

3장 더 나은 C로서의 C++ (2)

- ⌘ 인라인 함수
- ⌘ 참조(reference)의 이해
- ⌘ 함수에 대한 참조
- ⌘ 참조와 함수
- ⌘ 참조의 반환
- ⌘ linkage 지정
- ⌘ 선언과 정의
- ⌘ 객체지향 프로그래밍

1. 인라인 함수

예 : x, y 값 중 최소값을 반환하는 매크로와 함수 작성

```
// 매크로로 구현한 경우  
#define MIN(X, Y) ((X) < (Y) ? (X) : (Y))
```

X, Y 각각을 괄호() 안에 넣는 이유는?

```
int main(void)  
{  
    cout << MIN(4, 5) << endl;  
    cout << MIN((2 + 3), (1 + 2)) <<  
  
    return 0;  
}
```

main 함수는 동일!

```
// 함수로 구현한 경우  
int MIN(int X, int Y)  
{  
    return (X < Y ? X : Y);  
}
```

```
int main(void)  
{  
    cout << MIN(4, 5) << endl;  
    cout << MIN((2 + 3), (1 + 2)) << endl;  
  
    return 0;  
}
```

1. 인라인 함수

▣ 매크로

- 전처리 단계에서 해당 문장으로 대치 → 전처리기 담당
- 장점 : 실행 시 수행 속도 빨라짐
- 단점
 - 내용이 많은 경우 실행 파일의 크기가 커짐 → 내용이 짧은 경우 사용
 - 코드의 가독성(readability)이 떨어짐

▣ 함수

- 컴파일 시 해당 함수의 주소만 기록, 실행 시 해당 주소로 이동
- 단점 : 상대적으로 수행 속도가 느려짐

▣ 함수의 모양을 하면서도 매크로처럼 동작하게 하는 방법

- 인라인 함수 (inline function)
 - 작성은 함수처럼, 동작은 매크로처럼 → 수행 속도가 빨라짐
 - 인라인 함수로 만드는 방법 : 함수 정의 시 inline 키워드 추가
 - ✓ `inline int MIN(int X, int Y) { return X < Y ? X : Y; }`

1. 인라인 함수

⌘ 다음 중 제대로 동작하는 것은? ← 인라인 함수의 장점

- #define MULTI(X, Y) (X * Y)
- inline int MULTI(int X, int Y) { return (X * Y); }
- MULTI(1 + 2, 3 + 4)를 수행하는 경우!

⌘ 예 : Factorial을 계산하는 Fact 함수를 인라인 함수로.

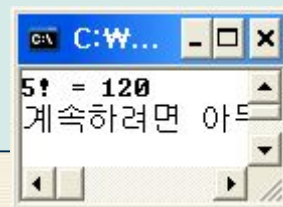
```
inline int Fact(int n) // 재귀 호출 함수
{
    return ((n <= 1) ? 1 : n * Fact(n-1));
}

int main(void)
{
    cout << "5! = " << Fact(5) << endl;

    return 0;
}
```

inline 선언은 인라인 함수로 동작하도록 요청하는 것
→ 실제 인라인 함수로의 동작(대치) 여부는
컴파일러에 의해 결정됨

똑똑한 컴파일러 : 120으로 대치
덜 똑똑한 컴파일러 : 5 * Fact(4)로 대치
멍청한 컴파일러 : 인라인 함수 포기



2. 참조의 이해

참조 변수

- 기존 변수의 또 다른 이름. 홀로 존재할 수 없음.
- 선언과 동시에 어떤 변수에 대한 다른 이름인지 초기화 필수
- 방법 : &
 - int Var = 2;
 - int &refVar = Var; // refVar와 Var는 동일한 변수

