# 개발자 차원의 버그 최소화

한동대학교 전산전자공학부
김인중
(ijkim@handong.edu)

# 목차

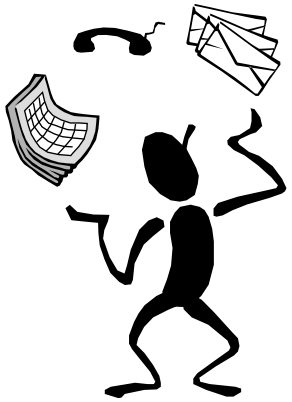
- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- 코딩 후 검토
- 버그 예방을 위한 **Tip** 모음
- 유용한 도구
- 질의 응답

# 버그에 대한 바람직한 접근 태도

- 증상이 아닌 원인을 이해하고 그에 대한 대책을 마련한다.
- 프로그램과 문제를 충분히 이해하기 전에는 수정하지 않는다.
- 버그 수정은 빠를 수록 좋다.
- 버그를 통해 자기를 계발할 수 있다.
  - 작업중인 프로그램과 자신의 문제를 이해할 수 있다.
  - 자신의 코드에 대한 비판적 관점을 경험할 수 있다.
  - 문제 해결 방법
  - 결함 발견 및 수정 방법

# 버그의 발생 원인과 패턴에 대한 이해

- 증상
  - 프로그램 halt
- 증상의 원인
  - Buffer overflow (버그)
- 버그의 발생 원인
  - 소재적 오류: 빈약한 기술과 예외처리
- 대책
  - 일관성 있는 예외처리 정책 수립
  - Buffer overflow 가능한 코드에 대한 검토



# 버그에 대한 위험한 접근 태도

- 사용 가능한 데이터들을 추론하는데 사용하지 않고 사소하고 불합리한 변경 작업을 수행한다. 그리고 그 결과를 복원하지 않는다.
- 확실한 원인을 모르는 상태에서 추론에 의해 해결하려 한다.
- 문제를 이해하려고 시간을 투자하지 않는다.
- 문제를 해결하지 않고 밀봉한다

```
Ex) x = Compute( y );
    if ( y = 17 )
    x = 25.15      --  Compute() doesn't work for y = 17, so fix it
```

- 생각하기보다는 디버깅 툴에 의존한다.

# 목차

- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- 코딩 후 검토
- 버그 예방을 위한 Tip 모음
- 유용한 도구
- 질의 응답

# 방어적 프로그래밍

- 방어 운전
  - 다른 운전자가 무엇을 하려고 하는지 절대로 확신하지 않는 마음가짐
  - 다른 운전자의 과실일 때도 스스로를 보호할 책임이 있다.

- 방어적 프로그래밍
  - 타당하지 않은 데이터, 절대로 발생할 수 없는 이벤트, 다른 프로그래머의 실수로부터 스스로를 보호하기 위한 프로그래밍 기법

# 방어적 프로그래밍

- 잘못된 입력을 처리하기 위한 방법
  - 외부로부터 들어오는 모든 데이터(함수의 입력 매개변수, 호출한 함수로부터의 리턴 값) 검사

```
int MyFunc(int arg1, int arg2){
    if(arg1 < 0){                          // check arg1
      // some codes
    }
    if(arg2 < 0 || arg2 >= Limit){         // check arg2
      // some codes
    }

    ret = OtherFunc(…);
    if(ret != SUCCESS){                    // check function result
      // some codes
    }

    // some codes
}
```

# Assertion

- **Assertion이란?**
  - 프로그램이 실행될 때 스스로를 검사할 수 있도록 개발 도중에 사용되는 코드 (함수, 또는 매크로)
  - 특정 루틴이 실행되기 위해 필요한 조건을 검사하는데 이용

구현 예) 두 개의 인자를 갖는 assertion

- 반드시 참이 되어야 하는 조건을 기술하는 Boolean 표현식
- 참이 아닐 경우 표시할 메시지

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

# Assertion을 이용한 방어적 프로그래밍

- 절대로 발생해서는 안 되는 조건을 위해서 **assertion**을 사용하라

```
Ex) int Factorial(int n)
    {
       ASSERT(n >= 0)
       // …
    }
```

