

# Chapter.4

## 와이어 헌터

4장부터는 지금까지 학습한 C언어를 이용하여 실제 게임을 제작하게 된다. 게임 제작을 위한 각 장의 C언어 문법은 1장에서 소개한 내용이 전부이며 그 내용만 이해하면 앞으로 제작하게 되는 모든 게임을 완성하고 이해할 수 있다.

C언어 문법을 완전히 이해하지 못해서 다시 처음부터 C언어 문법을 학습하고 이 부분을 공부하려는 사람도 있을 것이다. 하지만 그렇게 하지 않아도 된다. 앞으로 제작하는 모든 게임은 작은 부분에서부터 단계적으로 제작하여 전체를 완성하는 방향으로 진행되므로, 각 단계를 완성하면서 부족한 문법을 보완해 나간다면 무리 없이 게임을 제작할 수 있다.

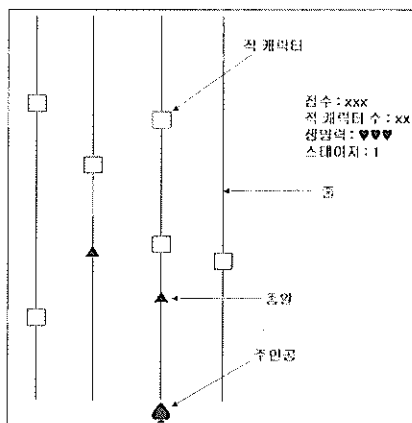
와이어헌터는 슈팅 게임의 기본적인 요소인 주인공, 적 캐릭터, 총알만을 포함했지만 적 캐릭터 수와 이동 속도에 변화를 준다면 다양한 스테이지의 게임을 만들 수 있다.

스트리

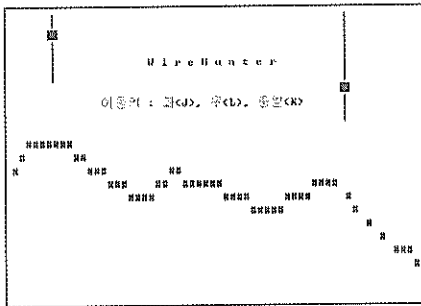
## 게임 방식

## 제한 사항

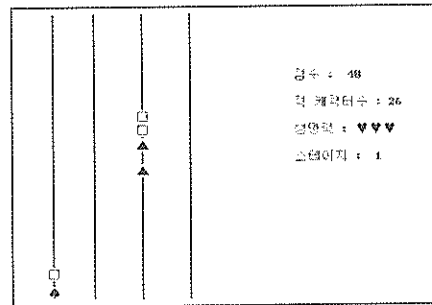
기회



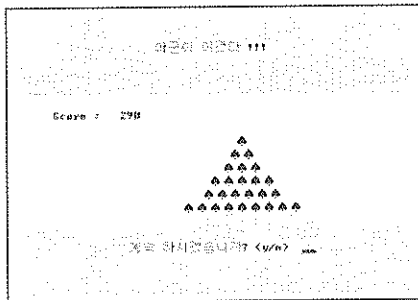
046 : C를 이용한 게임프로그래밍



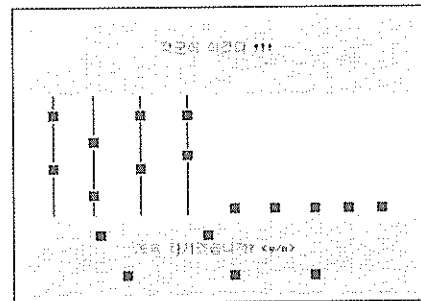
초기 화면



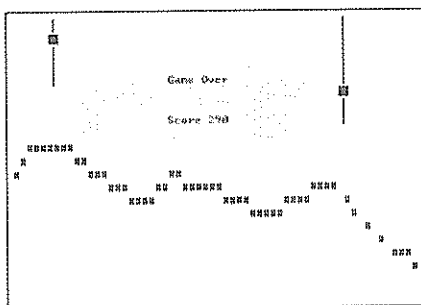
진행 화면



미션 성공 화면



미션 실패 화면



게임 종료 화면

[그림 4-2] 실행 화면

# Lesson 03



Game Programming Using C

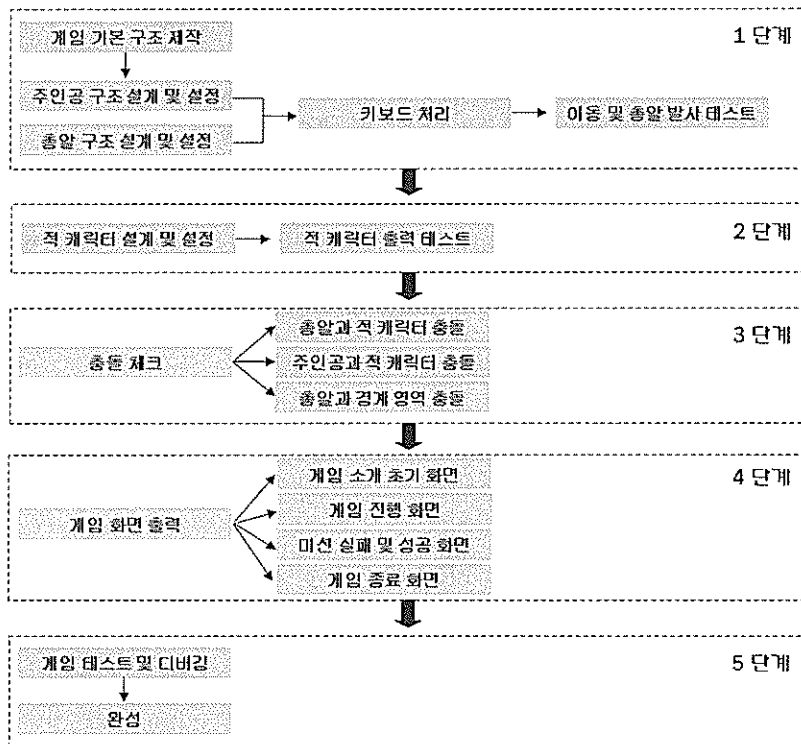
## 게임 제작 로드맵

와이어헌터 게임은 전체적으로 5단계로 나눈다. 제작 순서는 [그림 4-3]에 따라 단계별로 제작한다.

각 단계에서 제작된 각 모듈은 2장에서 제작한 게임의 기본 구조에 넣어 테스트한다.

로드맵에 따른 개발 방법은 구현 원리를 명확하게 하며 디버깅(debugging) 시간을 최소화시켜 주는 이점이 있다. 또한 '무엇을 어떻게 제작할 것인가'와 '제작된 부분을 어떻게 게임의 기본 구조에 탑재할 것인가'와 같은 사항을 전체적으로 볼 수 있도록 해 준다.

이제 아래의 로드맵을 살펴보면서 단계별로 제작해 보자.



[그림 4-3] 게임 제작 로드맵



Game Programming Using C

## 단계별 프로그래밍

Lesson  
04

1단계



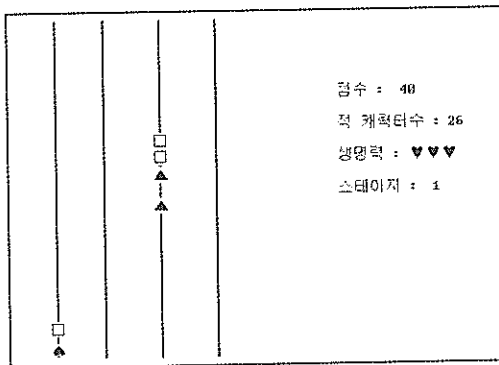
이 단계의 핵심은 앞에서 학습한 게임의 기본 구조를 먼저 제작하여 주인공과 총알을 적용해 보는 것이다.

### 🐟 게임의 기본 구조

앞으로 제작하는 게임의 기본 구조에 대한 설명은 2장의 내용과 동일하므로 생략한다.

### 🐟 주인공

#### ▶ 속성 정의



[그림 4-5]

[그림 4-5] 를 보면 주인공에게 필요한 데이터가 무엇인지를 생각해 볼 수 있다. 먼저 주인공은 키보드에 따라 좌우로 이동해야 하며 천하무적이 아니므로 적군의 공격에 의해 소멸되기도 해야 한다.

그래서 주인공은 이동 좌표와 생명이라는 속성이 있어야 한다. 이 두 가지 속성은 결국 데이터이므로 변수가 필요하다.

