 사항이 아니다. 다음 함수 중에서 컴파일러가 인라인으로 처리하기에 가장 바람직한 것은?**
5. 이 함수는 매우 간단하며, 단순한 비교 연산을 수행합니다. 이러한 간단한 함수는 인라인으로 처리하기에 적합합니다.
6. 이 함수도 상대적으로 간단하며, 반복문을 사용하여 합계를 계산합니다. 인라인으로 처리해도 코드 크기가 크게 늘어나지 않을 것입니다.
7. 이 함수는 정적 변수를 사용하여 값을 누적하고 반환합니다. 이 함수는 인라인으로 처리하기에도 적합할 수 있으며, 호출 빈도에 따라서 처리 여부를 결정할 수 있습니다.
8. 이 함수는 재귀 호출을 사용하여 합계를 계산하므로 인라인화하기에는 다소 복잡할 수 있습니다. 재귀 함수의 경우 컴파일러가 인라인 처리를 선택하기 어려울 수 있습니다.

따라서 컴파일러가 인라인으로 처리하기에 가장 바람직한 함수는 ‘big’ 또는 ‘add(int a)’일 수 있습니다.

1. **C++ 구조체(struct)에 대해 잘못 설명한 것은?**

1: C++에서 구조체를 둔 이유는 C언어돠의 호환성 때문이다. (틀린 설명)

C++에서 구조체는 C언어와 호환성을 유지하기 위한 목적이 아닙니다. C++에서 구조체는 클래스와 유사한 멤버 함수를 가질 수 있으며, 클래스와 거의 동일한 기능을 제공합니다. 따라서 C++에서 구조체는 C언어와는 다른 목적과 특징을 가지고 있습니다.

1. **다음 C++ 구조체를 동일한 의미를 가지는 클래스로 작성하라.**

주어진 구조체를 동일한 의미를 가지는 클래스로 작성하면 다음과 같습니다.

class Family {

private:

char tel[11];

public:

int count;

char address[20];

Family() {

// 생성자에서 초기화

count = 0;

address[0] = '\0'; // 빈 문자열로 초기화

tel[0] = '\0'; // 빈 문자열로 초기화

}

};

위 코드에서 ‘struct’의 멤버 변수들과 동일한 멤버 변수들을 ‘private’ 섹션에 정의하고, 생성자에서 초기화 작업을 수행했습니다. 따라서 클래스 ‘Family’는 동일한 의미를 가지는 구조체와 유사한 동작을 합니다.

1. **다음 클래스를 구조체로 선언하라.**

주어진 클래스를 구조체로 선언하면 다음과 같이 정의할 수 있습니다.

struct Universe {

private:

char dateCreated[10];

char creator[10];

int size;

// 구조체 내에서도 private 멤버 사용 가능

public:

Universe() {

// 생성자에서 초기화

creator[0] = '\0'; // 빈 문자열로 초기화

size = 0;

dateCreated[0] = '\0'; // 빈 문자열로 초기화

}

};

구조체와 클래스의 주요 차이점 중 하나는 접근 제어 지시어인데, 구조체 내에서도 private 멤버를 선언하고 사용할 수 있습니다. 따라서 주어진 클래스를 구조체로 변환하면서 private 멤버를 그대로 유지할 수 있습니다.