

구현 단계

경북대학교 IT대학 컴퓨터학부

이우진

053-950-6378, woojin@knu.ac.kr

Reference : Steve McConnell, Code Complete, 2nd Edition, Microsoft Press, 2004

Contents

- 변수 및 자료형 사용시 유의사항
- 구조적 프로그래밍 기법 개념
- 고급 루틴 작성하기
- 방어적 프로그래밍

변수의 범위

- 변수에 대한 참조를 지역화하라
- 변수의 “수명”을 가능한 짧게 유지하라

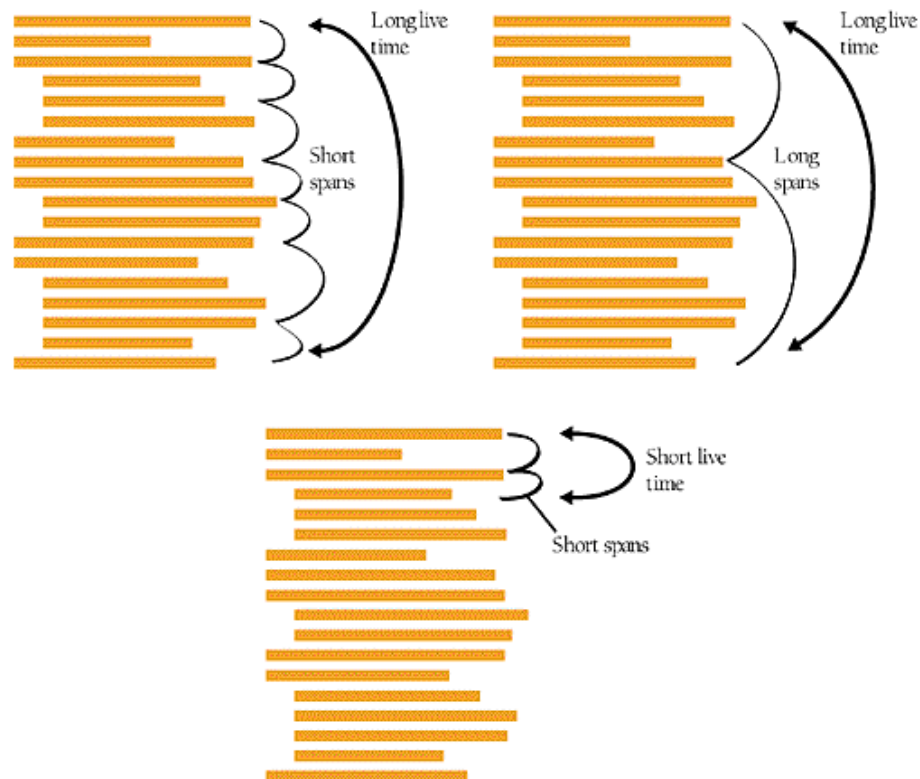


Figure 10-1. "Long live time" means that a variable is live over the course of many statements. "Short live time" means it's live for only a few statements.

짧은 변수 수명 예

Example 10-7. Java Example of Variables with Excessively Long Live Times

```
1  // initialize all variables
2  recordIndex = 0;
3  total = 0;
4  done = false;
   ...

26 while ( recordIndex < recordCount ) {
27   ...
28   recordIndex = recordIndex + 1;    <-- 1
   ...

64 while ( !done ) {
   ...
69   if ( total > projectedTotal ) {
70     done = true;    <-- 3
```

긴 변수 수명 코드

짧은 변수 수명 코드

Example 10-8. Java Example of Variables with Good, Short Live Times

```
   ...
25 recordIndex = 0;    <-- 1
26 while ( recordIndex < recordCount ) {
27   ...
28   recordIndex = recordIndex + 1;
   ...
62 total = 0;    <-- 2
63 done = false;    <-- 2
64 while ( !done ) {
   ...
69   if ( total > projectedTotal ) {
70     done = true;
```

범위 최소화를 위한 지침

- 루프에 사용되는 변수는 함수 시작이 아니라 루프 앞에서 초기화
- 변수를 사용하기 전까지 변수에 값을 할당하지 않음
- 연관된 명령문들을 그룹화함(아래 예제 참조)
- 연관된 명령문 그룹들을 별도의 루틴으로 나눔
- 처음에는 변수 영역을 최대한 제한하고 필요한 경우 영역을 늘임

```
void SummarizeData(...) {  
    ...  
    GetOldData( oldData, &numOldData );          <-- 1  
    GetNewData( newData, &numNewData );           |  
    totalOldData = Sum( oldData, numOldData );    |  
    totalNewData = Sum( newData, numNewData );    |  
    PrintOldDataSummary( oldData, totalOldData,  |  
numOldData );  
    PrintNewDataSummary( newData, totalNewData,  |  
numNewData );  
    SaveOldDataSummary( totalOldData, numOldData );  
    SaveNewDataSummary( totalNewData, numNewData );  
    <-- 1  
    ...  
}
```

```
void SummarizeData( ... ) {  
    GetOldData( oldData, &numOldData );          <-- 1  
    totalOldData = Sum( oldData, numOldData );    |  
    PrintOldDataSummary( oldData, totalOldData,  |  
numOldData );  
    SaveOldDataSummary( totalOldData, numOldData );  
    <-- 1  
    ...  
    GetNewData( newData, &numNewData );          <-- 2  
    totalNewData = Sum( newData, numNewData );    |  
    PrintNewDataSummary( newData, totalNewData,  |  
numNewData );  
    SaveNewDataSummary( totalNewData, numNewData );  
    <-- 2  
    ...  
}
```

편의성 vs 지적 관리성



편의성

- 프로그램 작성에 중점
- 변수 범위 최대화
- 전역변수 사용
- 프로그램 변경 어려움

지적관리성

- 프로그램 읽는 데 중점
- 변수 범위 최소화
- 매개변수 사용
- 프로그램 변경 쉬움

숫자 유의사항

- 매직 넘버를 피하라
 - 100, 47524와 같은 숫자는 모두 명명된 상수로 대신 사용
 - 만약 필요하다면, 0과 1은 그냥 사용함
- 0으로 나눔 오류를 미연에 방지함
- 형 변환을 명확하게 수행
 - $y = x + (\text{float}) i$, $y = x + \text{static_cast}\langle\text{float}\rangle i$
- 서로 다른 형을 비교하지 않음
- 컴파일러의 경고에 주의를 기울여라

정수 유의사항

- 정수 나눗셈을 검사함
 - $7/10 = 0.7$ (??)
- 정수 오버플로우를 검사함
 - n-bits int : $(-2^{n-1} \sim 2^{n-1} - 1)$
 - n-bits unsigned int : $(0 \sim 2^n - 1)$
- 중간 결과에서의 오버플로우를 검사함
 - 아래 product의 값은 ?
 - Long 형 정수나 부동소수점형으로 변경

Example 12-1. Java Example of Overflow of Intermediate Results

```
int termA = 1000000;  
int termB = 1000000;  
int product = termA * termB / 1000000;
```


부동소수점 유의사항 (1/2)

- 서로 크기가 매우 다른 수를 더하거나 빼지 않음
 - (32-bit 부동소수점) $1,000,000.00 + 0.01 = 1,000,000.00$
 - (32-bit 부동소수점) $5,000,000.02 - 5,000,000.01 = 0.0$
- 동치 비교를 피함 (아래 코드의 결과는 ?)