- ① 이동 좌표
- ② 생명

이동 좌표는 정수형 데이터이므로 int형으로 선언할 수 있다. 생명 속성은 적군으로부터 공격을 받게 되면 값을 1씩 감소하며 그 값이 0이 되면 주인공은 소멸하게 된다. 그래서 생명 속성의 데이터형은 int형으로 선언할 수 있다. 이 두 속성은 주인공에게 국한된 데이터이므로 다음과 같은 구조체로 정의한다.

```
typedef struct _PLAYER
{
    int nX, nY;
    int nLifePower;
} PLAYER;
```

#### ▶ 이동 및 키보드 처리

이동이란, 방향키에 따라 좌표 값을 증감해 주는 것을 말한다.

[그림 4-5] 를 보면 주인공 y 좌표는 고정이지만 x 좌표는 4개의 줄 좌표로 제한되어 있다. 그래서 좌우 키를 계속 눌러 이동을 하려고 해도 4개의 줄 좌표를 벗어날 수 없는 것이다. 이러한 점으로 알 수 있는 것은 주인공 캐릭터 x 좌표와 줄 좌표가 동일하다는 것이다. 그리고 모든 캐릭터가 줄과 줄 사이를 이동하기 때문에 줄을 출력하기 위한 x 좌표 값은 공유하는 값이라는 것을 알 수 있다. 그러므로 줄의 x 좌표가 변경된다면 모든 캐릭터의 x 좌표도 변경되어야 한다.

이와 같이 하나의 값을 공유하는 경우에는 변수를 전역 변수로 선언하며, 이 변수의 변화는 전체 이동 좌표에 영향을 주게 된다. 또한 전역 변수로 선언한다는 것은 이 변수가 전체 프로그램 안에 오직 하나만 존재한다는 의미가 된다.

줄은 4개의 x 좌표가 있어야 하므로 배열로 선언한다. 배열로 선언한 4개의 값에 주인공, 총알, 적 캐릭터가 상호 접근할 수 있는 방법에는 무엇이 있을까? 그것은 배열의 인덱스 값을 주인공, 총알, 적 캐릭터가 가지도록 하는 것이다. 그렇게 하면 배열에서 값을 쉽게 가져올 수 있으며 배열값이 바뀌더라도 주인공, 총알, 적 캐릭터 좌표를 일일이 수정할 필요가 없다.

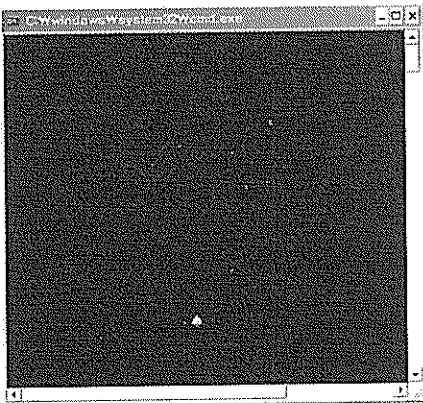
이러한 사항이 적용된 주인공 캐릭터의 속성을 정의하면 다음과 같다.

```
typedef struct _PLAYER
{
    int nX, nY;
    int nLifePower;
    int nIndex;    // Note: 줄 좌표에 대한 배열 인덱스
} PLAYER;
```



실행하기 : 4-1

키보드 입력에 따라 이동하는 주인공 캐릭터를 출력해 보자.



[그림 4-6] 실행 화면

```
01  #include "stdafx.h"
02  #include <windows.h>
03  #include <conio.h>
04  #include <time.h>
05
06  void gotoxy( int x, int y )
07  {
08      COORD CursorPosition = { x, y };
09      SetConsoleCursorPosition( GetStdHandle( STD_OUTPUT_HANDLE ), CursorPosition );
10  }
11
12  typedef struct _PLAYER
13  {
14      int nX, nY;
15      int nLifePower;
16      int nIndex;
```

◉ NEXT

```

17     } PLAYER;
18
19     PLAYER g_Player;
20     int g_XPos[4] = { 6, 12, 19, 26 }; // Note: 줄의 x 좌표 배열
21
22     int _tmain(int argc, _TCHAR* argv[])
23     {
24         int nKey;
25         clock_t CurTime, OldTime;
26         g_Player.nIndex = 1;
27         g_Player.nX = g_XPos[g_Player.nIndex];
28         g_Player.nY = 20; // Note: 주인공 Y값 고정
29         OldTime = clock();
30
31         while( 1 )
32         {
33             if( _kbhit() )
34             {
35                 nKey = _getch();
36                 switch( nKey )
37                 {
38                     case 'j' : // Note: 왼쪽 키
39                         if( g_Player.nIndex - 1 >= 0 )
40                             g_Player.nIndex--;
41                         break;
42                     case 'l' : // Note: 오른쪽 키
43                         if( g_Player.nIndex + 1 <= 3 )
44                             g_Player.nIndex++;
45                         break;
46                 }
47             }
48
49             system( "cls" );
50             g_Player.nX = g_XPos[g_Player.nIndex];
51             gotoxy( g_Player.nX, g_Player.nY );
52             printf( "♣" );
53
54             while( 1 )
55             {
56                 CurTime = clock();
57                 if( CurTime - OldTime > 33 )
58                 {
59                     OldTime = CurTime;
60                     break;
61                 }
62             }
63         }
64         return 0;
65     }

```

[소스 4-1]



○ 26~28행 : 주인공 캐릭터를 초기화하는 부분이다.

26행에서 g\_Player.nIndex의 값을 1로 초기화하는 것은 g\_XPos[ ]로부터 두 번째 좌표 값인 12를 가져오기 위함이다. 그리고 27행의 g\_Player.nX=g\_XPos[g\_Player.nIndex]는 g\_Player.nX=g\_XPos[1]과 같다.

○ 36 ~ 46행 : 키보드 입력에 따라 이동하는 부분이다.

이동은 좌표 변환이며 좌표 변환을 한다는 것은 값이 증감한다는 것이다. 그러나 우리는 g\_XPos[ ]로부터 이미 증감된 좌표를 가져와 설정하는 것이므로 좌우로 이동한다는 것은 배열의 인덱스를 증감하는 것을 의미한다. 그래서 왼쪽으로 이동하는 것은 인덱스가 1씩 감소하는 것이며 오른쪽으로 이동하는 것은 인덱스를 1씩 증가시키는 것이다. 주의할 것은 인덱스의 범위는 0~30이므로 증감을 하더라도 어 범위를 벗어나지 않도록 g\_Player.nIndex의 값을 설정하는 것이다. 그래서 39 ~ 40행의 내용을 보면 다음과 같이 g\_Player.nIndex-1 결과가 0 이상일 때만 1씩 감소시키고 있다.

```
if( g_Player.nIndex - 1 >= 0 )  
    g_Player.nIndex--;
```

이러한 코드는 다음과 같이 코딩할 수도 있다.

```
g_Player.nIndex--;  
if( g_Player.nIndex < 0 )  
    g_Player.nIndex = 0;
```

## 총알

총알은 k 키가 눌리면 발사되며 주인공이 있는 x 좌표 위치에서 출발하여 위로 이동한다.

### ▶ 속성

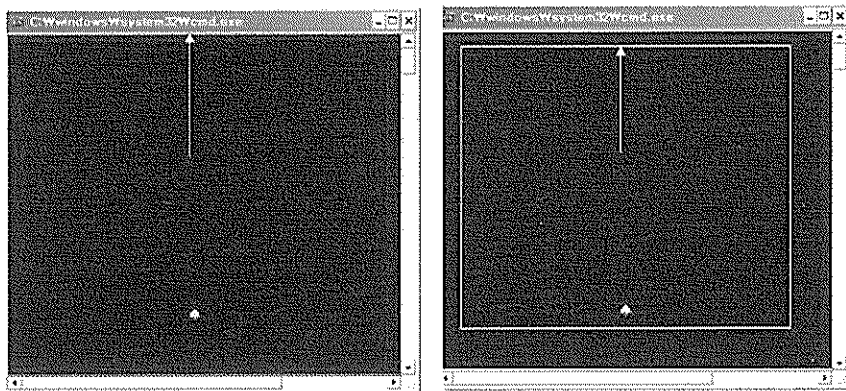
총알도 이동을 하므로 좌표 정보가 필요하다. 특히 총알은 주인공이 있는 위치가 출발 지점이 되므로 주인공의 x 좌표는 총알의 x 좌표가 된다. 또 프레임에 따라 y 좌표를 감소시키면 총알은 아래에서 위로 이동한다.