- 매우 견고한 코드를 작성하기 위해서는 **assert** 후 오류를 처리하라**.**

# 오류 처리

- **C 언어에서의 오류 처리 방법**
  - 상태코드(에러코드)를 리턴
    - 구현하기 간단함.
    - 에러타입들 뿐만 아니라 그것들의 열거(enumerate)된 값들도 표준화되지 않는다.
    - 프로그램 코드가 커진다
  - 에러코드를 전역변수로 할당하고 다른 함수들이 그것을 검사하도록 함
    - 에러코드의 표준화가 가능하다.
    - 다중 스레드 환경에서는 충돌 위험
    - 오류 값 처리 후 전역변수의 관리(reset)에 주의해야 함.
  - 프로그램 종료
    - 정확성은 강하나 견고성 취약

# 오류 처리

- **정확성 vs. 견고성**
  - 정확성: 절대로 부정확한 결과를 리턴하지 않음
  - 견고성: 부정확한 결과를 만들어내더라도 소프트웨어는 계속 작동
- **정확성을 확보하기 위한 오류 처리 방식**
  - 리턴 값 대신 오류 코드, 또는 Invalid 한 값 리턴
  - 프로그램 종료
- **견고성을 확보하기 위한 오류 처리 방식**
  - 적절한 대안을 리턴
    - 중립적인 값, 근사값, 이전 또는 다음 값 등

# 예외 처리

- 예외(exception)란?
  - 특정 상황 하에서 일반적인 실행 flow를 수정해야 하는 조건
    예) File open failure, 메모리 할당 실패, …

```
FILE *fp = NULL;
char *buffer = NULL;

buffer = malloc(BUFFER_SIZE);
if(buffer == NULL)
    return MEMORY_ALLOC_FAILURE;

fp = fopen(filename, “r”);
if(fp == NULL)
    return FILE_OPEN_FAILURE;

read(buffer, NoData, DataSize, fp);
```

C의 예외처리 예

```
iostream is;
char *buffer = NULL;

try{
    buffer = new char[BUFFER_SIZE];
    is.open(filename);
    is.read(buffer, BUFFER_SIZE];
} catch (...){
    // some codes
}

// some codes
```

C++의 예외처리 예

# 예외 처리

- 바람직한 예외 처리
  - 무시될 수 없는 오류를 알리기 위해 사용하라.
  - 지역적으로 처리될 수 있는 예외는 직접 처리하라.
  - 메시지는 친절하게 한다.
  - 모든 예외는 적절히 처리되어야 한다.

# 예외 처리

- **C++, Java, C# 에서의 예외 처리: try / catch / throw**

- **C 언어에서의 예외 처리**
  - 함수 리턴값, 전역 변수 이용
  - longjmp(), setjmp() 이용
    - C++, Java 의 throw-catch와 유사한 오류 처리 방식을 구현
    - 강력하나 구현하기 복잡함

```
jmp_buf jumper;

int Divide(int a, int b)
{
   if (b == 0) // can't divide by 0
      longjmp(jumper, -3);
   return a / b;
}
```