Example 12-2. Java Example of a Bad Comparison of Floating-Point Numbers

```
double nominal = 1.0;          <-- 1
double sum = 0.0;

for ( int i = 0; i < 10; i++ ) {
    sum += 0.1;                 <-- 2
}

if ( nominal == sum ) {        <-- 3
    System.out.println( "Numbers are the same." );
}
else {
    System.out.println( "Numbers are different." );
}
```

부동소수점 유의사항 (2/2)

- 실수 값의 동치 비교는 차이값으로 판단

```
final double ACCEPTABLE_DELTA = 0.00001;
boolean Equals( double Term1, double Term2 ) {
    if ( Math.abs( Term1 - Term2 ) < ACCEPTABLE_DELTA ) {
        return true;
    }
    else {
        return false;
    }
}
```

- 라운딩 오류를 예측한다
 - Float ➔ Double 형으로 변경
 - 부동소수점을 정수형 변수로 변경함
 - 달러 계산시, 100을 곱하여 정수 변환후 계산

문자와 문자열 유의사항

- 매직 문자와 문자열을 사용하지 않음

```
if ( input_char == 0x1B ) ...  
if ( input_char == ESCAPE ) ...
```

- 문자열을 CONSTANT+1로 선언함

```
char name[ NAME_LENGTH + 1 ] = { 0 };  
for ( i = 0; i < NAME_LENGTH; i++ )  
    name[ i ] = 'A';
```

- 문자열을 널(null)로 초기화함

```
char EventName[ MAX_NAME_LENGTH + 1 ] = { 0 };
```

- C언어에서 포인터 대신 문자 배열을 사용
- 끝없는 문자열을 피하기 위해 strcpy() 대신 strncpy() 를 사용함

Boolean 변수 유의사항

- 프로그램을 문서화하기 위해서 boolean 변수 사용

```
if ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) ||  
    ( elementIndex == lastElementIndex )  
    ) {
```

```
finished = ( ( elementIndex < 0 ) || ( MAX_ELEMENTS < elementIndex ) );  
repeatedEntry = ( elementIndex == lastElementIndex ); if ( finished ||  
repeatedEntry ) {
```

- 복잡한 테스트를 단순하게 하기 위해 boolean 변수 사용

Example 12.8.

```
If ( ( document.AtEndOfStream() ) And ( Not inputError ) ) And _  
    ( ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES ) ) And _  
    ( Not ErrorProcessing() ) Then
```

```
allDataRead = ( document.AtEndOfStream() ) And ( Not inputError )  
legalLineCount = ( MIN_LINES <= lineCount ) And ( lineCount <= MAX_LINES )  
If ( allDataRead ) And ( legalLineCount ) And ( Not ErrorProcessing() )
```

열거형 유의사항

- 가독성을 향상시키기 위해서 열거형을 사용하라
 - “if chosenColor == 1” vs. “if chosenColor == Color_Red”

```
int result = RetrievePayrollData( data, true, false, false, true );
```

```
int result = RetrievePayrollData(  
    data,  
    EmploymentStatus_CurrentEmployee,  
    PayrollType_Salaried,  
    SavingsPlan_NoDeduction,  
    MedicalCoverage_IncludeDependents  
);
```

- 신뢰성 향상을 위해 열거형을 사용하라
 - enum Color {Color_Red=1, Color_Green=2, Color_Blue=3 }
 - Color bgColor = Color_Green;
 - Color bgColor = 5; (컴파일 오류 ??)

배열의 유의사항

- 배열의 모든 인덱스가 배열의 경계 내에 있는지 확인
- 배열의 마지막 위치를 확인
- 배열 대신 스택, 큐와 같은 컨테이너의 사용을 고려
 - 배열 : 임의 접근, 스택과 큐 : 접근 제한
- 다차원 배열에서는 첨자 사용 순서가 정확한지 확인
 - Array[i][j] vs. Array[j][i]
- 중첩된 루프에서 인덱스(i,j) 혼선 주의
- 배열을 다루기 위한 ARRAY_LENGTH() 매크로 사용

Example 12-27. C Example of Defining an *ARRAY_LENGTH()* Macro

```
#define ARRAY_LENGTH( x ) (sizeof(x)/sizeof(x[0]))
```

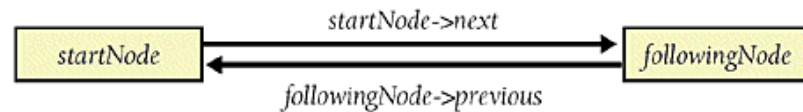
포인터 유의사항(1/2)

- 포인터 사용은 오류 유발 가능성이 가장 높은 분야의 하나임
- 포인터가 아닌 다른 기술을 사용하라
 - 다른 방법이 없을 때 최후의 수단으로 포인터 사용
- 포인터 연산을 루틴이나 클래스로 고립시킴
 - 연결리스트 예 : NextLink(), PreviousLink(), InsertLink()
- 포인터를 선언과 동시에 정의하라
- 포인터를 사용하기 전에 포인터의 타당성 검사 실시
- 포인터가 참조하는 변수를 사용 전에 타당성 검사 실시
- 임시 메모리를 할당한다
 - 메모리 부족 시, 임시 메모리 해제 후 작업을 정리하고 종료함
- 포인터와 일반 변수를 함께 사용하지 마라
 - `if (ps > value), sum = *ps + pg;`