기획을 살펴보면, 총알은 키 입력에 따라 무제한으로 발사되는 것이 아니라 한 화면에 4개까지 발사되도록 제한하고 있다. 총알의 이동은 현재 프레임과 이전 프레임 간의 차이로 결정한다. 이러한 사항을 기본으로, 총알의 속성을 정리하면 다음과 같다.

- ① 좌표 ( x, y )
- ② 생명
- ③ 이전 프레임

총알의 생명 속성은 주인공의 생명 속성과 약간의 차이가 있는데 총알이 소멸되는 두 가지 경우는 다음과 같다.

첫째, 총알이 화면의 경계 또는 임의로 정한 경계 영역에 닿았을 경우이다. 총알은 아래에서 위로 진행하며 총알의 y 좌표가 0이 되거나 임의로 정한 경계 좌표에 닿게 되는 경우에 총알은 소멸된다. 총알이 소멸되면 주인공은 총알을 다시 요구할 수 있으며, 소멸된 총알은 재사용된다.



[그림 4-7] 화면 경계 영역 부분과 임의의 경계 영역 부분

둘째, 총알과 적 캐릭터 사이에 충돌이 이루어졌을 경우이다. 총알은 주인공의 이동과는 상관없이 스스로 이동하며 일정한 속도로 이동한다. 이것은 일정한 시간 간격으로 y 좌표 값을 감소시키는 것을 의미한다.

여기서 현재 프레임 값과 이전 프레임 값과의 차이를 계산하여 이동 시점을 결정한다. 현재의 프레임 값은 현재 시점에서의 프레임 값이므로 언제든지 얻을 수 있지만 이전 프레임 값은 과거 시점이 되기 때문에 그때의 프레임 값을 저장하기 위한 변수가 따로 있어야 한다. 그래서 총알의 속성을 구조체로 정의하면 다음과 같다.

```
typedef struct _BULLET
{
    int    nLife;
    int    nX, nY;
    int    nOldFrame; // Note: 이전 프레임
} BULLET;
```

#### ▶ 총알 발사

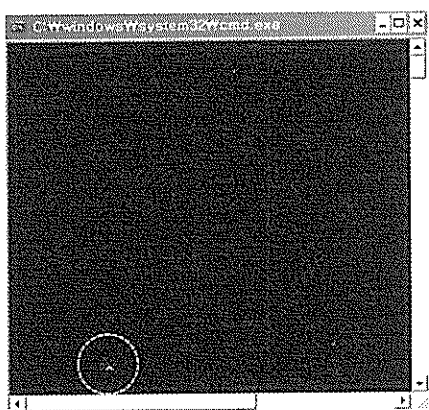
위의 총알 속성 구조체에서 생명 속성 변수인 nLife 값이 1이면 현재 사용 중이며 화면에 출력되고 있는 것을 의미한다. 반면에 nLife 값이 0이라면 사용되지 않는 총알로서 소멸된 총알을 말한다.

그러므로, 총알을 발사한다는 말은 총알 속성 중에서 nLife 값이 0인 것을 찾아내어 1로 만들고 좌표를 설정한다는 뜻이다.



## 실습하기 : 4-2

k 키를 눌렀을 때 총알이 출력만 되도록 프로그래밍해 보자.  
총알의 이동이 없기 때문에 4번 k 키를 눌러도 같은 좌표에 총알 4개가 겹쳐서 나타나게 된다. 결국, 총알 1개가 화면에 출력된 것처럼 보이는 것이다.



[그림 4-8] 총알 4개가 겹쳐서 출력된 모습



```
01  #include "stdafx.h"
02  #include <windows.h>
03  #include <conio.h>
04  #include <time.h>
05
06  int g_nFrameCount = 0;
07
08  typedef struct _BULLET
09  {
10      int    nLife;
11      int    nX, nY;
12      int    nOldFrame; // Note: 이전 프레임
13  } BULLET;
14  BULLET g_sBullet[4];
15
16  void gotoxy( int x, int y )
17  {
18      COORD CursorPosition = { x, y };
19      SetConsoleCursorPosition( GetStdHandle( STD_OUTPUT_HANDLE ), CursorPosition );
20  }
21
22  int _tmain(int argc, _TCHAR* argv[])
23  {
24      int i, nKey;
```

◉ NEXT

```

25     clock_t OldTime, CurTime;
26     OldTime = clock();
27
28     while( 1 )
29     {
30         if( _kbhit() )
31         {
32             nKey = _getch();
33             if( nKey == 'k' )
34             {
35                 for( i = 0 ; i < 4 ; i++ )
36                 {
37                     if( g_sBullet[i].nLife == 0 )
38                     {
39                         g_sBullet[i].nLife = 1;
40                         g_sBullet[i].nOldFrame = g_nFrameCount;
41                         g_sBullet[i].nX = 12;
42                         g_sBullet[i].nY = 20;
43                         break;
44                     }
45                 }
46             }
47         }
48         // Note: 총알 이동
49
50         // Note: 총알 출력
51         system( "cls" );
52         for( i = 0 ; i < 4 ; i++ )
53         {
54             if( g_sBullet[i].nLife )
55             {
56                 gotoxy( g_sBullet[i].nX, g_sBullet[i].nY );
57                 printf( "^" );
58             }
59         }
60
61         while( 1 )
62         {
63             CurTime = clock();
64             if( CurTime - OldTime > 33 )
65             {
66                 OldTime = CurTime;
67                 break;
68             }
69         }
70         g_nFrameCount++;
71     }
72     return 0;
73 }

```

[소스 4-2]

○ 35~45행 : k 키를 눌렀을 때 총알 4개 중에서 사용할 수 있는 총알을 찾는 부분이다.

총알을 사용할 수 있는 조건은 nLife 값이 0인 경우이며 nLife 값이 0이 되는 경우는 세 가지가 있다.

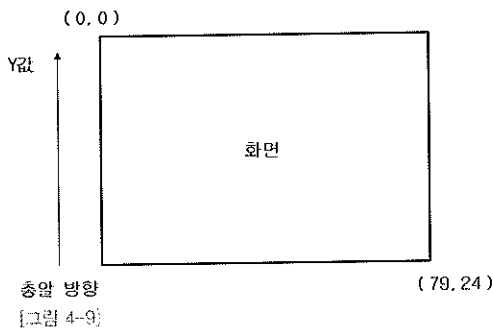
- ① 초기화
- ② 경계 영역에 충돌
- ③ 적 캐릭터와 충돌

14행에서 g\_sub[테4]와 같이 전역 변수를 선언하면 각 변수의 초기화 값은 0으로 설정된다. 총알 4개 중 사용이 가능한 총알을 찾았다면 39행과 같이 nLife 값을 1로 설정하고 현재 프레임 값을 40행과 같이 저장하여 다음의 이동 여부를 체크할 때 사용한다.

○ 52~59행 : 총알을 출력하는 부분이다.

총알의 출력 조건은 단 한 가지이다. 바로 총알의 nLife 값이 1인가 하는 점이다. 그래서 54행에서 이와 같은 출력 조건을 체크하고 있다.

## ▶ 총알 이동



총알 이동은 먼저 총알 속성인 nLife 값이 1인 변수를 대상으로 총알의 y 좌표를 [그림 4-9]와 같이 감소시킨다. 또한 살아 있는 총알은 주인공의 움직임과 상관없이 일정한 프레임 간격으로 y 좌표 값을 감소시킨다.

총알의 이동 조건은 일정한 프레임 간격이 되었을 때 좌표 증감을 해 주는 것인데, 다음의 계산식으로 일정한 프레임 간격을 구할 수 있으며, 2장에서도 이와 같은 사항을 이미 설명한 바 있다.

$$\text{프레임 간격} = \text{현재 프레임} - \text{이전 프레임}$$

총알이 발사되면 y 값이 감소하게 되고 화면의 경계 영역과 좌표가 같거나 작은 경우가 발생한다. 이런 경우에 총알은 경계 영역과 충돌한 것으로 판단한다.



### 실습하기 : 4-3

y 축의 경계 영역을 0 이라 하고 k 키를 눌렀을 때 y 축 경계 영역을 향하여 이동하는 총알을 프로그래밍해 보자.

