KWANGWOON UNIVERSITY

고급프로그래밍

- 과제 1 -

생명 게임(Game of Life)

담당 교수님 : 이강훈 교수님
제 출 날 짜 : 2017. 05. 27.
학 과 : 컴퓨터소프트웨어학과
학 번 : 2014707040
이 름 : 유 진 혁

KWANGWOON UNIVERSITY

목 차

1. 클래스 멤버 구성

```
class LifeGame
{
public:
    LifeGame();
    ~LifeGame();
    void initialize(int w, int h);
    int getWidth() const;
    int getHeight() const;
    void setState(int i, int j, bool s);
    bool getState(int i, int j) const;
    void clear();
    void update();
    friend ostream& operator <<(ostream& os, const LifeGame& game);

private:
    int width, height;
    bool* current_cells;          // 격자에 속한 각 칸의 현재 상태를 저장하는 배열의 첫 원소 가르키는 포인터
    bool* next_cells;             // 격자에 속한 각 칸의 다음 상태를 저장하는 배열의 첫 원소 가르키는 포인터
};
```

<그림 1. 클래스 멤버 구성>

가. public 영역

Default Constructor와 Destructor, main 함수에서 호출하는 멤버 함수들, << 연산자를 overloading 하는 friend 함수를 public 영역에 선언하였다.

나. private 영역

Information Hiding과 Encapsulation을 위해 멤버 변수들을 private 영역에 선언하였다.

2. 각 함수 별 소스코드 설명

```
LifeGame::LifeGame() : width(0), height(0)          // Default Constructor
{/*intentionally empty.*/
}

LifeGame::~LifeGame()                               // Destructor
{
    delete[] current_cells;
    delete[] next_cells;
}
```

<그림 2. Default Constructor와 Destructor의 정의>

가. LifeGame()

Default Constructor로써, LifeGame 클래스의 멤버 변수 width와 height를 각각 0으로 초기화한다.

나. ~LifeGame()

Destructor로써, 프로그램 실행 중에 동적 할당되는 배열을 가리키는 포인터 current_cells, next_cells의 메모리를 해제한다.

```
void LifeGame::initialize(int w, int h)          // 현재 상태 배열 동적 할당 및 초기화
{
    current_cells = new bool[w*h];
    for (int i = 0; i < w*h; i++)
        *(current_cells + i) = false;
    width = w;
    height = h;
}
```

<그림 3. initialize 함수 정의>

다. void initialize(int w, int h)

너비(w)와 높이(h)를 인자로 받아 그 크기만큼의 격자 생성을 위해 bool 타입 1차원 배열을 동적 할당한다. 그리고 포인터 변수 current_cells가 그 배열을 가리키도록 하였다. 반복문을 이용해 배열의 모든 원소에 참조하여 모든 칸의 값을 false로 초기화해주었다. 인자로 받은 w와 h 매개변수는 각각 멤버변수 width와 height에 할당해주었다.

2. 각 함수 별 소스코드 설명

```
int LifeGame::getWidth() const
{
    return width;
}

int LifeGame::getHeight() const
{
    return height;
}
```

<그림 4. getWidth와 getHeight 함수 정의>

라. int getWidth() const

멤버 변수 width의 값을 반환해주는 Accessor 함수이다. width를 return 한다.

마. int getHeight() const

멤버 변수 height의 값을 반환해주는 Accessor 함수이다. height를 return 한다.

```
void LifeGame::setState(int i, int j, bool s)       // j번째 행 i번째 열의 칸을 s로 설정
{
    *(current_cells + j*width + i) = s;
}

bool LifeGame::getState(int i, int j) const         // j번째 행 i번째 열의 칸 상태를 반환
{
    return *(current_cells + j*width + i);
}
```

<그림 5. setState와 getState 함수 정의>

바. void setState(int i, int j, bool s)

격자의 j번째 행 i번째 열의 칸 상태를 s로 설정하는 함수이다. 1차원 배열로 동적 할당했기 때문에 j번째 행은 배열 첫 원소에서 (width(격자의 너비) X j)를 더한 지점부터 시작한다. 과제 설명에서 격자의 (i,j) 위치에 해당하는 칸의 상태를 설정하라고 하였으나, main 함수에서 setState 함수를 호출할 때 넘겨주는 인자의 값과 예시 실행파일의 결과를 비교 · 확인한 결과, 위와 같이 정의하는 것이 정확한 결과를 출력해주어 i와 j의 위치를 바꿔주었다.