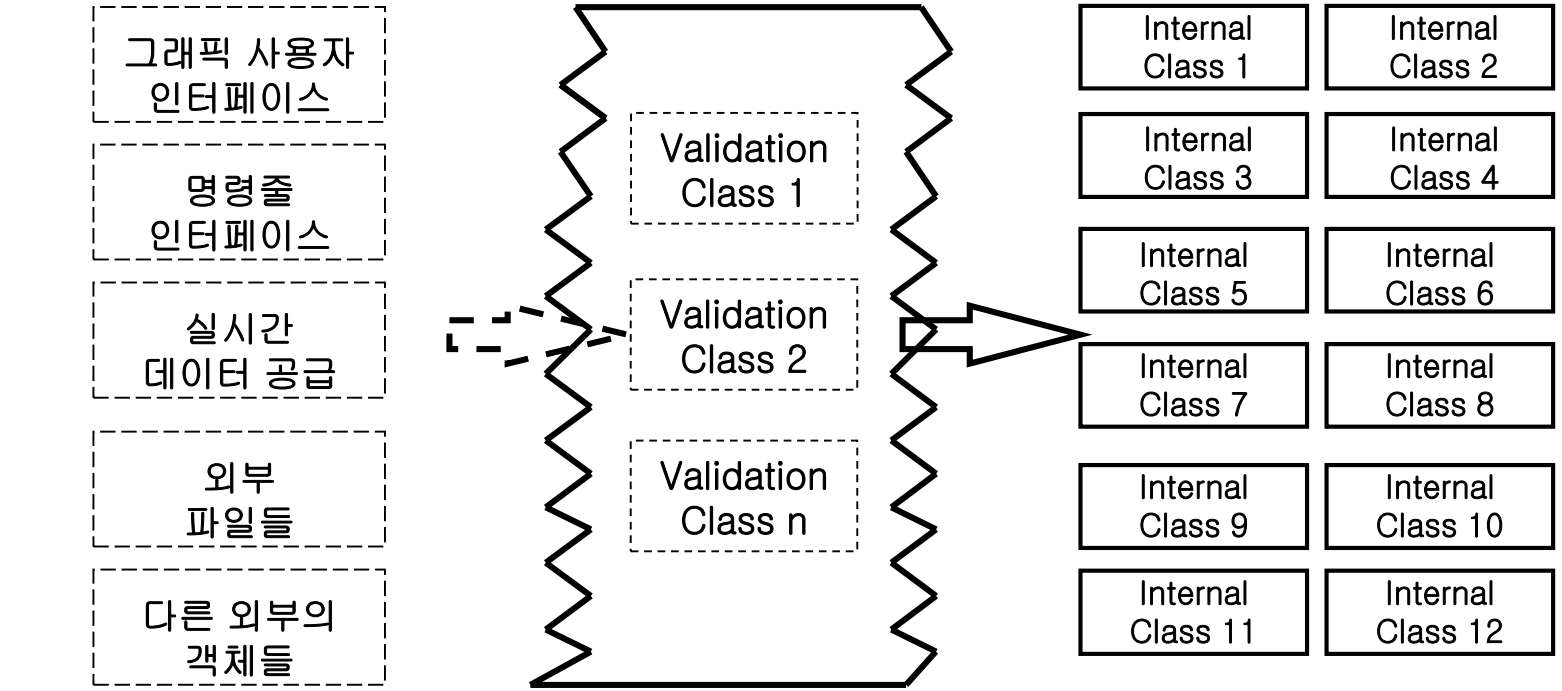
```
void main(void)
{
   if(setjmp(jumper) == 0){
      int Result = Divide(7, 0);
      // continue working with Result
   } else
      printf("an error occured₩n");
}
```

# 수술실 기법

- **수술실 기법**
  - 데이터는 수술실에 들어가는 것이 허용되기 전에 살균
  - 수술실에 있는 것은 모두 안전하다 가정
  - 일반적으로 데이터는 여러 단계에서 살균될 필요성 있기 때문에 여러 단계로 살균하는 것도 필요
  - 입력 시에 입력 데이터를 적절한 형을 변환
    예) “Blue” (string) → BLUE (Macro)
- **오류 처리 vs. assertion**
  - 방책의 외부에 있는 루틴: 오류 처리 사용
  - 방책의 내부에 있는 루틴: assertion 사용

# 수술실 기법

- 특정한 인터페이스를 “안전한” 지역의 경계로써 명시




여기에 있는 데이터들은 신뢰할 수 없고 불결한 것으로 여겨진다.

이 클래스들은 데이터를 정리하는 역할을 하고 방책을 만든다.

이 클래스들은 데이터를 신뢰할 수 있고 깨끗한 것으로 간주할 수 있다.