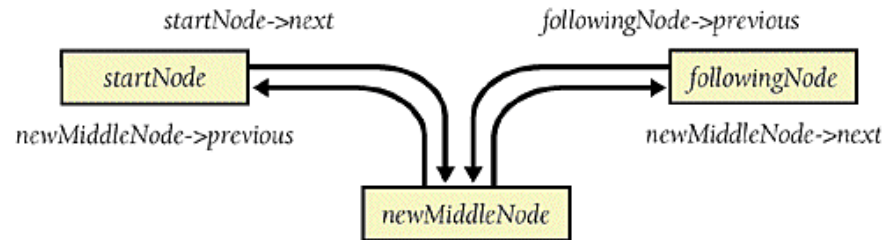
포인터 유의사항(2/2)

- 포인터 사용시
그림을 그려라

Initial Linkage



Desired Linkage



- 연결리스트에 있는 포인터들을 올바른 순서로 삭제함
- 포인터 해제 시, 쓰레기 데이터로 채워라 (예제 참조)
- 메모리를 삭제하거나 해제한 다음, 널(null)로 설정함 (예제 참조)

```
memset( pointer, GARBAGE_DATA, MemoryBlockSize( pointer ) );  
delete pointer;  
pointer = NULL;
```


포인터 사용 예

- 포인터를 리턴하는 아래 함수는 안전한가 ?

```
int *calculate(int array[], int size)
{
    int sum = 0;
    int *psum = sum;

    for(int i = 0; i < size; i++)
        sum += array[i];

    return psum;
}
```

구조적 프로그래밍 기법

구조적 프로그래밍 기법

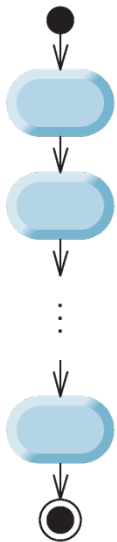
- 구조적 프로그래밍 기법은 단순성을 지향함
- Bohm and Jacopini: 아래 3가지 구조만으로 프로그램 생성
 - 순차(Sequence)
 - 선택(Selection)
 - 반복(Repetition)
- 구조적 프로그램 생성 규칙

Rules for forming structured programs

1. Begin with the simplest activity diagram (Fig. 5.22).
2. Any action state can be replaced by two action states in sequence.
(This is the stacking rule.)
3. Any action state can be replaced by any control statement (sequence of action states, if, if...else, switch, while, do...while or for).
(This is the nesting rule.)
4. Rules 2 and 3 can be applied as often as you like and in any order.

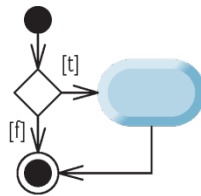
프로그램의 구조

Sequence

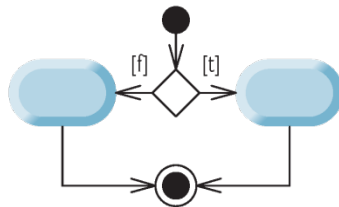


Selection

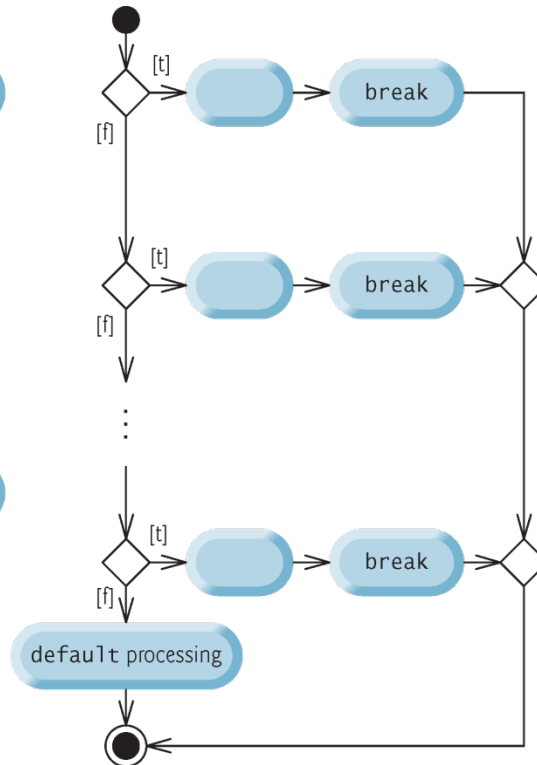
if statement
(single selection)



if...else statement
(double selection)

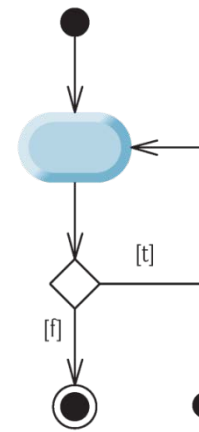


switch statement with breaks
(multiple selection)

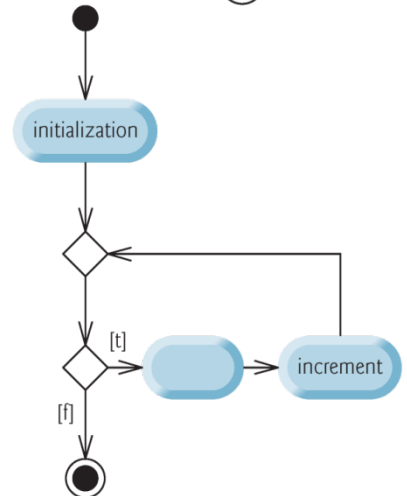
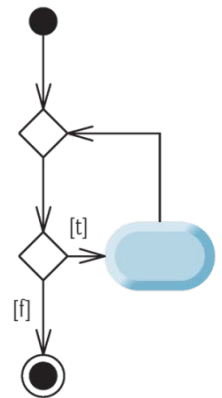