참고로 [실습 4-2] 에서 제작한 소스에 '총알 이동' 부분만 추가하면 된다.



```

01  #include "stdafx.h"
02  #include <windows.h>
03  #include <conio.h>
04  #include <time.h>
05
06  void gotoxy( int x, int y )
07  {
08      COORD CursorPosition = { x, y };
09      SetConsoleCursorPosition( GetStdHandle( STD_OUTPUT_HANDLE ), CursorPosition );
10  }
11
12  typedef struct _BULLET
13  {
14      int nLife;
15      int nX, nY;
16      int nOldFrame; // Note: 이전 프레임
17  } BULLET;
18
19  BULLET g_sBullet[4];
20  int g_nFrameCount;
21
22  int _tmain(int argc, _TCHAR* argv[])
23  {
24      int i, nKey;
25      clock_t OldTime, CurTime;
26      OldTime = clock();
27
28      while( 1 )
29      {
30          if( _kbhit() )
31          {
32              nKey = _getch();
33              if( nKey == 'k' )
34              {
35                  for( i = 0 ; i < 4 ; i++ )
36                  {
37                      if( g_sBullet[i].nLife == 0 )
38                      {
39                          g_sBullet[i].nLife = 1;
40                          g_sBullet[i].nOldFrame = g_nFrameCount;
41                          g_sBullet[i].nX = 12;
42                          g_sBullet[i].nY = 20;
43                          break;

```

```

44         }
45     }
46 }
47
48 for( i = 0 ; i < 4 ; i++ ) // Note: 총알 이동
49 {
50     if( g_sBullet[i].nLife )
51     {
52         if( g_nFrameCount - g_sBullet[i].nOldFrame > 5 )
53         {
54             if( g_sBullet[i].nY - 1 == 0 )
55                 g_sBullet[i].nLife = 0;
56             else
57             {
58                 g_sBullet[i].nY--;
59                 g_sBullet[i].nOldFrame = g_nFrameCount;
60             }
61         }
62     }
63 }
64
65 // Note: 총알 출력
66 system( "cls" );
67 for( i = 0 ; i < 4 ; i++ )
68 {
69     if( g_sBullet[i].nLife )
70     {
71         gotoxy( g_sBullet[i].nX, g_sBullet[i].nY );
72         printf( "^" );
73     }
74 }
75
76 while( 1 )
77 {
78     CurTime = clock();
79     if( CurTime - OldTime > 33 )
80     {
81         OldTime = CurTime;
82         break;
83     }
84 }
85 g_nFrameCount++;
86 }
87 return 0;
88 }

```

[소스 4-3]

○ 48~64행 : 총알의 이동에 관한 부분이다.

총알의 이동 조건은 다음의 두 가지 조건을 만족해야 한다.

- ① 사용하는 총알인가?
- ② 이동할 프레임 간격이 되었는가?

50행은 ① 조건에 해당하는 코드가 된다. 이 조건을 만족한다면 52행에서 ② 조건을 체크하는데 이러한 식은 화면 전환을 하기 위한 대기 상태의 코드와 유사하다. 다음의 표에서는 52행과 79행의 조건을 비교하고 있다.

이동 조건 체크	대기 상태 시간 체크
<code>if( g_nFrameCount - g_sBullet[i].nOldFrame &gt; 5 )</code>	<code>if( CurTime - OldTime &gt; 33 )</code>

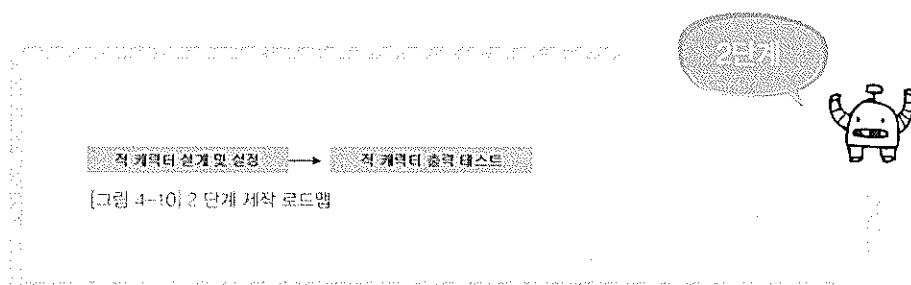
○ 54~55행 : 총알이 소멸되는 부분이다.

현재 총알이 소멸되는 조건은 화면의 경계 영역에 닿는 경우이다. 또한 총알이 소멸된다는 것은 화면에 총력되지 않는다는 뜻이며 총알의 nLife 값을 0으로 설정하는 것이다.

○ 58~59행 : 총알을 이동시키는 부분이다.

이동에 대한 조건이 성립되면 y 좌표를 감소시킨다. 그리고 좌표가 변경되었다면 다음 이동은 이 시점의 프레임부터 프레임 간격을 체크해야 한다.

그래서 58행을 보면 현재의 프레임(g\_nFrameCount)을 g\_sBullet[i].nOldFrame에 저장하고 있는 것이다.



## ❧ 적 캐릭터

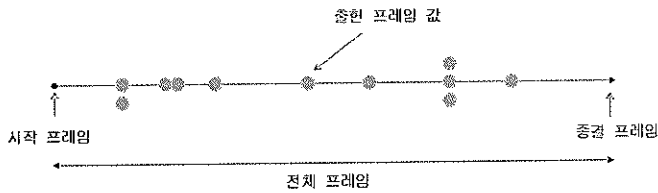
### ▶ 속성

적군도 캐릭터이므로 주인공과 유사한 속성을 가질 뿐만 아니라 적절한 때에 출현해야 하므로 총알의 속성과 유사한 점이 있다. 적 캐릭터와 총알의 차이점은, 적 캐릭터의 경우 적절한 시각에 스스로 출현해야 하지만 총알은 플레이어의 발사 키 입력에 따라 나타난다는 점이다.

적절한 시각에 출현시키기 위해서는 적 캐릭터마다 출현하는 프레임 값을 지정해야 한다. 이때 출현 프레임 값은 오름차순으로 설정한다. 이렇게 하면 현재 프레임 값과 출현 프레임 값을 순차적으로 비교할 수 있다.

적 캐릭터의 출현 프레임 값은 한 스테이지의 최대 프레임 값보다 작아야 한다. 예를 들어 20초 동안만 적 캐릭터를 출현하게 하려면, 1초에 약 30번의 프레임이 발생하므로 20초 동안의 프레임 수는 대략  $20 \times 30 = 600$  이 된다. 그래서 적 캐릭터의 모든 출현 프레임을 600프레임 안에서 오름차순으로 설정하는 것이다.





[그림 4-11]

위에서 언급한 적 캐릭터의 속성 항목을 정의하면 다음과 같다.

- ① 생명
- ② 좌표 (x, y)
- ③ 이동 거리
- ④ 이전 프레임
- ⑤ 출현 프레임

적 캐릭터의 속성을 구조체로 정의하면 다음과 같다.

```
typedef struct _ENEMY
{
    int      nLife;
    int      nX, nY;
    int      nXMoveDist;    // Note: 이동 거리
    int      nOldFrame;
    int      nAppearFrame;  // Note: 출현 프레임
} ENEMY;
```

#### ▶ 적 캐릭터의 출현 프레임 값 설정

아래를 보면 변수 선언과 동시에 적 캐릭터의 출현 프레임 값이 설정되어 있다.

ENEMY 구조체 배열의 멤버값 중에서 nAppearFrame 값이 5에서 850까지 오름차순으로 설정된 것을 주의 깊게 보기 바란다.

```
ENEMY g_sEnemy[ENEMY_COUNT] = { { 0, 1, 0, 0, 0, 5 }, { 0, 1, 0, 0, 0, 5 }, { 0, 1, 0, 0, 0, 5 }, { 0, 1, 0, 0, 0, 5 }, { 0, 1, 0, 0, 0, 30 }, { 0, 1, 0, 0, 0, 35 }, { 0, 1, 0, 0, 0, 50 }, { 0, 1, 0, 0, 0, 100 }, { 0, 1, 0, 0, 0, 150 }, { 0, 1, 0, 0, 0, 150 }, { 0, 1, 0, 0, 0, 200 }, { 0, 1, 0, 0, 0, 300 }, { 0, 1, 0, 0, 0, 320 }, { 0, 1, 0, 0, 0, 350 }, { 0, 1, 0, 0, 0, 400 }, { 0, 1, 0, 0, 0, 420 }, { 0, 1, 0, 0, 0, 470 }, { 0, 1, 0, 0, 0, 500 }, { 0, 1, 0, 0, 0, 530 }, { 0, 1, 0, 0, 0, 560 }, { 0, 1, 0, 0, 0, 600 }, { 0, 1, 0, 0, 0, 620 }, { 0, 1, 0, 0, 0, 650 }, { 0, 1, 0, 0, 0, 680 }, { 0, 1, 0, 0, 0, 700 }, { 0, 1, 0, 0, 0, 720 }, { 0, 1, 0, 0, 0, 750 }, { 0, 1, 0, 0, 0, 780 }, { 0, 1, 0, 0, 0, 800 }, { 0, 1, 0, 0, 0, 850 } };
```


#### ▶ 적 캐릭터의 출력 설정

적 캐릭터를 출력하기 위해 출현 프레임 값을 모두 조사해야 하지만 출현 프레임 값이 현

재 프레임 값보다 크게 설정된 적 캐릭터가 검색된다면 그 이후에 나오는 적 캐릭터의 출현 프레임 값은 조사할 필요가 없다. 왜냐하면, 이미 적 캐릭터의 출현이 오름차순으로 되어 있기 때문이다.

