A short introduction to system programming in C

Pointer programming, Storage Classes, Compiling, Linking, Loading, Debugging

Java vs. C

Java
- For application programming only
- Design goals:
  - Programmer productivity
  - Safety
  - Hardware completely hidden
  - Comfortable
  - E.g. automatic memory management
  - Protection
  - E.g. array bound checking
  - Slow

C
- For system programming mainly
- Design goals:
  - Direct control of hardware
  - High performance / real-time
  - Minimalistic design
  - Less comfortable
  - Little protection
  - "low-level"

A Short History of C

- C was developed in the early 1970's by Dennis Ritchie at Bell Labs (-> UNIX)
  - Objective: structured but flexible programming language e.g. for writing device drivers or operating systems
  - Book 1978 by Brian W. Kernighan and Dennis M. Ritchie ("K&R-C")
- "ANSI-C" 1989 standard by ANSI ("C89")
  - Became the basis for standard C++
  - Java borrowed much of its syntax
  - The GNU C compiler implemented a superset ("GNU-C")
- "ISO-C" 1999 standard by ISO, only minor changes ("C99")

Common organization of a C program

- A C program is a collection of C source files
  - Module files (extension .c) contain implementations of functionality (similar java)
  - Header files (extension .h)
    - contain declarations of global variables, functions, types
    - include’d from all module files that need to see them, using the C preprocessor (missing in Java)
  - When a module file is compiled successfully, an object code file (object module, binary module) is created (.o, corresponds to Java .class files)
  - An executable (program) is a single binary file (a.out) built by the linker by merging all needed object code files
  - There must be exactly one function called main()

Common organization of a C program (2) Example

- C run-time library is linked with the user code
Common organization of a C program (3)

Example

glob.h

abc.c

mymain.c

/**Comment: declaration of globally visible functions and variables: */
extern int incr( int );
extern int initval;

definition of var. initval:
int initval = 17;
definition of func. incr:
typedef struct
int incr( int k )
{
  return k+1;
}

#include <stdio.h>
#include "glob.h"
int counter;  // locally def.
void main( void )
{
  counter = initval;
  printf("new counter: %d",
  incr( counter ) );
}

Note the difference:
Declarations announce signatures (names+types).
Definitions declare (if not already done) AND allocate memory.
Header files should never contain definitions!

Data types in C

■ Primitive types
  • integral types: char, short, int, long; enum
    • can be signed or unsigned, e.g. unsigned int counter;
    • sizes are implementation dependent (compiler/platform),
      use sizeof( datatype ) operator to write portable code
  • floatingpoint types: float, double, long double
  • pointers

■ Composite data types
  • arrays
  • structs
  • unions
  • Programmer can define new type names with typedef

Constants and Enumerations

■ Constant variables:
  const int red = 2;
  const int blue = 4;
  const int green = 5;
  const int yellow = 6;

■ Enumerations:
  enum { red = 2, blue = 4, green, yellow } color;
  color = green;       // expanded by compiler to:  color = 5;

■ With the preprocessor:
  • symbolic names, textually expanded before compilation
    #define red 2
    ... (no =,  no semicolon)
  • In C, constants are often capitalized: RED, BLUE, ...

Composite data types (1): structs

struct My_IComplex  {
  int re, im;
};

typedef struct My_IComplex icplx;    // introduces new type name icplx
icplx y;

void main ( void )
{
  x.re = 2;
  y.im = 3 * x.re;
  printf("x needs %d bytes, an icplx needs %d bytes
", sizeof( x ), sizeof( icplx ) );
}

// Remark:  the sizeof operator is evaluated at compile time.

