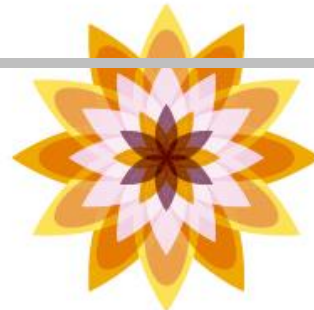


Chapter 02

벽돌 깨기 게임 Part1



1. 개요

- 우리가 만들 게임의 순서는 다음과 같다.



■ 공 그리기

- openFrameworks의 가장 기본적인 기능인 '도형(원) 그리기'를 다룬다.
- 컴퓨터 프로그램이라는 것은 어차피 규모가 크던 작던 상관없이 사용자의 입력에 반응해서 정해진 출력을 해주는 일을 한다.
- openFrameworks를 이용한 프로그램의 입력은 대부분 키보드 또는 마우스이며, 출력은 화면에 그림을 그리는 일이 대부분이다.
- 그 중에서도 원을 그리는 단계는 출력 부분의 가장 기본적인 요소이다.
- 본 강좌에서 공을 그리기 위해서 '원 그리기'를 이용한다.

■ 공 수평으로 움직이기

- 가만히 정지한 공은 심심하다.
- 그 공에 움직임을 부여하자.
- 가장 쉬운 좌·우로의 움직임이다.
- 이제 공이 좌·우로 움직이도록 해보자.

■ 공 좌우벽과 충돌 체크

- 공이 좌·우로 움직이다 보면 좌·우 경계를 벗어나서 사라져 버릴 수도 있다.
- 공이 움직이다가 좌·우 경계를 만나면 반대 방향으로 튕겨 나가도록 해보자.
- 충돌 여부를 감지 하기 위해서 '조건문'을 사용한다.

■ 공 2차원적으로 움직이기

- 1차원적으로 좌·우로만 움직이는 공은 여전히 심심하다.
- 2차원 화면 상에서 자유롭게 움직이는 공을 만들어보자.
- 1차원적인 좌-우로의 움직임과 그리 많이 다르지 않다.
- 다만 차원이 1차원에서 2차원으로 바뀌었을 뿐.

■ 라켓 움직여서 공 반사하기

- 이제 공 움직임은 완성되었다.
- 이 공을 튕겨낼 라켓을 만들고 마우스를 이용해서 움직여보자.
- 마우스를 따라 라켓이 좌·우로 움직이게 한다.
- 이제 마우스 사용이 가능해진다.
- 움직이는 공을 라켓으로 튕겨보자.
- 라켓과 공의 충돌 여부를 감지하기 위해서 '조건문'을 사용한다.
- 여기까지 하면 벽돌 깨기 게임은 거의 완성이다.

■ 1차원 배열을 이용한 벽돌 1줄 배치

- 이제 화면에 벽돌을 그린 후, 마우스로 라켓을 움직이고, 공과 벽돌의 충돌을 체크해보자.
- 벽돌을 배치하기 위해서 배열(array)을 사용한다.
- 배열을 효과적으로 이용하기 위해서는 '반복문'이 필요하다.

■ 2차원 배열을 이용한 벽돌 2줄 이상 배치(게임 완성)

- 이제 마지막이다.
- 2차원 배열을 이용하여 벽돌을 2줄 이상 배치하자.
- 2중 반복문을 사용한다.

■ 게임 업그레이드

- 키보드를 이용한 총알 발사, 게임 맵 구성 등 게임 업그레이드사항을 다룬다.

2. 공 그리기

- 게임 제작의 첫 단계에서 가장 기본적인 기능인 '도형 그리기'를 다룬다.
- 프로그램은 대부분 그 규모가 크던 작던 사용자의 입력에 반응해서 원하는 출력을 해주는 일을 한다.
- 지금 우리가 도전할 '원 그리기'를 이용해서 공을 그리는 단계는 출력 부분의 가장 기본적 요소이다.

2. 공 그리기

■ 원 그리기

- 이제 원을 그리는 프로그램을 시작한다.
- 새로운 프로젝트를 생성한다.

create / update

Project name:
drawCircle

Project path:
C:\openFrameworks\apps\myApps

Addons:
Addons...

Platforms:
Windows (Visual Studio 2017)

Generate

Success!

Your can now find your project in

[C:\openFrameworks\apps\myApps\drawCircle](#)

```
[notice ] -----  
[notice ] setting OF path to: C:\openFrameworks  
[notice ] from -o option  
[notice ] target platform is: vs  
[notice ] project path is: C:\openFrameworks\apps\myApps\drawCircle  
[notice ] setting up new project C:\openFrameworks\apps\myApps\drawCircle  
[notice ] saving addons.make
```

Open in IDE

Close

솔루션 작업 검토

프로젝트 대상 변경

다음 프로젝트에서는 이전 버전의 Visual C++ 플랫폼 도구 집합을 사용합니다. 최신 Microsoft 도구 집합을 대상으로 하도록 프로젝트를 업그레이드할 수 있습니다. 컴퓨터에 설치된 Windows SDK 버전 중에서 대상 버전을 선택할 수도 있습니다.

Windows SDK 버전: 10.0(최근 설치된 버전)

플랫폼 도구 집합: v142(으)로 업그레이드

☒ ..\drawCircle\drawCircle.vcxproj

확인

취소

2. 공 그리기

■ 원 그리기


- 아래의 내용을 ofApp.cpp 파일에 입력한 후 실행하자.

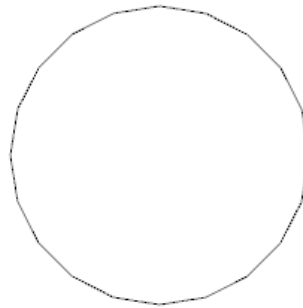
```
1  #include "ofApp.h"
2
3  //-----
4  void ofApp::setup(){
5      ofSetWindowTitle("Ball Drawing Program");
6      ofSetWindowShape(1024, 768);
7      ofSetFrameRate(60);
8      ofBackground(ofColor::white);
9      ofSetColor(ofColor::black);
10     ofSetLineWidth(1.0);
11     ofNoFill();
12 }
13
14 //-----
15 void ofApp::update(){
16
17 }
18
19 //-----
20 void ofApp::draw(){
21     ofCircle(200, 200, 100);
22
23 }
24
```

2. 공 그리기

■ 원 그리기

- [빌드] → [솔루션 빌드]하고 [디버그] → [디버그하지 않고 시작]을 클릭해서 실행하자.
- 아래는 콘솔 실행 화면이다.

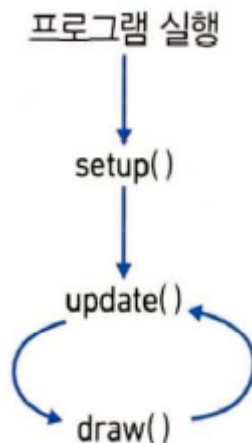
 Ball Drawing Program



2. 공 그리기

■ 원 그리기

- openFrameworks를 이용한 프로그래밍은 위와 같이 `setup()`, `update()`, `draw()` '함수'를 사용하게 된다.
 - `Setup()`: 프로그램을 실행시키면 가장 먼저 단 한번만 실행되는 함수이다. 프로그램의 초기화 작업을 수행하는 역할을 한다.
 - `Update()`: `setup()` 함수 이후에 내부적인 변수 값들의 반복적인 수정을 위한 함수이다.
 - `draw()`: `update()` 함수 이후 수행되는 함수이다. 화면에 그림을 그리는 부분이다. `update()` 함수와 `draw()` 함수는 연속해서 반복적으로 수행된다. 이 두 함수가 반복적으로 수행된다는 말이 이해되지 않을 수도 있는데, 자세한 설명은 다음 절에서 예를 들어서 설명한다.



2. 공 그리기

■ openFrameworks: 기본 도형 그리기

- 앞에서는 '원'을 그려보았다.
- 이제 openFrameworks에서 제공되는 다른 도형들을 그려보자.
- 직선
 - `ofLine(start_x, start_y, end_x, end_y);`
- 직사각형
 - `ofRect(left_top_x, left_top_y, width, height);`
- 삼각형
 - `ofTriangle(x1, y1, x2, y2, x3, y3);`
- 원
 - `ofCircle(center_x, center_y, radius);`
- 타원
 - `ofEllipse(center_x, center_y, width, height);`

2. 공 그리기

■ openFrameworks: 기본 도형 그리기

- 위의 내용을 응용해서 openFrameworks를 이용해서 다양한 도형을 그려보자
- 도형의 색깔을 바꾸고 싶다면 아래의 함수를 적절히 응용하면 된다.
 - `offsetColor(ofColor::black);`
 - `ofFill();`

2. 공 그리기

■ openFrarneworks: 기본 도형 그리기

1. 공을 2개 이상 다른 위치에 다른 크기로 그려보자.
2. 위에서 소개한 다양한 종류의 도형을 다양한 위치에 그려보자.
3. 도형의 색깔을 바꿔서 그려보자.

2. 공 그리기

■ openFrameworks: 기본 도형 그리기

- 공을 2개 이상 다른 위치에 다른 크기로 그려보자.

```
1  #include "ofApp.h"
2
3  //-----
4  void ofApp::setup(){
5      ofSetWindowTitle("Ball Drawing Program");
6      ofSetWindowShape(1024, 768);
7      ofSetFrameRate(60);
8      ofBackground(ofColor::white);
9      ofSetColor(ofColor::black);
10     ofSetLineWidth(1.0);
11     ofNoFill();
12 }
13
14 //-----
15 void ofApp::update(){
16
17 }
18
19 //-----
20 void ofApp::draw(){
21     ofCircle(200, 200, 100);
22     ofCircle(400, 500, 200);
23
24 }
```

2. 공 그리기

■ openFrameworks: 기본 도형 그리기

- 위에서 소개한 다양한 종류의 도형을 다양한 위치에 그려보자.

```
1  #include " ofApp.h"
2
3  //-----
4  void ofApp::setup(){
5      ofSetWindowTitle("Ball Drawing Program");
6      ofSetWindowShape(1024, 768);
7      ofSetFrameRate(60);
8      ofBackground(ofColor::white);
9      ofSetColor(ofColor::black);
10     ofSetLineWidth(1.0);
11     ofNoFill();
12 }
13
14 //-----
15 void ofApp::update(){
16 }
17
18
19 //-----
20 void ofApp::draw(){
21     ofCircle(100, 100, 100);
22     ofRect(100, 250, 100, 50);
23     ofTriangle(100, 700, 200, 700, 150, 600);
24     ofEllipse(500, 200, 400, 200);
25
26 }
```


2. 공 그리기

■ openFrameworks: 기본 도형 그리기

- 도형의 색깔을 바꿔서 그려보자.

```
1  #include "ofApp.h"
2
3  //-----
4  void ofApp::setup(){
5      ofSetWindowTitle("Ball Drawing Program");
6      ofSetWindowShape(1024, 768);
7      ofSetFrameRate(60);
8      ofBackground(ofColor::white);
9      ofSetColor(ofColor::paleVioletRed);
10     ofSetLineWidth(1.0);
11 }
12
13 //-----
14 void ofApp::update(){
15
16 }
17
18 //-----
19 void ofApp::draw(){
20     ofCircle(100, 100, 100);
21     ofRect(100, 250, 100, 50);
22     ofTriangle(100, 700, 200, 700, 150, 600);
23     ofEllipse(500, 200, 400, 200);
24
25 }
```

3. 공 수평 움직임(변수)

■ 공의 움직임

- 가만히 정지한 공은 너무 심심하다.
- 공에 움직임을 주자 아주 단순하게 오른쪽으로 쪽~ 움직여보자.
- 이 예제에서 중요한 부분은 '변수(variable)'라는 개념이다.
- 공의 위치가 변하기 때문에 변하는 위치 값을 저장할 변수가 필요하다.
- 공의 위치를 변수로 선언하고, 매 프레임마다 `update()` 함수 내부에서 변수 값을 1씩 더해주고, `draw()` 함수는 변수 값의 위치에 공을 다시 그렸을 뿐인데, 공이 움직이는 효과가 생긴다.
- 이 프로그램은 `ofSetFrameRate(60);` 함수에 의해서 초당 60 프레임씩 매 프레임마다 이전 프레임은 지워지고 매 프레임이 새로 그려지는데, 눈의 착시현상에 의해서 공이 움직이는 것처럼 보이는 것이다.