이미 출현된 적 캐릭터는 출현 검색에서 제외시켜 불필요한 검색을 하지 않도록 해야 한다. 그러기 위해서는 출현 조사를 할 때 반복문의 시작 인덱스를 출현한 적 캐릭터의 마지막 인덱스 다음부터 시작하면 된다.

이것을 코드로 작성하면 다음과 같다.



```
01  #define ENEMY_COUNT 30
02  int g_nEnemyIndex = 0; // Note: 적 캐릭터의 출현 인덱스
03  int g_nFrameCount = 0; // Note: 프레임 카운트
04  .....생략.....
05  for( i = g_nEnemyIndex; i < ENEMY_COUNT ; i++ )
06  {
07      if( g_sEnemy[i].nLife == 0 )
08      {
09          if( g_sEnemy[i].nAppearFrame == g_nFrameCount )
10          {
11              g_sEnemy[i].nLife = 1; // Note: 출현 설정
12              g_nEnemyIndex++; // Note: 출현 인덱스
13          }else{
14              break;
15          }
16      }
17  }
```

○ 2행 : g\_nEnemyIndex는 적 캐릭터가 출현하는 시작 인덱스를 저장하는 변수이다.

○ 7~16행 : 초기에 출현 시작 인덱스(g\_nEnemyIndex)는 0부터 시작하며 현재의 프레임 값(g\_nFrameCount)과 적 캐릭터가 출현하는 프레임 값이 같은 경우에는 g\_nEnemyIndex 값을 1 증가시키고 그 외의 값이 되면 이 반복문을 빠져 나오게 된다. 다시 5행의 반복문을 실행하게 되면 i 변수에는 출현하지 않았던 인덱스가 설정되므로 이미 출현한 적 캐릭터에 대해서는 검색을 하지 않게 된다. 14행의 'break' 문은 적 캐릭터가 출현하는 프레임 값이 현재 프레임 값과 다르다면 5행의 반복문을 빠져 나오게 하는 역할을 한다.

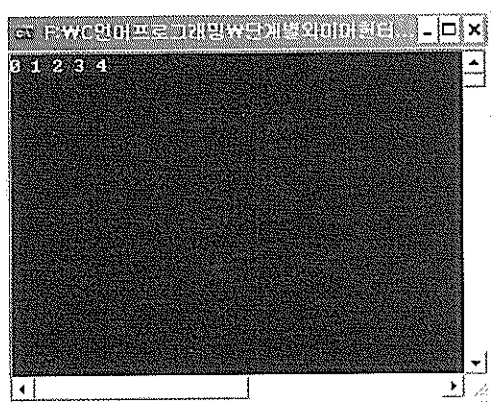


## 실습하기 : 4-4

다음의 배열은 출현 프레임이 저장된 배열이다.

```
int array[] = { 10, 40, 80, 120, 120, 300, 400 };
```

게임의 기본 구조 안에서 출현 프레임에 따라 [그림 4-12] 와 같이 프레임별로 array[ ] 배열의 인덱스를 출력하는 프로그램을 작성해 보자.



[그림 4-12] 출현 인덱스 출력



```
01  #include "stdafx.h"
02  #include <windows.h>
03  #include <time.h>
04  #include <conio.h>
05
06  int _tmain(int argc, _TCHAR* argv[])
07  {
08      int array[] = { 10, 40, 80, 120, 120, 300, 400 };
09      int index = 0, frame = 0, i;
10      clock_t OldTime, CurTime;
11      OldTime = clock();
12
13      while( 1 )
14      {
15          if( index == 7 )
16              break;
17          for( i = index ; i < 7 ; i++ )
18          {
19              if( array[i] == frame )
```

◉ NEXT

```

20         {
21             index++;
22         }else{
23             break;
24         }
25     }
26     system( "cls" );
27     for( i = 0 ; i < index ; i++ )
28     {
29         printf( "%d ", i );
30     }
31
32     while( 1 )
33     {
34         CurTime = clock();
35         if( CurTime - OldTime > 33 )
36         {
37             OldTime = CurTime;
38             break;
39         }
40     }
41     frame++;
42 }
43 return 0;
44 }

```

● 15 ~ 16행: 반복문 while( )을 빠져 나오게 하는 1순위 조건을 체크하는 부분이다.

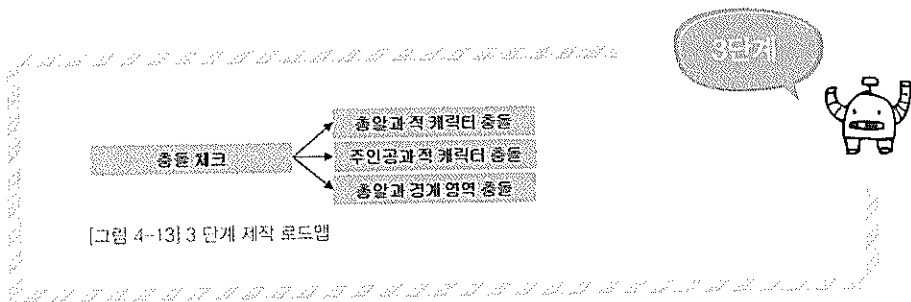
array[]에는 7개의 출현 프레임 값이 있으며 배열의 인덱스(index)는 0~6이 된다. 이 index의 값이 7이 되면 더 이상 반복문을 실행하면 안 되므로 16행의 break 문에 의해 13행의 while( ) 문을 빠져 나오게 된다.

● 17~25행 : 출현을 결정하는 부분이다.

17행의 반복문 초기화에서 i = index'는 출현을 조사하기 위한 시작 인덱스를 설정하는 부분이며 이미 출현한 인덱스를 조사하지 않도록 하는 가장 중요한 부분이다. 다음의 도표는 출현이 결정되고 난 후의 index의 변화를 나타내는 도표이다.

i 변수	출현 프레임	index++ 결과
0	10	1
1	40	2
2	80	3
3	120	4
4	120	5
5	300	6
6	400	7

위에서 index의 값이 2가 되었다는 것은 0, 1번째는 이미 출현된 상태라는 뜻이다. 다시 17행의 반복문을 실행할 때면 index의 값 2가 i 에 대입되기에 이미 출현된 0, 1번째 인덱스에 대한 출현 조사는 하지 않게 된다.

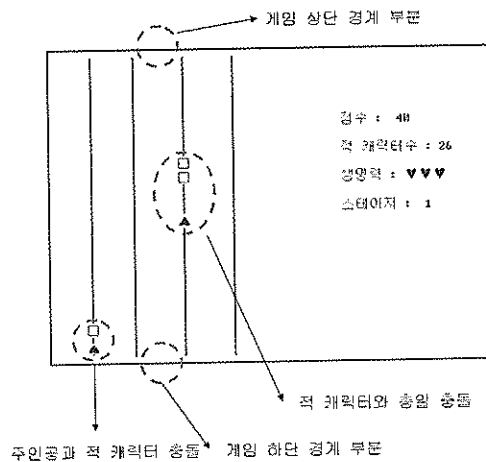


## 충돌

### ▶ 충돌하는 경우

충돌 대상을 살펴보면 총알, 적 캐릭터, 주인공, 게임 화면의 상단, 게임 화면의 하단이 있다. 이러한 충돌 대상들의 관계를 다음의 도표와 [그림 4-14] 를 통해 살펴보자.