사. bool getState(int i, int j) const

격자의 j번째 행 i번째 열의 칸 상태를 return 하는 함수이다. 현재 상태의 격자를 화면에 출력하기 위해 필요하다. setState 함수와 통일성을 위해 return 하는 식에서 i와 j의 위치를 바꾸었다.

2. 각 함수 별 소스코드 설명

```
void LifeGame::clear()                                  // 현재 모든 칸의 상태를 죽음(false)로 설정
{
    for (int i = 0; i < width*height; i++)
        *(current_cells + i) = false;
}
```

<그림 6. clear 함수 정의>

아. void clear()

반복문을 이용해 배열의 모든 원소에 참조하여 각 칸의 값을 모두 false로 설정하였다. 프로그램 진행 중, 격자의 초기 형태를 설정하기 전에 초기화를 위해 clear 함수를 호출한다.

```
void LifeGame::update()                                 // 모든 칸의 상태를 다음 상태로 변환
{
    int i, live_cell_num;
    next_cells = new bool[width*height];                // next_cells 배열 동적할당
    for (i = 0; i < width*height; i++)                  // next_cells를 currnet_cells와 같은 값으로 초기화
        *(next_cells + i) = *(current_cells + i);
```

<그림 7. update 함수 정의 중 지역 변수 선언과 next_cells 포인터 초기화 부분>

자. void update()

우선, 반복문의 index로 쓰일 int형 변수 i와 주위 칸들의 상태를 확인하여 살아있는 칸의 개수를 저장하는 int형 변수 live_cell_num를 선언하였다. 클래스의 멤버 변수 width와 height를 이용하여 current_cells가 가리키는 배열과 똑같은 크기의 배열을 동적 할당한 뒤, 멤버변수 포인터 next_cells가 가리키도록 하였다. 반복문으로 current_cells의 배열 모든 원소를 참조하여, current_cells 배열과 next_cells 배열 모든 원소가 같은 값을 갖도록 대입 연산하였다.

```
    for (i = 0; i < width*height; i++)
    {
        live_cell_num = 0;              // 주위 칸의 상태 저장

        if ((i - width) < 0)                    // 첫 번째 행인 경우
        {
            if (*(current_cells + i + (width * (height - 1))) == true)          // 위 칸 확인
                live_cell_num += 1;
            if (*(current_cells + i + width) == true)                           // 아래 칸 확인
                live_cell_num += 1;
            if (i == (width - 1))                           // 첫 번째 행 오른쪽 끝 칸
            {
                if (*(current_cells + i + (width * (height - 2)) + 1) == true)      // 오른쪽 위 칸 확인
                    live_cell_num += 1;
                if (*(current_cells + i + (1 - width)) == true)                     // 오른쪽 칸 확인
                    live_cell_num += 1;
                if (*(current_cells + i + 1) == true)                               // 오른쪽 아래 칸 확인
                    live_cell_num += 1;
                if (*(current_cells + i + (width * (height - 1)) - 1) == true)      // 왼쪽 위 칸 확인
                    live_cell_num += 1;
                if (*(current_cells + i - 1) == true)                               // 왼쪽 칸 확인
                    live_cell_num += 1;
                if (*(current_cells + i + width - 1) == true)                       // 왼쪽 아래 칸 확인
                    live_cell_num += 1;
            }
```