예 : 다음 프로그램의 출력 결과 분석

ref와 var1은
동일한 변수

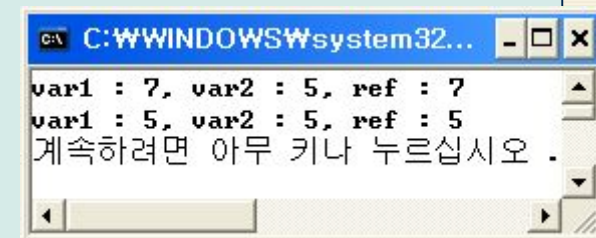
ref에 값 5 대입!

```
int main(void)
{
    int var1 = 3;
    int var2 = 5;
    int &ref = var1;

    ref = 7;
    cout << "var1 : " << var1
          << ", var2 : " << var2 << ", ref : " << ref << endl;

    ref = var2;
    cout << "var1 : " << var1
          << ", var2 : " << var2 << ", ref : " << ref << endl;

    return 0;
}
```



```
C:\WINDOWS\system32...
var1 : 7, var2 : 5, ref : 7
var1 : 5, var2 : 5, ref : 5
계속하려면 아무 키나 누르십시오 .
```

2. 참조의 이해

주의 사항 : 값이나 수식에 대한 참조 불가능

- `int &ref = a + b; // X`
- `int &ref = 4; // X`

예 : 출력 결과는 무엇인가?

```
int main(void)
{
    int var = 2;
    int &ref1 = var;
    int &ref2 = ref1;

    ref1 = 5;

    cout << "var  : " << var << endl;
    cout << "ref1 : " << ref1 << endl;
    cout << "ref2 : " << ref2 << endl;

    return 0;
}
```


3. 함수에 대한 참조

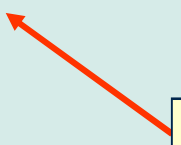
예 : 함수 포인터 만들기

```
int sum(int x, int y) { return (x + y); }

int main(void)
{
    int (*pSum)(int, int);           // 함수 포인터
    pSum = sum;                      // sum 함수를 가리킴

    cout << pSum(3, 4) << endl;

    return 0;
}
```



sum 함수와 동일하게 사용 가능

함수에 대한 참조

- `int (&rSum)(int, int) = sum;` // 기존 함수에 대한 참조
- `rSum(3, 4);` // sum 함수와 동일하게 사용 가능

3. 함수에 대한 참조

✦ 예 : 참조 변수(함수 참조 포함)의 다양한 사용 예

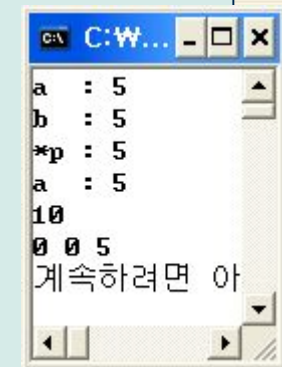
```
int sum(int x, int y) { return (x + y); }

int main()
{
    int a;
    int &b = a;
    int *p = &b;      // 여기서 &는 주소연산자임. 결국 a의 주소가 대입됨
    int &c = b;        // a, b, c는 동일한 변수
    c = 5;             // a = 5; 와 동일
    cout << "a : " << a << endl;
    cout << "b : " << b << endl;
    cout << "*p : " << *p << endl;
    cout << "c : " << c << endl;

    int (&rSum)(int, int) = sum;    // 함수에 대한 참조
    cout << rSum(c, 5) << endl;

    int ary[3] = { 0 };
    int (&rAry)[3] = ary;           // 배열에 대한 참조
    rAry[2] = c;
    cout << ary[0] << " " << ary[1] << " " << ary[2] << endl;

    return 0;
}
```



4. 참조와 함수

예 : 포인터를 이용한 swap 함수 작성

```
void swap(int *x, int *y)    // 참조에 의한 전달
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main(void)
{
    int a = 3, b = 4;
    swap(&a, &b);            // 주소 전달

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```

참조를 사용하여 swap 함수를 재작성한다면?!

4. 참조와 함수

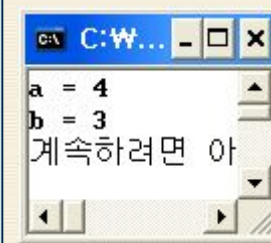
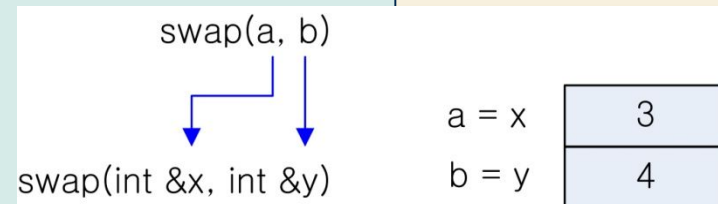
▣ 참조를 이용한 swap 함수 → 진정한 참조에 의한 전달