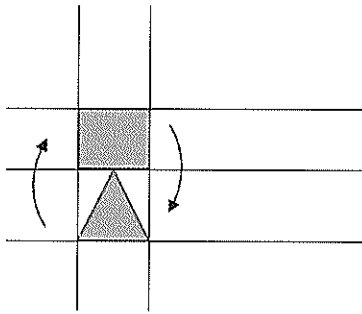
객체	대상
주인공	적 캐릭터
적 캐릭터	총알 게임 화면의 하단
주인공 총알	적 캐릭터, 게임 화면의 상단



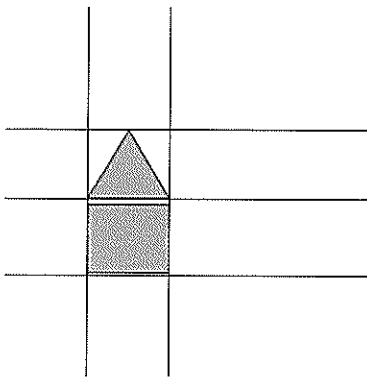
[그림 4-14] 충돌 포인트

충돌은 이동과 많은 관련이 있다. 왜냐하면, 이동은 좌표를 변경시키는 역할을 하며 좌표는 충돌 체크의 가장 중요한 역할을 하기 때문이다.

일반적인 적 캐릭터와 총알 충돌의 체크는 모든 좌표를 이동시켜 충돌 체크를 하는 방법을 사용하지만 와이어헌터와 같은 콘솔 게임에서는 좌표가 한 행씩 이동하므로 [그림 4-15], [그림 4-16] 과 같이 총알과 적 캐릭터가 동시에 이동하는 경우가 생긴다. 이것은 총알과 적 캐릭터가 충돌했음에도 불구하고 y 좌표가 같지 않으므로 충돌로 인정되지 않는다.



[그림 4-15] 적 캐릭터와 총알이 동시에 이동하는 경우



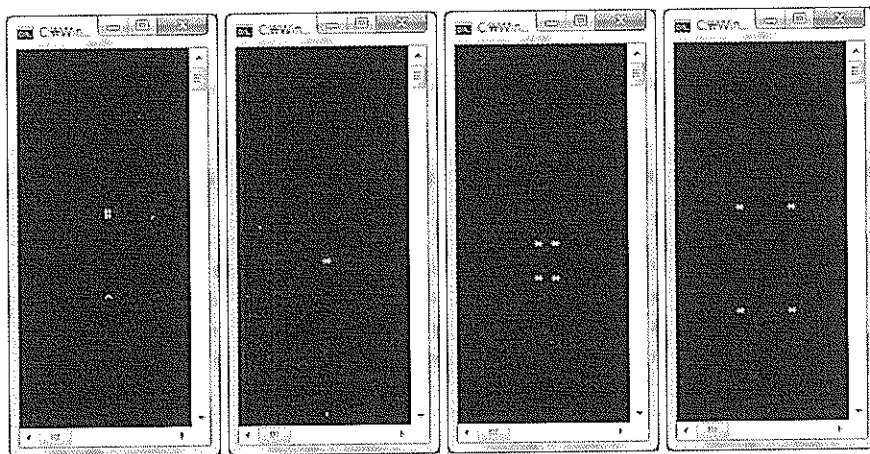
[그림 4-16] 동시 이동한 결과

이 부분을 해결하기 위한 몇 가지 방법이 있다. 적 캐릭터 또는 총알 중에서 어느 하나를 먼저 이동시키고 충돌 여부를 파악하는 방법과 [그림 4-15], [그림 4-16] 과 같이 x 좌표가 같고 y 좌표가 서로 엇갈린 경우를 충돌로 판단하는 방법이다.



## 실습하기 : 4-5

[그림 4-17] 과 같이 적 캐릭터와 총알을 강제적으로 충돌하게 하고 간단한 폭발 효과가 나타날 수 있도록 프로그래밍을 해 보자. 충돌을 체크하는 방법은 두 번째 방법인  $x$  좌표는 같고  $y$  좌표가 엇갈린 경우를 적용한다.



적과 총알 발사

충돌 후 폭발 1 단계

폭발 2 단계

폭발 3 단계

[그림 4-17] 폭발 효과 출력



```
01  #include "stdafx.h"
02  #include <windows.h>
03  #include <conio.h>
04  #include <time.h>
05
06  void gotoxy( int x, int y )
07  {
08      COORD CursorPosition = { x, y };
09      SetConsoleCursorPosition( GetStdHandle( STD_OUTPUT_HANDLE ), CursorPosition );
10  }
11
12  typedef struct _OBJECT
13  {
14      int nLife;
15      int nX, nY;
16      int nOldFrame;
17  } OBJECT;
18
19  typedef struct _EFFECT    // Note: 폭발 효과 속성
20  {
21      int nLife;
```

◉ NEXT

```

22         int nX, nY;
23         int nStepCount;
24         int nOldFrame;
25     } EFFECT;
26
27     OBJECT g_Enemy, g_Bullet;
28     EFFECT g_Effect;
29     int g_nFrameCount;
30
31     int _tmain(int argc, _TCHAR* argv[])
32     {
33         clock_t OldTime, CurTime;
34         g_Bullet.nLife = g_Enemy.nLife = 1;
35         g_Bullet.nX = g_Enemy.nX = 10;
36         g_Bullet.nOldFrame = 0;
37         g_Enemy.nOldFrame = 0;
38         g_Enemy.nY = 2;
39         g_Bullet.nY = 18;
40         OldTime = clock();
41
42         while( 1 )
43         {
44             if( g_Bullet.nLife )
45             {
46                 if( g_nFrameCount - g_Bullet.nOldFrame > 10 )
47                 {
48                     g_Bullet.nOldFrame = g_nFrameCount;
49                     g_Bullet.nY--;
50                 }
51             }
52
53             if( g_Enemy.nLife )
54             {
55                 if( g_nFrameCount - g_Enemy.nOldFrame > 5 )
56                 {
57                     g_Enemy.nOldFrame = g_nFrameCount;
58                     g_Enemy.nY++;
59                 }
60             }
61
62             if( g_Enemy.nLife && g_Bullet.nLife )
63             {
64                 if( g_Effect.nLife == 0 )
65                 {
66                     if( g_Enemy.nX == g_Bullet.nX && g_Enemy.nY >= g_Bullet.nY )
67                     {
68                         g_Enemy.nLife = 0;
69                         g_Bullet.nLife = 0;
70                         g_Effect.nLife = 1;
71                         g_Effect.nOldFrame = g_nFrameCount;
72                         g_Effect.nX = g_Enemy.nX;
73                         g_Effect.nY = g_Enemy.nY;
74                     }

```



```

75         }
76     }
77
78     if( g_Effect.nLife )
79     {
80         if( g_nFrameCount - g_Effect.nOldFrame > 7 )
81         {
82             // Note: 폭발 효과
83             g_Effect.nStepCount++;
84             g_Effect.nOldFrame = g_nFrameCount;
85             if( g_Effect.nStepCount > 3 )
86             {
87                 g_Effect.nLife = 0;
88                 g_Energy.nLife = 1;
89                 g_Bullet.nLife = 1;
90
91                 g_Energy.nX = g_Bullet.nX = 10;
92                 g_Energy.nY = 2;
93                 g_Bullet.nY = 18;
94                 g_Effect.nStepCount = 0;
95             }
96         }
97     }
98
99     system( "cls" );
100    // Note: 출력
101    if( g_Energy.nLife )
102    {
103        gotoxy( g_Energy.nX, g_Energy.nY );
104        printf( "#" );
105    }
106
107    if( g_Bullet.nLife )
108    {
109        gotoxy( g_Bullet.nX, g_Bullet.nY );
110        printf( "^" );
111    }
112
113    if( g_Effect.nLife )
114    {
115        gotoxy( g_Effect.nX - g_Effect.nStepCount, g_Effect.nY - g_Effect.nStepCount );
116        printf( "*" );
117        gotoxy( g_Effect.nX + g_Effect.nStepCount, g_Effect.nY - g_Effect.nStepCount );
118        printf( "*" );
119        gotoxy( g_Effect.nX - g_Effect.nStepCount, g_Effect.nY + g_Effect.nStepCount );
120        printf( "*" );
121        gotoxy( g_Effect.nX + g_Effect.nStepCount, g_Effect.nY + g_Effect.nStepCount );
122        printf( "*" );
123    }
124
125    while( 1 )
126    {
127        CurTime = clock();

```

◎ NEXT

```

128         if( CurTime - OldTime > 33 )
129         {
130             OldTime = CurTime;
131             break;
132         }
133     }
134     g_nFrameCount++;
135 }
136 return 0;
137 }
138

```

○ 19~25행 : 폭발 효과를 출력하기 위한 속성을 정의하는 부분이다.

폭발 효과의 속성 중에 nLife라는 변수가 있다. 이 변수는 효과를 소멸시킬 것인지 아니면 계속 애니메이션을 할 것인지를 결정하는 변수이다. 폭발 효과는 nLife를 1로 설정하면 생성되고 nLife를 0으로 설정하면 소멸한다. 폭발 효과는 충돌했을 때 '\*'가 사방으로 퍼져 나가도록 3단계로 애니메이션 한다.