3. 공 수평 움직임(변수)

■ 공의 움직임

- 이 코드를 보면서 이 점을 다시 한번 생각하자.
- openFrameworks에서는 setup() 함수 이후에 update(), draw() 함수가 계속적으로 반복 실행된다

```
1  #include " ofApp.h"
2
3  int xPos;
4
5  //-----
6  void ofApp::setup(){
7      ofSetWindowTitle("Ball Drawing Program");
8      ofSetWindowShape(1024, 768);
9      ofSetFrameRate(60);
10     ofBackground(ofColor::white);
11     ofSetColor(ofColor::black);
12     ofSetLineWidth(1.0);
13     ofSetColor(ofColor::paleVioletRed);
14     ofFill();
15
16     xPos = 0;
17 }
18
19 //-----
20 void ofApp::update(){
21     xPos++;
22 }
23
24 //-----
25 void ofApp::draw(){
26     ofCircle(xPos, 60, 30);
27 }
```

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 변수의 개념

- 컴퓨터 프로그램에서 어떤 데이터를 '저장'하려면 '변수(variables)'를 사용해야 한다.
- 변수는 어떤 값을 저장하고 있는 주 기억 장치(main memory)의 주소를 대신한다.
- 변수는 사용하기 전에 '어떤 이름의 변수를 어떤 자료형으로 사용할 것인지'를 명시해야 한다.
- 이를 변수(variable)를 '선언(declare)'한다고 한다.
- 아래에 정수형 (integer) 변수를 선언하는 예를 보인다.
 - `int x;`
 - `int y;`
 - `y = 10;`
 - `int z = 10;`

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

- 변수(variable)는 3가지 속성이 있다.
 - 이름(name): 메인 메모리의 주소를 대신하여 직관적으로 기억하기 쉬운 형태로 표현한 것을 말한다.
 - 자료형(data type, 데이터 타입): 변수가 가질 수 있는 값의 종류를 의미한다.
 - 값(value): 변수가 실제로 가지는 값이다.
- 예를 들어, 아래와 같은 문장은, '정수형 변수로 이름이 x인 변수를 만들고, 이 변수의 값을 10으로 초기화한다는 의미이다.
 - `int x = 10;`
- 변수가 담을 수 있는 "값의 범위"라는 것은 무슨 의미일까?
- 일반적으로 문자형 변수(char)는 8비트를 사용해서 -128 ~ +127까지 256개의 숫자를 표현한다.
- 그러나 아래의 예에서는 이 그릇(문자형 변수 a)에 +128을 담으려고 한다.
- 실제로 문자형 변수로는 +128을 표현할 수 없다.
- 그렇기 때문에 부정확한 값이 문자에 대입되는 것이다.
- 이를 overflow(넘침)라고 한다.
- 숫자 +128을 대입했지만 실제로는 -128이 대입되었다는 것을 확인할 수 있다.

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

```
#include <stdio.h>

int main(){
    char a;

    a = 128;

    printf("variable a is ... %d\n", a);

}
```

```
variable a is ... -128
```

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

- 실제로 내가 사용하는 컴퓨터에서는 각 자료형의 바이트 수가 어떻게 되는지 아래의 프로그램으로 확인해보자.
- 각자의 하드웨어나 운영체제의 환경에 따라서 자료형의 크기는 조금씩 달라질 수 있다.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("char의 크기 : %d\n", sizeof(char));
5      printf("short의 크기 : %d\n", sizeof(short));
6      printf("int의 크기 : %d\n", sizeof(int));
7      printf("long int의 크기 : %d\n", sizeof(long int));
8      printf("float의 크기 : %d\n", sizeof(float));
9      printf("double의 크기 : %d\n", sizeof(double));
10     printf("long double의 크기 : %d\n", sizeof(long double));
11
12     return 0;
13 }
```

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

- sizeof()라는 형태로 사용할 수 있어서 함수라고 생각하기 쉽지만, sizeof는 함수가 아니라 연산자(operator)이다.
- 이를 아래에서 확인하자.

```
1  #include <stdio.h>
2
3  int main() {
4      int a = 1;
5
6      printf(" %d\n", sizeof(a));
7      printf(" %d\n", sizeof a);
8      printf(" %d\n", sizeof(int));
9      //printf(" %d\n", sizeof int);
10     return 0;
11 }
```


3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

- 모든 기본 자료형에 'unsigned'라는 키워드를 붙여서 사용할 수 있다.
- Unsigned는 '부호 없는'이라는 뜻으로 0과 양수(+) 값만 사용하겠다는 의미이다.
- 따라서 음수를 표현하지 못하는 대신에 더 많은 양수값을 표현할 수 있다는 장점이 있다.
- printf() 함수를 사용할 때, unsigned 변수를 위한 포맷 문자는 %u이다.
- unsigned가 표현할 수 있는 값의 범위가 더 넓어서, %d는 절대값이 큰 경우에는 출력에 오류가 발생할 수 있기 때문이다.
- unsigned 자료형의 경우에는 %u를 사용하자.

3. 공 수평 움직임(변수)

■ C 언어: 변수(variables) #2

■ 기본 자료형

- 선언된 변수의 값은 '변수'라는 이름이 뜻하듯이 값을 '변경'할 수 있다.
- 그러나 `const` 형으로 선언된 변수는 변수 선언을 하면서 동시에 초기값을 할당한 이후에는 변경할 수 없다.
- 따라서 한번 변수의 값을 설정하고 변경할 필요가 없는 경우(아래와 같이 수학의 pi 값 같은 경우)에 사용하면, 실수(mistake)로 값을 변경하려는 경우 에러가 생겨서 이를 쉽게 감지할 수 있다는 장점이 있다.
- 즉, `const`를 사용함으로써 조금 더 에러에 안전한 코딩을 할 수 있게 된다.

```
#include <stdio.h>

int main() {
    const double pi=3.141592;

    pi = 100.0;

    return 0;
}
```

```
error: assignment of read-only variable 'pi'
pi = 100.0;
  ^
```

3. 공 수평 움직임(변수)

■ 문자: ASCII 코드/유니코드(UNICODE)

- 2진수를 표현하는 가장 기본적인 단위를 비트(Bit)라고 한다.
- 더 많은 상태를 표현하기 위해서 여러 비트를 뭉쳐서 사용하는데, 이를 바이트(Byte)라고 한다.
- 보통 8비트를 1바이트라고 하고 하는데 1비트로 $2^1(=2)$ 가지 상태를 나타내는 것처럼, 8비트로 $2^8(=256)$ 가지의 상태를 나타낼 수 있다.
- 각 바이트의 가장 오른쪽 비트를 LSB(Least Significant Bit), 가장 왼쪽의 비트를 MSB(Most Significant Bit)라고 한다.
- 이는 왼쪽으로 갈수록 단위가 2의 지수 승으로 커지기 때문에 붙여진 이름이다.
- 이러한 바이트는 주로 숫자를 표현하기 위해서 사용된다 2진수로 표현하면 숫자 $00000000_{(2)}$ 은 숫자 $0_{(10)}$, $11111111_{(2)}$ 은 숫자 $255_{(10)}$ 이런 방식으로 약속한 것이다.
- 문자를 표현하기 위해서도 유사한 방식의 약속을 한다.
- 예를 들면, $01000001_{(2)}$ 은 영어 대문자 'A', $01011010_{(2)}$ 은 영어 대문자 'Z'를 나타내기로 약속을 하는 식이다.
- 이러한 것을 문자의 코드 테이블(code table)이라고 하는데, 여러 종류가 있으며 그 중에 가장 일반적으로 많이 사용되는 것이 ASCII(American Standard Code for Information Interchange)이다.

3. 공 수평 움직임(변수)

■ 문자: ASCII 코드/유니코드(UNICODE)

▪ ASCII 코드 테이블

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

3. 공 수평 움직임(변수)

■ 문자: ASCII 코드/유니코드(UNICODE)

- C 언어에서 사용하는 문자형 데이터(char)도 아스키 코드를 사용한다.
- ASCII 코드는 처음은 7비트 만을 사용하여 $2^7(=128)$ 개의 문자를 사용할 수 있도록 하였으며, 8번째 비트는 원래 에러 검출용 비트로 사용되어서 정상적인 데이터 전송을 확인하는 역할을 하였다.
- 이후에는 8비트가 모두 사용되어서 $2^8(=256)$ 개의 문자를 표현하기도 한다.
- ASCII 문자의 앞부분의 32문자는 화면에 출력되는 문자가 아니라 줄바꿈, 탭 등의 특수한 목적을 위해서 할당되었다.
- C 언어에서 문자형 변수를 사용하려면 아래와 같이 하면 된다.
- ASCII 코드 숫자 값을 사용해도 되고, 작은 따옴표로 문자를 직접 사용해도 된다.
 - `char ch;`
 - `ch = 65;`
 - `Ch = 'A';`

3. 공 수평 움직임(변수)

■ 문자: ASCII 코드/유니코드(UNICODE)

- 유니코드(Unicode)는 1991년에 Xerox와 Apple사에 의해서 개발된 2바이트 코드이다.
- $2^{16}(=65,536)$ 개의 문자를 표현할 수 있어서 전 세계의 거의 모든 문자를 표현할 수 있다.
- 유니코드는 현재 다양한 프로그래밍 언어와 운영체제에 의해서 지원되고 있어서 세계 표준으로 자리 잡았다.
- 256개의 문자만 표현할 수 있기 때문에 국제적인 언어들의 사용에는 부족한 단점이 있었던 ASCII 코드와의 일관성을 지키기 위해서 유니코드 앞 부분의 256개의 문자는 정확하게 ASCII 코드와 일치하게 만들었다.
- 유니코드 2.0(국제 표준 ISO 10646, 한국 표준 1005-1)에는 한글이 11,172자라서 유니코드 전체 분량의 17%를 차지하면서 포함되어 있다.
- 유니코드 2.0에는 아래와 같이 AC00~D7AF까지의 코드를 한글을 위해서 사용하는 완성형 코드이다.
- 예를 들어서 한글 '각'이라는 글자는 유니코드로 16진수 'AC01'이다.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
AC0	가	각	갇	갓	간	갇	갸	갹	갈	갇	갸	갹	갺	갻	갼	갽
AC1	감	갇	갸	갹	강	갇	갸	갹	각	갇	갸	갹	개	객	괌	괎
AC2	갸	갹	갺	갻	갼	갽	갸	갹	갺	갻	갼	갽	갸	갹	갺	갻

3. 공 수평 움직임(변수)

■ 문자 입력시 주의 사항

- scanf() 함수를 이용하여 '문자'를 입력받는 경우를 살펴보자.
- 아래와 같이 문자 5개를 공백으로 구분해서 입력한 경우의 출력 결과를 보자.

```
#include <stdio.h>

int main(){
    char ch;
    int i;

    for(i=0; i<5; i++) {
        scanf("%c", &ch);
        printf("%c", ch);
    }

    return 0;
}
```

Input: a b c d e
Output: a b c

3. 공 수평 움직임(변수)

■ 문자 입력시 주의 사항

- 또한 아래와 같은 프로그램은 사용자가 'a'를 입력하고 엔터키를 입력하면, 엔터키가 두 번째 문자 입력으로 사용되기 때문에 2번째 문자 입력에서 원하는 결과를 얻을 수 없다.
- 이런 경우에는 `fflush(stdin)` 함수를 시용해야 한다.
- 아래의 코드에서는 주석 처리해둔 부분이다.

```
#include <stdio.h>

int main(){
    char ch;

    scanf("%c", &ch);
    printf("You entered %c\n", ch);
    // fflush(stdin);
    scanf("%c", &ch);
    printf("You entered %c\n", ch);

    return 0;
}
```

입력: a[Enter key]
출력: You entered a
입력: b[Enter key]
출력: You entered

3. 공 수평 움직임(변수)

■ 상수

- 상수(constant)란 프로그램이 실행되는 동안 변하지 않는 수이다.
- 우리가 일반적으로 사용하는 정수, 실수, 문자 등의 자료형의 상수들이 있다.
- 각 자료형의 상수 예는 다음과 같다.
 - 정수형 : -10, 203, 7103
 - 실수형 : 3.14, 309.21
 - 문자 : 'a', 'A', '0', '1'
 - 문자열 (string) : "Hello"
- 경우에 따라서는 16진수나 8진수를 사용해서 상수를 사용할 필요가 있는 경우가 있다.
- 정수형 상수에서 16진수와 8진수를 표현하려면 해당하는 접두사(prefix)를 사용하면 된다.

진법	접두사	예	10진수 값
16진수	0x	0x20	32
8진수	0	040	32

3. 공 수평 움직임(변수)

■ 상수

- 아래의 프로그램은 다양한 상수를 출력하는 예제이다.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("정수 %d\n", 16);
5      printf("정수 16진수 출력 %x\n", 16);
6      printf("정수 16진수 숫자 16진수 출력 %x\n", 0x10);
7      printf("정수 16진수 출력 %#x\n", 16);
8      printf("정수 8진수 숫자 8진수 출력 %o\n", 020);
9      printf("실수형 숫자 출력 %f\n", 3.14);
10     printf("문자열 %s\n", "Hello");
11     printf("문자 %c\n", 'a');
12     printf("문자 %d\n", 'a');
13
14     return 0;
15 }
```

3. 공 수평 움직임(변수)

