객체지향언어(11)

9주차 연습문제 과제

제출일: 2023.11.04

전공: 게임공학과

학번: 2019184020

성명: 윤은지

목차

1. C++의 함수 인자 전달 방식이 아닌 것은?

4)의 “묵시에 의한 호출”이라는 용어는 일반적인 C++ 함수 호출 방식 중 하나가 아닙니다.

1)값에 의한 호출: 함수에 인자로 값이 복사되어 전달됩니다.

2))주소에 의한 호출: 함수에 인자로 변수의 주소가 전달되어 해당 주소를 통해 변수에 접근할 수 있습니다.

3)참조에 의한 호출: 함수에 인자로 변수의 참조가 전달되어 참조를 통해 변수에 접근할 수 있습니다.

2. 일반적으로 함수 호출 시 가장 비용(cost) 부담이 큰 것은?

일반적으로 함수 호출 시 가장 비용 부담이 큰 것은 “값에 의한 호출” 1)입니다.

1) 값에 의한 호출은 함수에 인자로 변수의 복사본이 전달되므로 메모리 복사 작업이 필요하고, 이로 인해 메모리 사용과 시간이 더 소비됩니다. 따라서 인자가 큰 경우나 객체의 복사 생성 비용이 높은 경우에는 값에 의한 호출 비용 부담이 크다고 볼 수 있습니다.
2) 주소에 의한 호출은 주소나 참조를 전달하므로 실제 데이터를 복사하지 않습니다. 따라서 값에 의한 호출에 비해 비용이 낮습니다.
3) 묵시에 의한 호출은 C++에서 일반적으로 사용되지 않는 용어이며, 함수 호출 방식과 관련이 없으므로 비용 부담과 관련이 없습니다.

3. 다음에서 f() 함수가 호출될 때 사용되는 인자 전달 방식은 무엇인가?

```
void f(int n[]);

int main()
{
int m[3] = { 1,2,3 };
f(m);
}
```

주어진 코드에서 ‘f()’ 함수가 호출될 때 사용되는 인자 전달 방식은 “주소에 의한 호출” 또는 “배열의 이름에 의한 주소 전달” 방식입니다.

C++에서 배열을 함수에 전달할 때 배열 이름은 배열의 첫 번째 요소의 주소로 해석됩니다.

따라서 'f(m)'에서 'm'는 배열 'm'의 첫 번째 요소인 '1'의 주소로 전달됩니다. 이것은 주소에 의한 호출이며, 함수 'f()'는 이 주소를 통해 배열 'm'의 요소에 접근할 수 있습니다.

4. 다음 두 함수 선언은 같은 것인가?

1) `void f(int p[]);` 와 `void f(int* p);`
2) `void f(int* p);` 와 `void f(int& p);`

1) void f(int p[]);와 void f(int* p);는 서로 같은 함수 선언입니다. 배열 p의 이름은 배열의 첫 번째 요소를 가리키는 포인터로 해석되므로, 두 함수는 모두 정수 포인터 p를 인자로 받는 함수를 선언합니다.

2) void f(int* p);와 void f(int& p);는 서로 다른 함수 선언입니다. 첫 번째 함수는 정수 포인터 p를 인자로 받는 반면, 두 번째 함수는 정수 참조 p를 인자로 받습니다. 참조에 의한 호출은 포인터와 달리 참조된 변수 자체에 접근할 수 있고, 변경할 수 있습니다. 따라서 이러한 두 함수는 다른 동작을 합니다.

5. 다음 프로그램의 실행 결과는 무엇인가?

1)

```
#include<iostream>
using namespace std;

void square(int n) {
        n = n * n;
}

int main()
{
        int m = 5;
        square(m);
        cout << m;
}
```

2)

```
#include<iostream>
using namespace std;

void square(int& n) {
        n = n * n;
}
```

```
int main()
{
        int m = 5;
        square(m);
        cout << m;
}
```

1) 의 실행 결과는 5입니다. 이는 square 함수가 값을 전달하는 방식으로 호출되었기 때문입니다. 함수 내에서 n을 수정해도 호출자의 m에는 영향을 주지 않습니다.