위의 각 단계를 일정 시간 간격으로 퍼져 나가게 하기 위해서는 폭발 효과 속성에 각 단계를 저장하는 변수와 이전 프레임들을 저장하는 변수가 있어야 한다.

nStepCount와 nOldFrame은 이러한 용도로 사용하는 변수이다.

위의 같이 폭발 효과 애니메이션이 3단계라면 nStepCount의 값은 0~2가 되며 애니메이션은 nStepCount가 3이 되는 순간에 소멸 상태가 된다.

○ 44~51행 : 총알 이동에 대한 부분이다.

46행은 프레임 간격이 10 이상이면 1씩 이동하게 하는 조건이다.

○ 53~60행 : 적 캐릭터가 이동하는 부분이다.

프레임 간격이 5 이상이면 1씩 이동한다.

○ 62~76행 : 폭발 효과를 초기화하는 부분이다.

폭발 효과를 초기화하는 조건은 적 캐릭터와 총알이 살아 있고 충돌되었을 때이다.

68~73행의 내용은 위의 두 가지 조건이 모두 충족되어야 실행되는 부분이다.

중요한 것은 위의 조건이 충족되면 현재 적 캐릭터와 총알은 소멸되고 폭발 효과는 초기화되어 활성화 상태가 된다는 점이다. 그리고 폭발 효과의 초기 좌표는 충돌한 바로 그 위치가 시작 위치이므로 적 캐릭터 또는 총알의 좌표 중 한 가지를 설정하면 된다.

○ 78~97행 : 폭발 효과를 진행하게 한다.

80행의 조건이 충족된다면 속성에서 미리 설정한 3단계를 진행해야 한다. 물론 직접 좌표 이동을 해줄 수 있으나 폭발 효과가 같은 간격으로 퍼져 나가므로 간단히 nStepCount 변수의 값을 상하좌우로 증감해 주어 퍼져 나가는 효과를 만든다.

85행은 폭발 효과를 소멸시키는 조건이다.

그래서 이 조건을 만족한다면 폭발 효과의 속성 중 nLife 값을 0으로 만들고 총알과 적 캐릭터를 초기화한다.


94행에서 nStepCount를 0으로 설정해 주는 것은 상당히 중요하다.

이 값을 폭발 효과가 다시 생성될 때 1~3단계의 애니메이션을 하기 위한 초기화 값이 되기 때문이다.

○ 113~123행 : 폭발 효과를 출력하는 부분이다.

이 행의 소스들은 단계별로 폭발 효과를 출력하며 모두 1씩 이동하므로 nStepCount 값을 이동 거리로 사용한다. 1단계에서는 nStepCount가 모두 00이므로 4개의 '\*'가 같은 좌표에 출력된다. 그래서 하나의 '\*'가 출력된 것으로 보인다. 2단계와 3단계에서는 nStepCount가 증가하므로 x, y 좌표 값도 증감되어 사방으로 퍼져 나가는 폭발 효과가 출력된다.

다음의 소스는 와이어헌터 게임 소스 중에서 이동된 적 캐릭터와 총알의 충돌 체크 부분이다.



```

01  #define BULLET_COUNT    4
02  ..... 생략 .....
03  for( i = 0 ; i < g_nEnemyIndex ; i++ )
04  {
05      if( g_sEnemy[i].nLife == 1 )
06      {
07          for( j = 0 ; j < BULLET_COUNT ; j++ )
08          {
09              if( g_sBullet[j].nLife == 1 )
10              {
11                  if( (g_sBullet[j].nX == g_sEnemy[i].nX) &&
12                      g_sEnemy[i].nY >= g_sBullet[j].nY )
13                  {
14                      sBullet[j].nLife = 0;
15                      g_sEnemy[i].nLife = 0;
16                      g_nDeadEnemy++; // Note: 적 캐릭터 죽은 개수
17                      g_nGoal--;      // Note: 목표 개수
18                      g_nGrade += 10; // Note: 득점
19
20                      if( g_sPlay.nLifePower > 0 && g_nGoal == 0 )
21                      {
22                          g_GameState = SUCCESS;
23                          return ;
24                      }
25                      break;
26                  }
27              }
28          }
29      }
30  }
  
```

○ 5행, 9행 : 적 캐릭터와 총알의 충돌을 조사하기 위한 조건으로, 생존하는 것만 충돌 대상이 된다.

○ 20행 : 미션 성공이 되는 조건이다.

미션 성공은 주인공이 스테이지마다 할당된 목표 적군을 모두 소멸한 후 살아 있는 경우이다. g\_nGoal은 소멸할 적군의 수이며 주인공이 적 캐릭터를 소멸할 때마다 1씩 줄어든다. 0 이 되면 적군이 전부 소멸된 상태가 된다. 이와 같은 상태가 되면 게임 상태 변수인 g\_GameState의 값을 SUCCESS로 설정하여 다음 스테이지를 진행한다.

다음의 소스는 주인공과 적 캐릭터 간의 충돌 체크 소스이다.



```
01  for( i = 0 ; i < g_nEnemyIndex ; i++ )
02  {
03      if( g_sEnemy[i].nLife == 1 )
04      {
05          // Note: 좌표가 같으면
06          if( g_sPlay.nX == g_sEnemy[i].nX && g_sPlay.nY == g_sEnemy[i].nY )
07          {
08              g_sEnemy[i].nLife = 0;
09              g_sPlay.nLifePower--;
10              g_nGoal--;
11              g_nDeadEnemy++;
12              if( g_sPlay.nLifePower == 0 && g_nGoal > 0 )
13              {
14                  g_GameState = FAILED;
15                  return ;
16              }
17              if( g_sPlay.nLifePower > 0 && g_nGoal == 0 )
18              {
19                  g_GameState = SUCCESS;
20                  return ;
21              }
22              break; //
23          }
24      }
25  }
```

○ 6행, 22행 : 주인공과 적 캐릭터 간의 충돌 조건과 충돌 체크가 종료되는 부분이다.

22행의 break 문은 충돌 체크를 종료하게 하는 역할을 한다.

○ 12행 : 미션이 실패하는 조건이다.

미션이 실패하는 경우는 적 캐릭터의 소멸 수가 남아 있고 주인공이 소멸된 경우이다.

주인공이 소멸되는 경우는 주인공의 생명력 변수인 nLifePower 값이 0 이 되는 경우이다.


주인공이 소멸되면 더 이상 뒤에 나오는 코드를 실행할 필요가 없으므로 15행과 같이 return 문을 사용하여 현재 Update( ) 함수를 종료한다.

총알이 경계 화면의 상단과 충돌하는 경우의 소스는 다음과 같다.



```
01  for( i = 0 ; i < BULLET_COUNT ; i++ )
02  {
03      if( g_sBullet[i].nLife == 1 && g_sBullet[i].nY == 0 )
04      {
05          g_sBullet[i].nLife = 0;
06      }
07  }
```

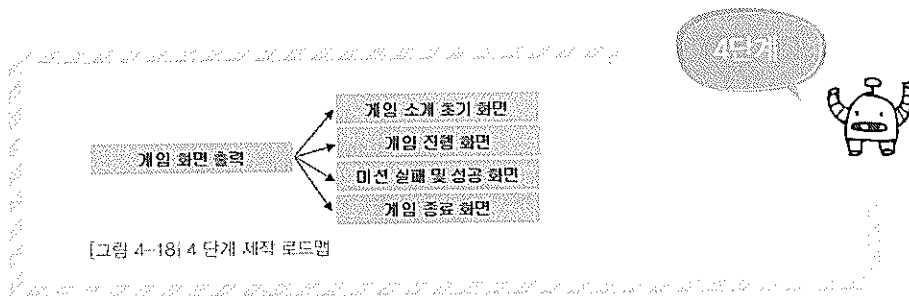
적 캐릭터가 경계 화면의 하단과 충돌하는 경우의 소스는 다음과 같다.



```

01     for( i = 0 ; i < g_nEnemyIndex ; i++ )
02     {
03         if( g_sEnemy[i].nLife == 1 && g_sEnemy[i].nY == END_LINE )
04         {
05             g_sEnemy[i].nLife = 0;
06             g_sPlay.nLifePower--;
07             g_nDeadEnemy++; // Note: 적 캐릭터 죽은 수
08             if( g_sPlay.nLifePower == 0 && g_nGoal > 0 )
09             {
10                 g_GameState = FAILED;
11                 return ;
12             }
13             if( g_sPlay.nLifePower > 0 && g_nGoal == 0 )
14             {
15                 g_GameState = SUCCESS;
16                 return ;
17             }
18         }
19     }

```



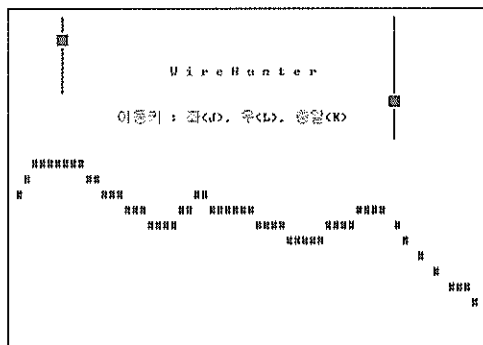
## 4. 게임 소개 화면 출력

게임 소개 화면은 게임을 실행하면 처음에 출력되는 화면을 말한다.

