Advanced C Programming
Memory, Code Review, Matching Replacement Resolution,
Multi-Platform Code Management

Sebastian Hack
hack@cs.uni-sb.de
Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

Winter Term 2008/09
## Memory Management

### Own Memory Management May Pay Off

<table>
<thead>
<tr>
<th>Memory Management</th>
<th>Example</th>
<th>Time (s)</th>
<th>Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Own Module</td>
<td>ALG196+1</td>
<td>560</td>
<td>254819</td>
</tr>
<tr>
<td>Standard</td>
<td>ALG196+1</td>
<td>689</td>
<td>254819</td>
</tr>
</tbody>
</table>

### Conclusion

- about 20% faster
- own module can be faster when many small objects are involved and it is well done
Example 1: Code Organization

Meaningless Loop

do {
    /* Here is the DPLL main algorithm */
    /* manipulating "finished" */
} while (!finished);

Guidelines

- code meaningful statements
- reflect abstract algorithm
Example 2: Standard Data Structures

Non-Standard Lists

typedef struct LIST_HELP {
    int elem;
    int guessed;
    struct LIST_HELP* next;
} LIST_NODE;

typedef LIST_NODE* LIST;

/***************************/
/*This is the structure to implement linked lists where  */
/*elem is the content of the current list node  */
/*next is the pointer to the next element, possibly NULL  */
/***************************/

Guidelines

- documentation is part of programming
- do not abuse standard notions
Example 3: Efficiency

Lists for Assignments

```c
else { /* part of the DPLL mainloop */
    /* guess a literal and add it to M */
    DBG((MOD_SOLVER,3,"\t\t\t\tguess\t%d\n",undefined_literal));
    list_add(&M,undefined_literal,1);
    finished = 0;
    break;
}
```

Guidelines

- dealing with memory is expensive
- prefer assignment over address operator
- if the size of a structure is a priori constant implement it that way
Example 4: Efficiency & Encapsulation

Clause Set Evaluation

```c
/* part of the main DPLL loop */
for (i = 0; i < clauses_count; i++) {
    undefined_count = 0;
    undefined.literal = 0;
    /* evaluate each clause */
    /* clause set is an array of clauses */
    /* a clause is a list of literals */
    for (clause = N[i]; clause != NULL; clause = clause->next) {
        /* check if clause is true under M */
        if (listContains(M, clause->elem) == 1) {
```
Example 5: User Interface

SAT Solver Usage

```
lecture/ex2> ./SAT
ERROR: No file name given
USAGE: ./SAT <cnf-file>
       ./SAT -h
lecture/ex2> ./SAT -h
SAT solver for CNF formulas using the DPLL algorithm

USAGE: ./SAT <cnf-file> [OPTIONS]
       ./SAT -h

Options:
-h                Print this help screen and exit

lecture/ex2>
```

Guidelines

- deliver useful information
Economical Memory Usage

```c
typedef struct LIST_HELP {
    struct LIST_HELP * next;
    struct LIST_HELP * prev;
    void * data;
} LIST_NODE;

typedef LIST_NODE * LIST;

typedef struct CLAUSE_HELP {
    LIST literals;
    LIST watch[2];
} CLAUSE_NODE;
```
typedef struct LITCOUNT {
    int cnt_pos;
    int cnt_neg;
    int literal;
    int rev_idx;
} LITCOUNT;
typedef struct LITERALS_HELP {
    int size;
    int capacity;
    long * data;  /* Array with literals */
    LITCOUNT * count;
    LIST * clauses;
} LITERALS_NODE;
typedef LITERALS_NODE * LITERALS;

Guidelines

- less memory consumption typically means faster code
- draw ASCII picture of structures with references
Example 7: Filenames

## Source Files

```bash
lecture/ex3> ls
algorithm.c  datastructures.c  debug.c  Makefile
memory.h     misc.h          parser.h  parser_main.h
algorithm.h  datastructures.h debug.h   memory.c
misc.c       parser.c        parser_main.c

lecture/ex3>
```

