

반환 값

- ❖ 함수는 약속된 기능을 수행하고 그 결과를 호출함수에게 반환하는 것이 일반적이다

호출 함수	피 호출 함수
<pre>int main() { const float result = getAverageUpTo(10) ; cout << result << endl ; // 5.5 }</pre>	<pre>float getAverageUpTo(const int max) { int sum = 0 ; for (int i = 1 ; i <= max ; i ++) sum += i ; return static_cast<float>(sum) / max ; }</pre>

반환 값의 전달: return 문

- ❖ 피호출 함수에서는 return문을 이용해서 호출 함수에게 자신의 수행 결과를 반환

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

int find(const int* const data, const int size, const int target) {
    for ( int i = 0 ; i < size ; i ++ )
        if ( data[i] == target ) return i ; // 발견된 경우
    return -1 ; // 발견되지 않은 경우
}

int main() {
    const int SIZE = 6 ;
    int scores[SIZE] = {10, 20, 30, 40, 50, 60} ;
    cout << find(scores, SIZE, 50) << endl ; // 4
    cout << find(scores, SIZE, 55) << endl ; // -1
}
```

반환 값의 사용 여부

- ❖ 피호출함수가 반환한 값을 호출 함수에서는 활용
- ❖ 피호출함수의 반환 값을 호출 함수에서 사용하지 않는 것도 가능

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

int readPositiveNumbers(vector<int>& numbers) {
    while ( true ) {
        int n ;
        cin >> n ;
        if ( n <= 0 ) break ;
        numbers.push_back(n) ;
    }
    return numbers.size() ;
}
```

```

int printLargeNumbers(const vector<int>& numbers, const int base) {
    typedef vector<int>::const_iterator iterator ;
    int count = 0 ;
    for ( iterator it = numbers.begin() ; it != numbers.end() ; ++ it ) {
        if ( *it > base ) {
            cout << *it << endl ;
            count ++ ;
        }
    }
    return count ;
}

int main() {
    vector<int> positiveNumbers ;
    // 피호출 함수가 반환한 값을 호출함수에서 사용하고 있지 않음
    readPositiveNumbers(positiveNumbers) ;
    printLargeNumbers(positiveNumbers, 50) ;
}

```

Good Design:

오류를 뜻하는 반환값은 호출함수에서 처리

- ❖ 반환한 값이 정상적인 계산/처리 결과가 아니라 비정상적인 처리 즉 오류를 뜻하는 경우에는 호출함수에서 이를 처리

```
int find(const int* const data, const int size, const int target) {  
    for ( int i = 0 ; i < size ; i ++ )  
        if ( data[i] == target ) return i ; // 발견된 경우  
    return -1 ; // 발견되지 않은 경우  
}
```

```
char* values = "Hello, C++!" ;  
char target = '+' ;  
int result = find(values, target) ;  
if ( result == -1 ) // -1 즉 오류 상황 발생에 대한 처리를 해야 함  
    cout << target << " Not Found in " << values << endl ;
```

반환 값 타입: 기본 타입

- ❖ 자신의 기능에 따라서 int, float, char, bool 등의 기본 타입의 반환 값을 호출 함수에게 전달

```
bool isEqualPoint(const Point& pt1, const Point& pt2) ;  
int getSum(const int max) ;  
float getAverage(const int max) ;  
int readPositiveNumbers(vector<int>& numbers) ;  
int printLargeNumbers(const vector<int>& numbers, const int base) ;
```

- ❖ void 타입은 아무 값도 반환하지 않음

```
void upgrade(Grade& grade) ;  
void printStudent(const Student& st) ;
```

반환 값 타입: 나열형

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;

Grade upgrade(const Grade now) {
    return ( now == SENIOR ) ? SENIOR : static_cast<Grade>(now+1) ;
}

int main() {
    const Grade current = FRESH ;
    const Grade next = upgrade(current) ;
    cout << current << 'Wt' << next << endl ; // 1 2
}
```

반환 값 타입: 구조체

❖ 구조체를 함수에서 반환하는 것도 가능

```
struct Student {  
    string name ;  
    Grade grade ;  
    float gpa ;  
};  
  
Student readStudent() ;  
  
Student findStudentByName(const vector<Student>& sts, const string& name) ;
```


반환 값 타입: 구조체