[그림 4-19] 와 같이 출력하기 위해 먼저 gotoxy( ) 함수로 위치를 설정하고 printf( )를 이용하여 출력한다.

앞으로 제작하는 모든 게임의 출력은 이 두 함수를 이용한다.

참고적으로 gotoxy( ) 함수는 Win32 API로 제작한 함수이므로 'windows.h'를 선언해야 한다.



[그림 4-19] 게임 소개 화면



```

01 void IntroScreen()
02 {
03     gotoxy( 0, 0);
04     printf( " \n");
05     printf( " | \n");
06     printf( " | ■ \n");
07     printf( " | \n");
08     printf( " | 줄을 타고 침투하는 적군을 섬멸하라 | \n");
09     printf( " | \n");
10     printf( " | ■ \n");
11     printf( " | 이동키 : 좌(L), 우(R), 총알(K) | \n");
12     printf( " | \n");
13     printf( " | \n");
14     printf( " | ##### \n");
15     printf( " | # # \n");
16     printf( " | # ### # \n");
17     printf( " | # # # # # # # # \n");
18     printf( " | # # # # # # # \n");
19     printf( " | # # # # # \n");
20     printf( " | # \n");
21     printf( " | # \n");
22     printf( " | # \n");
23     printf( " | # \n");
24     printf( " | \n");
25     printf( " | \n");
26     printf( " | \n");
27 }

```

[소스 4-1]

## Ⅲ 게임 진행 제어

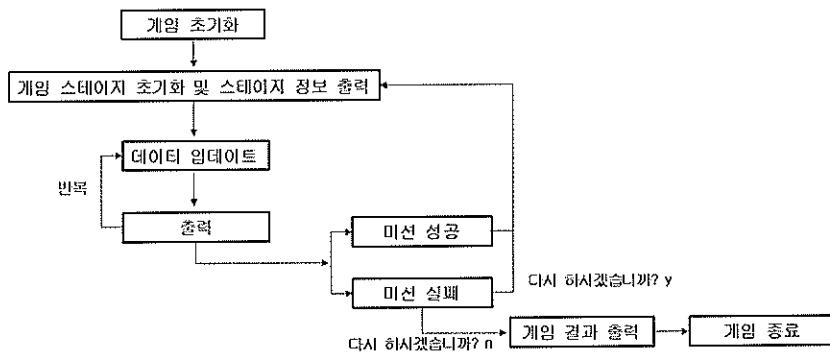
게임 상태는 기본적으로 초기, 진행, 미션 성공, 미션 실패, 결과, 종료라는 여섯 가지 상태로 나눌 수 있다. 상태별로 처리하는 방법은 변수에 고유값을 할당하고 고유값이 변경

되면 이것에 따라 처리하도록 하는 것이다. 각 고유값은 열거형 데이터로 다음과 같이 정의하면 고유값의 의미를 쉽게 파악할 수 있으므로 프로그래밍하기가 쉬워진다.

```
enum GAME_STATE { INIT, RUNNING, SUCCESS, FAILED, RESULT, END };
GAME_STATE g_GameState = INIT;
```

초기 상태	진행 상태	미션 성공	미션 실패	결과 출력	종료
INIT	RUNNING	SUCCESS	FAILED	RESULT	END

위의 고유값에 따른 처리 흐름을 나타내면 [그림 4-20] 과 같다.



[그림 4-20] 와이어한터 게임 흐름도



실습문제 4-5

상태를 나타내는 고유값에 따라 [그림 4-20] 과 같이 처리되도록 프로그래밍해 보자. 각 흐름의 상태가 다음과 같이 출력될 수 있도록 하며 s 키는 강제적으로 미션 성공 상태로, f 키는 미션 실패로 설정되게 프로그래밍해 보자.





```
01  #include "stdafx.h"
02  #include <conio.h>
03  #include <stdlib.h>
04
05  typedef enum _GAME_STATE { INIT, RUNNING, SUCCESS, FAILED, RESULT, END } GAME_STATE;
06  GAME_STATE g_GameState = INIT;
07
08  int _tmain(int argc, _TCHAR* argv[])
09  {
10      int nKey, nYN;
11      int nStage = 0; // 스테이지 변수
12      printf( "게임 초기화 실행\n" );
13      _getch();
14
15      while( 1 )
16      {
17          if( g_GameState == END )
18              break;
19          if( _kbhit() )
20          {
21              nKey = _getch();
22              if( nKey == 's' )
23                  g_GameState = SUCCESS;
24              if( nKey == 'f' )
25                  g_GameState = FAILED;
26          }
27          // Note: 데이터 업데이트
28          if( g_GameState == RUNNING )
29              printf( "데이터 업데이트\n" );
30
31          // Note: 출력
32          system( "cls" );
33          switch( g_GameState )
34          {
35              case INIT :
36                  if( nStage == 0 )
37                  {
38                      printf( "게임 소개 화면 출력\n" );
39                      nStage = 1;
40                      _getch();
41                  }
42                  printf( "게임 %d 스테이지 화면 출력 및 초기화\n", nStage );
43                  _getch();
44                  g_GameState = RUNNING;
45                  break;
46              case RUNNING :
47                  printf( "게임 진행 화면 출력\n" );
48                  break;
49              case SUCCESS :
50                  printf( "미션 성공\n" );
51                  nStage++;
```



```

52         _getch();
53         g_GameState = INIT;
54         break;
55     case FAILED :
56         printf( "미션 실패\n" );
57         printf( "계속 하시겠습니까? (y/n) " );
58         while( 1 )
59         {
60             nYN = _getch();
61             fflush( stdin ); // 키보드 버퍼를 모두 비움
62             if( nYN == 'y' || nYN == 'Y' )
63             {
64                 g_GameState = INIT;
65                 break;
66             }
67             if( nYN == 'n' || nYN == 'N' )
68             {
69                 g_GameState = RESULT;
70                 break;
71             }
72         }
73     case RESULT :
74         printf( "게임 결과 출력\n" );
75         _getch();
76         g_GameState = END;
77         break;
78     }
79 }
80 printf( "게임 종료\n");
81 return 0;
82 }
83

```

- 13행, 40행, 43행, 52행, 76행 : 게임의 진행 상태를 확인하기 위해 입력이 있을 때까지 정지 상태가 되도록 `_getch()`를 사용한다.
- 17행 : 게임의 종료 상태는 15행의 반복문을 빠져 나오게 하는 최상위 순위가 된다.
- 22행, 24행 : s 키가 입력되면 게임은 강제적으로 미션 성공 상태가 되며 f 키가 입력되면 미션 실패 상태가 된다.
- 36~41행 : 초기화하는 부분이다.

여기에서는 두 가지 초기화가 존재한다. 첫 번째는 게임 전체를 초기화하는 부분이며 두 번째는 각 스테이지를 초기화하는 부분이다. 첫 번째 초기화는 게임이 실행될 때 1회만 실행되는 부분을 말한다. 예를 들어 게임 소개 화면 출력, 시운드 초기화, 맵 파일 로딩, 스테이지 파일 로딩 등이 될 수 있다. 두 번째 초기화는 해당 스테이지에만 적용되는데 적 캐릭터의 초기화, 화면 초기화, 점수 초기화 등이 그것이다.

이와 같은 부분은 `nStage` 변수 값에 따라 초기화를 구분하여 실행할 수 있는데 0이면 첫 번째 초기화가 되며 1 이상의 값이면 각 스테이지별로 초기화가 된다.

`nStage`는 전체 프로그램에서 오로지 한 번만 0으로 설정되므로, 매번 `g_GameState`의 값이 `INIT`가 되어 35~45행을 반복한다 할지라도 이 두 가지는 구분하게 된다.