2) 의 실행 결과는 25입니다. 이번에는 square 함수가 참조에 의한 호출을 사용하여 호출되었기 때문입니다. 함수 내에서 n을 수정하면 호출자의 m도 동일한 값을 갖게 됩니다. 즉, square 함수에서 n을 제곱하면 호출자의 m도 제곱된 값으로 업데이트됩니다.

6. 다음 프로그램의 실행 결과는 무엇인가?

```
#include<iostream>
#include<string>
using namespace std;

void square(int n[], int size) {
        for (int i = 0; i < size; i++)n[i] = n[i] * n[i];
}

int main()
{
        int m[3] = { 1,2,3 };
        square(m, 3);
        for (int i = 0; i < 3; i++)cout << m[i] << ' ';
}
```

주어진 프로그램은 배열 m을 함수 square로 전달하여 각 요소의 제곱을 계산한 후, 배열의 내용을 출력하는 프로그램입니다.

프로그램의 실행 결과는 다음과 같을 것입니다.

1 4 9

m배열의 초기 값은 {1,2,3}이며, square함수를 통해 각 요소가 제곱되어 {1,4,9}로 변경됩니다. 그런 다음 반복문을 통해 변경된 배열의 내용이 출력되며, 결과로써 각 요소의 제곱이 출력됩니다.

7. char 형 변수 c가 선언되어 있을 때, 참조 변수 r의 선언 중 틀린 것은?

```
//1)
char & r = c;

//2)
char r & = c;

//3)
char& r = c;

//4)
char &r = c;
```

선언 중에서 올바른 것은 1)과 3)입니다. 1)과 3)은 올바른 참조 변수 선언입니다.

3) 와 4)는 올바르지 않습니다. 올바른 참조 변수 선언 방식은 변수 이름 뒤에 &을 붙이고, 변수 형식 앞에 &을 붙이는 것이 맞습니다.

따라서 2)와 4)는 올바른 참조 변수 선언 방식이 아니며, 틀린 것입니다.

8. 변수 c에 'a' 문자를 기록하지 못하는 것은?

```
char c;
char* p = &c;
char q = c;
char& r = c;

//1) c='a';
//2) q='a';
//3) r='a';
//4) *p='a';
```

변수 c에 'a'문자를 기록하지 못하는 것은 다음 중 2)와 3)입니다.

1) q='a' ;는 q에 'a'문자를 저장하는 것이며, q는 c의 복사본이기 때문에 c에는 영향을 주지 않습니다.

2) r='a' ;는 참조 변수 r을 통해 c에 접근하려는 시도이지만, 참조 변수는 초기화될 때만 다른 변수를 참조할 수 있고, 이후에는 참조 대상을 변경할 수 없습니다. 즉, r은 c의 별칭이 되어 'a'문자를 직접 c에 대입할 수 없습니다.

3) c='a' ;는 변수 c에 'a'문자를 기록하는 것입니다. c자체에 값을 할당하는 것이므로 가능합니다.

4) *p='a' ;는 포인터 p를 통해 변수 c에 접근하고 'a'문자를 저장하는 것이므로 가능합니다.

9. 다음 중 컴파일 오류가 발생하는 문장은?

```
int n = 10;
int &refn;//1)
refn = n;//2)
refn++;//3)
int & m = refn;//4)
```

컴파일 오류가 발생하는 문장은 다음과 같습니다.

1) int &refn; 은 참조 변수 refn을 선언하는 것이지만 초기화되지 않았기 때문에 컴파일 오류가 발생합니다. 참조 변수는 반드시 선언과 동시에 초기화되어야 합니다.

2) refn = n;은 refn을 이미 선언하지 않았기 때문에 컴파일 오류가 발생합니다.

3) refn++ 역시 refn이 선언되지 않았으므로 컴파일 오류가 발생합니다.

4) int & m = refn;은 refn이 선언되지 않았기 때문에 컴파일 오류가 발생합니다.

따라서 1,2,3,4 모두 컴파일 오류가 발생합니다.

10. 다음의 각 문제가 별도로 실행될 때 array 배열은 어떻게 되는가?

```
int array[] = { 0,2,4,6,8,10,12,14,16,18 };
int& f(int n) {
        return array[n];
}

//1) f(9) = 100;
//2) for(int i=1;i<9;i++) f(i)=f(i)+2;
//3) int v = f(0); v=100;
//4) f(f(2)) = 0;
```

1) f(9)=100;을 실행하면 array[9]값이 100으로 변경됩니다. array 배열은 다음과 같이 변경됩니다: {0,2,4,6,8,10,12,14,16,100}.