```
Student readStudent() { // 구조체 Student 반환
    string name ; int grade ; float gpa ;
    cin >> name >> grade >> gpa ;
    Student st ;
    st.name = name ;
    st.grade = static_cast<Grade>(grade) ;
    st.gpa = gpa ;
    return st ;
}

Student findStudentByName(const vector<Student>& sts, const string& name) {
    for ( unsigned int i = 0 ; i < sts.size() ; i ++ )
        if ( sts[i].name == name ) return sts[i] ;
    return Student() ;
}

int main() {
    cout << "Enter the number of students !" << endl ;
    int no ; cin >> no ;
    vector<Student> students(no) ; // 구조체 Student의 vector 생성
    for ( int i = 0 ; i < no ; i ++ )
        students[i] = readStudent() ; // 반환된 구조체 Student의 대입
    // 반환된 구조체 Student를 이용한 stJames의 초기화
    Student stJames = findStudentByName(students, "James") ;
}
```

반환 값 타입: 구조체

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
Grade upgrade(const Grade now) {
    return ( now == SENIOR ) ? SENIOR : static_cast<Grade>(now+1) ;
}

string getGradeLabel(const Grade grade) {
    const string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
    return gradeLabels[grade-1] ;
}

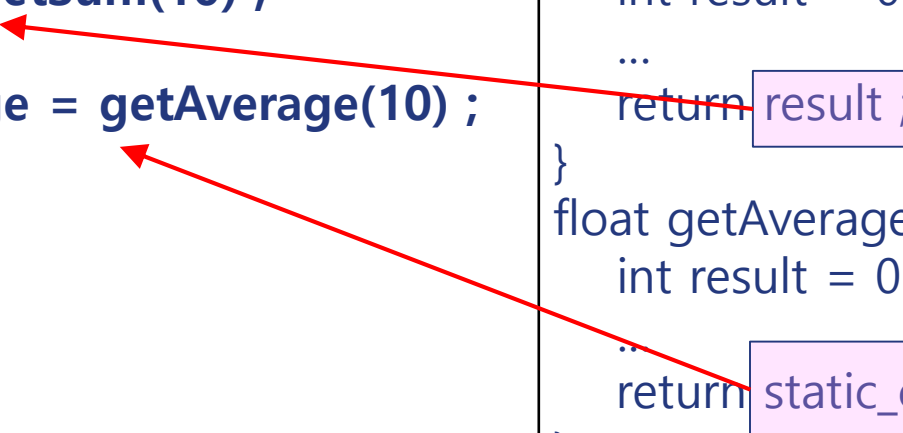
int main() {
    const Grade current = SOPHOMORE ;
    const Grade next = upgrade(current) ;
    cout << getGradeLabel(current) << endl ;
    cout << getGradeLabel(next) << endl ;
}
```

반환 값 전달: 값에 의한 전달

- ❖ 기본적으로 피호출 함수에서 생성한 반환 값은 호출 함수로 복사된다. 즉 값에 의한 전달이 발생한다

```
int main() {  
    int sum = getSum(10) ;  
  
    float average = getAverage(10) ;  
}
```

```
int getSum(const int max) {  
    int result = 0 ;  
    ...  
    return result ;  
}  
  
float getAverage(const int max) {  
    int result = 0 ;  
    ...  
    return static_cast<float>result / max ;  
}
```



반환 값 전달: 값에 의한 전달

- ❖ 값으로 복사되는 방식은 반환 타입이 구조체/클래스와 같이 많은 메모리를 사용할 때 문제를 유발

```
Student readStudent() ;
```

```
Student findStudentByName(const vector<Student>& sts, const string& name) ;
```

```
string getGradeLabel(const Grade grade) ;
```

반환 값 전달: 참조에 의한 전달

```
Student& readStudent() ;  
Student& findStudentByName(const vector<Student>& sts, const string& name) ;  
string& getGradeLabel(const Grade grade) ;
```