# 목차

- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- 코딩 후 검토
- 버그 예방을 위한 Tip 모음
- 유용한 도구
- 질의 응답

# 디버깅

- 디버깅이란?
  - 오류의 근본적인 원인을 규명하여 수정하는 과정
  - 오류를 감지하는 과정인 테스트와는 반대되는 개념

# 디버깅 시 주의사항

- 디버깅은 소프트웨어의 품질을 향상시키는 방법이 아니라, 결함을 진단하는 방법
- 고급 제품을 만드는 가장 좋은 방법은 디버깅이 아니라 요구 사항을 주의 깊게 계발 하고 설계를 잘하고, 고급 코드 작성 방법을 사용하는 것
- 문제를 충분히 이해한 후 수정하라
- 디버거에 지나치게 의존하지 말고 실행 결과로부터 적극적으로 추론하라.

# 과학적인 디버깅 방법

**1. 일관적으로 오류를 발생시키는 가장 간단한 테스트 케이스를 찾는다.**

- 예측 불가능한 오류의 원인들: 초기화, 타이밍 문제, dangling pointer.
- 증상이 즉시 나타나지 않는 문제들: memory / resource leak

**2. 오류의 원인을 찾아낸다**

- 데이터를 분석을 통해 결함에 대한 가설을 세운다.
- 프로그램을 테스트하거나 코드를 살펴봄으로써 그러한 가설을 증명하거나 반증할 방법을 결정한다.
- 절차를 사용하여 가설들을 증명하거나 반증한다.

**3. 결함을 수정한다.**

**4. 수정한 내용을 테스트한다.**

**5. 유사한 오류를 검사한다.**

# 목차

- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- **코딩 후 검토**
- 버그 예방을 위한 Tip 모음
- 유용한 도구
- 질의 응답

# 코딩 후 검토

- 코딩과 검토를 분리해야 하는 이유
  - 코딩 시에는 문제 해결에 집중한다.
  - 검토 시에는 버그의 검출 및 수정에 집중한다.
  - 버그에 대한 전문적 대응 방안을 적용


- 소스코드 정말 검토의 효과
  - 결함 발견의 측면에서 테스트보다 소스코드 검토가 더 효과적
  - 실행 테스트 시 발견되지 않는 결함도 발견된다.

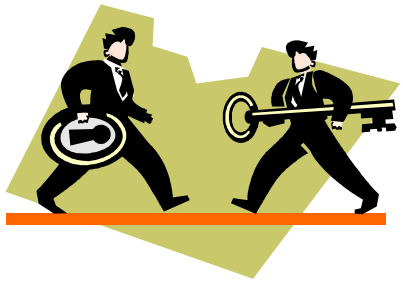
# 코딩 후 검토

- 개발자 검토
  - 코드 정밀 검토
    - 비판적 시각에서의 개발자 테스트
  - 자주 발생하는 버그들에 대한 체크
    - 체크 리스트 이용
  - 적절한 도구 이용
    - Debugger
    - Static code checker
    - Dynamic checker

# 코딩 후 검토

- **짝 프로그래밍 (Pair programming)**
  - 개발자들이 짝을 이루어 함께 개발하는 개발 방식
    - 짝 개발, 짝 검토, 짝 학습


- **짝 프로그래밍의 효과**
  - 스트레스에 더 잘 견딘다.
    - 서로 격려할 수 있다.
  - 가독성과 이해성이 우수한 개발자 수준으로
  - 일정 단축
    - 속도는 빠르고 오류는 적다
  - 협력 문화 보급, 프로그래머 교육



# 코딩 후 검토

- **짝 프로그래밍 시 주의할 점**
  - 강요하거나 감시가 되지 않도록 한다.
  - 코딩 스타일 표준을 이용한다.
  - 짝 프로그래밍이 효과적인 곳에만 부분적으로 이용한다.
  - 정기적으로 짝을 교대한다.
  - 초보자끼리 짝을 이루지 않도록 한다.

