Storage Class in C++

- The storage class specifiers control two independent properties of the name: its **storage duration** and **its linkage**

Keyword (기본값)	Storage Duration	Linkage	Linkage 예	Lifetime	Visibility (name lookup)	Initial Vaule
auto (지역변수)	automatic (block-scope)	no linkage	- 함수 파라미터 - non-extern block-scope 변수	Block ({ })	Local	Garbage
extern (모든함수, 파일범위 변수)	static (file-scope)	external linkage	- non-static 함수 - extern 변수 - file-scope non-static 변수	Whole Program	translation unit 외부 (다른 소스 파일)	Zero
static	static (file-scope)	internal linkage	- static file-scope 식별자(함수, 변수)	Whole Program	translation unit 내부 (동일한 소스 파일)	Zero
register	automatic	no linkage		Block ({ })	Local	Garbage

https://en.cppreference.com/w/c/language/storage_duration

block, file scope and external linkage

// local예제_1.cpp
(block scope)

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

void globalBFunction() ;

void localFunction() {
  //scope of the 'localCounter' begins
  int localCounter = 10 ;
  localCounter ++ ;
} //scope of the 'localCounter' ends
```

// static예제_1.cpp
(file scope)

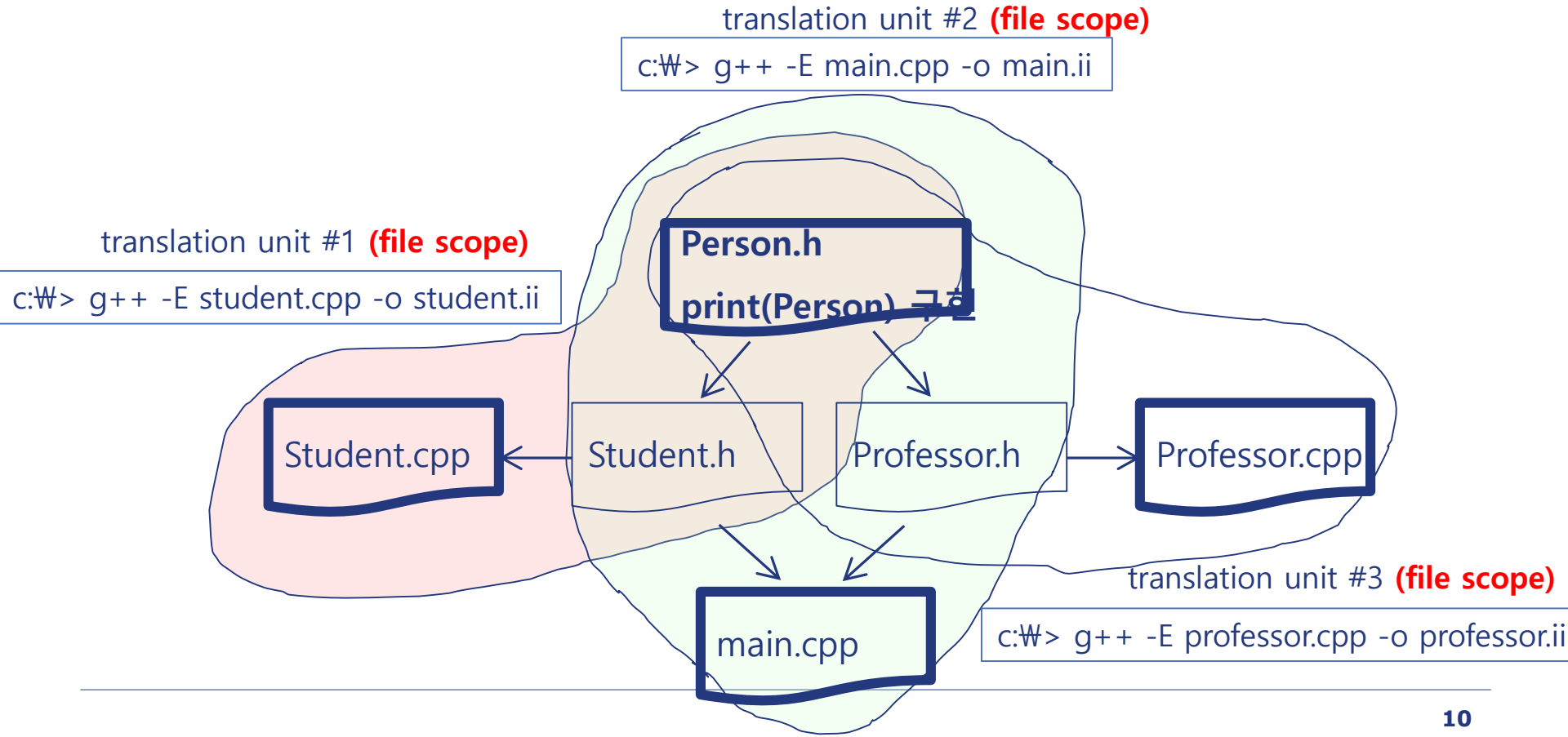
```
# include <iostream>
using namespace std ;
void globalBFunction() ;

