Defensive Programming–Outline

1. Asserts, and their use
2. What is an assert?
3. How does it work?
4. Examples of use
5. Where not to use asserts
6. General Tips on defensive programming

What is an `assert`?

An `assert` is a way of specifying something that should be true at a certain point in a program.

Example: *What can go wrong?*

```
void strlen(char *inp) {
     int i=0;
     while(inp[i] != 0) i++;
     return (i);
}
```

Example: *Is this a good fix??*

```
void strlen(char *inp) {
     int i=0;
     if (inp == (char *) NULL) {
        fprintf(stderr,'' NULL pointer argument!'');
        exit(1);
     }
     while(inp[i] != 0) i++;
     return (i);
}
```

So how about?

1. Run Time efficiency?
2. Malformed strings (no null termination?)

Another Fix

```
void strlen(char *inp) {
     int i=0;
#ifdef DEBUG
     if (inp == (char *) NULL) {
        fprintf(stderr,'' NULL pointer argument!'');
        exit(1);
     }
#endif
     while(inp[i] != 0)
#ifdef DEBUG
        if (i > MAXSTRINGSIZE) {
           fprintf( stderr, ``String is too long'');
           exit(1);
        }
#endif
        i++;
     return (i);
}
```

If something goes wrong, will these messages be helpful?

What additional information would be helpful?

Typical definition of `assert`:

```
#ifdef DEBUG
#define asssert(EXP) \
        (void)((EXP) || (__assert(#EXP, __FILE__, \
                                  __LINE__), \
                         0))
#else
#define assert(EXP) ((void) 0)
#endif
void __assert(const char *cond, const char *fn,
                                int ln) {
     fflush(stdout);
     fprintf(stderr,``%s failed at File:  %s, Line:  %d'',
                    cond, fn, ln);
     fflush(stderr);
     exit(1);
     }
/* —Version of strlen with assert— */
void strlen(char *inp) {
     int i=0;
     assert(inp != (char *) NULL);
     while (inp [i] != 0) {
           i++;
           assert(i <= MAXSTRINGSIZE);
     }
     assert(inp[i] == 0);
     return (i);
}
```

Where to use `asserts`

To validate arguments in subroutines: either invalid arguments, or cases where behaviour is undefined

```
RETURNTYPE myFun(TYPE1 arg1, ...TYPEn argn) {
            assert (...some condition about arg1... argn);
            ...
            ...
            ...
}
```

To validate return value from subroutines

```
RETURNTYPE myFun(TYPE1 arg1, ... TYPEn argn) {
            ...
            ...
            ...
            assert (... some condition about EXP);
            return ((RETURNTYPE) EXP);
}
```

Use asserts in loop bodies to avoid possible pointer overflow

Use asserts at the bottom of loops to ensure correct termination of loops **Why?**

Use asserts to catch incomplete switch statements **How?**

asserts are not the same as ERRORS!!

Some run time errors have to be explicitly handled, and are not good candidates for asserts*. Are these good uses of asserts?*

```
char *strdup(char *str) {
     assert(str != NULL); /* USE 1 */
     strNew = (char *) malloc(strlen(str) +1);
     assert(strNew != NULL); /* USE 2 */
     strcpy(strNew, str);
     return(strNew);
}

FILE *safeOpen4Write(char *fileName) {
     FILE *newFile;
     assert(fileName != NULL); /* USE 3 */
     newFile = fopen(fileName, ``w'');
     assert(newFile != NULL); /* USE 4 */
     return(newFile);
}

void getLine(char *bufPtr) {
     int ch;
     do
       assert((ch =getchar()) != EOF); /* USE 6 */
     while ((*bufPtr++ = ch) != '\n');
}
```

How do we decide what is an assert *and what is an error to be handled?*

Asserts in Java

Form 1: `assert` *Expression1*

Form 2: `assert` *Expression1 Expression2*

Run Like this:

`java -ea:`*class1* `-da:`*class2 mainclass*

Recommended uses:

1. Internal/Class Invariants (e.g., doubly-linked list, balanced tree)
2. Post conditions of public or private methods
3. Pre-conditions of private methods
4. Necessarily unreachable
5. Termination Condition

`http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html`

More tips on defensive Coding

1. Give code lots of opportunities to fail. *Example:* before freeing memory, put **trash** in it. Likewise right after mallocing, or reallocing. *What kind of trash??*

2. Force infrequent occurrences to happen frequently in your code. *Example:* Throw in artificial operations that force data structures into strange states, and cancel them out if necessary.

3. Inspect, inspect, inspect!!! Inspect every path! And don't forget "||" and "&&" short cuts.

4. *Read code a lot even before compiling* Your goal should be to write code that compiles first time without errors.

More Tips

For proper information hiding, and abstractness, a function interface should deal with error conditions and normal functioning distinctly

1. For C++, use exceptions. Don't return -1, NULL etc, `throw` an exception.
2. In C, use return code to indicate status of processing. Use a reference or pointer value to return actual argument. *how would* `malloc` *be done right?*

Use the simplest algorithm adequate for the job many of the searches can be linear search (rather than binary).

Don't use "fancy tricks". Such as using `memset(&i, 0, 3*sizeof(int))` to clear a set of variables, obscure initialization tricks, etc.

Conclusion: "Attitude"s of a super programmer

1. *Bugs don't go away nicely*. Which is it: Not a bug? Already fixed? Still exists? Not reproduced?

2. *Bugs are very patient.* Postponing bug fixes always costs more money.

3. *Root them bugs OUT!* Always ask "why is this bad value being passed here?" Keep chasing back.

4. *Don't let them hide!* Always ask, "Can I rewrite my code so that compiler could have caught this bug ?"

5. *Being Assertive* Always ask, "what `assert` would have caught this bug?"

6. *Be a Couch Potatoe- Use a tool!* If you are building some assumptions into your code, try to build tools (like `lint`) that can check for them automatically;

or comment it liberally, and use *assert*s if possible. Examples of tools: MIcrosoft SLAM, Association rule mining, clone detection, etc.