Repetition

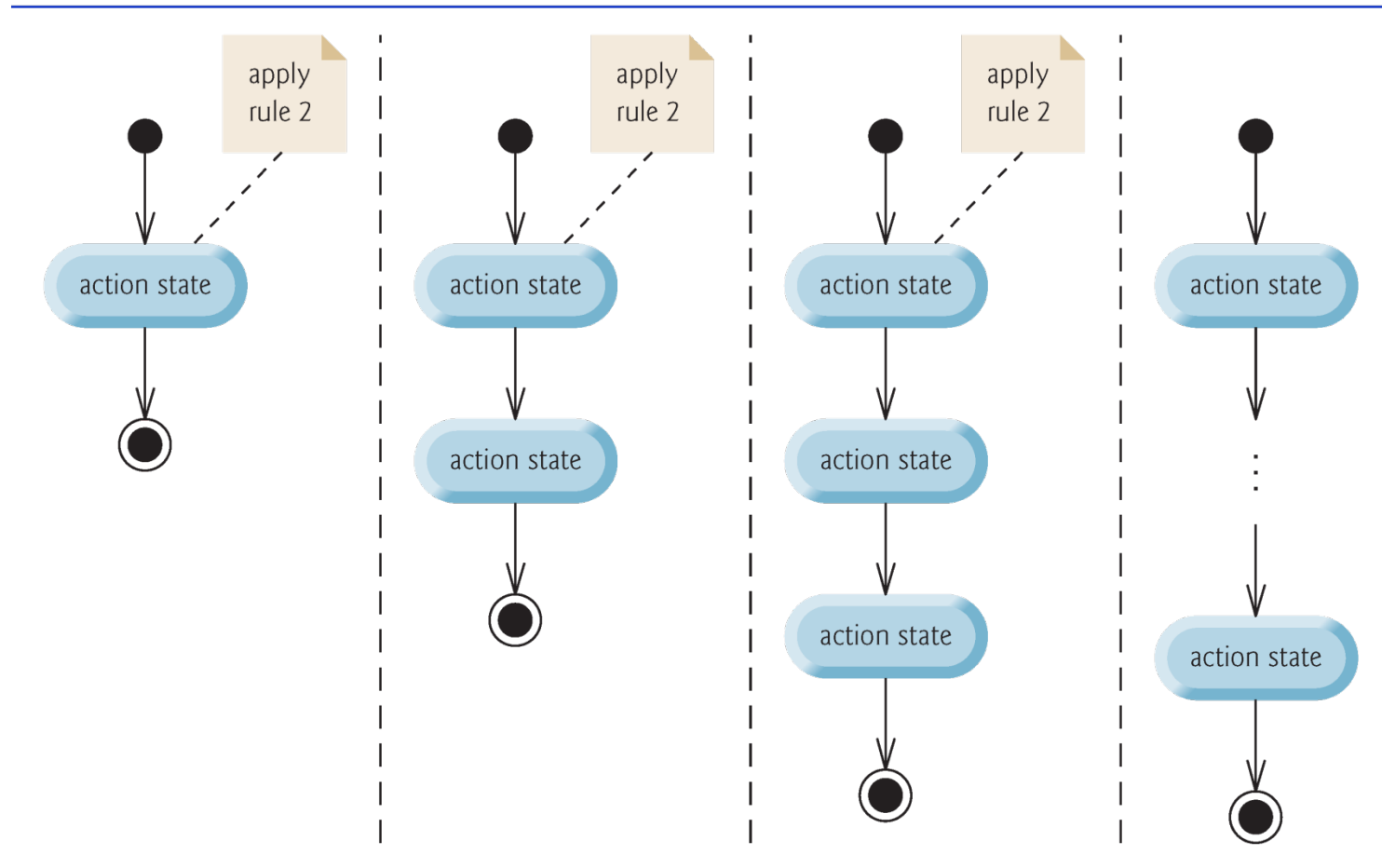
do...while statement



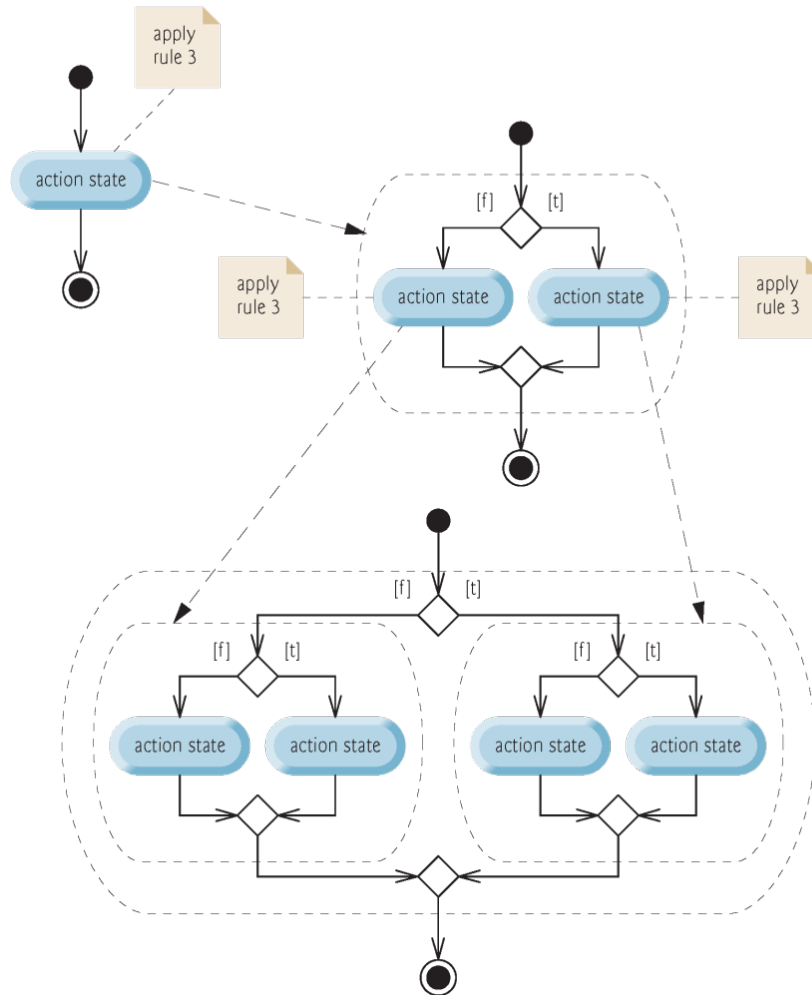
while statement



규칙 적용 예 1



규칙 적용 예 2



고급 루틴 작성하기

루틴(routine)이란 무엇인가

- 루틴은 한가지 목적을 위해서 호출 가능한 개별적인 메서드나 프로시저
- 아래 코드의 문제점은 ?