Composite data types (2): unions

■ Unions implement variant records:
  all attributes share the same storage (start at offset 0)
■ Unions break type safety
  • can interpret same memory location with different types
■ If handled with care, useful in low-level programming

union My_flexible_Complex  {
  struct My_IComplex ic;    // integer complex (re, im) – see above
  float f;   // floatingpoint value – overlaps with ic
};

c.ic.re = 1141123417;
printf("Float value interpreted from this memory address contents is %f
", c.f);
// writes 528.646057

Arrays

■ Declaration/definition very similar to Java:
  int a[20];
  int b[] = { 3, 6, 8, 4, 3 };
  icplx c[4];
  float matrix [3] [4];

■ Addressing:
  Location of element a[i] starts at: (address of a) + i * elsize
  where elsize is the element size in bytes

■ Use:
  a[3] = 1234567;
a[21] = 345;
  // ??, there is no array bound checking in C

■ Dynamic arrays: see later
■ Arrays are just a different view of pointers
Pointers

- Each data item of a program exists somewhere in memory
- Hence, it has a (start) memory address.
- A variable of type `pointer_to_type_X` may take as a value the memory address of a variable of type X

```c
int a;
int *p; // defines just the pointer p, not its contents!
p = 0x14a236f0; // initialize p
```
- Dereferencing a pointer with the * operator:
  ```c
  *p = 17;   // writes 17 to whatever address contained in p
  ```
- The address of a variable can be obtained by the & operator:
  ```c
  p = &a; // now, p points to address of a
  ```
- Use the NULL pointer (which is just an alias for address 0) to denote invalid addresses, end of lists etc.

```

Pointers and structs

```
struct My_IComplex { int re, im; } c, *p;
p = &c;
p is a pointer to a struct
p->re is shorthand for *(p + (offset of re))
Example: as p points to c, &(p->re) is the same as &(c.re)
Example: elem->next = NULL;
```

```
Why do we need pointers in C?

```
Pointers to functions (1)