■ 상수

- 문자열에 대한 아래의 코드를 보자.
- 문자열은 "Hello"인데 크기가 6으로 나온다.
- 문자열의 끝에는 자동으로 ASCII 코드 0번 문자(null character)가 삽입되어서 문자열의 끝을 표시하기 때문이다.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("%d", sizeof("Hello"));
5
6      return 0;
7  }
```

3. 공 수평 움직임(변수)

■ 상수

- 상수를 사용할 때 접미사(postfix)를 붙여서 상수의 타입을 명확하게 할 수 있다.
 - `float f1 = 3.14;`
- 위와 같은 문장은 경고(warning) 메시지가 나올 수도 있다.
- 이는 기본적으로 실수를 상수로 사용하면 `double` 타입으로 저장되기 때문에, 자동으로 `double` 형으로 선언되는 상수 3.14를 `float` 타입에 저장하면 정밀도가 잘려나갈 수 있기 때문이다.
- 아래와 같이 선언하면,
 - `float f1 = 3.14F;`
- 숫자 뒤에 붙은 'F'라는 접미사가 `float` 타입으로 3.14를 사용하라는 의미이다.
- 이와 같은 접미사는 `U`, `L`, `UL`, `LL`, `ULL`, `F` 등 아주 다양하게 존재한다.

접미사	자료형	예
U 또는 u	unsigned int	77u 또는 77U
L 또는 l	long	77l 또는 77L
UL 또는 ul	unsigned long	77ul 또는 77UL

3. 공 수평 움직임(변수)

■ 상수

- C 언어에서는 문자열을 상수로 사용할 수 있지만 이를 위한 변수 자료형은 제공하지 않고 아래와 같이 문자 배열(array)을 이용해서 사용한다.
- 배열(array)이라는 구조에 대해서 간단히 알아보자.
 - `Char str[] = {'A', 'B', 'C'};`
- 여기서 `str`이라는 변수를 배열(array) 변수라고 한다.
- 이렇게 선언하면 아래와 같이 할당한 효과가 있다.
 - `str[0] == 'A'`
 - `Str[1] == 'B'`
 - `Str[2] == 'C'`
 - `printf("%s", str);`
- 즉, 이렇게 이름이 동일하고 `[]` 사이에 인덱스를 넣어서 사용할 수 있는 편리함이 있는 구조가 배열이다.
- 이렇듯 C 언어에서는 `char` 배열이라는 형식으로 문자열을 사용할 수 있다.

3. 공 수평 움직임(변수)

■ 변수 활용

- 이제 C 언어의 변수에 대한 본격적인 공부를 해보자
- 아래의 프로그램은 변수가 표현할 수 있는 최대값에 1을 더했을 때 일어나는 현상을 보이고 있다.
- 프로그램의 출력 결과는 다음과 같다.
- 이 프로그램에서는 도대체 무슨 일이 일어난 것일까?

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int cnt;
6
7     printf("no of bytes of int type : %d\n", sizeof(int));
8
9     cnt = pow(2, 31) - 1;
10    printf("maximum of integer variable = %d\n", cnt);
11    cnt++;
12    printf("maximum + 1 = %d", cnt);
13
14    return 0;
15 }
```

Microsoft Visual Studio 디버그 콘솔

```
no of bytes of int type : 4
maximum of integer variable = 2147483647
maximum + 1 = -2147483648
C:\Users\j\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수 활용

- 위의 프로그램에서 마지막 출력결과는 음수(negative)다.
- 2,147,483,647이라는 값에 1을 더한 값이 음수가 되었다.
- 이 이유는 4바이트를 이용해서 표현할 수 있는 가장 큰 양의 정수는 2,147,483,647인데, 이 숫자에 '1'을 더하려고 하니까 컴퓨터가 표현할 수 있는 최대값을 넘어서서 오동작을 일으킨 것이다.
- 이러한 상황을 오버플로우(over-flow)라고 한다.

3. 공 수평 움직임(변수)

■ 변수 활용

- 이제 실수(float, double)에 대한 이야기를 해보자.
- 아래의 프로그램은 'Not Equal'을 출력한다.
- 상수 0.2를 대입했던 변수 a와 상수 0.2가 다르다나?
- 그 이유는 무엇일까?

```
1  #include <stdio.h>
2
3  int main() {
4      float a = 0.2;
5
6      if (a == 0.2)
7          printf("Equal");
8      else
9          printf("Not Equal");
10
11     return 0;
12 }
```

Microsoft Visual Studio 디버그 콘솔

```
Not Equal
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```


3. 공 수평 움직임(변수)

■ 변수 활용

- 이 문제를 이해하기 위해서는 숫자가 컴퓨터 내부에서 어떻게 표현되는지를 이해해야 한다.
- 컴퓨터는 2진수로 숫자를 표현한다는 점을 기반으로 생각하면 $0.2_{(10)}$ 는 컴퓨터 내부에서는 2진수의 형태로 저장될 것이다.
- $0.2_{(10)}$ 를 2진수로 바꾸면 다음과 같이 표현되는데, 컴퓨터 내부에서 $0.2_{(2)}$ 는 '0011'이 무한 번 반복되는 무한 순환 소수가 된다.
 - $0.2_{(10)} = 0.0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ \dots_{(2)}$
- 컴퓨터는 반도체 하드웨어를 이용해서 만든 물건이니 만큼 표현할 수 있는(사용할 수 있는) 비트 수에 제한을 받는다.
- 일반적으로 PC에서 실수를 표현하기 위해서 단-정밀도(single precision)는 4바이트(32비트, float 형)를, 배-정밀도(double precision)는 8바이트(64비트, double 형)를 할당한다.
- 그렇지만 몇 바이트를 할당하든지 상관없이 컴퓨터를 이용해서 10진수 $0.2_{(10)}$ 를 정확하게 표현한다는 것은 불가능하다!
- 10진수로 표현된 $0.2_{(10)}$ 는 무한 소수가 아니지만, 이를 2진수로 변환하면 무한 순환 소수가 되기 때문이다.
- 컴파일러에 따라 다를 수 있지만 대부분의 C 언어 컴파일러는 상수(constant number)는 8바이트를 할당해서 표현한다.
- 즉, 배-정밀도(double precision)로 표현한다.
- 따라서 위의 경우에 $0.2_{(10)}$ 를 가지고 있는 '변수 a'는 4바이트를 사용하고, '상수 $0.2_{(10)}$ '는 8바이트를 사용하기 때문에 if 문에서 비교할 때 다른 값으로 인식이 되는 것이다.

3. 공 수평 움직임(변수)

■ 변수 활용

- 그럼 위의 프로그램을 조금 더 정확하게 수행하게 하려면 프로그램을 어떻게 수정해야 할까?
- 아래와 같이 변수 선언에서 float 대신 double을 사용하면 된다.

```
1  #include <stdio.h>
2
3  int main() {
4      double a = 0.2;
5
6      if (a == 0.2)
7          printf("Equal");
8      else
9          printf("Not Equal");
10
11     return 0;
12 }
```

Microsoft Visual Studio 디버그 콘솔

```
Equal
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수 활용

- 또 다른 해결 방법은 없을까?
- 조금 변경한 아래의 프로그램을 보자.

```
1  #include <stdio.h>
2
3  int main() {
4      float a = 0.2;
5
6      printf("%10.8f\n", (a - 0.2));
7      printf("%15.13f\n", (a - 0.2));
8      printf("%25.23f\n", (a - 0.2));
9      printf("%35.33f\n", (a - 0.2));
10
11     if (a - 0.2 == 0)
12         printf("Equal(Exactly) \n");
13     else
14         printf("Not Equal(Exactly) \n");
15
16     if (a - 0.2 < 0.00000001)
17         printf("Equal(Approximately) \n");
18     else
19         printf("Not Equal(Approximately) \n");
20
21     return 0;
22 }
```

Microsoft Visual Studio 디버그 콘솔

```
0.00000000
0.0000000029802
0.00000000298023222766730
0.000000002980232227667301003748435
Not Equal(Exactly)
Equal(Approximately)
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\Console
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수 활용

- $a - 0.2 == 0$ 임을 검사하기 위해서는 연산 결과가 위의 프로그램에서 사용한 값과 같이 “아주 작은 값”보다 작다면 0으로 간주하는 방법이다.
- 실제로 `double 0.2`와 `float 0.2`를 뺀 값은 ‘0’이 아니고, 아주 작은 값이다.
- 이렇게 실수 값은 표현할 수 있는 값의 범위도 제한되고, 정확한 값도 표현하지 못할 수 있기 때문에, 컴퓨터를 이용하여 다양한 계산을 할 때는 주의해야 한다.

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 변수는 전역 변수(global variables)와 지역 변수(local variables)가 있다.
- 특정 함수(또는 블록) 내부에서 선언된 변수는 지역 변수, 그렇지 않은 변수를 전역 변수라고 한다.
- 지역 변수는 그 함수(또는 블록) 내에서만 사용할 수 있다.
- 예를 들어, 아래의 변수 i는 test() 함수 '안'에서만 사용 가능하다는 말이다.
- test() 함수 '밖'에서는 이 변수 i를 사용할 수 없다.

```
void test() {  
    int i;  
  
    i = 10;  
}
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 그렇다면, 아래와 같은 프로그램은 어떻게 동작할까?
- `main()` 함수에서 `test()` 함수를 2번 호출한다.
- 각 호출시에 출력되는 값을 살펴보자.

```
1  #include <stdio.h>
2
3  void test() {
4      int i = 10;
5
6      printf("%d\n", i);
7      i++;
8      printf("%d\n", i);
9  }
10
11 int main() {
12     int i = 100;
13
14     test();
15     printf("%d\n", i);
16     test();
17
18     return 0;
19 }
```

Microsoft Visual Studio 디버그 콘솔

```
10
11
100
10
11

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 전역 변수와는 다르게 지역 변수는 일정한 블록(block)에서만 효력이 있다.
- 간단한 예제 프로그램으로 확인해보자.

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5
6      a = 10;
7      printf("a @ main function = %d\n", a);
8      {
9          int a = 30;
10         printf("a @ inner block = %d\n", a);
11     }
12
13     printf("a @ main function = %d\n", a);
14
15     return 0;
16 }
```

Microsoft Visual Studio 디버그 콘솔

```
a @ main function = 10
a @ inner block = 30
a @ main function = 10
```

```
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 전역 변수와 지역 변수의 초기화 과정을 살펴보기 위해서 아래의 프로그램을 보자.
- 아래와 같이 전역 변수는 컴파일러에 의해서 자동으로 0으로 초기화된다.

```
1  #include <stdio.h>
2
3  int globalInt;
4  float globalFloat;
5  char globalChar;
6
7  int main() {
8      printf("Global int: %d\n", globalInt);
9      printf("Global float: %f\n", globalFloat);
10     printf("Global char: %c\n", globalChar);
11
12     return 0;
13 }
```

Microsoft Visual Studio 디버그 콘솔

Global int: 0

Global float: 0.000000

Global char:

C:\Users\j\inu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 위와 다르게 지역 변수는 별도로 명시적으로 초기화하지 않으면 에러를 발생시킨다.
- 따라서 지역 변수를 사용할 때는 특히 변수의 초기화를 신경써야 한다.

```
1  #include <stdio.h>
2
3  int main() {
4      int localInt;
5      float localFloat;
6      char localChar;
7
8      printf("Local int: %d\n", localInt);
9      printf("Local float: %f\n", localFloat);
10     printf("Local char: %c\n", localChar);
11
12     return 0;
13 }
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

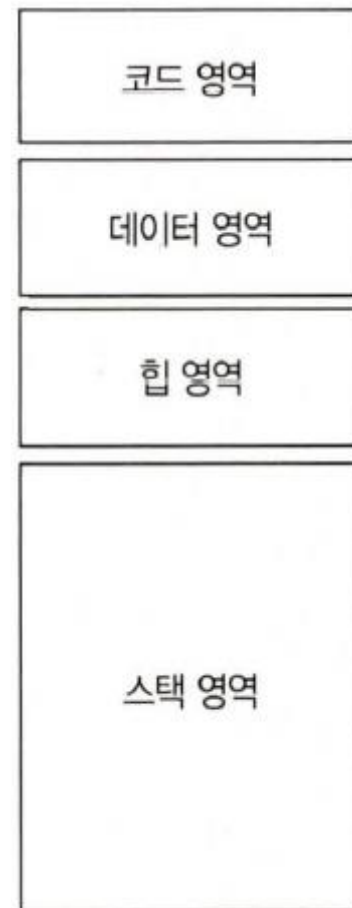
- 아래에 전역 변수와 지역 변수의 생존 기간(life time)과 접근 가능 영역(scope), 할당되는 메모리 영역, 초기값을 표로 정리하였다.

	전역 변수	지역 변수
생존 기간 (life time)	프로그램 실행 시작부터 종료시까지: 전역 변수는 프로그램이 실행되면서 변수가 생성되어 종료시까지 남아있다.	블록(함수) 실행 시작부터 종료시까지: 지역 변수는 프로그램 실행 중 블록이 시작될 때 생성되어서 블록이 수행을 마칠 때 삭제된다.
접근 영역 (scope)	프로그램의 어디서나 사용할 수 있다.	변수가 선언된 블록 내부에서만 사용할 수 있다.
할당되는 메모리 영역	데이터 영역	스택 영역
초기값	프로그래머가 별도로 초기화하지 않으면 컴파일러에 의해서 자동으로 0으로 초기화된다.	지역 변수는 컴파일러에 의해 자동으로 0으로 초기화되지 않는다. 아무 의미없는 쓰레기 값이 들어있다.

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 컴파일된 프로그램은 프로그램의 각 부분들(코드, 데이터 등)이 영역별로 나누어서 메모리에 자리잡게 된다.
- 아래는 프로그램이 사용하는 메모리 영역에 대한 설명이다.
 - 코드 영역(code segment): 프로그램의 인스트럭션(명령어)이 위치.
 - 데이터 영역(data segment): 전역 변수와 static 변수들이 위치. 이 영역에 할당되는 변수들은 프로그램 시작과 동시에 메모리 공간에 할당되어 프로그램 종료시까지 메모리에 남아있다.
 - 스택 영역(stack segment): 지역 변수와 매개 변수들이 위치. 함수(블록)가 실행될 때 메모리를 할당받음.
 - 힙 영역(heap segment): 프로그램 실행 도중에 동적으로 할당되는 메모리.



3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 전역 변수와 static 변수는 컴파일 하는 시점에서 주소를 정할 수 있다.
- 따라서 전역 변수와 static 변수는 데이터 영역에 메모리를 할당받으며 컴파일러가 값을 0으로 자동으로 초기화해준다.
- 그러나 지역 변수는 프로그램 실행 중에 동적으로 할당받는 스택 영역에 메모리가 잡힌다.
- 따라서 자동 초기화가 되지 않고 쓰레기 값이 들어갈 수 있다.
- 그러니까 지역 변수는 꼭 명시적으로 초기화하는 것이 안전하다.
- 컴파일러에 따라서는 초기화하지 않은 변수를 사용하려는 경우 컴파일 시점에 에러가 발생할 수도 있다.
- 지역 변수 뿐만 아니라 전역 변수도 프로그래머가 명시적으로 초기화하는 버릇을 들이는 것이 좋다.

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 전역 변수의 장점은 무엇일까?
- 아래의 2 프로그램은 동일한 작업을 한다.
- 하나는 지역 변수로 숫자를 입력받고, 다른 하나는 전역 변수로 숫자를 입력 받는다.
- 바로 아래의 지역 변수를 사용하는 프로그램이 훨씬 좋은 구조다.

▪ 왜 그럴까?