❖ 상황에 따라서 위험한 상황을 초래

참조에 의한 반환의 문제: 예 1

```
struct Student {
    string name ;
    Grade grade ;
    float gpa ;
};

Student& readStudent() {
    string name ; int grade ; float gpa ;
    cin >> name >> grade >> gpa ;
    Student st ;
    st.name = name ;
    st.grade = static_cast<Grade>(grade) ;
    st.gpa = gpa ;
    return st ; // 지역변수 st에 대한 참조(주소)가 반환되므로 위험함
}

const Student& findStudentByName(
    const vector<Student>& sts, const string& name) {
    for ( unsigned int i = 0 ; i < sts.size() ; i ++ )
        if ( sts[i].name == name ) return sts[i] ; // 안전함
    return Student() ; // 위험함
}

int main() {
    Student newSt = readStudent() ;
    vector<Student> students ;
    const Student& stJames = findStudentByName(students, "James") ;
}
```

참조에 의한 반환의 문제: 예 2

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;

Grade upgrade(const Grade now) {
    return ( now == SENIOR ) ? SENIOR : static_cast<Grade>(now+1) ;
}

string& getGradeLabel(const Grade grade) {
    string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
    return gradeLabels[grade-1] ; // 지역 변수에 대한 참조를 반환하므로 위험함
}

int main() {
    const Grade current = FRESH ;
    const Grade next = upgrade(current) ;
    cout << getGradeLabel(current) << endl ;
    cout << getGradeLabel(next) << endl ;
}
```

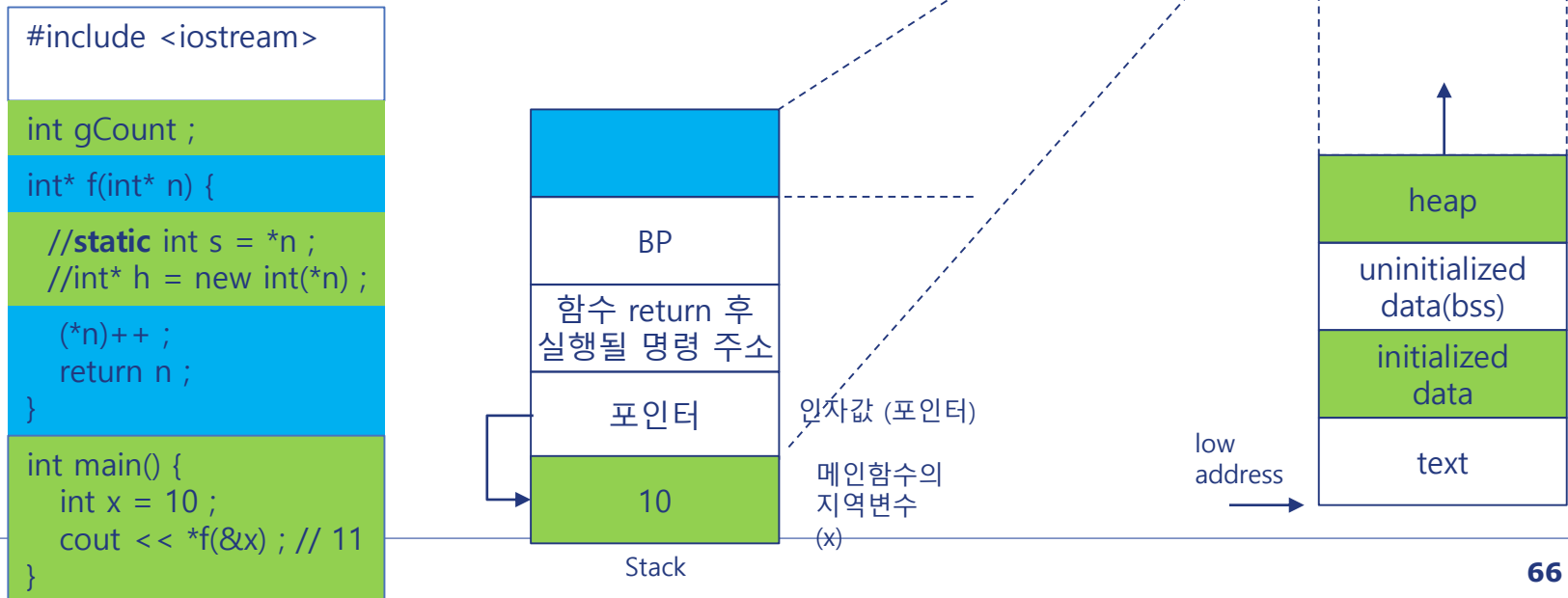
Good Design – 지역변수에 대한 참조/포인터 반환은 위험

지역변수에 대한 참조 반환	지역변수에 대한 주소(포인터) 반환
<pre>int& f() { int local ; ... return local ; }</pre>	<pre>int* g() { int local ; ... return &local ; }</pre>

안전한 참조 및 포인터 반환

❖ 호출 함수에서도 유효한 메모리에 대한 참조 또는 포인터를 전달하는 것은 안전

- global (전역) 변수에 대한 참조/포인터 반환
- static 지역 변수에 대한 참조/포인터 반환
- 할당된 동적 메모리(new 키워드)에 대한 포인터 반환
- 참조 매개변수에 대한 참조/포인터 반환



전역 변수에 대한 참조/포인터 반환

전역변수에 대한 참조 반환	전역변수에 대한 포인터 반환
<pre>int gCount ; int& f() { gCount ++ ; return gCount ; } int main() { int x = f() ; cout << x ; // 1 }</pre>	<pre>int gCount ; int* f() { gCount ++ ; return &gCount ; } int main() { int* x = f() ; cout << *x ; // 1 }</pre>

정적 지역 변수에 대한 참조/포인터 반환

- ❖ 정적 지역 변수에 대한 참조 및 포인터를 반환하고 이를 호출 함수에서 정적 지역 변수의 값을 접근하는 것은 안전

static 지역 변수에 대한 참조 반환	static 지역 변수에 대한 포인터 반환
<pre>int& f() { static int n = 0 ; n ++ ; return n; } int main() { cout << f() ; // 1 cout << f() ; // 2 }</pre>	<pre>int* f() { static int n = 0 ; n ++ ; return &n ; } int main() { cout << *f() ; // 1 cout << *f() ; // 2 }</pre>

할당된 동적 메모리에 대한 포인터 반환

- ❖ 피호출함수에서 `new`를 이용하여 동적으로 메모리를 할당하고 이에 대한 포인터를 호출함수에게 전달하는 것은 안전