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
    double & estimRevenue, double ytdRevenue, int screenX, int screenY,
    COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
    int expenseType )
{
    int i;
    for ( i = 0; i < 100; i++ ) {
        inputRec.revenue[i] = 0;
        inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
    }
    UpdateCorpDatabase( empRec );
    estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
    newColor = prevColor;
    status = SUCCESS;
    if ( expenseType == 1 ) {
        for ( i = 0; i < 12; i++ )
            profit[i] = revenue[i] - expense.type1[i];
    }
    else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
    }
    else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
}
```


나쁜 루틴의 예

- 루틴의 이름이 나쁘다. `HandleStuff()` 이름은 정보가 없음
- 루틴이 문서화되어 있지 않음
- 루틴의 배치가 나쁨
- 루틴의 입력값인 `inputRec`가 변경됨
- 루틴이 전역변수를 쓰고 있음(예: `corpExpense`, `profit`)
- 루틴의 목적이 하나가 아님
- 루틴이 잘못된 데이터로부터 자신을 방어하지 못함: `divide-by-zero`
- 루틴이 여러 가지 매직 넘버(100, 4.0, 12, 2, 3)를 사용함
- 루틴의 변수 중 몇 개가 사용되지 않음(예 : `screenX`, `screenY`)
- 루틴의 매개변수 중 하나가 잘못 전달됨(예: `prevColor`)
- 루틴의 매개 변수가 너무 많음
- 루틴의 매개 변수가 잘못 정렬되어 있으며 문서화되어 있지 않음

루틴을 작성하는 이유

- 복잡성을 줄임
 - 루틴을 작성하는 가장 중요한 이유는 프로그램의 복잡성 줄임
 - 루틴의 내부 작업에 대해서 전혀 모른 채 사용할 수 있어야 함
- 코드의 중복을 피함
- 이해하기 쉬운 중단 단계의 추상화 도입(예제 참조)
 - 코드 섹션을 좋은 이름을 갖는 루틴으로 작성하는 것은 코드의 목적을 설명하는 최고 방법중의 하나임

```
if ( node <> NULL ) then
  while ( node.next <> NULL ) do
    node = node.next
    leafName = node.name
  end while
else
  leafName = ""
end if
```

```
leafName = GetLeafName( node )
```

- 이식성을 향상시킴
- 복잡한 불린 테스트를 단순화함
- 여러 곳에 있는 코드 대신, 한 곳에 있는 코드를 최적화할 수 있음

너무 단순한 연산의 루틴

- 효율적인 루틴 작성의 정신적 장애물
 - 간단한 목적을 위한 간단한 루틴 작성을 꺼리는 마음

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```



```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer  
    DeviceUnitsToPoints = deviceUnits *  
        ( POINTS_PER_INCH / DeviceUnitsPerInch() )  
End Function
```

```
points = DeviceUnitsToPoints( deviceUnits )
```



```
Function DeviceUnitsToPoints( deviceUnits: Integer ) Integer;  
    if ( DeviceUnitsPerInch() <> 0 )  
        DeviceUnitsToPoints = deviceUnits *  
            ( POINTS_PER_INCH / DeviceUnitsPerInch() )  
    else  
        DeviceUnitsToPoints = 0  
    end if  
End Function
```

좋은 루틴 이름

- 루틴이 하는 모든 것을 표현하라
- 의미가 없거나 모호하거나 뚜렷한 특징이 없는 동사들을 피하라
 - `HandleOutput()` ➔ `FormatAndPrintOutput()`
- 루틴의 이름을 숫자만으로 구분하지 마라
- 필요한 길이만큼 루틴의 이름을 만들어라
- 함수의 이름을 지을 때, 리턴값에 대한 설명을 사용하라
 - `printer.IsReady()`
- 반의어를 정확하게 사용

add/remove	increment/decrement	open/close
begin/end	insert/delete	show/hide
create/destroy	lock/unlock	source/target
first/last	min/max	start/stop
get/put	next/previous	up/down
get/set	old/new	

루틴의 길이

- 이론적으로 가장 좋은 길이는 한 화면이나 한 두 페이지(50~150줄)
- IBM은 한때 50줄로 제한, TRW는 두 페이지로 제한
- 프로그램들은 다수의 매우 짧은 루틴과 몇 개의 긴 루틴 혼재
- 극히 예외상황 : 4,000줄 루틴, 12,000줄 루틴

루틴의 매개변수 사용 방법(1/3)

- 루틴간의 인터페이스는 가장 오류가 발생하기 쉬운 영역임
 - 전체 오류의 39%가 내부 인터페이스 오류
- 매개 변수를 입력-수정-출력으로 입력함 (예제1)
- 고유한 in과 out 키워드 생성을 고려하기 바람 (예제2)

```
procedure InvertMatrix(  
    originalMatrix: in Matrix;  
    resultMatrix: out Matrix  
);  
...  
  
procedure ChangeSentenceCase(  
    desiredCase: in StringCase;  
    sentence: in out Sentence  
);  
...  
  
procedure PrintPageNumber(  
    pageNumber: in Integer;  
    status: out StatusType  
);
```

예제1

```
#define IN  
#define OUT  
void InvertMatrix(  
    IN Matrix originalMatrix,  
    OUT Matrix *resultMatrix  
);  
...  
  
void ChangeSentenceCase(  
    IN StringCase desiredCase,  
    IN OUT Sentence *sentenceToEdit  
);  
...  
  
void PrintPageNumber(  
    IN int pageNumber,  
    OUT StatusType &status  
);
```

예제2

루틴의 매개변수 사용 방법(2/3)

- 만약 여러 루틴들이 유사한 매개변수들을 사용한다면, 유사한 매개변수들을 일관된 순서로 입력하라
 - printf()와 fprintf(), puts()와 fputs()
- 모든 매개 변수들을 사용하라
 - 미사용 변수가 하나 이상인 루틴의 17~29% 오류 없음
 - 미사용 변수 없는 루틴은 46% 오류 없음
- 상태나 오류 변수를 마지막에 입력함
- 루틴의 매개변수를 작업용 변수로 사용하지 마라 (예제 참조)

```
int Sample( int inputVal ) {  
    inputVal = inputVal * CurrentMultiplier( inputVal );  
    inputVal = inputVal + CurrentAdder( inputVal );  
    ...  
}
```



```
int Sample( int inputVal ) {  
    int workingVal = inputVal;  
    workingVal = workingVal * CurrentMultiplier( workingVal );  
    workingVal = workingVal + CurrentAdder( workingVal );  
    ...  
}
```

루틴의 매개변수 사용 방법(3/3)

- 아래와 같은 매개 변수에 대한 인터페이스 가정을 문서화하라
 - 매개 변수가 입력, 수정, 출력을 위한 것인지
 - 숫자 매개 변수의 단위(인치, 피트, 미터 등)
 - 열거형이 사용되지 않는다면, 상태코드와 오류 값의 의미
 - 기댓값의 범위
 - 절대로 나타나서는 안 되는 특정 값들
- 루틴의 매개 변수의 수를 7개 정도로 제한하라
- 매개 변수에서 사용할 입력, 수정, 출력 이름 규약을 고려함
 - 매개 변수의 접두사(i_, m_, o_) 사용
- 실질적인 매개변수가 형식적인 매개변수와 일치하는지 확인하라
 - 부동소수점 매개변수에 정수 값을 사용

함수와 프로시저의 차이

- 함수 : 오직 입력 매개변수들만을 취하며 오직 하나의 값을 리턴함
- 프로시저 : 입력, 수정, 출력 매개 변수를 원하는 만큼 취할 수 있음
- 일반적인 프로그래밍 습관
 - 프로시저처럼 다양한 매개변수 입력 받고 상태 값을 반환함