```
1  #define _CRT_SECURE_NO_WARNINGS
2
3  #include <stdio.h>
4
5  int add(int a, int b) {
6      return (a + b);
7  }
8
9  int main() {
10     int answer, input1, input2;
11
12     scanf("%d", &input1);
13     scanf("%d", &input2);
14     answer = add(input1, input2);
15     printf("answer = %d\n", answer);
16
17     return 0;
18 }
```

Microsoft Visual Studio 디버그 콘솔

```
10
20
answer = 30

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\Con
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 아래의 프로그램은 전역 변수를 사용하는 경우이다.
- 그러나, 가능하면 전역 변수는 사용하지 않는 것이 프로그래밍에 좋은 습관이다.

```
1  #define _CRT_SECURE_NO_WARNINGS
2
3  #include <stdio.h>
4
5  //global variables;
6  int globalA;
7  int globalB;
8
9  int add() {
10     return (globalA + globalB);
11 }
12
13 int main() {
14     int answer;
15
16     scanf("%d", &globalA);
17     scanf("%d", &globalB);
18     answer = add();
19     printf("answer = %d\n", answer);
20
21     return 0;
22 }
```

Microsoft Visual Studio 디버그 콘솔

```
10
20
answer = 30

C:\Users\j\inu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 전역 변수의 또 다른 단점은 무엇일까?
- 전역 변수를 사용하면 아래와 같이 하나의 함수에서 값을 수정한 결과가 다른 함수에 영향을 미칠 가능성이 많아진다는 점이 단점이다.

```
1  #include <stdio.h>
2
3  int i;
4
5  void printStar() {
6      // int i;
7
8      for (i = 0; i < 10; i++) {
9          printf("*");
10     }
11     printf("\n");
12 }
13
14 int main() {
15     //int i;
16
17     i = 1;
18
19     printStar();
20     i = i + 10;
21     printf("%d\n", i);
22
23     return 0;
24 }
```

Microsoft Visual Studio 디버그 콘솔

20

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 위의 예에서와 같이 가능하면 전역 변수를 사용하지 않는 것이 좋다.
- 전체적인 프로그램의 복잡도를 높일 수 있기 때문이다.
- 그러나 아래와 같이 전역 변수를 써야 하는 경우도 있다.
- 아래와 같이 특정 함수가 2개 이상의 계산 결과를 반환해야 한다면 전역 변수를 써야 한다.

Microsoft Visual Studio 디버그 콘솔

```
10
20
sum = 30
mul = 200

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe(
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

```
1  #define _CRT_SECURE_NO_WARNINGS
2
3  #include <stdio.h>
4
5  int sum;
6  int mul;
7
8  void addAndMul(int a, int b) {
9      sum = a + b;
10     mul = a * b;
11 }
12
13 int main() {
14     int in1, in2;
15
16     scanf("%d", &in1);
17     scanf("%d", &in2);
18
19     addAndMul(in1, in2);
20
21     printf("sum = %d\n", sum);
22     printf("mul = %d\n", mul);
23
24     return 0;
25 }
```


3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 이러한 경우 이외에도 여러 함수가 하나의 변수를 공유할 필요가 있는 경우에도 전역 변수를 사용하면 편하게 프로그래밍할 수 있다
- 특정 함수가 호출된 횟수를 알고 싶다.
- 어떻게 하면 될까?
- 아래의 2개의 프로그램을 보자 출력 결과는 동일하다.
- 이 두 프로그램의 차이점은 무엇인가?
- 왼쪽은 함수의 호출 횟수를 전역 변수로 카운트하고, 오른쪽 프로그램은 지역 변수로 카운트한다.

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 그런데 static이라는 단어가 눈에 띈다.
- static 지역 변수는 전역 변수와 동일하게 프로그램이 시작될 때 변수가 선언되고 초기화된다.
- 그리고 전역 변수와 동일한 생존 기간을 가지며, 지역 변수와 동일한 접근 영역 특성을 가진다.

```
1  #include <stdio.h>
2
3  int count = 1;
4
5  void printStar(){
6      printf("***** %d\n", count);
7      count++;
8  }
9
10 int main() {
11     printStar();
12     printStar();
13     printStar();
14     printStar();
15
16     return 0;
17 }
```

```
1  #include <stdio.h>
2
3  void printStar(){
4      static int count = 1;
5
6      printf("***** %d\n", count);
7      count++;
8  }
9
10 int main() {
11     printStar();
12     printStar();
13     printStar();
14     printStar();
15
16     return 0;
17 }
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 그러면 아래와 같이 코딩하면 어떻게 될까?
- 음영으로 표시한 부분이 바뀐 부분이다.
- 출력 결과에서 알 수 있듯이, static 지역 변수는 변수 선언 문장만 프로그램이(함수가 아니다) 처음 실행될 때 선언되고 초기화됨을 확인할 수 있다.
- 즉, 아래와 같이 사용하면 원하는 결과를 얻을 수 없다.

```
1  #include <stdio.h>
2
3  void printStar(){
4      static int count;
5
6      count = 1;
7      printf("***** %d\n", count);
8      count++;
9  }
10
11 int main() {
12     printStar();
13     printStar();
14     printStar();
15     printStar();
16
17     return 0;
18 }
```

3. 공 수평 움직임(변수)

■ 변수의 범위(scope)과 생존 기간(life time)

- 아래에서 전역 변수, static 지역 변수, 지역 변수의 생존 기간과 접근 영역을 정리해보자.
- static 지역 변수는 전역 변수와 지역 변수의 특성을 혼합해서 가지고 있다.

	전역 변수	static 지역 변수	지역 변수
생존 기간 (life time)	프로그램 실행 시작부터 종료시까지	프로그램 실행 시작부터 종료시까지	블록(함수) 실행 시작부터 종료시까지
접근 영역 (scope)	어디서나	변수가 선언된 블록 내부에서만	변수가 선언된 블록 내부에서만

3. 공 수평 움직임(변수)

■ 변수 기타: 형 변환과 파생 자료형

- 다양한 자료형의 데이터 값들을 필요한 경우에는 서로 변환하여 사용할 수 있다.
- 변환 방법은 크게 2가지가 있다.
 - 명시적(explicit) 형 변환(type conversion)
 - 묵시적(implicit) 형 변환(type conversion)
- 명시적 형 변환은 아래와 같이 값 앞에 원하는 데이터 타입을 괄호 안에 적어주면 되고, 묵시적 형 변환은 자동으로 이루어지는 형 변환을 의미한다.

```
1  #include <stdio.h>
2
3  int main() {
4      int noi = 5;
5      float nof;
6
7      printf("noi in integer: %d\n", noi);
8      printf("noi in float(explicit conversion): %f\n", (float)noi);
9
10     nof = noi;
11     printf("nof(implicit conversion): %f\n", nof);
12
13     return 0;
14 }
```

Microsoft Visual Studio 디버그 콘솔

```
noi in integer: 5
noi in float(explicit conversion): 5.000000
nof(implicit conversion): 5.000000

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApp
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수 기타: 형 변환과 파생 자료형

- 위와 같은 경우는 명시적 형 변환이든 묵시적 형 변환이든 작은 자료형(int)에서 더 큰 자료형(float)으로의 변환이었다.
- 따라서 손실되는 값은 없다.
- 아래를 한번 보자.
- 아래는 반대의 경우이다.
- 큰 자료형에서 작은 자료형으로 변환되면서 값의 손실이 있다.

```
1 #include <stdio.h>
2
3 int main() {
4     int noi;
5     float nof = 5.2;
6
7     printf("nof in float: %f\n", nof);
8
9     noi = nof;
10    printf("noi(implicit conversion): %d\n", noi);
11
12    noi = (int)nof;
13    printf("noi(explicit conversion): %d\n", noi);
14
15    return 0;
16 }
```

Microsoft Visual Studio 디버그 콘솔

```
nof in float: 5.200000
noi(implicit conversion): 5
noi(explicit conversion): 5
```

```
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApp1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공수평움직임(변수)

■ 변수 기타: 형 변환과 파생 자료형

- 이러한 자동 형 변환(묵시적 형 변환)은 모든 연산자(대입 연산자 포함)를 이용한 연산 과정에서 일어난다.
 - 대입 연산 시에는 등호의 오른쪽 값이 자동으로 왼쪽의 자료형으로 변환되어서 대입된다.
 - 수식 연산 시에는 피 연산자 중에 더 범위가 넓은 쪽으로 자동형 변환되어서 계산된다.

```
1  #include <stdio.h>
2
3  int main() {
4      int i;
5      float f;
6
7      f = 3.14;
8
9      printf("%f\n", f+10);
10
11     i = f + 10;
12
13     printf("%d", i);
14
15     return 0;
16 }
```

Microsoft Visual Studio 디버그 콘솔

```
13.140000
13
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

3. 공 수평 움직임(변수)

■ 변수 기타: 형 변환과 파생 자료형

- C에서 제공되는 기본 자료형 이외에도 다양한 자료형이 제공된다.
- 이러한 자료형을 파생 자료형이라고 한다.
 - 기본 자료형: char, short, int, float, double 등
 - 파생 자료형 : 배열(array), 나열형(enumeration), 구조체(struct), 공용체(union)

3. 공 수평 움직임(변수)

1. 프로그램을 수정하여서 공을 오른쪽 끝에서 왼쪽으로 움직여 보자.

```
1  #include "ofApp.h"
2
3  int xPos;
4
5  //-----
6  void ofApp::setup(){
7      ofSetWindowTitle("Ball Drawing Program");
8      ofSetWindowShape(1024, 768);
9      ofSetFrameRate(60);
10     ofBackground(ofColor::white);
11     ofSetColor(ofColor::black);
12     ofSetLineWidth(1.0);
13     ofSetColor(ofColor::paleVioletRed);
14     ofFill();
15
16     xPos = 1024;
17 }
18
19 //-----
20 void ofApp::update(){
21     xPos--;
22 }
23
24 //-----
25 void ofApp::draw(){
26     ofCircle(xPos, 60, 30);
27 }
```

3. 공 수평 움직임(변수)

2. 2개 이상의 공을 움직여보자. 각 공들의 움직이는 속도를 다르게 할 수 있을까?

```
1  #include "ofApp.h"
2
3  int xPos1, xPos2;
4
5  //-----
6  void ofApp::setup(){
7      ofSetWindowTitle("Ball Drawing Program");
8      ofSetWindowShape(1024, 768);
9      ofSetFrameRate(60);
10     ofBackground(ofColor::white);
11     ofSetColor(ofColor::black);
12     ofSetLineWidth(1.0);
13     ofSetColor(ofColor::paleVioletRed);
14     ofFill();
15
16     xPos1 = 0;
17     xPos2 = 0;
18 }
19
```

```
20  //-----
21  void ofApp::update(){
22     xPos1++;
23     xPos2 = xPos2 + 5;
24 }
25
26  //-----
27  void ofApp::draw(){
28     ofCircle(xPos1, 60, 30);
29     ofCircle(xPos2, 200, 30);
30 }
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ if 문(조건문)

- 앞에서 공이 오른쪽 경계를 벗어난 후 없어져서 황당했을 것이다.
- 이제는 조금 전의 프로그램을 공이 좌 · 우의 경계에서 계속적으로 반사되면서 움직이도록 수정해보자
- 이를 위해서는 '매번 공을 움직일 때마다' 좌우의 경계에 부딪히는지 체크해야 한다.
- 그리고 경계에 부딪히면 공을 반사(방향 변경)시켜줘야 한다.
- 이를 위해서 xDir이라는 이름의 변수를 만들었다.
- 변수 xDir은 공의 움직이는 방향을 나타내는 새로운 변수인데, 오른쪽으로 움직이면 +1, 왼쪽으로 움직이면 -1 값을 가지는 것으로 정했다.
- 즉 xDir 변수는 X축 상에서 움직이는 방향을 의미한다.
- 그래서 이제는 공을 움직이기 위해서는 아래와 같이 해야 한다.
 - $xPos = xPos + xDir;$

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ if 문(조건문)

- 공이 왼쪽으로 움직이다가 xPos가 0보다 작아지면 방향을 바꾸고, 공이 오른쪽으로 움직이다가 xPos가 화면의 우측 끝을 만나면 방향을 바꾸면 된다.
- 아래와 같이 xDir의 부호를 바꾸어서 공이 움직이는 방향을 바꿀 수 있다.
 - if (xPos < 0) xDir = xDir * -1;
 - if (xPos > ofGetWidth()) xDir = xDir * -1;
- 위의 2 문장은 아래와 같이 if else if .. 문으로 병합하는 것이 더 효율적이다.
 - if (xPos < 0) xDir = xDir * -1;
 - else if (xPos > ofGetWidth()) xDir = xDir * -1;
- 또는 좌-우 경계와의 충돌을 체크하는 위의 두 조건문을 || (or 연산자)를 사용하여 아래와 같이 합칠 수도 있다.
 - if (xPos < 0 || xPos > ofGetWidth()) xDir = xDir * -1;

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ if 문(조건문)

- 아래에 최종적인 ofApp.cpp 파일을 보자