2) for(int i=1;i<9;i++) f(i)=f(i)+2;를 실행하면 배열 array의 요소 1부터 8까지를 2씩 증가시킵니다. array 배열은 다음과 같이 변경됩니다: {0,4,6,8,10,12,14,16,100,100}.

3) int v = f(0); v=100;을 실행하더라도 array 배열은 변경되지 않으며, v에는 array[0]의

값을 복사한 후에 v가 100으로 설정됩니다.

4) f(f(2)) =0;을 실행하면 f(2)는 array[2]의 참조를 반환하고, 이후 f(array[2])는 array[2]의 참조를 반환합니다. 그러나 이렇게 반환된 참조에 값을 대입하는 것은 array[2]의 값을 변경시키지 않습니다. 따라서 array배열은 변경되지 않으며 여전히 {0,4,6,8,10,12,14,16,100,100}로 유지됩니다.

11. 다음 copy() 함수는 src 값을 dest에 복사하는 함수이다.

```
void copy(int dest, int src) {
dest = src;
}

//copy()를 이용하여 b 값을 a에 복사하고자 하지만, b값이 a에 복사되지 않는다.
int a = 4, b = 5;
copy(a, b);//b값을 a에 복사
```

복사되지 않는 이유가 무엇인지 설명하고, 복사가 잘 되도록 copy() 함수만 고쳐라.

copy()함수의 현재 구현에서는 dest와 src 매개변수가 값에 의한 호출로 전달되므로 copy()함수 내에서 dest를 변경해도 원래의 변수 a에는 영향을 미치지 않습니다. 값에 의한 호출은 함수 내에서 인사의 복사본을 다루기 때문에 dest와 src는 a와 b와 무관한 별개의 변수가 됩니다.

복사가 원활하게 이루어지려면 copy()함수에 참조에 의한 호출을 사용해야 합니다.

아래는 수정된 copy()함수와 사용 예제입니다:

```
void copy(int& dest, const int& src) {
        dest = src;
}
```

```
int a = 4, b = 5;
copy(a, b); // b의 값이 a로 복사됩니다.
```

copy() 함수의 매개변수 dest와 src는 이제 참조로 전달되므로 함수 내에서 dest에 src의 값을 복사하여 a로 올바르게 복사됩니다.

12. 비슷하게 생긴 다음 두 함수가 있다.

```
int& big1(int a, int b) {
        if (a > b)return a;
        else return b;
        }
int& big2(int& a, int& b) {
        if (a > b)return a;
        else return b;
        }

//다음 코드를 실행하였을 때, x, y의 값이 어떻게 변하는지 예측하고, 그 이유를 설명하라.
        int x = 1, y = 2;
        int& z = big1(x, y);
        z = 100;
        int& w = big2(x, y);
        w = 100;


        int x = 1, y = 2;
        int& z = big1(x, y); // x와 y 중에서 큰 값을 반환받아 z에 연결되지만, z는 함수
//내에서 생성된 변수에 연결되므로 x와 y에 영향을 주지 않음.
        z = 100; // z는 함수 내의 지역 변수에 연결되므로 x와 y에는 영향을 미치지 않음.

        int& w = big2(x, y); // x와 y 중에서 큰 값을 반환받아 w에 연결되며, w는 x 또는
//y에 연결됨.
        w = 100; // w는 x 또는 y에 연결되므로 x와 y 중에서 큰 값을 100으로 변경함.

        // 결과: x = 100, y = 2, z = 100, w = 100
```

13. MyClass 클래스의 기본 생성자(디폴트 생성자)와 복사 생성자의 원형은 무엇인가?

```
class MyClass {
public:
    MyClass(); // 기본 생성자
    // 다른 멤버 변수 또는 함수들...
};
class MyClass {
public:
    MyClass(const MyClass& other); // 복사 생성자
    // 다른 멤버 변수 또는 함수들...
};
```

14. 클래스 MyClass가 있다고 할 때, 복사 생성자가 필요한 경우가 아닌 것은?