# 개발자 검증을 위한 체크리스트

- 일반적으로 자주 발생하는 버그들
  - Divide by zero
  - Infinite loops
  - Arithmetic overflow or underflow
  - Exceeding array bounds
  - Using uninitialized variables
  - Accessing memory not owned (Access violation)
  - Memory leak or Handle leak
  - Stack overflow or underflow
  - Buffer overflow
  - Deadlock
  - Off by one error
  - Race condition
  - Loss of precision in type conversion

# 개발자 검증을 위한 체크리스트

- **변수 및 배열**
  - 변수 초기화 (모든 변수에 대하여)
  - 배열의 경계를 침범하지 않는가?
- **자원 할당 및 해제**
  - 동적 메모리 할당, file open 시 NULL 체크
  - 동적 메모리 할당 코드(malloc)과 해제 코드(free)가 짝을 이루는가?
  - 메모리 해제 후 참조를 시도하지 않는가?
  - 프로그램의 Heap 및 stack 사용량은?

# 개발자 검증을 위한 체크리스트 (계속)

- 산술 오류
  - 나눗셈 연산 시 분모가 0이 될 가능성
  - Type변환에 의한 정보 손실
  - 연산에서 overflow / underflow 가능성
- 제어
  - 루프의 종료 조건은 언젠가 참이 되는가?
- 방어적 프로그래밍
  - 함수의 입력 파라미터와 함수 리턴 결과 체크 및 예외처리

# 목차

- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- 코딩 후 검토
- **버그 예방을 위한 Tip 모음**
- 유용한 도구
- 질의 응답

# 버그 예방을 위한 Tip 모음

- 실수를 통해 배우라
- 기억해야 할 정보를 최소화하라
- 예측 불가능한 오류를 예방하라
- 배열 경계 조건을 통일하라
- 동적 메모리 할당은 함수 단위로
- 리턴 값이 올바른지 체크하라
- 다중 중첩을 최소화 하라
- **Source Code**를 모듈화하라
- 심리적 거리가 크도록 명명하라.
- 적당한 종료 시점을 정의하라

# 실수를 통해 배우라

- 증상이 아닌 문제를 치료하라.

- 동일한 버그를 예방하기 위한 스스로의 기법을 구축하라.

# 기억해야 할 정보를 최소화하라

- 전역 변수의 사용은 가능한 한 피하라.
  - 가능한 경우 정적 변수(static)를 이용하여 정보를 지역화 하라
- **Loop** 변수는 **loop** 밖에서 다른 목적으로 사용하지 말라.
- 변수 선언은 가능한 한 **block** 내에서 한다.

```
for(i = 0; i < strlen(pString); i++) {
        char c = pString[loop];
         ...
}
```

# 기억해야 할 정보를 최소화하라

- **명시적 데이터 형 변환(type casting)은 최소화하라.**
  - Type casting은 개발자가 강요한 형 변환
  - 컴파일러는 개발자가 그 의미를 정확히 안다고 가정한다. 그러나 환경이 바뀔 경우 문제를 일으킨다.

- **sizeof()의 파라미터로 데이터형 보다는 변수를 사용하라.**

예) int nVar; // nVar 는 long 형으로 바뀔 수 있음

```
DumpHex(&nVar, sizeof(int) );
DumpHex(&nVar, sizeof(nVar));
```

# 기억해야 할 정보를 최소화하라

- 방법 보다는 하는 일 자체를 표현하라.

  Ex1) while (*pDst++ = *pSrc++);

  Ex2) strcpy(pDst, pSrc);

  - Never reinvent the wheel !

# 예측 불가능한 오류를 예방하라

- ☞ 주의! 예측 불가능한 오류의 주요 원인
  - 변수 초기화 누락
  - Dangling pointer

- 모든 변수는 초기화하라.