```
1  #include "ofApp.h"
2
3  int xPos;
4  int xDir;
5
6  //-----
7  void ofApp::setup(){
8      ofSetWindowTitle("Ball Drawing Program");
9      ofSetWindowShape(1024, 768);
10     ofSetFrameRate(60);
11     ofBackground(ofColor::white);
12     ofSetColor(ofColor::black);
13     ofSetLineWidth(1.0);
14     ofFill();
15
16     xPos = 0;
17     xDir = 1;
18 }
19
```

```
20  //-----
21  void ofApp::update(){
22     xPos += xDir;
23
24     if (xPos < 0) xDir = xDir * -1;
25     if (xPos > ofGetWidth()) xDir *= -1;
26 }
27
28  //-----
29  void ofApp::draw(){
30     ofCircle(xPos, 60, 30);
31 }
32
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ c 언어: 기본 연산자

- 연산자(operator)란 실제 연산(계산)에 사용되는 기호들을 의미한다.
- 우리가 자주 사용하는 $+$, $-$, $*$, $/$ 같은 사칙연산에 사용하는 기호들도 프로그래밍에서 연산자로 사용된다.
- 연산자는 크게 다음과 같은 종류들이 있다.
 - 산술 연산자
 - 관계 연산자
 - 비트 연산자
 - 논리 연산자
 - 대입 연산자

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 모든 연산자는 연산 결과값을 반환(return)한다.
- 아래 문장의 실행 순서를 알아보자.
 - $a = 3 + 4 * 5;$
- $4 * 5$
 - 가장 먼저 연산되는 것은 $*$ 연산자이다. $4 * 5$ 는 20을 반환한다.
 - 일반적인 사칙 연산에서도 곱셈이 덧셈보다 우선한다.
- $3 + 20$
 - $*$ 연산자 이후의 결과값 20을 사용해서 $3 + 20$ 연산을 한다.
 - 이 연산은 23를 반환한다.
- $a = 23$
 - 변수 a에 23를 대입(assignment)한다.
 - $=$ 연산자는 대입 결과인 23를 반환한다.
 - 즉, 대입 연산자도 결과값을 반환한다.
 - 즉, 위의 문장의 최종 결과값은 23이다.

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 아래의 코드로 그 결과를 확인해보자.

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5
6      a = 3 + 4 * 5;
7      printf("a: %d\n", a);
8      printf("a: %d\n", 3 + 4 * 5);
9      printf("a: %d\n", a = 3 + 4 * 5);
10
11     return 0;
12 }
```

Microsoft Visual Studio 디버그 콘솔

a: 23
a: 23
a: 23

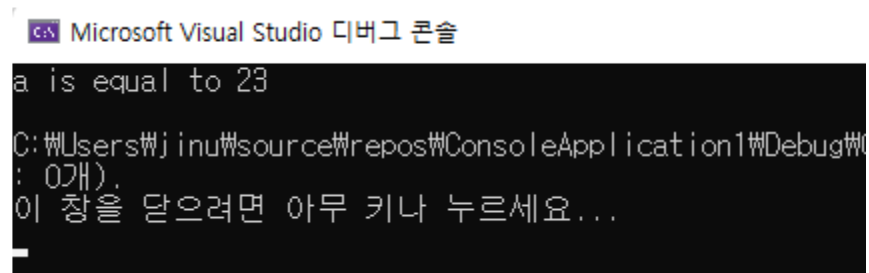
C:\Users\jinu\source\repos\ConsoleApplication1\Debug
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 그렇기 때문에 아래와 같은 문장 형식도 사용할 수 있다.

```
1  #include <stdio.h>
2
3  int main() {
4      int a;
5
6      if ((a = 3 + 4 * 5) == 23) {
7          printf("a is equal to 23\n");
8      }
9
10     return 0;
11 }
```



Microsoft Visual Studio 디버그 콘솔

```
a is equal to 23
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

- 연산 결과의 자료형은 피연산자(operand)의 자료형과 일치한다.
- 즉, 연산자에 의한 연산 결과는 피연산자의 자료형의 결과와 동일한 자료형을 반환한다.

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 아래는 얼마를 출력할까?
- 모든 연산자의 연산 결과는 피연산자의 자료형과 일치한다.
- 따라서 아래의 결과는 '3'을 출력한다.

```
int n1=10, n2=3;  
printf("나눗셈 결과: %d\n", n1/n2);
```

나눗셈 결과: 3

- 아래의 나눗셈은 float 형을 나눗셈하기 때문에 3.333333을 출력한다.

```
float n1=10.0, n2=3.0;  
printf("나눗셈 결과: %f\n", n1/n2);
```

나눗셈 결과: 3.333333

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 아래의 출력문의 결과를 하나씩 유심히 보자.

```
1  #include <stdio.h>
2
3  int main() {
4      int i;
5      float f;
6
7      printf("i: %d\n", 3.3 + 4);
8      printf("i: %f\n", 3.3 + 4);
9
10     i = 3.3 + 4;
11     printf("i: %d\n", i);
12
13     printf("f: %f\n", 3.3 + 4);
14
15     f = 3.3 + 4;
16     printf("f: %f\n", f);
17
18     return 0;
19 }
```

C:\ Microsoft Visual Studio 디버그 콘솔

```
i: 858993459
i: 7.300000
i: 7
f: 7.300000
f: 7.300000

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 아래의 프로그램을 컴파일하면 3, 4번째 `printf()` 문에서 "warning" 메시지가 출력된다.
 - warning (4477 : 'printf' format string '7od' requires an argument of type 'int' , but variadic argument 1 has type 'double'
- warning 메시지가 출력되더라도 프로그램은 실행될 수 있다.
- 그러나 정확한 결과값을 보장하지는 못한다.
- 아래에 엉뚱한 결과가 출력된 것을 확인할 수 있다.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("%d\n", 10 / 3);
5      printf("%d\n", 10 % 3);
6
7      printf("%d\n", 10.0 / 3.0);
8      printf("%d\n", 10 / 3.0);
9
10     printf("%f\n", 10.0 / 3.0);
11     printf("%f\n", 10 / 3.0);
12
13     //printf("%d\n", 10.0 % 3.0);
14
15     return 0;
16 }
```

Microsoft Visual Studio 디버그 콘솔

```
3
1
-1431655765
-1431655765
3.333333
3.333333

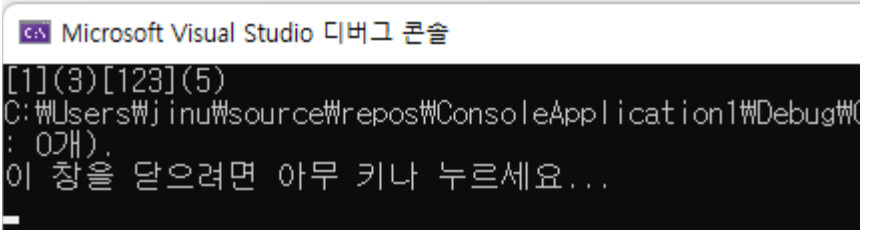
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 이제 아래의 프로그램을 보자 조금 기이하다고 느낄 수도 있겠다.
- 출력값이 왜 이렇게 나오는지 이해할 수 있는가?

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b;
5
6      printf("(%d)", printf("[%d]", 1));
7      printf("(%d)", printf("[%d]", 123));
8
9      return 0;
10 }
```



Microsoft Visual Studio 디버그 콘솔

[1](3)[123](5)
C:\Users\winu\source\repos\ConsoleApplication1\Debug\0 : 0개).
이 창을 닫으려면 아무 키나 누르세요...

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 위의 내용을 이해하려면 printf() 함수의 반환값을 알아야 한다.
- 아래와 같이 printf() 함수도 출력하는 함수이기는 하지만 어떤 값을 반환한다.
- 아래에서 printf() 함수의 반환값(return value)이 무엇인지 찾아보자.
- 즉, printf() 함수는 무엇을 반환할까?

```
int printf ( const char * format, ... );
```

Print formatted data to stdout

Writes the C string pointed by *format* to the standard output (stdout).
If *format* includes *format specifiers* (subsequences beginning with %),
the additional arguments following *format* are formatted and inserted
in the resulting string replacing their respective specifiers.

Return Value

On success, the total number of characters written is returned.
If a writing error occurs, the *error indicator* (ferror) is set and a
negative number is returned.

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- printf () 함수는 출력을 정상적으로 성공하였을 때는 '출력된 문자의 개수'를 반환한다.
- 즉, 아래와 같이 프로그래밍할 수 있다.

```
1  #include <stdio.h>
2
3  int main() {
4      if (printf("%s", "Hello World!") == 12) printf("Success");
5
6      return 0;
7  }
```

Microsoft Visual Studio 디버그 콘솔

```
Hello World!Success
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 연산자의 결과 값

- 이와 같이 프로그래밍 언어를 새롭게 배울 때는 모든 문법적인 정의(definition)를 정확히 알아두어야 활용할 때 실수하지 않는다.
- 이제 위의 프로그램에서 아래의 문장의 출력 결과를 이해할 수 있을까?
 - `printf("(%d)", printf("[%d]", 1));` → 출력: [1](3)

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 대입 연산자

- 위에서 예를 든 설명은 연산자와 관련된 극히 일부분이다.
- 중요한 몇 가지 연산자를 아래에서 예를 들어서 살펴보자.
 - `a = 10;` `// 변수 a에 10을 대입한다.`
- 위의 `=` 연산자는 대입 연산자이다.
- 그리고 아래와 같이 대입 연산자와 산술 연산자를 합쳐 놓은 연산자도 있다.
- 이를 '복합 대입 연산자'라고 한다.
 - `a += 10;` `// a의 값이 10 증가된다. a = a+10과 동일`
 - `a *= 10;` `// a의 값이 10배가 된다. a = a * 10과 동일`
 - `a /= 10;` `// a의 값이 1/10이 된다. a = a / 10과 동일`
 - `a %= 10;` `// a에 a를 10으로 나눈 나머지가 대입. a = a % 10과 동일`

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 대입 연산자

- 대입 연산자는 다른 대부분의(콤마 연산자 제외) 연산자에 비해서 우선 순위가 낮다.

```
1  #include <stdio.h>
2
3  int main() {
4      int i, j, k;
5
6      i = 1 + 2 * 3;
7      j = 10, k = 20;
8      printf("i: %d, j: %d, k: %d", i, j, k);
9
10     return 0;
11 }
```

Microsoft Visual Studio 디버그 콘솔

```
i: 7, j: 10, k: 20
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

- 대입 연산자의 왼쪽에 올 수 있는 것을 lvalue(left value)라고 한다.
- 일반적인 변수라고 생각하면 된다.
- 당연히 상수(constant)는 lvalue가 될 수 없다.
- 아래와 같은 문장은 에러가 발생한다.
 - `3 = 5 + 10;`
 - Error: lvalue required as left operand of assignment

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

- 산술 연산자에는 피연산자(operand)가 2개인 '이항 연산자'와, 피연산자가 1개인 '단항 연산자'가 있다.
- 이항 산술 연산자(*, /, %, +, -)
 - 산술 연산자 중에서도 이항 연산자란 피연산자가 두 개인 연산자를 의미하며, 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 연산자를 의미한다.
 - 덧셈, 뺄셈, 곱셈, 나눗셈과 같은 대부분의 산술 연산자의 의미는 이미 알고 있을 것이고, 연산자의 우선 순위에만 신경쓰면 별 어려움은 없을 것이다.
 - C 언어에는 여기에 나머지 연산자(%)가 추가된다.
 - 나머지 연산자는 '나머지'라는 연산의 특성상 피 연산자가 정수형 값이어야 한다는 점만 유의하면 된다.
 - 즉, 나머지 연산자는 피연산자를 정수값으로 하면서 나머지를 정수값으로 반환한다.

```
1  #include <stdio.h>
2
3  int main() {
4      int n1 = 10, n2 = 3;
5
6      printf("나머지: %d\n", n1 % n2);
7
8      return 0;
9  }
```

Microsoft Visual Studio 디버그 콘솔

나머지: 1

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe : 0개).
이 창을 닫으려면 아무 키나 누르세요...