```
if ( report.FormatOutput( formattedReport ) = Success ) then ...
```



```
report.FormatOutput( formattedReport, outputStatus )  
if ( outputStatus = Success ) then ...
```

```
outputStatus = report.FormatOutput( formattedReport )  
if ( outputStatus = Success ) then ...
```

- 루틴의 일차적인 목적이 함수 이름이 가리키는 값을 반환하는 경우에는 함수 사용, 그렇지 않으면 프로시저 사용

방어적 프로그래밍 (Defensive Programming)

The idea of defensive programming is based
on **defensive driving**

Invalid 입력에 대한 보호

- “Garbage in, garbage out”
- 좋은 프로그램에서는
 - “Garbage in, nothing out”, “Garbage in, error message out”
 - “No garbage allowed in”
 - 작은 문제들로부터 스스로를 보호하도록 함(예제 참조)
- 쓰레기 입력을 처리하는 세가지 방법
 - 외부로부터 들어오는 모든 데이터 값을 검사함
 - 루틴의 모든 입력 매개변수 값을 검사함
 - 잘못된 입력을 어떻게 처리할 것인지 결정함



시애틀의 90번
고속도로 부교

Assertion

- Assertion
 - 프로그램 실행 시 스스로를 검사할 수 있도록 개발 중 사용되는 코드
 - Assertion이 참이면, 모든 것이 예상대로 동작함을 의미
 - 거짓이면, 코드에서 예상치 못한 오류가 감지되었음을 의미
- Assertion의 용도
 - 크고 복잡한 프로그램과 신뢰성이 높은 프로그램에 특히 유용
 - 인터페이스 가정과 일치하지 않는 경우나 코드가 수정될 때 코드에 포함된 오류 등을 보다 빠르게 찾을 수 있음
- Java Assertion의 이용법
 - `assert expression;`
 - `assert expression1 : expression2`
 - Denominator가 0이 아니라는 것을 보장하기 위한 코드 (예제 참조)

Example 8-1. Java Example of an Assertion

```
assert denominator != 0 : "denominator is unexpectedly equal to 0.";
```

자신만의 Assertion 구축하기

- C++, Java, Visual Basic 은 Assertion 개념 기본 제공
- 언어에서 지원하지 않는 경우, 매크로로 작성 가능(예제 참조)

```
#define ASSERT( condition, message ) {           \
    if ( !(condition) ) {                         \
        LogError( "Assertion failed: ",          \
                  #condition, message );          \
        exit( EXIT_FAILURE );                     \
    }                                              \
}
```

Assertion 사용 지침(1/3)

- 오류 처리 코드: 발생할 것이라고 예상하는 상황들에 사용
- **Assertion** : 절대로 발생해서는 안 되는 조건에 사용
- 실행 가능한 코드를 Assertion 내에 입력하지 않음
 - Assertion 내부에 코드를 넣으면, assertion을 사용하지 않을 때 컴파일러가 코드를 제거할 확률이 높아짐

Example 8-3. Visual Basic Example of a Dangerous Use of an Assertion

```
Debug.Assert( PerformAction() ) ' Couldn't perform action
```



Example 8-4. Visual Basic Example of a Safe Use of an Assertion

```
actionPerformed = PerformAction()  
Debug.Assert( actionPerformed ) ' Couldn't perform action
```

Assertion 사용 지침(2/3)

- 선행조건과 후행조건을 문서화하고 검증하기 위해 Assertion 사용
 - **선행조건(pre-condition)**: 호출 및 생성 전의 루틴이나 클래스를 사용하는 클라이언트 코드의 참인 특성
 - **후행조건(post-condition)**: 실행을 마쳤을 때 루틴이나 클래스의 참인 특성

```
Private Function Velocity ( _  
    ByVal latitude As Single, _  
    ByVal longitude As Single, _  
    ByVal elevation As Single _  
    ) As Single  
  
    ' Preconditions  
    Debug.Assert ( -90 <= latitude And latitude <= 90 )  
    Debug.Assert ( 0 <= longitude And longitude < 360 )  
    Debug.Assert ( -500 <= elevation And elevation <= 75000 )  
    ...  
    ' Postconditions Debug.Assert ( 0 <= returnVelocity And  
returnVelocity <= 600 )  
  
    ' return value  
    Velocity = returnVelocity  
End Function
```

Assertion 사용 지침(3/3)

- 매우 견고한 코드 작성을 위해서는 Assertion 후, 오류를 처리하라
 - 일반적으로는 assertion이나 오류처리 코드 중의 하나를 사용함
 - MS Words의 소스 코드에서 assertion이 실패한 경우, 오류 처리 코드로 처리

```
Private Function Velocity ( _  
    ByRef latitude As Single, _  
    ByRef longitude As Single, _  
    ByRef elevation As Single _  
    ) As Single  
  
    ' Preconditions  
    Debug.Assert ( -90 <= latitude And latitude <= 90 )          <-- 1  
    Debug.Assert ( 0 <= longitude And longitude < 360 )          |  
    Debug.Assert ( -500 <= elevation And elevation <= 75000 )    <-- 1  
    ...  
  
