Safe Programming (in C) and Debugging

Administrivia:

- Freely ask questions by raising your hand throughout this class.
- Subscribe to the class mailing list. Use it to post questions (but not source code) and get answers from your classmates (or TAs) - this is the fastest way to get feedback if you're stuck.
- Send us questions you want us to address ***prior*** to sections (so we have some time to prepare; we don't know all the answers by heart).
- Email Valentin by the end of the week with the following info:
    - Which section you are in;
    - Prior C experience (yes/no; courses/industry jobs);
    - Prior Unix experience (yes/no; beginning/advanced);
    - Expectations from this class (1 sentence);
    - Other 400-level CS classes you've taken and currently taking.
- Modify project deliverables (2-3 min)
    - Make sure your code has comments in it for someone other than you to understand (Linux code you see is a ***bad*** example of how to write code that others can read.);
    - Comment all functions briefly (what they are supposed to do, what their arguments mean), e.g.:

```
/* converts input string pcStr to the integer value */
/* it contains (if correct) */
int Str2Int(char *pcStr, int *pnInt);
```

    - Include a README file with your submission to discuss all of the following:
        - Briefly describe any non-standard structures and algorithms you employ;
        - Write a 2-3 sentence assessment of the state of the project you are submitting: does it work, how did you test it, what problems are there still (if any), what would you like us to look for (if there's a problem you weren't able to resolve);
            - If you skip this or lie to us, we won't be able to help you as effectively as we otherwise could, so you will lose in the end.
            - If you worked for a company, do you seriously believe that someone will accept your code without these simple descriptions?

Questions:

- Answering questions about the project

Writing Solid Code:

- Assume as little as possible about the environment, compiler, operating system, etc. This will make your code much more robust, albeit sometimes a bit less efficient. Verify all conditions your program depends on.

Example:

Use `sizeof(int)` instead of `4`, since you'd be having non-portable code otherwise.

- Use assertions (from `assert.h`) to make sure your program behaves correctly. If the condition fails, the program exits immediately. Assertions are used for debugging, and incur no runtime overhead.

Example:

```
int getArrayElem(int pnArr[], int nLen, int nInd)
{
  assert(nInd < nLen && nInd >= 0);
  return(pnArr[nInd]);
}
```

- Type checking is not enforced in C (like it is in Java). To help yourself "enforce" it, use Hungarian notation: prefix pointer (and array) names by 'p', charaters by 'c', integers by 'n', long by 'l', etc. (You can invent your own shorthand.)

Examples:

```
int nIndex; char cInitial; char *pcString;
int pnArray[10]; int **ppnMatrix;
```

This allows you to avoid type mismatching - a common error - which C silently accepts) by looking at the types in an expression even if you don't know what they mean.

Example:

```
if (*pn == nIndex)       -> looks good (int = int)
*pcString = cInitial     -> looks good (char = char)
pnArray[3] = nIndex      -> looks good (int = int)
*ppnMatrix = pnArray[3] -> doesn't look good (int * <> int)
```

- Document pre-conditions and post-conditions for your functions. If a function can only be executed under a certain assumption (see above example with an array element by index), this is a pre-condition. Verify that it holds. If a function can only be successfully completed if it produces a certain result, this is a post-condition. Verify it too.

Example:

```
int searchIndex(int nElem, int pnArray[], int nLen)
{
   int nIndex;

   ... /* finds the index of nElem */
       /* within array pnArray of length nLen */

   assert(nIndex >= 0 && nIndex < nLen);
   return(nIndex);
}
```

- Compile with all warnings turned on, especially if you sense something fishy. For `gcc`, the right compiler option is `-Wall`. This may produce some false warnings, but will also give you a hint about where a problem may be lurking.

Debugging:

"It's a painful thing
To look at your own trouble and know
That you yourself and no one else has made it."
-- Sophocles, Ajax

- How it works in principle
    - The debugger loads the executable and rewrites it dynamically in memory, inserting an interrupt instruction before every program statement that is to be monitored. The function handling that interrupt displays what the user requested, including variable and expression values.
    - Main user primitives (interfaces) most debuggers offer:
        - step over
        - step into
        - step out
        - run to cursor
        - back trace (examine call stack)
        - set a breakpoint
        - evaluate an expression
        - watch a value
        - etc.

- Available debuggers
    - Debuggers integrated in development environments
        - MSVC for C/C++ on Windows, Eclipse for Java, etc.
            - Issues with portability (make sure you compile on the target system before you submit, so that what we get is what you saw before)
    - Standalone debuggers
        - gdb for C/C++ on UNIX; command-line tool, installed on instructional machines
            - ddd is a user friendly front end of gdb (and other Unix debuggers);
                - Not installed on instructional machines
                - Available at http://www.gnu.org/software/ddd/
                - See article "Visual Debugging Using DDD" (http://www.ddj.com/documents/s=869/ddj0103a/0103a.htm)
        - Features of gdb (and debuggers in general) you will find most useful;
        - How to use gdb; where to find it and how to get help – `man` pages on Linux.
    - Which ones have you used?