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

■ 이항 산술 연산자(*, /, %, +, -)

- 아래와 같이 float와 int가 섞인 나눗셈은 어떻게 될까?
- 일반적으로 나눗셈 연산자의 피연산자가 서로 다른 타입의 값이면 값의 손실이 적은 방식으로 묵시적 형변환(implicit type conversion)이 일어난다.
- 이러한 사항은 다른 연산자에도 동일하게 적용된다.

```
1  #include <stdio.h>
2
3  int main() {
4      int n1 = 10, n2 = 3;
5
6      printf("나눗셈 결과: %d\n", n1 / n2);
7
8      return 0;
9  }
```

Microsoft Visual Studio 디버그 콘솔

```
나눗셈 결과: 3
C:\Users\jinu\source\repos\ConsoleApplica
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

```
1  #include <stdio.h>
2
3  int main() {
4      float n1 = 10.0;
5      int n2 = 3;
6
7      printf("나눗셈 결과: %f\n", n1 / n2);
8
9      return 0;
10 }
```

Microsoft Visual Studio 디버그 콘솔

```
나눗셈 결과: 3.333333
C:\Users\jinu\source\repos\ConsoleApplica
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

▪ 단항 산술 연산자(++ , --)

- ++나 --와 같은 단항 연산자에 대해서 이야기해보자.
- 단항 연산자란 피연산자(operand)가 한 개인 연산자를 말한다.
- C 언어에서 다른 '대부분'의 연산자는 피연산자가 2개이고 단항 연산자는 몇 개 되지 않는다.
- 아래의 예를 보자 단항 연산자를 변수의 앞에 사용(prefix)하면 단항 연산자를 먼저 수행하고 그 결과값을 사용하고, 단항 연산자를 변수의 뒤에 사용(postfix)하면 현재 값을 사용한 후 한 단항 연산자를 수행한다.
 - ++a; 는 a가 증가된 후 사용되고,
 - a++; 는 a가 사용되고 나서 증가된다.
- 즉, 변수 a의 값이 현재 10이라면, 아래에서는 x에 10이 대입된 후, a가 11이 되는데,
 - a = 10;
 - x = a++;
- 아래의 경우는 a가 11이 된 후, 이 값이 x에 대입된다.
- 즉, x는 11이 된다.
 - a = 10;
 - x = ++a;

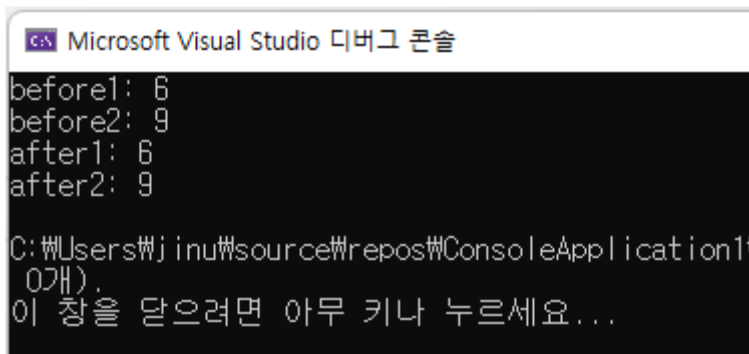
4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

▪ 단항 산술 연산자(++ , --)

- 이러한 사실을 예를 통해서 살펴보자.
- 아래는 단항 연산자인 ++과 --이 prefix에 위치하기 때문에 단항 연산자가 연산된 후 대입된다.

```
1  #include <stdio.h>
2
3  int main() {
4      int before1 = 5;
5      int before2 = 10;
6      int after1, after2;
7
8      after1 = ++before1;
9      after2 = --before2;
10
11     printf("before1: %d\n", before1);
12     printf("before2: %d\n", before2);
13     printf("after1: %d\n", after1);
14     printf("after2: %d\n", after2);
15
16     return 0;
17 }
```



Microsoft Visual Studio 디버그 콘솔

```
before1: 6
before2: 9
after1: 6
after2: 9

C:\Users\jinu\source\repos\ConsoleApplication1\
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

▪ 단항 산술 연산자(++ , --)

- 그러나 아래와 같이 postfix로 사용하면 결과가 달라진다.
- 단항 연산자는 대입 연산자 보다 우선 순위가 빠르다.
- 그러나 postfix로 사용하면 증가되기 전의 값이 사용된다는 특성이 있다.

```
1  #include <stdio.h>
2
3  int main() {
4      int before1 = 5;
5      int before2 = 10;
6      int after1, after2;
7
8      after1 = before1++;
9      after2 = before2--;
10
11     printf("before1: %d\n", before1);
12     printf("before2: %d\n", before2);
13     printf("after1: %d\n", after1);
14     printf("after2: %d\n", after2);
15
16     return 0;
17 }
```

Microsoft Visual Studio 디버그 콘솔

```
before1: 6
before2: 9
after1: 5
after2: 10
```

C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

▪ 단항 산술 연산자(++ , --)

- 아래의 예는 일반적으로 이렇게 사용하는 경우는 없겠지만, postfix 단항 연산자의 작동 방식을 이해하기 위해서 일 부러 만들어본 코드이다.
- postfix 연산자는 값은 변동하지만 변동된 값은 그 다음 문장부터 유효하다.

```
1  #include <stdio.h>
2
3  int main() {
4      int before = 5;
5      int after;
6
7      after = (before--) + 5;
8
9      printf("before: %d\n", before);
10     printf("after: %d\n", after);
11
12     return 0;
13 }
```

Microsoft Visual Studio 디버그 콘솔

before: 4
after: 10

C:\Users\jinu\source\repos\ConsoleApplication1\Debug
0개),
이 창을 닫으려면 아무 키나 누르세요...

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 산술 연산자

▪ 단항 산술 연산자(++ , --)

- 아래의 예를 보면 postfix로 사용된 단항 연산자의 연산 순서에 대해 감을 잡을 수 있을 것이다.

```
1  #include <stdio.h>
2
3  int main() {
4      int before = 5;
5
6      printf("before: %d\n", before--);
7      printf("before: %d\n", before);
8
9      return 0;
10 }
```

Microsoft Visual Studio 디버그 콘솔

```
before: 5
before: 4
```

```
C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 관계 연산자

- 관계 연산자는 피 연산자 값들의 대소 동일성 등을 검사하는 연산자로서, 연산 결과는 true 또는 false를 생성하는 연산자이다.

- 예

- if (a == b) // a와 b의 값이 같으면 if 문의 조건이 참.
- if (a != b) // a와 b의 값이 다르면 if 문의 조건이 참.
- if (a > b) // a가 b보다 크면 if 문의 조건이 참.
- if (a >= b) // a가 b보다 크거나 같으면 if 문의 조건이 참.
- if (a < b) // a가 b보다 작으면 if 문의 조건이 참.
- if (a <= b) // a가 b보다 작거나 같으면 if 문의 조건이 참.

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 관계 연산자

- 관계 연산자는 참(true) 또는 거짓(false)이라는 값을 반환하는데, C 언어에서는 참은 1, 거짓은 0으로 출력된다.
- 아래의 프로그램을 살펴보자.

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b;
5
6      printf("%d\n", a = 3);
7      printf("%d\n", a == 3);
8      printf("%d\n", a != 3);
9      printf("%d\n", a > 3);
10     printf("%d\n", a < 3);
11
12     b = 4;
13     printf("%d\n", a < b);
14     printf("%d\n", a == 3 && b == 4);
15     printf("%d\n", a == 3 && b != 4);
16
17     return 0;
18 }
```

Microsoft Visual Studio 디버그 콘솔

```
3
1
0
0
0
0
1
1
0

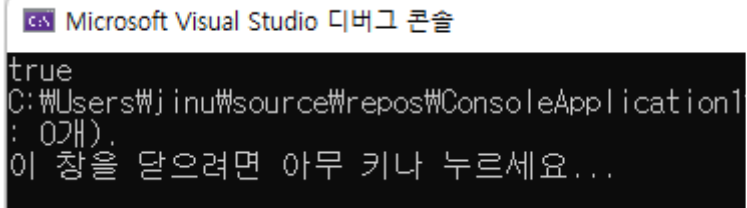
C:\Users\jinu\source\repos\ConsoleApplication1\
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 관계 연산자

- 관계 연산자의 반환값은 1 또는 0으로써 참과 거짓을 표시하지만, 아래의 if문과 같이 조건을 체크하는 부분 또는 관계 연산자나 논리 연산자의 피 연산자에서는 0이 아닌 모든 값은 모두 참이라고 간주한다.

```
1  #include <stdio.h>
2
3  int main() {
4      if (5) printf("true");
5      else printf("false");
6
7      return 0;
8  }
```



Microsoft Visual Studio 디버그 콘솔

```
true
C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 비트 연산자

- 비트 연산자는 피 연산자의 값을 2진수 비트열로 생각해서 연산을 수행한다.
- 실제로 컴퓨터 내부에서는 2진수로 표현되고, 이 2진수 값들간의 연산이 비트 연산이다.
- 비트 연산자는 영상 처리 (image processing)와 같은 분야에서 실제로 많이 사용되는 표현이다.
- 정수형 연산보다 수행 속도가 빠르기 때문이다.
- 비트 연산자는 아래와 같다.

연산자	기능
&	두 개의 피 연산자가 모두 1이면 결과가 1
	두 개의 피 연산자 중 하나라도 1이면 결과가 1
^	두 개의 피 연산자 중 하나만 1이면 결과가 1
~	피 연산자가 1개인 비트 연산자이다. 비트 값을 반전시킨다.
<<	비트열을 왼쪽으로 이동한다(shift left).
>>	비트열을 오른쪽으로 이동한다(shift right).

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 비트 연산자

- 예를 들어, $a = 17$, $b = 21$ 일 때, 2진수로 보면 $a = 00010001$, $b = 00010101$ 이다.

& 연산	^ 연산(exclusive or)	연산	~ 연산
<pre>00010001 & 00010101 ----- 00010001</pre>	<pre>00010001 ^ 00010101 ----- 00000100</pre>	<pre>00010001 00010101 ----- 00010101</pre>	<pre>~ 00010001 ----- 11101110</pre>

- 아래와 같이 비트 시프트(shift) 연산은 2의 지수승의 곱셈 또는 나눗셈과 같은 효과가 있다.
- 아래는 2비트를 좌 또는 우측으로 움직인 예이다.

<<연산	>>연산
<pre>00010001 << 2 ----- 01000100 = 68₍₁₀₎</pre>	<pre>00010001 >> 2 ----- 00000100 = 4₍₁₀₎</pre>

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 비트 연산자

- 비트 연산을 위한 프로그램 예를 살펴보자.

```
1  #include <stdio.h>
2
3  int main() {
4      unsigned int a = 60;
5      unsigned int b = 13;
6      int c = 0;
7
8      c = a & b;
9      printf("%u\n", c);
10
11     c = a | b;
12     printf("%u\n", c);
13
14     c = a ^ b;
15     printf("%u\n", c);
16
```

```
17     c = ~a;
18     printf("%u\n", c);
19
20     c = a << 2;
21     printf("%u\n", c);
22
23     c = a >> 2;
24     printf("%u\n", c);
25
26     return 0;
27 }
```

Microsoft Visual Studio 디버그 콘솔

```
12
61
49
4294967235
240
15

C:\Users\jinu\source\repos\ConsoleApplication1\
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 논리 연산자

- 논리 연산자는 논리(logic, 참(true) 또는 거짓(false)으로 표현할 수 있는 데이터)값을 피 연산자로 하는 연산자이다.
- 참/거짓으로 표현되는 데이터 형을 부울 데이터 형(Boolean data type)이라고 하는데, C 언어에서는 부울 데이터 형을 별도로 제공하지는 않는다.
- 다만 관계 연산자의 연산 결과로 나오는 참/거짓, 또는 다른 데이터 형(int, char, float 등)의 값이 0이면 거짓(false), 0이 아니면 모두 참(true)으로 간주한다.
- 아래에 논리 연산자&&와 ||의 예를 살펴보자.
 - if (a<b && b<c) // a < b이고 b < c이면 if 문의 조건이 참
 - If (a<b || b<c) // a < b거나 b < c이면 if 문의 조건이 참

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 논리 연산자

- 아래의 예제는 if 문과 관계 논리 연산자의 사용을 보이는 예이다.

```
1  #include <stdio.h>
2
3  int main() {
4      int input;
5
6      printf("Enter one integer number.\n");
7      scanf_s("%d", &input);
8
9      printf("Your entered number %d is .... \n", input);
10     if (input >= -9 && input <= 9) {
11         printf("single digit number.\n");
12     }
13     else if((input >= -10 && input <= -99) || (input >= 10 && input <= 99)) {
14         printf("double digit number.\n");
15     }
16     else {
17         printf("its absolute value is more than or equal to 100.\n");
18     }
19
20     return 0;
21 }
```

선택 Microsoft Visual Studio 디버그 콘솔

```
Enter one integer number.
200
Your entered number 200 is ....
its absolute value is more than or equal to 100.

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplic
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 조건 연산자

- 물음표 즉, ?로 표시되는 연산자도 있는데, 이를 '조건 연산자'라고 한다.
- 아래와 같이 사용할 수 있다.
- 아래의 결과로 변수 result는 1의 값을 가진다.

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b, result;
5
6      a = 20;
7      b = 10;
8
9      result = (a > b) ? 1 : 0;
10
11     printf("%d", result);
12
13     return 0;
14 }
```

Microsoft Visual Studio 디버그 콘솔

```
1
C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 조건 연산자

- 조건 연산자는 아래와 같이 3개의 피 연산자를 가진다.
- 조건을 위한 값(condition), 조건이 만족했을 때 가지는 값(x), 조건이 만족하지 않을 때 가지는 값(y).
 - (condition)? x : y;
- 이러한 조건 연산자는 아래와 같은 경우에 많이 사용된다.