    ' Sanitize input data. Values should be within the ranges asserted above,  
    ' but if a value is not within its valid range, it will be changed to the  
    ' closest legal value  
    If ( latitude < -90 ) Then          <-- 2  
        latitude = -90                 |  
    ElseIf ( latitude > 90 ) Then       |  
        latitude = 90                  |  
    End If                             |  
    If ( longitude < 0 ) Then           |  
        longitude = 0                  |  
    ElseIf ( longitude > 360 ) Then     <-- 2  
        ...
```


오류 처리 기법(1/2)

- **중립적인 값을 리턴함**
 - 아무런 문제가 없다고 알려진 값을 리턴하고 작업을 계속 수행함
 - 숫자 : 0, 문자열 : 빈 문자열, 포인터 : Null
 - 색상 값을 잘못 입력받은 게임의 그리기 루틴은 기본 배경색 또 전경색
 - 주의사항 : 암환자의 X-ray를 보여주는 그리기 루틴에는 적용 불가
- **다음에 오는 타당한 데이터로 대체함**
 - DB에서 깨진 레코드를 읽으면, 타당한 레코드까지 계속해서 읽음
 - 초당 100번씩 온도를 재고 있을 경우, 타당하지 않은 값은 다음 값으로 대체
- **이전과 동일한 값을 리턴함**
 - 온도를 읽는 소프트웨어가 온도를 제대로 읽지 못하면, 이전 값 이용
 - 비디오 게임의 타당하지 않는 색상 감지하면, 이전 색상으로 반환
 - 주의사항 : 현금인출기 거래 인증 오류 시에 이전 은행 계좌 번호 사용 ?
- **가장 가까운 타당한 값으로 대체함**
 - 0 ~ 100 사이의 온도계에서 0보다 작은 값은 0으로, 100보다 큰 값을 100으로
 - 자동차가 후진을 할 때, 속도계 값은 0으로 표시

오류 처리 기법(2/2)

- **경고 메시지를 파일에 기록함**
 - 경고 메시지를 기록하고 계속 실행하는 방법
 - 타 방법과 병행 사용 가능
- **오류 코드를 리턴함**
 - 오류를 자체 처리하지 않는 경우, 오류 감지를 보고함
 - 오류 발생 사실을 알리는 구체적인 매커니즘
 - 상태 변수에 값을 설정함
 - 함수의 리턴 값으로 상태 값을 리턴함
 - 언어의 예외처리 메커니즘을 사용하여 예외를 던짐(throw)
- **오류 처리 루틴이나 객체를 호출함**
 - 오류 처리를 전역적인 오류 처리 루틴이나 객체에게 집중시킴
 - 장점 : 오류 처리가 집중되므로 디버깅이 쉬워짐
 - 단점 : 전체 프로그램이 오류 처리 기능과 연계되므로 재사용이 어려워짐
 - 단점 : 공격자가 핸들러 루틴의 주소를 손상시키는 보안 문제 가능

견고성 vs 정확성

- **정확성** : 절대로 부정확한 결과를 리턴할 수 없음
- **견고성** : 비록 부정확한 결과를 만들어내더라도 소프트웨어가 작동할 수 있도록 계속해서 무언가를 하려고 애쓰는 것을 의미함
- **안전성이 중요한 프로그램(예 : 방사능 치료기)**
 - 잘못된 결과를 리턴하는 것보다 아무것도 리턴하지 않는 것이 더 좋음
 - 견고성보다 정확성을 선호하는 경향이 있음
- **개인용 응용 프로그램**
 - 일반적으로 무엇이든지 간에 결과가 있는 것이 종료되는 것보다 더 좋음
 - 워드 프로세서의 화면 하단에 텍스트 구분 선이 생기는 오류는 종료하는 대신에 refresh 시에 정상으로 하는 것이 좋음
 - 정확성보다 견고성을 선호함

예외(Exception)

- 예외(exception)
 - 코드의 오류나 예외적인 이벤트를 루틴을 호출한 코드에 전달할 수 있는 특수한 방법
 - 만약 어떤 루틴이 어떻게 처리해야 할 지 모르는 예외적인 상황에 부딪히면, 예외를 던짐(throw)

```
try
{
    statements
    resource-acquisition statements
} // end try
catch ( AKindOfException exception1 )
{
    exception-handling statements
} // end catch
...
catch ( AnotherKindOfException exception2 )
{
    exception-handling statements
} // end catch
finally
{
    statements
    resource-release statements
} // end finally
```

예외 처리 유의사항(1/2)

- 무시되어서는 안 되는 오류를 프로그램의 다른 부분에 알리기 위하여 예외를 사용하라
 - 예외의 장점은 절대 무시되지 않음
- 정말로 예외적인 조건인 경우에만 예외를 던져라
 - 예외는 예상치 못한 조건을 처리하기 위한 강력한 방법과 복잡성 증가 사이에서의 트레이드 오프를 나타냄
 - 예외는 루틴을 호출하는 코드가 호출된 코드 내부에서 어떤 예외를 던질 것인지 알고 있어야 하기 때문에 캡슐화 약화 및 코드의 복잡성 증가시킴
- 책임을 전가하기 위해서 예외를 사용하지 말라
 - 만약 오류 조건이 지역적으로 처리될 수 있다면, 지역적으로 처리함
- 만약 생성자와 소멸자에서 예외를 잡을 수 없다면, 생성자와 소멸자에서 예외를 던지지 말라
 - 생성자에서 예외를 던진다면, 소멸자 비호출로 리소스 누수 발생 가능

예외 처리 유의사항(2/2)

- 예외를 야기한 모든 정보를 예외 메시지에 포함시켜라
- 비어있는 catch 블록을 피하라

```
try {  
    ...  
    // lots of code  
    ...  
} catch ( AnException exception ) {  
}
```



```
try {  
    ...  
    // lots of code  
    ...  
} catch ( AnException exception ) {  
    LogError( "Unexpected exception" );  
}
```

- 라이브러리 코드가 던지는 예외를 파악하라
- 집중된 예외 보고자의 구축을 고려하라

```
Sub ReportException( _  
    ByVal className, _  
    ByVal thisException As Exception _  
)
```

```
    Dim message As String  
    Dim caption As String
```

```
    message = "Exception: " & thisException.Message & "." & ControlChars.CrLf & _  
        "Class: " & className & ControlChars.CrLf & _  
        "Routine: " & thisException.TargetSite.Name & ControlChars.CrLf  
    caption = "Exception"  
    MessageBox.Show( message, caption, MessageBoxButtons.OK, _  
        MessageBoxIcon.Exclamation )
```