<그림 8. update 함수 정의 중 현재 자신의 칸이 첫 줄 오른쪽 끝 칸일 경우에 대한 부분>

2. 각 함수 별 소스코드 설명

반복문으로 첫 번째 원소부터 마지막 원소까지 자신의 칸을 옮겨가며 자신의 칸 주변 8개의 칸 상태를 확인하고, 살아있는 칸의 수를 live_cell_num에 저장하게 하였다. 반복문 첫 부분에서 live_cell_num 변수를 0으로 초기화하여 자신의 칸이 바뀔 때마다 live_cell_num 값이 다시 0이 되도록 하였다. 본 과제에서 격자의 왼쪽 경계와 오른쪽 경계, 그리고 위쪽 경계와 아래쪽 경계가 서로 맞닿아 있다고 가정했으므로 자신의 칸이 각 경계에 있을 때별로 분기를 나누어 코딩하였다. 자신의 칸이 첫 번째 행에 있는 경우, 마지막 행에 있는 경우, 중간 행에 있는 경우로 나누고, 여기에 각각 다시 오른쪽 끝 칸인 경우, 왼쪽 끝 칸인 경우, 가운데 칸인 경우로 나누어 자신의 칸 위치에 맞는 주변 8개 칸의 상태를 확인하도록 하였다. <그림 8>은 자신의 칸이 첫 번째 행 오른쪽 끝 칸일 경우, 그 주변 8개의 칸 상태를 확인하는 과정이다. 이 외의 경우에 대한 소스코드 캡쳐 그림은 생략하도록 하겠다. (첨부된 소스코드 파일 참고)

```
        if (*(current_cells + i) == true && live_cell_num <= 1)                  // 자신이 현재 살아있고, 주위에 1개 이하의 칸만 살아있다면,
            *(next_cells + i) = false;                                           // 죽는다.
        else if (*(current_cells + i) == true && live_cell_num >= 4)             // 자신이 현재 살아있고, 주위에 4개 이상의 칸이 살아있다면,
            *(next_cells + i) = false;                                           // 죽는다.
        else if (*(current_cells + i) == true && (live_cell_num == 2 || live_cell_num == 3))   // 자신이 현재 살아있고, 주위에 2~3개의 칸이 살아있다면,
            *(next_cells + i) = true;                                            // 계속 산다.
        else if (*(current_cells + i) == false && live_cell_num == 3)            // 자신이 현재 죽어있고, 주위에 정확히 3개의 칸이 살아있다면,
            *(next_cells + i) = true;                                            // 살아난다.
    }
```

<그림 9. update 함수 정의 중 게임 규칙 적용 부분>

현재 자신의 칸 상태와 live_cell_num에 저장된 값을 참고하여 조건문을 이용해 생명 게임 규칙이 적용되도록 하였다. 현재 상태가 저장된 current_cells의 배열 값을 수정하지 않고, next_cells의 배열을 수정하여 next_cells의 배열이 그 다음 상태의 격자를 저장하도록 하였다. 이는 '순서 의존성'에 의해 생기는 문제를 해결하기 위함이다.

```
    for (i = 0; i < width*height; i++)                    // next_cells에 보관된 값을 current_cells로 복사
        *(current_cells + i) = *(next_cells + i);
}
```

<그림 10. update 함수 정의 중 마지막 부분, next_cells의 배열 값으로 current_cells의 배열로 대입>

마지막으로, next_cells의 배열에 저장된 값들을 다시 current_cells 배열에 대입해 현재 격자 형태를 다음 격자 형태로 변경시켰다. 반복문으로 모든 원소를 참조하여 각 원소 하나하나 값을 차례로 대입 연산시켰다.