- 모든 포인터는 올바른 주소, 또는 **NULL**
  - 모든 포인터는 NULL로 초기화
  - free후 반드시 NULL로 리셋
  - 디버깅 시 편리 (die or not)

# 배열 경계 조건을 통일하라

- ☞ **주의! 많은 버그는 배열의 경계에서 발생한다.**

  Ex) for(i = 0; i <= n; i++);  // n까지 포함 → 경계 침범

  for(i = 10; i > 0; i--);  // 0은 제외됨

- **범위를 표시할 때 하위 경계(lower bound)는 닫힌 경계를, 상위 경계(upper bound)는 열린 경계를 이용하라.**
  - 다음 중 { 0, 1, 2, 3, 4, 5 }의 표현 방법으로 가장 좋은 것은?
    - [0, 5]
    - (-1, 6)
    - [0, 6)
    - (-1, 5]

# 배열 경계 조건을 통일하라

- **Why?**
  - Loop을 정의하기 쉽다.
    for(i = 0; i < 6; i++)…
  - 원소의 개수가 <상위 경계> - <하위 경계>와 동일해 진다.
    - # of elements: 6 – 0 = 6
  - 고정된 규칙에 의해 배열을 참조하기 편리하다.

```
for(i = 0; i < 6; i++)                    // int a[6];
  printf("%d ", a[i]);
```

예) { 0, 1, 2, 3, 4, 5 }로부터 2, 3, 4를 삭제
  - Remove [2, 4]: 남은 원소의 수 = 6 – (4 – 2 **+ 1**)
  - Remove [2, 5): 남은 원소의 수 = 6 – (5 – 2)

- 많은 라이브러리에서 채택된 방식
  - STL(Standard Template Library), MFC(Microsoft, Foundation Class)

# 동적 메모리 할당은 함수 단위로

- 각 함수에서 할당된 동적 메모리는 그 함수 종료 전에 반드시 해제한다.
  - ***malloc***와 ***free*** 는 반드시 함수 내에서 쌍을 이루도록 한다.

  단, Creator와 Destructor에서는 예외

# 동적 메모리 할당

- **Bad style**

```
int *AllocAndInit(int len);

void fun1()
{
   int size = 10;
   int *pi = NULL;
   pi = AllocAndInit(size);
   pi[0] = 10;
   …
   free(pi);          // easy to forget
}

int *AllocAndInit(int len)
{
   int i = 0;
   int *a = malloc(len*sizeof(int));

   for(i = 0; i < len; i++)
      a[i] = 0;

   return a;
}
```

- **Good style**

```
void Init(int a[], int n);

void fun1()
{
   int size = 10;
   int *pi = NULL;

   pi = malloc(size * sizeof(int));

   Init(pi, size);
   pi[0] = 10;
   …
   free(pi);     // make pair with malloc
}

void Init(int a[], int n)
{
   int i = 0;

   for(i = 0; i < len; i++)
      a[i] = 0;
}
```

# 리턴 값이 올바른지 체크하라

- 스택의 배열은 리턴하지 않는다.

```
char *myitoa( int nNumber )
{
   char buffer[80];
   sprintf (buffer, "%d", nNumber);
   return (buffer);
}
```

# 다중 중첩을 최소화 하라

- 다중 중첩

```
void DeepNestFunction( void )
{
  if (test1) {
    // more code
    if (test2) {
     // more code
     if (test3) {
       // more code
       if (test4) {
         // more code
       }
      }
    }
  }
} /* DeepNestFunction */
```

- 조건 변수를 이용한 다중 중첩 제거

```
void UnrollingDeepNesting( void )
{
  BOOL bVar=(test1);
  if (bVar) {
    // more code
    bVar = (test2);
  }
  if(bVar) {
    // more code
    bVar = (test3);
  }
  //...
} /* UnrollingDeepNesting */
```

# Source Code를 모듈화하라

- **Source Code 모듈화**
  - 모듈 별 독립성 확보
  - 외부에서 호출 가능하도록 잘 정의된 API

- **초기에 API를 심사숙고하여 결정한다.**
  - API의 수정은 매우 비용이 크다.