int globalCounter ;
static int staticCounter = 10 ;
static void staticFunction() {
  staticCounter ++ ;
}
void globalAFunction() {
  globalCounter ++ ;
  staticFunction() ;
  cout << globalCounter << '\t'
       << staticCounter << endl ;
}
int main() {
  localFunction();
  globalAFunction() ;
  globalBFunction() ;
}
/* end of this translation unit,
end of file scope */
```

// static예제_2.cpp
(external linkage)

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

extern int globalCounter ;

static int staticCounter = 20 ;
static void staticFunction() {
  staticCounter ++ ;
}

void globalBFunction() {
  globalCounter ++ ;
  staticFunction() ;
  cout << globalCounter << '\t'
       << staticCounter << endl ;
}
```

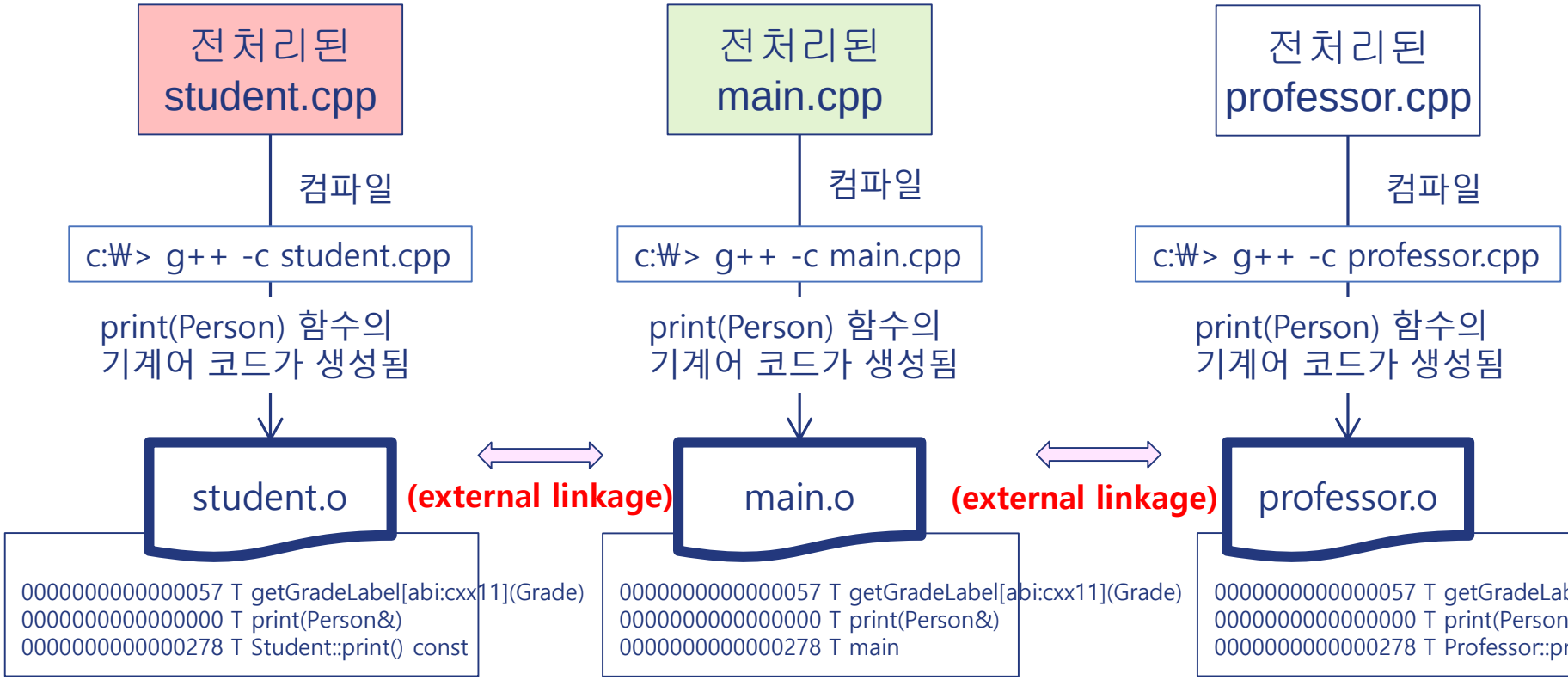
(추가) 전처리, 컴파일, 그리고 링크

- 소스 코드(cpp 파일)을 전처리(preprocess) 하면 #include 했던 헤더 파일의 내용이 모두 합쳐진 translation unit 파일을 생성함
- cpp 파일이 3개면 세개의 translation unit 파일이 생성됨