2. 각 함수 별 소스코드 설명

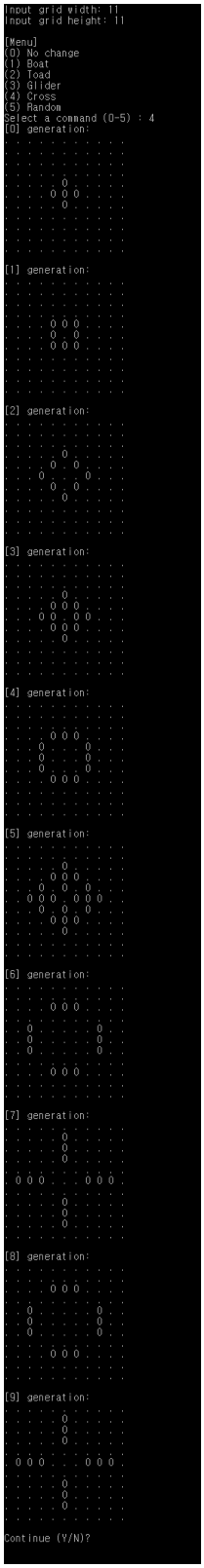
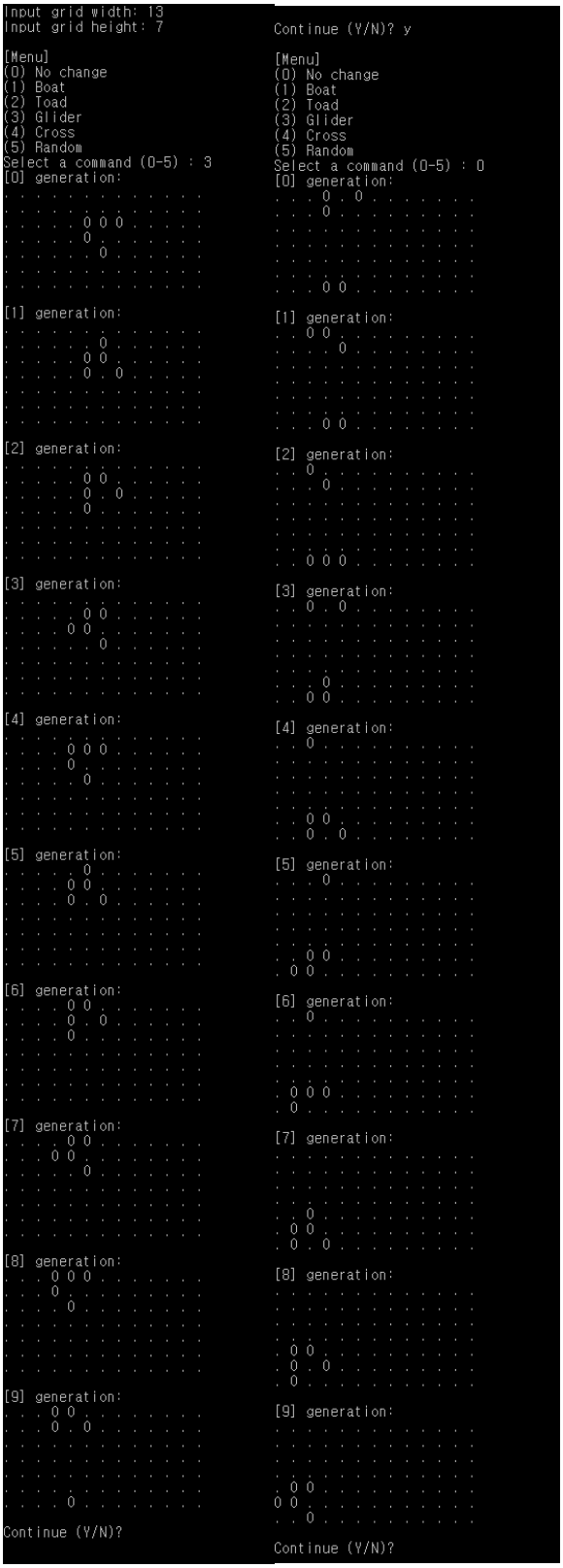
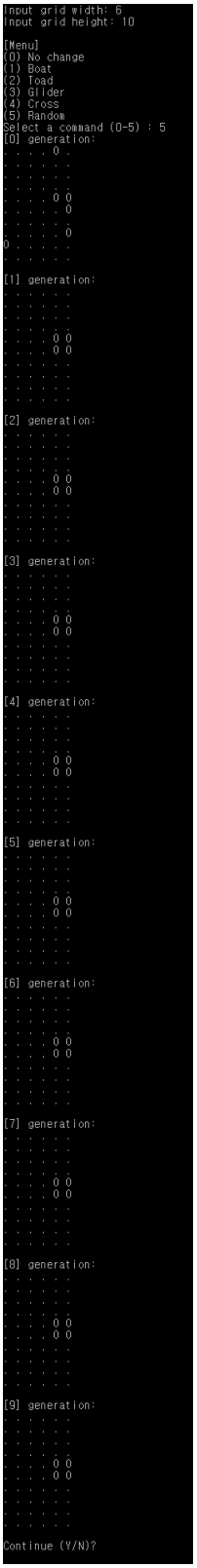
```
ostream& operator <<(ostream& os, const LifeGame& game)     // 객체를 cout으로 출력하기 위한 "<<" Overloading
{
    for (int j = 0; j < game.height; j++)                   // j번째 행
    {
        for (int i = 0; i < game.width; i++)                // i번째 열
        {
            if (game.getState(i, j) == true)                // 칸이 살아있으면 0 출력
                os << "0 ";
            else                                            // 칸이 죽어있으면 . 출력
                os << ". ";
        }
        os << endl;                                         // j번째 행의 열 다 출력하면 개행
    }
    return os;
}
```

<그림 11. << 연산자 Overloading 하는 friend 함수 정의>

차. ostream& operator <<(ostream& os, const LifeGame& game)

위에서 정의한 getState(int i, int j) 함수를 호출하여 해당 칸의 상태를 반환하였다. 칸의 상태가 true이면 “0 ”를, 그렇지 않으면 “. ”를 cout으로 출력되게 했다. 또, 반복문의 인덱스 j가 행을 나타내므로 j 값이 증가하기 직전에 개행을 하여 2차원 모양의 격자로 출력되게 하였다.