```
1  #include <stdio.h>
2
3  int max(int a, int b) {
4      return (a > b) ? a : b;
5  }
6
```

```
7  int main() {
8      int a, b, result;
9
10     a = 20;
11     b = 10;
12
13     result = max(a, b);
14     printf("%d", result);
15
16     return 0;
17 }
```

Microsoft Visual Studio 디버그 콘솔

```
20
C:\Users\jinu\source\repos\ConsoleApplication
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

4. 공 좌·우 벽과 충돌 체크(조건문: if 문)

■ 콤마 연산자

- 콤마(,)도 연산자이다.
- 이제까지 아주 무의식적으로 사용했었는데, 변수를 동시에 선언하거나, 둘 이상의 문장을 하나의 문장으로 표현하는 경우에 사용하는 연산자이다.
- 이 연산자는 결합 순서가 left-to-right이다.
- 아래에서 콤마 연산자를 찾아보자.

```
1 | #include <stdio.h>
2 |
3 | int main() {
4 |     int a = 1, b = 2;
5 |
6 |     a++, b++;
7 |     printf("a: %d\n", a), printf("b: %d\n", b);
8 |
9 |     for (a = 1, b = 1; a < 10 && b < 10; a++, b++) {
10 |         printf("for body: %d, %d\n", a, b);
11 |     }
12 |
13 |     return 0;
14 | }
```

Microsoft Visual Studio 디버그 콘솔

```
a: 2
b: 3
for body: 1, 1
for body: 2, 2
for body: 3, 3
for body: 4, 4
for body: 5, 5
for body: 6, 6
for body: 7, 7
for body: 8, 8
for body: 9, 9

C:\Users\jinu\source\repos\ConsoleApplication1\
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

- x축, 즉 좌-우로만 움직이는 공은 재미가 없다.
- 공을 2차원 공간 상에서 마음대로 움직여 보자 2차원으로 움직이는 공도 1차원과 다른 점이 별로 없다.
- 다만 공의 x축 좌표와 더불어서 y축의 좌표도 동일한 방식으로 변경해주면 된다.
- 그렇지만 차원이 하나 더 늘면서 더욱 재미있는 공의 움직임을 볼 수 있다.
- 따라서, 이전 단계에서는 x축 관련 변수, xPos와 xDir 변수만 만들어서 사용했었는데, 이제는 y축 관련 변수들이 필요하다.

- int xpos; // 공의 x축 위치
- int ypos; // 공의 y축 위치
- int xDir; // 공의 x축(가로방향)의 이동거리
- int yDir; // 공의 y축(세로방향)의 이동거리

5. 공의 2차원 움직임(변수)

■ 새로운 y축 관련 변수 사용

- xDir과 yDir, 이 두 개의 값이 연결되어서 2차원 벡터로 동작하는 셈이다.
- 이 두 값을 동일한 값으로 설정하면 45도 각도로 공이 움직이게 된다.



```
1  #include "ofApp.h"
2  #define BALLSIZE 30
3
4  int xPos, yPos;
5  int xDir, yDir;
6
7  //-----
8  void ofApp::setup(){
9      ofSetWindowTitle("Ball Drawing Program");
10     ofSetWindowShape(1024, 768);
11     ofSetFrameRate(60);
12     ofBackground(ofColor::white);
13     ofSetColor(ofColor::black);
14     ofSetLineWidth(1.0);
15     ofFill();
16
17     xPos = ofGetWidth() / 2;
18     yPos = ofGetHeight() / 2;
19     xDir = yDir = 10;
20 }
21
22 //-----
23 void ofApp::update(){
24     xPos += xDir;
25     yPos += yDir;
26
27     /* 공의 크기를 고려하지 않는 충돌 검사
28     if (xPos < 0 || xPos > ofGetWidth()) xDir *= -1;
29     if (yPos < 0 || yPos > ofGetHeight()) yDir *= -1;
30     */
31
32     // 공의 크기를 고려한 충돌 검사
33     if (xPos < BALLSIZE || xPos > ofGetWidth() - BALLSIZE) xDir *= -1;
34     if (yPos < BALLSIZE || yPos > ofGetHeight() - BALLSIZE) yDir *= -1;
35 }
36
37 //-----
38 void ofApp::draw(){
39     ofCircle(xPos, yPos, BALLSIZE);
40 }
41
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

- 지금까지는 단순히 [Build] 메뉴로 프로그램을 컴파일했었다.
- 이제 컴파일 과정을 조금 더 자세하게 알아보자.
- C 언어의 컴파일은 아래와 같은 순서로 수행된다.
 - 전처리(pre-process): C 언어로 작성된 소스 코드에서 전처리 명령어를 처리하여 최종적인 코드로 재조합한 후,
 - 컴파일(compile): 컴파일을 통해서 목적 코드(object code)를 만들고(*.c → *.obj),
 - 링크(link): 목적 코드들을 모아서 서로 결합(link)하면 최종적인 실행 파일이 생성된다(*.obj → *.exe).



5. 공의 2차원 움직임(변수)

■ c 언어: 전처리기 지시자(compiler directive)

- 전처리기 명령들은 여러가지가 있는데, 대표적으로 많이 사용하는 전처리 명령어는 아래와 같다.
 - #include : 특정 파일의 코드 포함
 - #define: 매크로 정의
 - #undef: 정의된 매크로를 해제
 - #if, #ifdef, #ifndef, #else, #endif: 전처리 조건문
 - #pragma : 컴파일 옵션 설정
- #include
 - #include <파일명>: 컴파일러의 옵션에 의해 미리 지정된 위치에서 파일을 찾게 된다. 기본적으로 제공되는 C 라이브러리 대부분이 이에 해당한다.
 - #include "파일명" 파일명을 큰 따옴표로 표시하면 기본적으로는 소스 파일과 같은 폴더 안에서 해당 파일을 찾게 된다. 같은 폴더 안에 파일이 없다면, 정의된 위치(<파일명> 이렇게 한 경우와 동일하게)에서 파일을 찾는다. 즉, 소스 파일과 같은 폴더 안에 파일이 없으면 컴파일 옵션에서 지정한 위치에서 파일을 찾게 된다.

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리기 지시자(compiler directive)

■ #define

- 전처리 명령어 중에서 #define은 매크로(macro)라고도 한다. #define은 프로그램 내에서 자주 사용하는 문맥들을 간결하게 정의함으로써, 상수(constant) 등을 한꺼번에 바꿔야 할 때 매크로를 이용하면 편리하다.
- #define의 사용 방법은 아래와 같다.
 - #define [매크로 이름] [매크로 내용]
- 아래와 같은 매크로 선언에 의해서 PI라는 심볼이 3.14로 전처리에 의해서 자동으로 바뀌게 된다.
 - #define PI 3.14
- 즉, 아래 좌측과 같이 PI라는 이름의 매크로를 지정하면, 컴파일하기 전에 전처리에 의해서 아래의 우측과 같이 코드가 변환된다.
- 프로그래밍하는 사람의 눈에는 변환 과정이 보이지 않지만, 컴파일러가 컴파일 전에 "매크로 이름"을 "매크로 내용"으로 자동으로 바꾼 후에 컴파일해주는 것이다.

전처리 전	전처리 후
<pre>#define PI 3.14 void main(){ printf ("%d\n", PI); }</pre>	<pre>//#define PI 3.14 void main() { printf ("%d\n", 3.14); }</pre>

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- #defin에서 2줄 이상을 사용하려면 줄 끝에 백 슬래시로 줄 바꿈을 하면 된다.

```
1  #include <stdio.h>
2  #define MULTI_LINE_MACRO printf("11111\n"); \
3                                printf("22222\n");
4
5  int main() {
6      MULTI_LINE_MACRO
7
8      return 0;
9  }
```

Microsoft Visual Studio 디버그 콘솔

11111
22222

C:\Users\jinu\source\repos\ConsoleApplication1\Debug
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- #define은 숫자를 써도 되고 문자열을 써도 되고, 때로는 함수처럼 사용할 수도 있다.
- 매크로를 함수처럼 사용하면 컴파일 하기 전에 함수를 풀어서 변경해주니까 실제 함수를 사용하는 것보다 속도 면에서 약간의 이득을 얻을 수도 있다.
- 실행 시점에 함수 호출에 소요되는 시간을 줄일 수 있기 때문이다.
- 이러한 함수를 "매크로 함수"라고 한다.
- 아래 예를 보자 아래의 코드는 별 무리 없이 동작하는 듯이 보인다.

```
1  #include <stdio.h>
2  #define square(x) x*x
3
4  int main() {
5      printf("square(3) : %d \n", square(3));
6
7      return 0;
8  }
```

Microsoft Visual Studio 디버그 콘솔

```
square(3) : 9
C:\Users\jinu\source\repos\ConsoleApplication1\Debug
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 그런데 #define를 사용할 때는 주의해야 한다.
- 아래와 같은 파라미터를 사용하는 경우에는 의도와 다르게 치환되기 때문이다.
 - $\text{square}(2+3) \rightarrow 2 + 3 * 2 + 3$

```
1  #include <stdio.h>
2  #define square(x) x*x
3
4  int main() {
5      printf("square(2 + 3) : %d \n", square(2 + 3));
6
7      return 0;
8  }
```

Microsoft Visual Studio 디버그 콘솔

```
square(2 + 3) : 11
C:\Users\jinu\source\repos\ConsoleApplication1\
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 그래서 매크로를 선언할 때 아래와 같이 괄호를 이용해서 고쳐야 한다.

```
1  #include <stdio.h>
2  #define square(x) (x) * (x)
3
4  int main() {
5      printf("square(2 + 3) : %d \n", square(2 + 3));
6
7      return 0;
8  }
```

Microsoft Visual Studio 디버그 콘솔

square(2 + 3) : 25

C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...

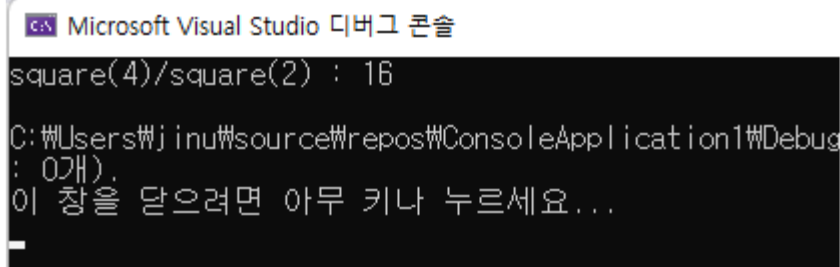
5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 그럼 아래의 경우는 어떻게 될까?

```
1  #include <stdio.h>
2  #define square(x) (x)*(x)
3
4  int main() {
5      printf("square(4)/square(2) : %d \n", square(4)/square(2));
6
7      return 0;
8  }
```



Microsoft Visual Studio 디버그 콘솔

```
square(4)/square(2) : 16
C:\Users\jinu\source\repos\ConsoleApplication1\Debug
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 이처럼 매크로 함수는 편리하고, 일반 함수를 호출하는 경우보다 시간면에서 더 효율적인 장점이 있지만 사용에 주의해야 한다.
- 위와 유사한 경우로써 아래의 경우도 한 번 보자.

```
1  #include <stdio.h>
2  #define add(x, y) x+y
3
4  int main() {
5      printf("add(1, 2) : %d \n", add(1, 2) );
6      printf("add(1, 2)*add(3, 4) : %d \n", add(1, 2)*add(3, 4));
7
8      return 0;
9  }
```

Microsoft Visual Studio 디버그 콘솔

```
add(1, 2) : 3
add(1, 2)*add(3, 4) : 11

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 위와 같이 2번째의 출력 결과는 잘못된 결과이다.
- 그 이유가 뭘까?
 - $\text{add}(1, 2) * \text{add}(3, 4)$
- 위의 문장은 아래와 같이 변형된다 우리가 의도한 결과가 아니다.
 - $1+2 * 3+4$
- 이러한 실수를 없애려면 아래와 같은 습관을 들이자.

```
1  #include <stdio.h>
2  #define add(x, y) (x+y)
3
4  int main() {
5      printf("add(1, 2) : %d \n", add(1, 2) );
6      printf("add(1, 2)*add(3, 4) : %d \n", add(1, 2)*add(3, 4));
7
8      return 0;
9  }
```

C:\ Microsoft Visual Studio 디버그 콘솔

```
add(1, 2) : 3
add(1, 2)*add(3, 4) : 21
```

```
C:\Users\jinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
0개).
이 창을 닫으려면 아무 키나 누르세요...
```


5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 매크로 함수는 다음과 같은 장점이 있다.
 - 매크로 함수를 사용하면 일반 함수에 비해서 실행 속도가 빠르다. 일반 함수를 호출할 때는 함수를 실행하는 시점에 함수 호출과 파라미터 복사와 관계된 스택(stack) 관련 작업으로 시간이 소요되기 때문이다.
 - 매크로 함수를 사용하면 자료형에 따라 별도로 함수를 정의하지 않아도 된다. 아래의 예를 보자 max() 매크로 함수는 파라미터로 정수값이든 실수값이든 상관없이 수행된다.

```
1  #include <stdio.h>
2  #define max(a, b) ((a)>(b)? (a) : (b))
3
4  int main() {
5      int maxi;
6      float maxf;
7      char maxch;
8
9      maxi = max(1, 3);
10     printf("%d\n", maxi);
11
12     maxf = max(1.0, 3.0);
13     printf("%f\n", maxf);
14
15     maxch = max('a', 'd');
16     printf("%c\n", maxch);
17
18     return 0;
19 }
```

Microsoft Visual Studio 디버그 콘솔

```
3
3.000000
d
C:\Users\jinu\source\repos\ConsoleApplication1
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 위의 프로그램을 일반 함수를 이용해서 구현해보자.
- 아래와 같이 서로 다른 자료형에 대해서 별도로 함수를 구현해야 한다.