```
int* f(int n) {  
    int* const p = new int[n] ;  
    for ( int i = 0 ; i < n ; i ++ )  
        p[i] = i+10 ;  
    return p ;  
}  
  
int main() {  
    const int* const pVal = f(10) ;  
    cout << pVal[0] ; // 10  
    delete [] pVal ;  
}
```

참조 매개변수에 대한 참조/포인터 반환

- ❖ 전달받은 참조 매개변수에 대한 참조/포인터를 반환하는 것은 안전

참조 매개변수에 대한 참조 반환	참조 매개변수에 대한 포인터 반환
<pre>int& f(int& n) { n++ ; return n; } int main() { int x = 10 ; cout << f(x) ; // 11 }</pre>	<pre>int* f(int& n) { n++ ; return &n ; } int main() { int x = 10 ; cout << *f(x) ; // 11 }</pre>

안전한 참조 및 포인터 반환 1

```
Student* readStudent() {           // 할당된 동적 메모리에 대한 포인터 반환
    string name ; int grade ; float gpa ;
    cin >> name >> grade >> gpa ;
    Student* const pSt = new Student ;
    pSt->name = name ;
    pSt->grade = static_cast<Grade>(grade) ;
    pSt->gpa = gpa ;
    return pSt ;
}
// 참조 매개변수에 대한 포인터 반환
const Student* findStudentByName(const vector<Student*> & sts, const string& name) {
    for ( unsigned int i = 0 ; i < sts.size() ; i ++ )
        if ( sts[i]->name == name ) return sts[i] ;
    return 0 ;
}
int main() {
    cout << "Enter the number of students !" << endl ;
    int no ; cin >> no ;
    vector<Student*> students(no) ;
    for ( int i = 0 ; i < no ; i ++ ) students[i] = readStudent() ;
    const Student* stJames = findStudentByName(students, "James") ;
    for ( unsigned int i = 0 ; i < students.size() ; i ++ ) delete students[i] ;
}
```

안전한 참조 및 포인터 반환 2

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

enum Grade { FRESH=1, SOPHOMORE, JUNIOR, SENIOR } ;
Grade upgrade(const Grade now) {
    return ( now == SENIOR ) ? SENIOR : static_cast<Grade>(now+1) ;
}

// static 지역 변수에 대한 참조 반환
string& getGradeLabel(const Grade grade) {
    static string gradeLabels[] = { "Fresh", "Sophomore", "Junior", "Senior" } ;
    return gradeLabels[grade-1] ;
}

int main() {
    const Grade current = FRESH ;
    const Grade next = upgrade(current) ;
    cout << getGradeLabel(current) << endl ;
    cout << getGradeLabel(next) << endl ;
}
```