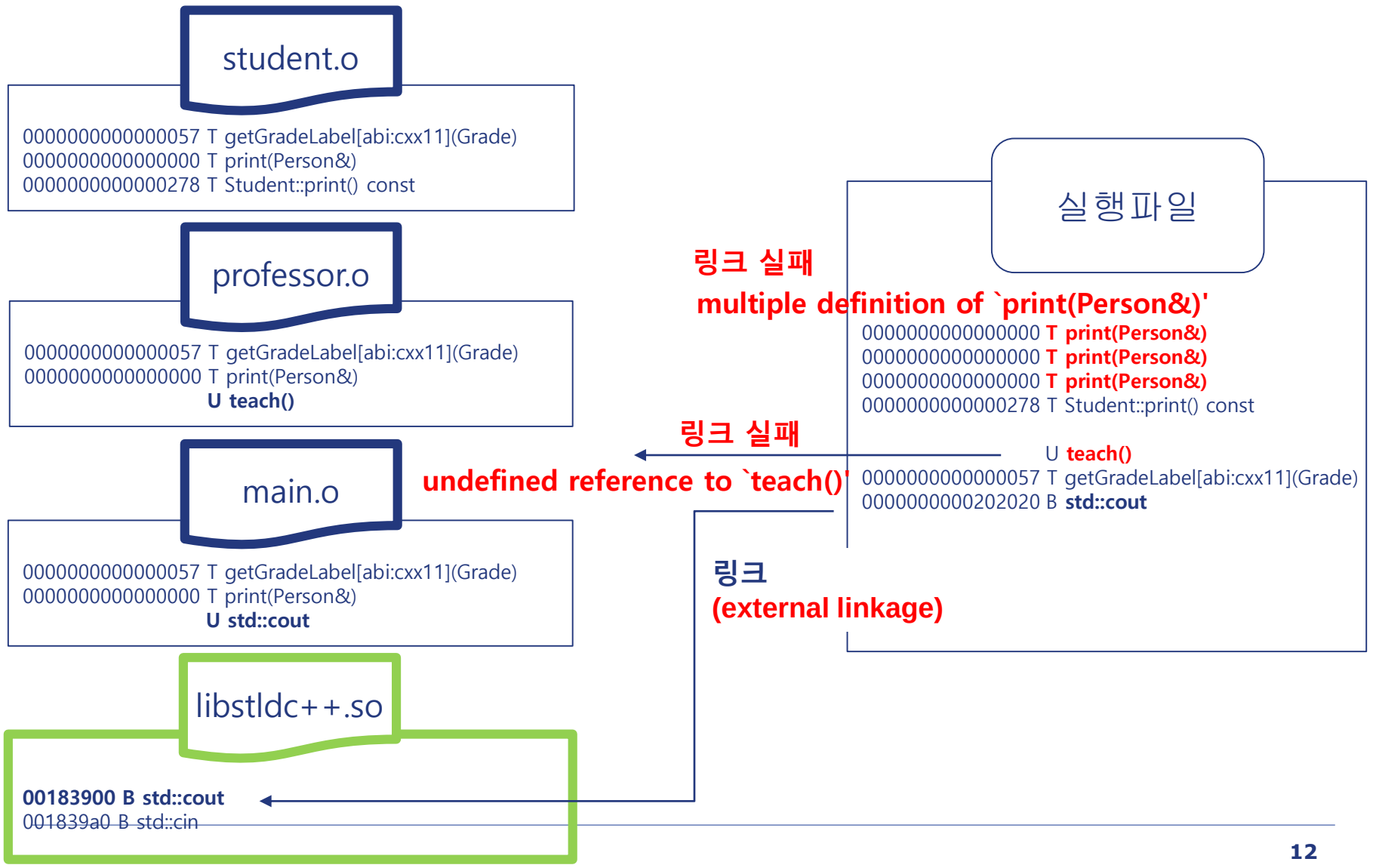


전처리, 컴파일, 그리고 링크

- 전처리된 소스코드(cpp 파일)은 컴파일 후 목적 파일(object code)을 생성함 – Windows 에서는 *.obj, Ubuntu에서는 *.o 파일
- 목적 코드 안에는 심볼 테이블 (함수명 등), 기계어 코드 등이 있음



전처리, 컴파일, 그리고 링크



Good Design: 헤더 파일

- self-contained : 이 헤더 파일 하나만 include 하면 컴파일이 정상적으로 되어야 함
  - 템플릿 및 인라인 함수에 대한 정의를 선언과 동일한 파일에 배치
- #define 헤더 가드를 사용하라! : 여러 번 포함되는 것을 방지
  - <PROJECT>_<PATH>_<FILE>_H_
- 사용하는 헤더 파일만 include 하라!
  - 다른 곳에 정의된 symbols(함수, 변수 등)을 참조하는 경우 해당 symbols 의 선언 혹은 정의를 제공하는 헤더 파일만 포함해야 함
  - transitive inclusions에 의존하면 안됨 – foo.h에 bar.h가 include 됐어도 cpp 파일에서 bar.h에 있는 symbols를 사용하면 직접 bar.h를 include 해야함
- forward declarations: 가능한 경우 전방 선언을 사용하지 말고, 대신 필요한 헤더를 include 하라!
  - 전방 선언 장점: 컴파일 타임 절약, 불필요한 재컴파일을 줄여줌
  - 전방 선언 단점: 종속성을 숨겨 재컴파일 시 건너 뛸 수 있음, #include 문과 반대되는 전방 선언은 자동화 도구가 symbol 을 발견하는 것을 어렵게 함, 라이브러리가 나중에 변경됐을 때 전방 선언이 깨질 수 있음
- 인라인 함수: 10 줄 이하일 때 사용하자! (overuse 경계)

https://google.github.io/styleguide/cppguide.html#Header_Files

정적 지역 변수

- 값이 함수 호출 간에 유지

```
# include <iostream>
using namespace std ;
int incorrectAdd(const int val) {
  int sum = 0 ; // 일반 지역 변수
  sum += val ;
  return sum ;
}
int correctAdd(const int val) {
  static int sum = 0 ; // 정적 지역 변수
  sum += val ;
  return sum ;
}
int main() {
  cout << incorrectAdd(3) << endl ;     // 3
  cout << incorrectAdd(5) << endl ;     // 5
  cout << correctAdd(3) << endl ;       // 3