```
1  #include <stdio.h>
2
3  int maxi(int a, int b) {
4      return ((a) > (b) ? (a) : (b));
5  }
6
7  float maxf(float a, float b) {
8      return ((a) > (b) ? (a) : (b));
9  }
10
11 int main() {
12     int i;
13     float f;
14
15     i = maxi(1, 3);
16     printf("%d\n", i);
17
18     f = maxf(1.1, 3.1);
19     printf("%f\n", f);
20
21     return 0;
22 }
```

```
Microsoft Visual Studio 디버그 콘솔
3
3.100000
C:\Users\jinu\source\repos\ConsoleApplication1\Debug
0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리 지시자(compiler directive)

■ #define

- 매크로는 이러한 장점이 있는 반면에 매크로를 정의하기가 까다롭고, 전처리된 파일을 일일이 살펴보는 것은 아주 불편한 작업이라서 디버깅하기가 힘들다는 단점이 있다.
- 그래서 아주 아주 조심스럽게 사용해야 한다.
 - 전처리 조건문 : 조건부 컴파일
- “전처리 조건문”은 “조건부 컴파일”이라고도 하는데, 전처리 조건문에 따라서 컴파일될 부분을 지정할 수 있다.
- 전처리 조건문은 아래와 같은 것들이 있다.
- 전처리 조건문을 사용하는 용도는 대량으로 코멘트를 처리하기 위해서 사용하기도 하고, 플랫폼이 다른 컴퓨터 간에 크로스 컴파일(cross compile)이 되도록 하기 위해서도 사용하는 등의 목적으로 사용한다.
- 사용법은 if 문과 유사하다.
 - #ifdef
 - #ifndef
 - #if
 - #else
 - #endif

5. 공의 2차원 움직임(변수)

■ c 언어: 전처리기 지시자(compiler directive)

■ #define

- 아래 예를 보자 아래와 같이 매크로 선언을 바꿈으로써 컴파일될 코드 부분을 다르게 할 수 있다.

```
1  #include <stdio.h>
2  #define TEST_1
3
4  int main() {
5      #ifdef TEST_1
6          printf("TEST_1\n");
7      #endif
8          printf("ETC\n");
9
10     return 0;
11 }
```

Microsoft Visual Studio 디버그 콘솔

```
TEST_1
ETC
C:\Users\jinu\source\repos\ConsoleApp1\ConsoleApp1\Debug>
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

```
1  #include <stdio.h>
2
3  int main() {
4      #ifdef TEST_1
5          printf("TEST_1\n");
6      #endif
7          printf("ETC\n");
8
9      return 0;
10 }
```

Microsoft Visual Studio 디버그 콘솔

```
ETC
C:\Users\jinu\source\repos\ConsoleApp1\ConsoleApp1\Debug>
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 컴퓨터 내부에서 정수(integer)는 실제로 어떻게 표현될까?
- 아래 프로그램은 입력받는 정수를 2진수로 변환해서 출력한 것과, 입력받은 값을 16진수(%X 포맷을 통해서)로 출력한 값이 동일함을 보인다.
- 즉, 아래의 결과로부터 컴퓨터 내부에서 정수가 2진수의 비트열로 저장됨을 눈으로 확인할 수 있다.

```
1  #include <stdio.h>
2
3  int decimal_binary(int n);
4
5  int main() {
6      int n;
7      scanf_s("%d", &n);
8      printf("%d in decimal = %d in binary\n", n, decimal_binary(n));
9      printf("%d in decimal = %X in hexadecimal\n", n, n);
10
11     return 0;
12 }
13
```

Microsoft Visual Studio 디버그 콘솔

```
126
126 in decimal = 1111110 in binary
126 in decimal = 7E in hexadecimal
```

```
C:\Users\j\inu\source\repos\ConsoleApplication1\WD
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

```
14 int decimal_binary(int n) {
15     int remains, i = 1, binary = 0;
16
17     while (n != 0) {
18         remains = n % 2;
19         n /= 2;
20         binary += remains * i;
21         i *= 10;
22     }
23     return binary;
24 }
```

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 이번에는 실수는 컴퓨터 내부에서 어떻게 표현되는지 살펴보자.
- 아래의 프로그램에서는 정수 표현을 살펴본 위와 비슷한 방식으로 표현하려고 하였지만 %X에 의한 실수의 16진수 표현이 '0'으로 출력되었다.
- 이는 "%X" 표현이 unsigned int형을 출력하기 때문에 실수값에는 옳게 작동하지 않기 때문이다.
- 따라서 다른 방식으로 16진수 표현을 출력해야 한다.

```
1  #include <stdio.h>
2
3  int main() {
4      float n = 1.5;
5
6      printf("%f in decimal is %X in hexadecimal\n", n, n);
7
8      return 0;
9  }
10
```

Microsoft Visual Studio 디버그 콘솔

1.500000 in decimal is 0 in hexadecimal

C:\Users\jinu\source\repos\ConsoleApplication1\Debug\Con
: 0개).

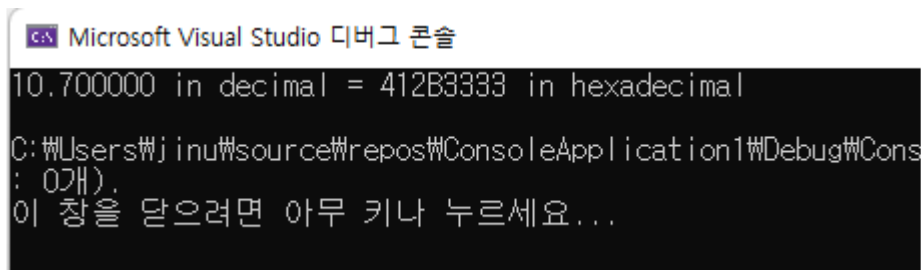
이 창을 닫으려면 아무 키나 누르세요...

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- C 언어의 union(공용체) 형식을 사용해서 실수값의 16진수를 출력해보자.
- 아래의 예에서 sample이라는 union type은 4바이트 메모리 공간인데, 이름을 i라고 사용하면 int 형으로 해석하고 이름을 f라고 사용하면 float 형으로 해석하라는 것이다.
- 즉, 동일한 공간을 서로 다른 데이터 형으로 사용할 수 있도록 하는 것이 공용체 자료형 (union)이다.

```
1  #include <stdio.h>
2
3  union sample {
4      int i;
5      float f;
6  };
7
8  int main() {
9      union sample n;
10     n.f = 10.7;
11     printf("%f in decimal = %X in hexadecimal\n", n.f, n.i);
12
13     return 0;
14 }
15
```



Microsoft Visual Studio 디버그 콘솔

10.700000 in decimal = 412B3333 in hexadecimal

C:\Users\Wjinu\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe
: 0개).

이 창을 닫으려면 아무 키나 누르세요...

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 이제 실수 표현의 정확도를 이야기해보자.
- 아래 프로그램은 실수 $1/3$ 을 30,000번 더하는 것과, 실수 $1/3$ 에 30,000을 곱하는 값의 차이를 보인다
- 이 두 값은 이론적으로는 같아야 한다.
- 그렇지만 실제로는 다를 수도 있다.

1/3의 3만 번 덧셈

$$\frac{1}{3} * 30,000 = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \cdots + \frac{1}{3} + \frac{1}{3} = \sum_{i=1}^{30,000} \frac{1}{3}$$

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 아래 프로그램은 실수 1/3을 30,000번 더하는 것과, 실수 1/3에 30,000을 곱하는 값의 차이를 보인다

```
1  #include <stdio.h>
2
3  double toAdd() {
4      float a = 1.0 / 3.0;
5      float sum = 0;
6      int i;
7
8      for (i = 0; i < 30000; i++) {
9          sum += a;
10     }
11     return sum;
12 }
13
```

```
14 double toMultiply() {
15     float a = 1.0 / 3.0;
16     float sum;
17
18     sum = a * 30000;
19     return sum;
20 }
21
22 int main() {
23     printf("%f\n", toAdd());
24     printf("%f\n", toMultiply());
25
26     return 0;
27 }
```

Microsoft Visual Studio 디버그 콘솔

```
9999.832031
10000.000000
```

```
C:\Users\jinu\source\repos\ConsoleApplication1\
: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 위의 결과와 같이 30,000번 더하는 것과, 30,000을 곱한 결과가 차이를 보인다.
- 이러한 차이는 2가지를 시사한다.
 - 실수값의 덧셈과 곱셈 연산은 내부적으로 다르게 작동한다. 즉, 정수값의 곱셈은 반복적인 덧셈 연산(예: 3×10 은 3을 10번 더하는 연산으로 수행)으로 수행되지만, 실수값의 곱셈은 단순한 덧셈 연산의 반복이 아니다.
 - 실수값은 컴퓨터에서 정확하게 표현되지 않을 수 있다.

5. 공의 2차원 움직임(변수)

■ c 언어: 변수(variables)의 #3

- 정수값 설명에서 오버 플로우(over-ftow)를 이야기했었다.
- 실수값에도 당연히 오버 플로우가 발생할 수 있다.
- 이제는 실수값에만 있는 언더 플로우(under-ftow)에 대한 예제를 보자.
- 아래의 예제는 1/2을 계속 2로 나누어가면서 중간값들을 출력하는데, 어느 시점에는 0이 되버린다.
- 즉, 표현할 수 있는 최소값이 정해져있다는 의미이다.
- 시스템 내부에서 표현할 수 있는 가장 작은 절대값을 넘어서는 것을 언더 플로우라고 한다.

```
1  #include <stdio.h>
2
3  int main() {
4      double sample = 1.0 / 2.0;
5      int i;
6
7      for (i = 0; i < 100; i++) {
8          printf("%f\n", sample);
9          sample = sample / 2;
10     }
11
12     return 0;
13 }
14
```

Microsoft Visual Studio 디버그 콘솔

```
0.000977
0.000488
0.000244
0.000122
0.000061
0.000031
0.000015
0.000008
0.000004
0.000002
0.000001
0.000000
0.000000
```

5. 공의 2차원 움직임(변수)

■ 실전연습문제

- 공이 상 · 하 · 좌 · 우 벽에 충돌할 때 공의 색깔이 바뀌도록 해보자.
- 즉 상단에 충돌하면 공이 빨간색으로 하단에 충돌하면 파란색 등으로 바뀌게 하자.

5. 공의 2차원 움직임(변수)

■ 실전연습문제

- 공이 상 · 하 · 좌 · 우 벽에 충돌할 때 공의 색깔이 바뀌도록 해보자.
- 즉 상단에 충돌하면 공이 빨간색으로 하단에 충돌하면 파란색 등으로 바뀌게 하자.

```
1  #include "ofApp.h"
2  #define BALLSIZE 30
3
4  int xPos, yPos;
5  int xDir, yDir;
6
7  //-----
8  void ofApp::setup(){
9      ofSetWindowTitle("Ball Drawing Program");
10     ofSetWindowShape(1024, 768);
11     ofSetFrameRate(60);
12     ofBackground(ofColor::white);
13     ofSetColor(ofColor::black);
14     ofSetLineWidth(1.0);
15     ofFill();
16
17     xPos = ofGetWidth() / 2;
18     yPos = ofGetHeight() / 2;
19     xDir = yDir = 10;
20 }
21
```

```
22  //-----
23  void ofApp::update(){
24      xPos += xDir;
25      yPos += yDir;
26
27      // 공의 크기를 고려한 충돌 검사
28      if (xPos < BALLSIZE || xPos > ofGetWidth() - BALLSIZE) {
29          if (xDir >= 0) ofSetColor(ofColor::forestGreen);
30          else ofSetColor(ofColor::paleVioletRed);
31      }
32      xDir *= -1;
33  }
34      if (yPos < BALLSIZE || yPos > ofGetHeight() - BALLSIZE) {
35          if (yDir >= 0) ofSetColor(ofColor::blue);
36          else ofSetColor(ofColor::red);
37      }
38      yDir *= -1;
39  }
40  }
41
42  //-----
43  void ofApp::draw(){
44      ofCircle(xPos, yPos, BALLSIZE);
45  }
```

5. 공의 2차원 움직임(변수)

■ 실전연습문제

- 공이 벽에 충돌하면 공의 크기가 커지거나 작아지도록 공의 크기를 변화시켜 보자.
- 즉 상단에 충돌하면 공이 커지고, 하단에 충돌하면 공이 작아지게 하자.

5. 공의 2차원 움직임(변수)

■ 실전연습문제

- 공이 벽에 충돌하면 공의 크기가 커지거나 작아지도록 공의 크기를 변화시켜 보자.
- 즉 상단에 충돌하면 공이 커지고, 하단에 충돌하면 공이 작아지게 하자.

```
1  #include "ofApp.h"
2  #define BALLSIZE_BIG 60
3  #define BALLSIZE_SMALL 10
4  #define BALLSIZE 30
5
6  int xPos, yPos;
7  int xDir, yDir;
8  int bsize;
9
10 //-----
11 void ofApp::setup(){
12     ofSetWindowTitle("Ball Drawing Program");
13     ofSetWindowShape(1024, 768);
14     ofSetFrameRate(60);
15     ofBackground(ofColor::white);
16     ofSetColor(ofColor::black);
17     ofSetLineWidth(1.0);
18     ofFill();
19
20     xPos = ofGetWidth() / 2;
21     yPos = ofGetHeight() / 2;
22     xDir = yDir = 10;
23     bsize = BALLSIZE;
24 }
25
26 //-----
27 void ofApp::update(){
28     xPos += xDir;
29     yPos += yDir;
30
31     // 공의 크기를 고려한 충돌 검사
32     if (xPos < BALLSIZE || xPos > ofGetWidth() - BALLSIZE) {
33         if (xDir >= 0) bsize = BALLSIZE_SMALL;
34         else bsize = BALLSIZE_BIG;
35
36         xDir *= -1;
37     }
38     if (yPos < BALLSIZE || yPos > ofGetHeight() - BALLSIZE) {
39         if (yDir >= 0) bsize = BALLSIZE_SMALL;
40         else bsize = BALLSIZE_BIG;
41
42         yDir *= -1;
43     }
44 }
45
46 //-----
47 void ofApp::draw(){
48     ofCircle(xPos, yPos, bsize);
49 }
```



Thank You