```
End Sub
```

```
Try
```

```
    ...
```

```
Catch exceptionObject As Exception
```

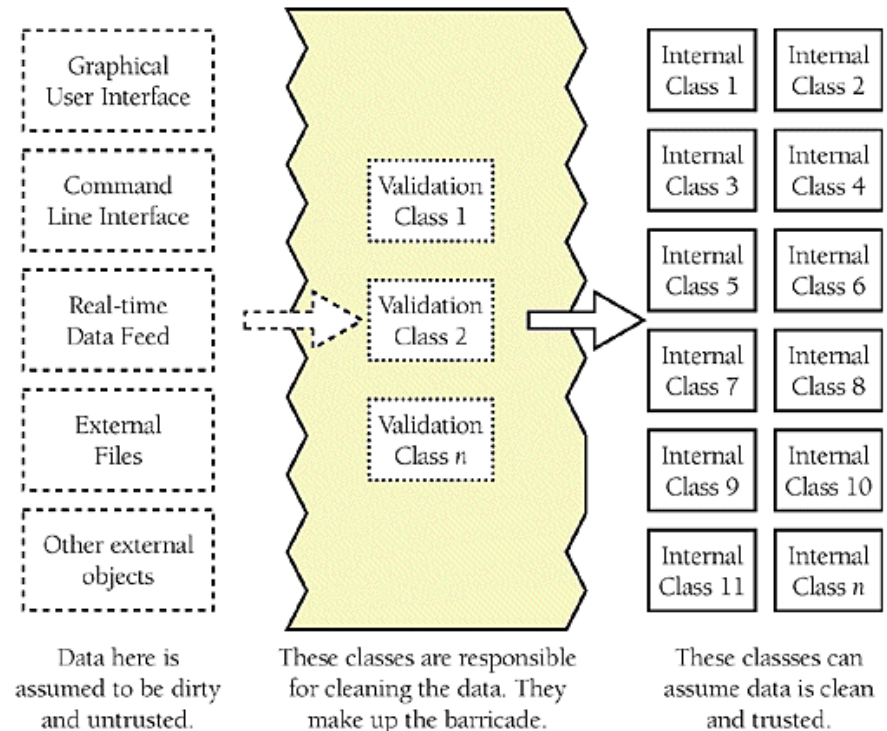
```
    ReportException( CLASS_NAME, exceptionObject )
```

```
End Try
```

오류로 인한 손해 억제 대책

- 방책(Barricade)
 - 방책은 오류 손실 억제 전략임
 - 선체에서 객실을 분리하거나 빌딩의 방화벽 설치와 유사
 - 특정한 인터페이스를 안전한 지역의 경계로서 명시하는 것
 - 수술실 비유 : 데이터는 수술실에 들어가기 전에 살균됨

- 방책과 Assertion
 - 방책 외부 루틴: 오류 처리
 - 방책 내부 루틴: **Assertion** 사용



디버깅 보조 도구

- 디버깅 보조 도구의 초기 사용
 - 오류를 빠르게 발견하기 위한 도구를 초기에 적용
- 제품의 제약사항을 개발 버전에 무의식적으로 적용하지 마라
 - 제품 버전은 실행 속도가 빨라야 하지만, 개발 버전은 느려도 상관 없음
 - 제품 버전은 자원을 아껴야 하지만, 개발 버전은 마음껏 쓸 수 있음
- 공격적인 프로그래밍 기법 사용
 - `Assertion` 실패 시에 프로그램을 중단시킴
 - 메모리를 완벽하게 채워서 메모리 할당 오류를 발견함
 - 파일이나 스트림을 완벽하게 채워서 파일 형식 관련 오류를 발견함
 - `Default`절이나 `else` 절이 심각한 문제를 일으켜 종료되지 않는지 확인
 - 객체나 동적 할당 메모리를 삭제하기 전에 쓰레기 데이터로 채움
 - 적합한 경우에, 오류 로그 파일을 이메일로 보내도록 프로그램을 설정함

디버깅에서의 심리적 측면

- 아래 싸인의 의미는 ?



- 대부분의 사람은 지금의 현상이 이전 유사한 현상이기를 기대함
 - 앞서본 SYSTSTS 변수와 방금 본 SYSSTSTS는 같은 것으로 봄
 - 아래 두 코드는 같은가 ?

```
if ( x < y ) {  
    swap = x;  
    x = y;  
    y = swap;  
}
```

```
if ( x < y )  
    swap = x;  
    x = y;  
    y = swap;
```

디버깅 명령어 제거하기(1/4)

- 레벨 0 : 디버깅 명령어(cout, printf()) 삭제
- 레벨 1 : 디버깅 명령어 comment out 하기

```
#define TEAM_SIZE 8

void main()
{
    bool team[TEAM_SIZE];
    for(int i = 0; i < TEAM_SIZE; i++)
        team[i] = false;

    srand(time(0));
    for( i = 0; i < TEAM_SIZE; i++) {
        int select = rand() % TEAM_SIZE;
        // cout << "Selected = " << select << endl;
        while ( team[select] )
            select = rand() % TEAM_SIZE;

        team[select] = true;
        cout << "Team " << select+1 << endl;
    }
}
```

디버깅 명령어 제거하기(2/4)

- 레벨 2 : 전처리 명령어 사용(#define DEBUG 1)

```
#define DEBUG 1
#define TEAM_SIZE 8

void main()
{
    bool team[TEAM_SIZE];
    for(int i = 0; i < TEAM_SIZE; i++)
        team[i] = false;

    srand(time(0));
    for( i = 0; i < TEAM_SIZE; i++) {
        int select = rand() % TEAM_SIZE;

        if (DEBUG)
            cout << "Selected = " << select << endl;
        while ( team[select] )
            select = rand() % TEAM_SIZE;

        team[select] = true;
        cout << "Team " << select+1 << endl;
    }
}
```

디버깅 명령어 제거하기(3/4)

- 단계 3 : 전처리기 명령어 사용 (#define DEBUG)
 - 컴파일 옵션으로 디버그 코드 포함/제거 가능(-D Debug)

```
#define DEBUG
#define TEAM_SIZE 10

void main()
{
    bool team[TEAM_SIZE];
    for(int i = 0; i < TEAM_SIZE; i++)
        team[i] = false;

    srand(time(0));
    for(int i = 0; i < TEAM_SIZE; i++) {
        int select = rand() % TEAM_SIZE;

        #if defined(DEBUG)
            cout << "Selected = " << select << endl;
        #endif

        while ( team[select] )
            select = rand() % TEAM_SIZE;

        team[select] = true;
        cout << "Team " << select+1 << endl;
    }
}
```

디버깅 명령어 제거하기(4/4)

- 레벨 4 : 전처리기 명령어 사용(DebugCode(code_fragment))

```
#if defined(DEBUG)
#define DebugCode( code_fragment ) { code_fragment }
#else
#define DebugCode( code_fragment )
#endif

void main()
{
    bool team[TEAM_SIZE];
    for(int i = 0; i < TEAM_SIZE; i++)
        team[i] = false;

    srand(time(0));
    for(int i = 0; i < TEAM_SIZE; i++) {
        int select = rand() % TEAM_SIZE;

        DebugCode( cout << "Selected = " << select << endl; );

        while ( team[select] )
            select = rand() % TEAM_SIZE;

        team[select] = true;
        cout << "Team " << select+1 << endl;
    }
}
```