## Guidelines

- assign meaningful names to files
Function Definition

```c
#ifdef TWO_WATCH
struct VAL* solveSAT(struct VAL *val, struct CNF *cnf)
#else
struct VAL* solveSAT(struct VAL *val, struct CNF *cnf,
                      struct WATCH_LIST* wl)
#endif
{
#ifdef TWO_WATCH
int unitLiteral;
/* continues ... */
```
Example 9: Efficiency

### Pick Next Undefined Variable

```c
int pickUndefinedVariable(struct VAL *val, struct CNF *cnf) {
    for (i=0; i < cnf->numberOfVariables; ++i) {
        /* grab literal */
        while (valLit != NULL) {
            /* check if defined */
            valLit = valLit->next;
        }
    }
    return result;
}
```

### Guidelines

- has to be done in (almost) constant time
Efficient SAT Implementation

Hints

- no call to malloc after input phase, i.e., during search
- prefer arrays over lists
- push crucial operations to constant time (if possible)
- profile
Definition: Resolution

From $C_1 \lor L$ and $C_2 \lor \neg L$ conclude $C_1 \lor C_2$.

Definition: Merging Replacement Resolution

Consider two clauses $C_1 \lor L$ and $C_2 \lor \neg L$ such that $C_1 \subseteq C_2$. Then replace $C_2 \lor \neg L$ with $C_2$.

Examples

- Replace $P \lor Q$ by $P$ in the presence of $P \lor \neg Q$
- Replace $P \lor \neg Q \lor \neg R$ by $P \lor \neg Q$ in the presence of $\neg Q \lor R$
Merging Replacement Resolution: Implementation

Hints

▶ Given a literal $L$ find fast ways getting all clauses containing $\neg L$
▶ Given two clauses $C_1 \lor L$, $C_2 \lor \neg L$ find constant time criteria for $C_1 \nsubseteq C_2$
▶ Find an at most linear implementation for $C_1 \subseteq C_2$ (recall marking algorithms)
The Dimensions: Products

- shared files (e.g., parser)
- several developers
- several versions (e.g., two watched literals)
- several configurations (e.g., debug/optimized)
- several programs (e.g., SAT, normalization)
- several platforms
<table>
<thead>
<tr>
<th>The Don’ts</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ ifdef</td>
</tr>
<tr>
<td>▶ excessive makefiles</td>
</tr>
<tr>
<td>▶ code duplication</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>The Dos</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ keep it simple</td>
</tr>
<tr>
<td>▶ share what can be shared</td>
</tr>
<tr>
<td>▶ separate what is different</td>
</tr>
</tbody>
</table>
### Key Idea: Two Level Set Up

- a directory structure holding exactly what is needed for one product: sources, makefiles, libraries, test bed, etc.
- dynamic generation of this structure out of a given template structure

### The Concept

Solve the problem by code organization and standard processes.
# Multi-Platform Code Management: Directory Structure

## CMTREE - Code Management Tree

- hold all makefile templates

## CMHTREE - Code Management Help Tree

- tools for maintaining the trees
- test data/procedures
- actual releases

## Source Trees

- pi - platform independent source code
- pd - platform dependant source code
- pid - mixed source code
Multi-Platform Code Management: CMTREE

Makefile Structure

- makefile - top-level, includes all others, simple tasks
- makefile.tre - defines standard macros pointing to locations in the different trees
- platform-name.plt - defines platform specific information
- imports.imp - program specific information, get the source
- makefile.pi/pd/pid - dependency rules for the software
- makefile.llb/xxe/sse - building libraries, executables, script products

CMTREE

```
CMTREE
  |
  -- PLT
    |
    -- SUNOS.PLT
    |
    -- X86LINUX.PLT
```
## Multi-Platform Code Management: Processes

### Simple Start

- makenode - open node for programming
- getmakes - fetch and compose the makefile(s) for the node
- make import - fetch the sources
- start working