3. 프로그램 실행 예

너비 11, 높이 11, 초기 형태 크로스, 1회 실행	너비 13, 높이 7, 초기형태 글라이더, 2회 실행	너비 6, 높이 10, 초기형태 랜덤, 1회 실행
		

4. 고 찰

제일 먼저, 과제 설명을 참고하여 클래스 선언문에 멤버 변수와 함수를 구성하였다. Principles of OOP를 고려해 멤버 변수는 private 영역에, 멤버 함수와 friend 함수는 public 영역에 선언하였다. 그러고 나서, 제공된 main() 함수를 참고하여 멤버 함수를 정의하였다. Default Constructor와 Destructor, getWidth() 함수와 getHeight() 함수와 같은 기본적인 함수 구성에는 어려움이 없었다. initialize(int w, int h) 함수도 배열 동적 할당을 위해 new 연산자를 사용했지만 큰 어려움은 없었고, << 연산자를 오버로딩하는 friend 함수의 경우에도, C++를 배우면서 처음 연산자 오버로딩을 해본거지만 수업 시간에 배운 내용에서 크게 벗어나지 않아 무리 없이 짤 수 있었다. 그런데, 여기서 격자의 행과 열을 구분하는데 있어 조금 문제가 있었다. 이 때문에 setState(int i, int j, bool s) 함수를 구성할 때 첫 난관에 부딪혔다. 처음에 과제 설명에서 준 힌트대로 *(current_cell + i*width + j)로 i번째 행, j번째 열에 접근하려고 했다. 그러나 main() 함수에서 초기형태를 지정할 때, setState(int i, int j, bool s) 함수로 넘겨주는 인자를 보니 *(current_cell + i*width + j)를 이용하면 예제 실행파일과 다른 모양의 초기형태를 형성했다. 이 사실을 몰랐을 땐, 과제 설명에 나온 힌트를 단순히 믿고, 그대로 코드를 작성했기 때문에 어디서 문제가 생긴 건지 발견하기 어려웠다. 마지막에 main() 함수 부분을 확인하다 위와 같은 문제가 있다는 것을 발견했고, main() 함수는 수정할 수 없으니 *(current_cell + j*width + i)를 이용해 j번째 행, i번째 열을 접근하는 방법으로 수정을 했다. 그 결과, 예제 실행파일과 같은 정상적인 초기 형태가 출력되었다. 행 구분은 j, 열 구분은 i를 사용하기로 하면서 getState(int i, int j)함수와 << 연산자 오버로딩 함수도 수정이 조금 필요했다. update() 함수를 제외한 나머지 함수들을 구성한 뒤, 컴파일하여 실행해보았더니, 입력받은 너비와 높이대로 올바른 초기형태를 갖춘 격자가 출력되었다.

마지막으로 update() 함수를 정의하는데 가장 오랜 시간이 걸렸다. 같은 크기의 배열을 동적 할당하여 next_cells 포인터로 가리키게 한 다음, 그 배열에 다음 상태를 저장하고 마지막에 current_cells의 배열로 각 원소 값을 덮어씌워줘야 한다는 건 알고 있었다. 그러나 각 경계가 맞닿아 있다는 가정을 적용하는 부분이 쉽지 않았다. 종이에 배열을 그려가며 어떤 값을 더하고 빼야할지 자신의 칸을 한 칸씩 옮겨가며 주변 8칸을 확인해보았다. 일반화된 간단한 식이 구해지길 기대했지만, 생각처럼 나오지 않았다. 결국, 자신의 칸이 어디 위치하는가에 따라 분기를 나눠 그 칸에 맞는 주변 8칸을 확인하도록 하였다. 처음 구성할 땐 '왼쪽 열이 존재할 때, 존재하지 않을 때, 오른쪽 열이 존재 할 때, 존재하지 않을 때…….' 같이 각 분기를 너무 상세하게 나누어 코드의 양도 많고 가독성도 낮았다. 하지만 공통된 조건의 분기끼리 묶어 조건문의 수를 줄여 나가다보니 보다 깔끔한 코드를 구성할 수 있었다.

예제 실행파일과 동일한 결과를 출력하도록 프로그램을 완성했지만 구현에 있어 아쉬운 점이 있다면, 내가 짠 프로그램과 예제 실행파일 사이에 출력속도 차이가 있다는 것이다. 나름 코드를 단순화하기 위해 노력했지만 속도를 더 단축시킬 수 있는 방법이 존재하는데 찾지 못했다는 점이 아쉽다. 어떻게 하면 코드를 더 최적화할 수 있는지, 코드 최적화 방법에는 어떤 것들이 있는지 더 알아봐야겠다.