```
void swap(int &x, int &y)    // 진정한 참조에 의한 전달
{
    int temp = x;
    x = y;
    y = temp;
}

int main(void)
{
    int a = 3, b = 4;
    swap(a, b);              // 변수 자체 전달

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

    return 0;
}
```



▣ 형식매개변수로 참조 변수를 사용하는 경우

- 장점 : 속도 향상
- 장점이자 단점 : 형식매개변수 변경 시 실매개변수까지 변경됨
 - void sum(const int &x, const int &y); // x, y 값 변경 시 에러 발생

5. 참조의 반환

변수값 반환 원리

```
int sum(int x, int y)
{
    int z = x + y;
    return z;
}

int main(void)
{
    int a = 3, b = 4;
    int result = sum(a, b);
    cout << result << endl;

    return 0;
}
```

- ① z에 대한 임시 변수 생성
- ② 지역 변수 z 메모리 해제
- ③ 임시 변수 전달
- ④ 임시 변수 대입 후 임시변수 메모리 해제

5. 참조의 반환

■ 참조의 반환이 가능한가? 이것은 무엇을 의미하는가?

- 다음 예는 변수값의 반환임! 참조의 반환이 아님!

```
int sum(int x, int y)
{
    int z = x + y;
    int &refZ = z;      // refZ와 z는 동일한 변수
    return refZ;        // return z;와 동일, z값 반환
}
```

- 다음 예가 바로 참조의 반환임!

```
int &sum(int x, int y) // 참조의 반환
{
    int z = x + y;
    return z;          // z 변수 자체 반환
}
```

이 경우 z 변수 자체를 반환함
그러나 지역 변수는 사라지게
되므로 유효하지 않은 예.
→ 참조를 반환하는 유효한 예는?
→ 어디에 쓰는가?

5. 참조의 반환

참조를 반환하는 유효한 예

- 전역 변수의
참조 반환

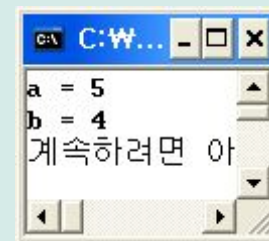
```
int z;  
  
int &sum(int x, int y)  // 참조의 반환  
{  
    z = x + y;  
    return z;           // z 변수 자체 반환  
}
```

- 참조로 전달된 변수의 참조 반환

```
int &min(int &x, int &y)  // 매개변수 참조 전달 및 참조 반환  
{  
    return ((x < y) ? x : y);  
}
```

```
int main(void)  
{  
    int a = 3, b = 4;  
    min(a, b) = 5;  
    cout << "a = " << a << endl;  
    cout << "b = " << b << endl;  
  
    return 0;  
}
```

함수 호출이 대입문의 왼쪽에 올 수 있는가?
→ 참조(변수 그 자체)를 반환하기 때문에
가능!
→ 결국 **a = 5;**와 동일



5. 참조의 반환

✦ 참조의 반환은 어디에 쓰는가?

- 7장 연산자 오버로딩

✦ 참조 반환의 사용 예

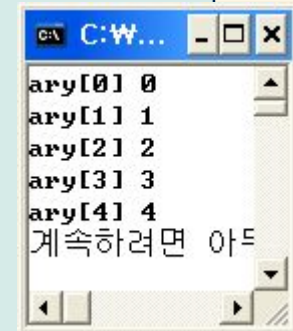
```
int &GetArray(int *ary, int index) // 참조 반환
{
    return ary[index];           // index에 해당하는 변수 자체 반환
}

int main(void)
{
    int i;
    int ary[5];

    for (i = 0; i < 5; i++)
        GetArray(ary, i) = i;    // ary[i] = i;와 동일

    for (i = 0; i < 5; i++)
        cout << "ary[" << i << "] " << GetArray(ary, i) << endl;

    return 0;
}
```



5. 참조의 반환

다음 프로그램의 문제점은 무엇인가?