```
```
Points to functions (2)
- Most frequent use: generic functions
- Example: Ordinary sort routine
  ```c
  void bubble_sort( int arr[], int asize )
  {
    int i, j;
    for (i=0; i<asize-1; i++)
      for (j=i+1; j<asize; j++)
        if ( arr[i] > arr[j] )
          ... // interchange arr[i] and arr[j]
  }
  ```
- Need to rewrite this for sorting in a different order?
  - Idea: Make bubble_sort generic in the compare function

Points to functions (3)
- Most frequent use: generic functions
- Example: Generic sort routine
  ```c
  void bubble_sort( int arr[], int asize, int (*cmp)(int,int) )
  {
    int i, j;
    for (i=0; i<asize-1; i++)
      for (j=i+1; j<asize; j++)
        if ( cmp ( arr[i], arr[j] ) )
          ... // interchange arr[i] and arr[j]
  }
  ```
  ```c
  int gt ( int a, int b )  {
    if (a > b)
      return 1;
    else return 0; }
  ```
  ```c
  bubble_sort ( somearray, 100, gt );
  bubble_sort ( otherarray, 200, lt );
  ```

Storage classes in C (1)
- Automatic variables
  - Local variables and formal parameters of a function
  - Exist once per call
  - Visible only within that function (and function call)
  - Space allocated automatically on the function’s stack frame
  - Live only as long as the function executes
  ```c
  int *foo ( void )    // function returning a pointer to an int.
  {
    int t = 3;     // local variable
    return &t;   // ??  t is deallocated on return from foo,
      // so its address should not make sense to the caller...
  }
  ```

Storage classes in C (2)
- Global variables
  - Declared and defined outside any function
  - Unless made globally visible by an `extern` declaration, visible only within this module (file scope)
  ```c
  int x;    // at top level – outside any function
  extern int y; // y seen from all modules; only declaration
  int y = 9; // only 1 definition of y for all modules seeing y
  ```
- Static variables
  - `static` int counter;
  - Allocated once for this module (i.e., not on the stack) even if declared within a function!
  - Value will survive function return: next call sees it

Dynamic allocation of memory in C
- `malloc( N )`
  - Allocates a block of N bytes on the heap
  - And returns a generic (void *) pointer to it;
  - This pointer can be type-casted to the expected pointer type
  - Example: icplx *p = (icplx *)malloc ( sizeof ( icplx ));
- `free( p )`
  - Deallocates heap memory block pointed to by p

Can be used e.g. for simulating `dynamic arrays`:
- Recall: `arrays are pointers`
- `int *a = (int *) malloc ( k * sizeof(int) );`  `a[3] = 17;`

Run-Time Memory Organization
- Automatic variables
  - (formal parameters, local variables of functions)
  - Allocated on the run-time stack at function call and removed on return
  - Indexed by frame pointer (or stack pointer)
- Global and static variables
  - Reside in the .data segment
  - Indexed by absolute address (or global pointer)
- Dynamically allocated objects
  - Reside on the heap
  - Indexed via pointer variables
- Compiled program code (for function bodies)
  - Reside in the .text segment
  - Branch target addresses: by absolute address or PC-relative

NB: This is a simplification.
**C: There is much more to say...**

- Type conversions and casting
- Bit-level operations
- Operator precedence order
- Variadic functions (with a variable number of arguments, e.g., printf())
- C standard library
- C preprocessor macros
- I/O in C

... 

**Compiling and Linking**

- include
- abc.h
- def.h
- stdio.h
- print.h
- glob.h
- mymain.c
- def.c
- compile
- link
- a.out (executable)

**Compiling C Programs**

- Example: GNU C Compiler: gcc (Open Source)
- One command calls preprocessor, compiler, assembler, linker

**Single module:** Compile and link (build executable):
  - gcc mymain.c executable: a.out
  - gcc -o myprog mymain.c rename executable

**Multiple modules:** Compile and link separately
  - gcc -c -o mymain.o mymain.c compiler only
  - gcc -c -o other.o other.c compiles other.c
  - gcc other.o mymain.c call the linker, -> a.out

**make** (e.g., GNU gmake)
- automates building process for several modules
- Check the man pages!

**How to relocate code?**

- Needed in both static linking and loading
- Patching
  - Absolute Jumps
  - Adresses of global variables

**Symbolic addressing:**
- Definition of data X
- If (*X) goto P

**Relative addressing:**
- 0: ...
- 1: Space for data
- 2: ...
- 3: If (*1) goto 5
- 4: ...
- 5: ...

**Absolute addressing:**
- 243: ...
- 244: Space for data
- 246: ...
- 246: If (*244) goto 248
- 247: Static loading
- 248: ...

**Linking Multiple Modules (1)**

- Compiler (-c) created an **(object) module file** (.o, .bin)
  - Binary format, e.g. COFF (UNIX), ELF (Linux)
  - Non-executable (yet)
  - **Segments** for code, global data, stack / heap space
  - List of global symbols (e.g., functions, **extern** variables)
  - Addresses in each segment start at 0

**Relocation table:**
- List of addresses/instructions that the linker must patch when changing the start address of the module

**Static relocation** (at compile/link time):
- Merge all object modules to a single object module, with consecutive addresses in each segment type

**Linking Multiple Modules (2)**

- C run-time library is linked with the user code

- Linker
- Loader
- Binary module (program, e.g., ...
Background: How the Linker Works

- Read all object modules to be linked (including library archive modules if necessary)