  cout << correctAdd(5) << endl ;       // 8
}
```

정적 지역 변수 예

```
# include <iostream>
using namespace std ;
int incrementCounter() {
   static int counter = 0 ;
   counter ++ ;
   return counter ;
}
int sumUpTo(const int upTo) {
   int sum = 0 ;
   for ( int i = 1 ; i <= upTo ; i ++ ) sum += i ;
   return sum ;
}
int main() {
   while ( true ) {
      int number ;
      cin >> number ;
      if ( number <= 0 ) break ;
      const int counter = incrementCounter() ;
      cout << counter << " integer: " << number << endl ;
      const int sum = sumUpTo(number) ;
      cout << "Sum of 1 to " << number << " : " << sum << endl ;
   }
}
```

정적 지역 변수 예

```
enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
string getGradeLabel_1(const Grade grade) { // 방법 1: 낮은 성능 유발
  const string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
  return gradeLabels[grade-1] ;
}
string& getGradeLabel_2(const Grade grade) { // 방법 2: 안전하지 않음
  string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
  return gradeLabels[grade-1] ;
}
string& getGradeLabel_3(const Grade grade) { // 방법 3: 가장 바람직함
  static string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
  return gradeLabels[grade-1] ;
}
int main() {
  cout << getGradeLabel_1(FRESH) << endl ;
  cout << getGradeLabel_2(FRESH) << endl ;
  cout << getGradeLabel_3(FRESH) << endl ;
}
```