```
int &GetVar(void)
{
    int var = 5;
    return var; // 지역 변수 var 자체 반환
}
```

```
int main(void)
{
    int &value = GetVar();
    value = 3;

    cout << value << endl;

    return 0;
}
```

실행은 될 수도 있으나
지역 변수의 참조 반환은
올바른 사용이 아님

6. linkage 지정

- # C와 C++의 함수 호출문에 대한 컴파일 결과 → linkage
 - C : 함수 이름만 저장 ← 함수 오버로딩을 허용하지 않으므로 구별 가능
 - C++ : 함수 이름 + 매개변수 정보 저장 ← 함수 오버로딩 허용
→ mangled name
- # C++에서 C 라이브러리의 함수를 호출하는 경우
 - C : `int func(int a);` → `func`라는 이름으로 연결 가능
 - C++ : `func(3);` → `func`+매개변수정보로 linkage 작성
 - 함수 호출과 함수 사이의 mismatch 발생 → 호출 불가!
- # C++ 컴파일 시 함수 호출문을 C 형식으로 컴파일하기
 - linkage 지정 : extern “C” 사용
 - `extern “C” int func(int a);` → 함수명(`func`)만으로 linkage 작성

6. linkage 지정

linkage 지정 방법

```
extern "C" int func(int a);
```

기존 라이브러리에 지정

```
extern "C" {  
    #include "clib.h"  
}
```

한꺼번에 지정

```
extern "C" {  
    int func(int a);  
    int myfunc(void);  
    void mysort(int *ary, int count);  
}
```

C++에서는 어떻게 C 라이브러리의 사용이 가능할까?

- 예 : #include <stdio.h>
- stdio.h 파일 내에는 이미
extern "C" 선언이 되어 있음
 - 필요에 따라 C++ 형식의
컴파일 시에는 자동으로 붙게 됨

stdio.h

```
#ifdef __cplusplus  
extern "C" {  
#endif  
.....  
#ifdef __cplusplus  
}  
#endif
```

C++ 컴파일 시
_cplusplus가
자동 정의됨

6. linkage 지정

- ✦ 다음 프로그램에서 어느 한 함수에만 extern “C”를 지정하는 경우와 두 함수 모두 extern “C”를 지정하는 경우에 대해 컴파일해 보라.

```
int sum(int x, int y)
{
    return (x + y);
}

double sum(double x, double y)
{
    return (x + y);
}

int main(void)
{
    cout << sum(1, 2) << endl;
    cout << sum(1.1, 2.2) << endl;

    return 0;
}
```

7. 선언과 정의

⌘ 선언(declaration)

- 식별자의 타입 정보를 컴파일러에게 알림
- 컴파일이 되기 위해서는 식별자 사용 이전에 선언이 되어야 함
- 선언의 예

```
int a;  
extern int count = 1;    // extern이라도 값이 대입된다면 선언 & 정의가 됨  
int func(int x) { return (x * x); } // 함수의 몸체가 오는 경우 선언 & 정의  
struct Point { int x; int y; };  
struct XCount {  
    int x;  
    static int count;    // static 멤버변수는 선언일 뿐 정의는 아님  
};  
int XCount::count = 1;    // static 멤버변수는 명시적으로 외부에 정의해야 됨  
                        // 초기화는 값은 없어도 됨 => 이 경우 0으로 초기화  
XCount anX;  
enum { up, down };  
namespace NS { int var; }  
namespace NAnother = NS; // NAnother와 NS는 동일한 네임스페이스임
```

이 경우를 제외하고 모두 선언인 동시에 정의!

7. 선언과 정의

⌘ “정의이면 선언이다” 성립O, “선언이면 정의이다” 성립X

⌘ 선언이지만 정의가 아닌 경우

- ① 함수 몸체를 기술하지 않은 함수, 즉, 함수 프로토타입
- ② extern 변수. 단, 초기화 구문을 포함하면 정의가 됨
- ③ extern 함수. 단, 함수 몸체를 포함하면 정의가 됨
- ④ 클래스 또는 구조체 내에 선언된 static 멤버 변수
- ⑤ 클래스명 또는 구조체명 선언 (구조체 전체를 의미하는 것이 아님)
- ⑥ typedef 선언
- ⑦ using 선언

정의가 아닌 선언의 예