- Merge the code, data, stack/heap segments of these into a single code, data, stack/heap segment
- Resolve global symbols (e.g., global functions, variables): check for duplicate globals, undefined globals
- Write the resulting object module, with a new relocation table
- and mark it executable.

Variants of static linking: (need hardware support)
- Dynamic linking (on demand at run time, as in Java)
- Shared libraries

Variants of static linking: (need hardware support)
- Dynamic linking (on demand at run time, as in Java)
- Shared libraries

Sources of information

- Lab-Kickoff Session
  - Programming exercise: C pointer programming and debugging with DDD
- DDD tutorial
- Books on C and C++
  - ... and many other good books
- Open IDA course on C and C++
  - TDDB89 Advanced Programming in C++. Given every semester.

Homework

- Practise!
- Attend the Lab-Kickoff session
- Read the DDD tutorial on the course homepage
  - The following material may also be of some interest.

Appendix: Debugging

Debugging vs. Testing

Testing: may detect existence of a programming error (but cannot guarantee absence of errors!)
- Compare output of testing candidate with reference output (e.g. of an older, working version – Regression testing, e.g. DEJAGNU)

Debugging: localise error: Cause ➔ Effect
- Iterative Process
- Systematic Isolation

Tools for manual debugging:
- Symbolic Debugger
  - e.g. dbx, gdb, jdb, ddd
- Debug-Problem Documentation
  - e.g. BUGZILLA (Bug database + Web interface)
- Runtime-Protection Monitoring, esp. for memory access errors:
  - ElectricFence, VALGRIND, Java VM, INSURE++, PURIFY,...

Debugging Techniques and Tools (1)

- Manual Methods
  - Static: Code-Inspection
  - Dynamic: printf-statements, Validation of assertions (in C: assert(p) macro in assert.h)

Tools for manual debugging:
- Symbolic Debugger
  - e.g. dbx, gdb, jdb, ddd
- Debug-Problem Documentation
  - e.g. BUGZILLA (Bug database + Web interface)
- Runtime-Protection Monitoring, esp. for memory access errors:
  - ElectricFence, VALGRIND, Java VM, INSURE++, PURIFY,...
Debugging Techniques and Tools (2)
- Automatic Debugging:
  - Formal Verification against a formal specification of the program
    - Often no or incomplete specification available
    - Perhaps can derive specification (but then itself error-prone)
  - Search source program for language-specific error idioms
    - e.g., lint, splint, jlint
    - Incomplete
  - Reduce scope of searching by static program analysis:
    - Program Slicing [Weiser'82] [Lyle, Weiser'87]
    - Program Dicing (Difference of 2 slices) [Lyle, Weiser'87]
    - e.g., UNRAVEL slicer [Lyle'95]
    - Needs good static analysis (data flow analysis, points-to-analysis)
    - Conservative static approximation – Slices quickly get large
  - Delta-Debugging
    - Automatic scope reduction by binary partitioning
    - (input data, code)

Symbolic Debugger (1)
- Needs information about name and type of memory locations on source code level
  - i.e., the symbol table and type table of the compiler
  - added to generated binary file on request (e.g., line nr.)
- Needs tight similarity of control flow between source code structure and generated machine code
  - Incompatible with aggressive compiler optimizations
  - e.g., Prefetching, Loop-invariant code hoisting, Loop transformations, Scheduling
- Trade-Off Code-Efficiency Debugger-Transparency
  - Can under certain circumstances lead to "Heisenbugs" - Error does not occur if run with debugger
  - (may even happen with printf-debugging)
- Graphic User Interface (e.g., ddd, Eclipse Debug-View)

Symbolic Debugger (2)
- Post-Mortem-Debugging
  - After crash: Read core-file; inspect memory contents, variables values
- Interactive Debugging
  - Stop computation
  - Set / clear breakpoints
  - Step-by-step execution
  - Print values, evaluate expressions (Interpret)
  - Change values of variables
  - Inspect stack contents (call chain)
  - Walk call chain upwards / downwards

Debugging concurrent programs
- Problem: Occurrence of a bug can depend on the schedule
  - Run 1: CPU
    - Thread 1
    - Thread 2
    - Thread 1
    - Thread 2
  - Run 2: CPU
    - Thread 2
    - Thread 1
    - Thread 2
  - Non-determinism → hard to reproduce the error

Solution 1: Deterministic replay
- Record inputs and schedule
  - e.g., DEJAVU for Java

Solution 2: Static Analysis (potential concurrency, "Data-races")

Solution 3: Dynamic Analysis
- identifies shared-memory accesses at runtime

Solution 4: Test-based Approach with Delta-Debugging [Choi, Zeller ’02]
- In combination with DEJAVU

Debugger-Technique with OS/HW-Support
- Debugger-Process
  - fork() (via OS)
  - ptrace() (via OS): "trace me"
  - signal() (via OS): "stop"
  - wait() (via OS)
  - ptrace() (via OS)
  - read, write values in address space, add breakpoints (special instructions)
  - …
  - signal() (via OS): "resume"
  - signal(): "continue"

- OS
  - trap

- Find breakpoint:
  - signal(): "Breakpoint"
  - …
  - signal(): "continue"

- Interpreters

Debugging - Summary and Literature
- Testing vs. Debugging
- Debugging Methods
- Debugger Technology
- Debugging Concurrent Programs

DDD: www.gnu.org/software/ddd/
- see also the DDD tutorial on the course home page

Chapter on Debugging

Srikant, Shankar: Compiler Design Handbook, CRC press 2003,
Chapter 9 on debugger technology (by Aggarwal and Kumar)


Morgan Kaufmann, 2005.