2)void f(MyClass *p); 이 함수는 MyClass 포인터를 인자로 받습니다. 포인터를 전달할 때는 객체의 복사가 발생하지 않으므로 복사 생성자가 필요하지 않습니다.

15. 다음 클래스에 대해 물음에 답하여라.

```
class MyClass {
        int size;
        int* element;
public:
        MyClass(int size) {
                this->size = size;
                element = new int[size];
                for (int i = 0; i < size; i++)element[i] = 0;
        }
};
//1)적절한 소멸자를 작성하라.
//2)컴파일러가 삽입하는 디폴트 복사 생성자 코드는 무엇인가?
//3)MyClass에 깊은 복사를 실행하는 복사 생성자 코드를 작성하라.
```

1)

```
~MyClass() {
    delete[] element;
}
```

2)

```
MyClass(const MyClass& other) {
    size = other.size;
    element = other.element; // 얕은 복사: 같은 메모리 블록을 가리킴
}
```

3)

```
MyClass(const MyClass& other) {
    size = other.size;
    element = new int[size];
    for (int i = 0; i < size; i++) {
```

```
            element[i] = other.element[i];
        }
    }
```

16. 복사 생성자에 대해 설명한 것 중 틀린 것은?

1) 복사 생성자는 중복 가능하여 필요에 따라 여러 개 선언될 수 있다.

2) 복사 생성자가 선언되어 있지 않은 경우, 컴파일러가 디폴트 복사 생성자를 삽입한다.

3) 디폴트 복사 생성자는 얕은 복사를 실행한다.

4) 포인터 멤버가 없는 경우 디폴트 복사 생성자는 거의 문제가 되지 않는다.

3)디폴트 복사 생성자는 얕은 복사를 실행한다.

디폴트 복사 생성자는 얕은 복사를 실행하는 경우도 있지만, 항상 그렇지는 않습니다. 디폴트 복사 생성자는 클래스에 포인터 멤버가 있는 경우에 문제를 일으킬 수 있으며, 이 경우에는 얕은 복사로 인해 메모리 관리 문제가 발생할 수 있습니다. 따라서 포인터 멤버가 있는 클래스의 경우, 복사 생성자를 직접 정의하여 깊은 복사를 실행하도록 하는 것이 바람직합니다.

17. 다음 클래스에서 컴파일러가 삽입하는 디폴트 복사 생성자는 무엇인가?

```
class Student {
        string name;
        string id;
        double grade;
};
```

```
Student(const Student& other) {
    name = other.name;
    id = other.id;
    grade = other.grade;
}
```

18. 다음 클래스에서 컴파일러가 삽입하는 디폴트 복사 생성자는 무엇인가?

```
class Student {
        string* pName;
        string* pId;
        double grade;
    };
```

```
Student(const Student& other) {
    pName = other.pName;
    pId = other.pId;
    grade = other.grade;
}
```

이렇게 얕은 복사를 수행하는 디폴트 복사 생성자는 pName 및 pId가 동일한 메모리를 가리키게 하므로 문제가 발생할 수 있습니다. 따라서 포인터 멤버 변수를 사용하는 클래스의 경우, 깊은 복사를 수행하도록 사용자 정의 복사 생성자를 구현하는 것이 좋습니다.

19. 문제 15의 클래스에 대해 다음 치환문이 있다면 어떤 문제가 발생하는가?

```
int main()
{
        MyClass a(5), b(5);
        a = b;// 여기
}
```

주어진 클래스 MyClass에 대한 대입 연산자 a = b를 사용하면 얕은 복사가 수행됩니다. 즉, a 객체의 element 포인터가 b 객체의 element 포인터를 가리키게 되며, 이로 인해 두 객체가 동일한 메모리를 가리키게 됩니다. 이렇게 얕은 복사로 인해 문제가 발생할 수 있습니다.

1) 메모리 누수: 객체가 소멸될 때 메모리를 해제하는 책임이 여러 객체에 걸쳐지면 메모리 누수가 발생할 수 있습니다.

2) 두 객체가 동일한 데이터를 수정할 수 있음: 한 객체가 데이터를 변경하면 다른 객체에도 영향을 미칩니다.