```
extern int count;  
int func(int);  
struct Point;  
typedef int MyInt;  
using NS::var;
```


7. 선언과 정의

▣ 컴파일이 되기 위해서는 ← 파일 단위

- 파일 내에 사용하고자 하는 식별자의 선언이 **1번 이상** 등장
- 여러 번 등장할 경우 식별자의 타입은 동일해야 함

```
extern int count;  
extern int count;    // (O) 선언은 2번 이상 나올 수 있음  
extern char count;   // (X)
```

▣ 링크가 되기 위해서는 ← 프로젝트 단위

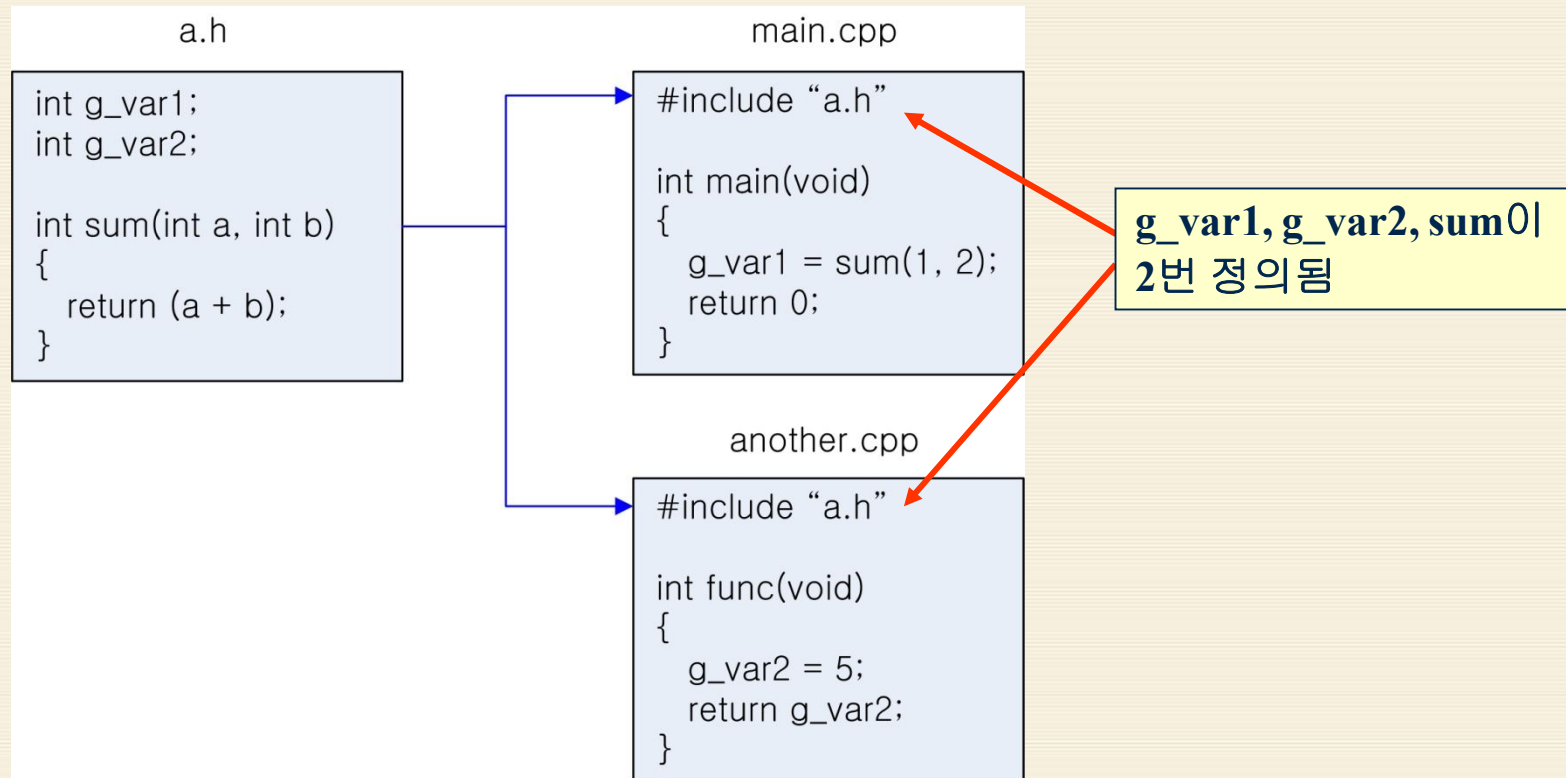
- 해당 식별자에 대한 정의가 소스 파일들 중 어딘가에 **단 한번** 등장
- ODR(One-Definition Rule)