# 심리적 거리가 크도록 명명하라.

- 변수/함수의 이름은 “심리적 거리”가 크도록 지정한다.

| 첫 번째 변수 | 두 번째 변수 | 심리적 거리 |
|---|---|---|
| stoppt | stcppt | 거의 알아보기 힘들다 |
| shiftrn | shiftrm | 거의 없다 |
| dcount | bcount | 좁다 |
| claims1 | claims2 | 좁다 |
| product | sum | 멀다 |

# 적당한 종료 시점을 정의하라

- 품질 관리와 관련된 개발 목표를 설정 후 이를 달성하면 종료하라.
  - 확실한 목표가 없는 프로그램 개발은 종료되지 않는다.
  - “The 90 Percent Done Syndrome”

# 목차

- 버그에 대한 접근 태도
- 버그 예방을 위한 프로그래밍 기법
  - 방어적 프로그래밍
  - 오류 및 예외 처리
- 디버깅
- 코딩 후 검토
- 버그 예방을 위한 Tip 모음
- **유용한 도구**
- 질의 응답

# 유용한 도구: 개발자용

- **정적 코드 체크 (static code checker)**
  - 소스코드 중 버그, 또는 실수로 의심할 만한 부분을 자동 검색

  Ex) Splint, Flawfinder, McCabe, …

- **Debugger / dynamic checker**
  - VC++ IDE, Debugging Tools for Windows, Bound Checker, Purifier, …

- **Coverage checker**
  - 테스트 시 코드의 각 부분이 실제로 실행되었는지 검사

  Ex) TrueCoverage, PureCoverage, …

# 유용한 도구: 조직 및 시스템용

- 소스 코드 관리 시스템
  - 개발자의 중대한 실수로부터 프로젝트 소스를 보호할 수 있음

  Ex) CVS, Visual Source Safe, ClearCase, RCS, …

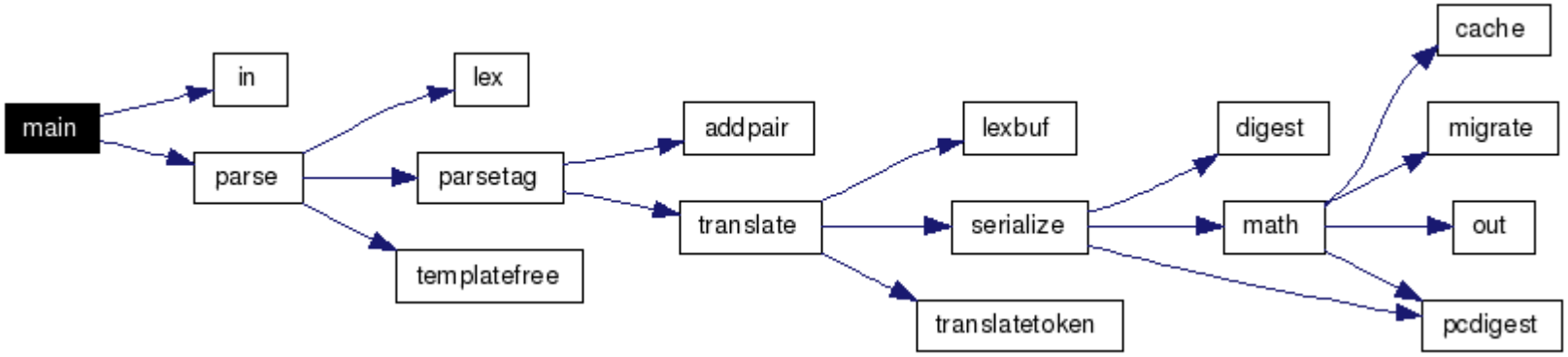
- 버그 추적 시스템
  - 발견된 버그 등록, 담당자 배정, 수정 확인, 수정 기록 및 통계 등

  Ex) Bugzilla, Mantis, …

# 유용한 도구: 기타

- 매뉴얼 생성기
  - 간단한 주석으로부터 자동 문서화
  - 함수간 call graph 자동 생성

Ex) Doxygen, …



# 질의 & 응답

# 감사합니다!