Good Design : 함수를 범용적으로 사용하려면 스마트 포인터 보다 raw 포인터나 참조를 인자로 사용하라! (F.7)

- ❖ 스마트 포인터를 인자로 사용하면 소유권이 이전되거나 공유됨
 - lifetime semantics 을 의도적으로 표현한 경우에만 인자로 사용함
- ❖ 함수 호출 시 스마트 포인터만 사용할 수 있는 제약이 발생
- ❖ shared_ptr 의 경우에 성능상의 비용이 추가로 발생 (참조 카운터)

```
// accepts any int*  
void f(int*);
```

```
// can only accept ints for which you want to transfer ownership  
void g(unique_ptr<int>);
```

```
// can only accept ints for which you are willing to share ownership  
void g(shared_ptr<int>);
```

```
// doesn't change ownership, but requires a particular ownership of the caller  
void h(const unique_ptr<int>&);
```

```
// accepts any int  
void h(int&);
```


Good Design : 함수가 객체의 소유권을 가진다는 것을 표현하기 위해 `unique_ptr`를 파라미터로 사용하라! (R.32)

- ❖ 함수 호출시 객체의 소유권을 이전한다고 강제 및 문서화함을 의미

```
void sink(unique_ptr<widget> widget); // takes ownership of the widget  
  
void uses(widget*);                // just uses the widget
```

- ❖ 함수가 스마트 포인터로 다른 객체를 가리킨다고 표현하기 위해서는 `unique_ptr<T>&`를 사용하라! (R.33)
 - `reset`: 포인터 또는 스마트 포인터가 다른 객체를 참조하도록 하는 것

```
void reset(unique_ptr<widget>&); // "will" or "might" reset pointer
```

Good Design : 함수가 부분 소유자라고 표현하기 위해 `shared_ptr`를 파라미터로 사용하라! (R.34)

- ❖ 함수 호출시 객체의 소유권을 공유한다는 것을 명시적으로 표현

```
void share(shared_ptr<widget>);           // share -- "will" retain refcount  
  
void may_share(const shared_ptr<widget>&); // "might" retain refcount  
  
void reseal(shared_ptr<widget>&);         // "might" reseal ptr
```

- ❖ 함수가 `shared_ptr` 포인터를 `reseal` 한다는 것을 표현하기 위해서는 `shared_ptr<T>&` 를 파라미터로 사용하라! (R.35)
 - `reseal`: 포인터 또는 스마트 포인터가 다른 객체를 참조하도록 하는 것

```
void share(shared_ptr<widget>);           // share -- "will" retain refcount  
  
void may_share(const shared_ptr<widget>&); // "might" retain refcount  
  
void reseal(shared_ptr<widget>&);         // "might" reseal ptr
```

Good Design : 엘리어싱된 스마트 포인터로부터 얻은 포인터나 참조를 파라미터로 전달하지 마라! (R.37)

- ❖ 위반 시 참조 카운터를 잃어 버리고, 댕글링 포인터를 사용할 수 있음
- ❖ 함수 호출 체인 (예, f()->g()->h()) 에서는 raw 포인터나 참조를 아래로 전달하는 것을 선호해야 함
- ❖ 호출 체인의 최상단에서 스마트 포인터로부터 포인터나 참조를 획득하여 객체의 생존을 계속 유지해 줌

```
// global (static or heap), or aliased local ...
shared_ptr<widget> g_p = ...;

void f(widget& w) {
    g();
    use(w); // A
}

void g() {
    // 여기가 마지막 참조면 함수 종료 시 객체는 사라짐
    g_p = ...; // oops!
}
```

```
void my_code() {
    // 참조를 1 증가
    auto pin = g_p;

    // 괜찮음
    f(*pin);

    // 괜찮음
    pin->func();
}
```

엘리어싱 포인터 (aliasing pointer): 두 개의 포인터가 같은 곳을 가리킴. f (int* a, int* b) 함수 내부에서 a, b가 같은 곳을 가리킨다면? 함수의 기능은 정상적으로 동작하는가?

Q&A