```
int x;  
int x;    // (X) 중복해서 정의되어 있음, C 언어에서는 됨
```

```
int x = 5;  
int x = 3; // (X), C 언어에서도 안됨
```

7. 선언과 정의

다중 파일 프로그래밍에서 ODR을 위반하는 예



7. 선언과 정의

다중 파일 프로그래밍에서 ODR의 달성 예

```
#ifdef __MYMAIN
#define EXTERN
#else
#define EXTERN extern
#endif
```

header.h

```
EXTERN int count;    // __MYMAIN 정의 여부에 따라 extern 추가 또는 삭제
struct Point {
    int x;
    int y;
};
```

main.cpp

```
#define __MYMAIN
#include "header.h"

int main(void)
{
    Point P1;
    count = 1;
}
```

point.cpp

```
#include "header.h"

void func(int a)
{
    Point P2;
    count = 5;
}
```

7. 선언과 정의

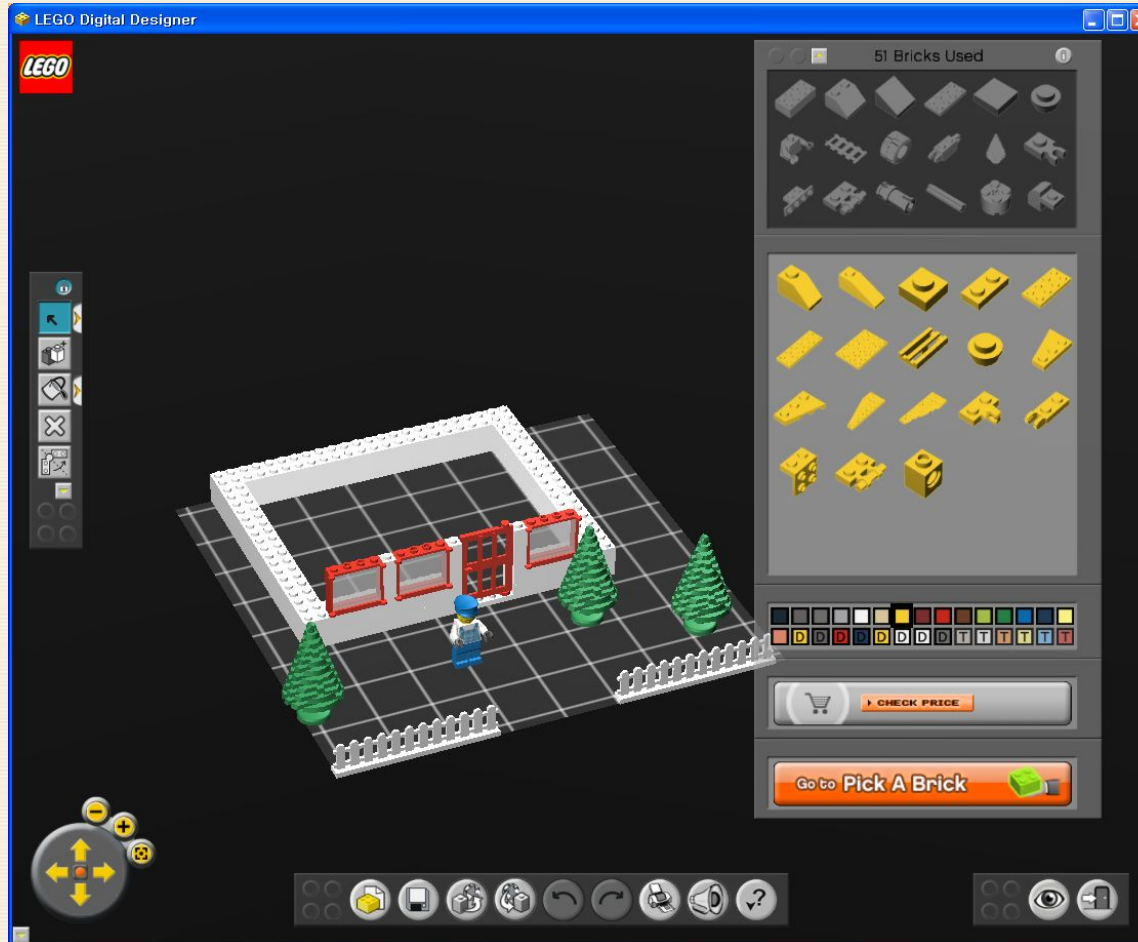
ODR의 예외

- 프로그래밍의 편의를 위해 다음에 대한 정의는 ODR을 따르지 않아도 됨
 - 구조체, 열거체, 인라인 함수, 클래스, 템플릿
- 하나의 소스 파일 내에는 단 한 번의 정의가 와야 됨
- 하나의 프로그램 내에는 2번 이상 정의 가능
 - 단, 구조가 동일해야 함
 - 다음의 경우에는 서로 다른 구조체로 인식

```
a.cpp : struct Point { int x; int y; };  
b.cpp : struct Point { int x; };
```

8. 객체지향 프로그래밍

예 1 : “LEGO Digital Designer”



개별 블록들을 조립하여
더 큰 구조물 형성

개별 블록을 잘 만들어
놓는다면 ... !



8. 객체지향 프로그래밍

■ 예 2 : 컴퓨터 조립

- 메인 보드, CPU, 메모리, 그래픽카드, 사운드카드, 하드디스크 ...
- 기존 부품을 새로운 부품으로 교체 가능 → 인터페이스만 같다면

■ 객체지향 프로그래밍

- 데이터 추상화(data abstraction) → 추상 데이터형
 - 데이터와 메서드로 구성됨
 - C++에서의 추상 데이터형 : 클래스 → 생성된 변수 : 객체
 - ✓ 레고 : 클래스 - 블록을 찍어내는 틀, 객체 - 각 블록
 - ✓ 컴퓨터 : 클래스 - 부품을 찍어내는 틀, 객체 - 각 부품
- 상속(inheritance)
 - 컴퓨터 : 기존 부품과 유사한 새로운 부품을 만들 경우
 - ✓ 기존 부품의 기능을 그대로 유지하고 추가 또는 변경된 부분만 수정
 - ✓ 기존 부품도 동작하고 새로운 부품도 동작함

8. 객체지향 프로그래밍

▣ 객체지향 프로그래밍 (계속)

■ 다형성(polymorphism)

- 컴퓨터 : 부품 A로부터 상속받아 부품 B, 부품 C를 만든 경우, 모두 f라는 기능을 가지고 있다. 메인 보드 입장에서는 A에 대한 f를 수행하면 실제 부품이 A, B, C냐에 따라 각 부품에 맞는 f가 수행되도록 하고 싶다.
- 동적 바인딩 기술 적용

▣ C++의 일반화 프로그래밍(generic programming)

■ 객체지향 프로그래밍과는 또 다른 특징

- 하나의 함수나 클래스가 특정 타입에 구속되지 않고 다양한 타입에 적용될 수 있도록 하는 것

■ 방법 : 템플릿

▣ 교재의 구성

- 클래스와 객체 : 4~7장
- 상속 : 8장
- 다형성 : 9장
- 템플릿 : 10장