

# Lecture 12: C and UNIX Programming

## CSCI0320: Introduction to Software Engineering

July 13, 2009  
Steven P. Reiss

### I. Learning a New Language

#### A. What does it take to pick up a second language?

##### 1. For natural languages

- a) Learn the grammar and vocabulary is sufficient
- b) But this takes a long time

##### 2. For computer languages

- a) Grammar and vocabulary are trivial
- b) But they are not sufficient
- c) Must learn to “think” in the new language
  - (1) Set of rules for a game
- d) Must understand how the language is used
  - (1) How it is used most effectively
  - (2) How it is used in practice
  - (3) How it should be used for particular applications

#### B. Learning C

##### 1. Assume that you know Java

- a) You know how to design in terms of objects & interfaces
- b) You know how to build methods, classes, & packages

##### 2. What do you need to know to code in C

- a) What does it mean to think in “C”
- b) What do C programs look like

##### 3. C programs vary all over the book

- a) C as Pascal, Fortran, assembler
- b) Naming conventions have changed over the years
- c) But there is a C standard that has evolved
- d) That is what we will teach

## **II. Procedural Programming**

### **A. The first difference between C & Java**

- 1. Java deals with classes and their fields and methods**
- 2. C deals with variables and functions**
  - a) Variables are similar to fields
  - b) Functions are similar to methods
- 3. What is missing is the class organization**
  - a) Including class hierarchy, formal interfaces, etc.
  - b) Also Java's package structure and libraries
- 4. This doesn't mean you can't think in terms of objects**

### **B. For simple programs this doesn't matter**

- 1. Single class is equivalent to a C program**
- 2. But for complex programs it does matter**

## **III. Write a LIFE program**

### **A. Basics of the game of life**

- 1. Array of squares**
- 2. Generations**
  - a) Count neighbors
  - b) Full with 0,1 neighbor => die
  - c) Full with 4+ neighbors => die
  - d) Empty with 3 neighbors => create

### **B. How would you design this in Java**

- 1. Some of you already have (cs15?)**
- 2. Classes, etc.**

### **C. C works procedurally**

- 1. Overall structure of system**
- 2. Does this mean there are no data structures?**

### **D. C thinks in terms of modules**

- 1. Module == class without inheritance**
  - a) Separate interface and implementation
  - b) But there is a 1-1 relationship between the two

(1) Not independent (or not as much as Java)

## **2. Abstract data structures**

- a) Similar to classes
- b) But no this variable and no constructors
- c) Routine that returns a hidden type
- d) All other routines take a parameter of that type as the first argument

## **E. Levels of abstraction in C**

### **1. What are the levels of abstraction in Java**

- a) Namespaces based on package - class
- b) Namespaces within a class
- c) Nested classes
- d) Interfaces with multiple implementations

### **2. Global name space**

- a) Variables and functions defined here are available everywhere
- b) Single top-level name space, no overloading
- c) Conflicting names are not allowed

### **3. File name space**

- a) Variables and functions can be restricted to a single file
  - (1) Only defined & accessible in that file
- b) Nothing really inbetween global and file-local
  - (1) Contrast to Java: package, protected, interfaces

### **4. Much of C programming is designed to deal with this**

## **F. Outline of the program**

- 1. Set up board routine**
- 2. Board management**
- 3. Play routine**
- 4. Output routine**

## **G. Bit twiddling**

- 1. What is the most efficient board representation?**
- 2. How to compute the next generation efficiently**

## IV. File Structure

### A. Compiler's point of view

#### 1. Program consists of multiple files

- a) Each is compiled separately
  - (1) Diagram
- b) Each is internally complete
  - (1) All referenced variables are defined in that file
  - (2) All referenced functions are defined in that file
  - (3) External definitions are allowed
- c) Compiler reads the file from start to finish
  - (1) Variables & functions used must be defined first
  - (2) Within the file, before their use
  - (3) Single pass compiler -- how does Java differ?
- d) Files are bound together in a separate operation

#### 2. Loading

- a) Takes all the separately compiled files
- b) Takes all libraries that are explicitly listed
- c) Creates a single executable from this
- d) Show in the diagram

### B. Programmer's point of view

#### 1. Each file and library needs to work with others

- a) Its global definitions must be accessible to other files
- b) Its global definitions must be distinct from others

#### 2. C deals with this w/ preprocessor and header files

- a) Each file/library defines a header file that includes its global definitions
- b) This file then becomes part of the compilation of any accessing file
- c) Using `#include <file>` or `#include "file"`
- d) Header files are `<package>.h`

#### 3. Standard include files

- a) `assert.h`

- b) ctype.h
- c) stdio.h
- d) math.h
- e) errno.h
- f) stdlib.h
- g) signal.h
- h) string.h (copy ops)

#### 4. Lower case names

### C. C Preprocessor (cpp)

#### 1. Meta language outside of C

#### 2. Cpp is run over the source file first

- a) The result is what is actually compiled
- b) Show this in the compilation diagram

#### 3. Simple text processing language

- a) Based on macro assemblers (macros)

#### 4. Statements

- a) #include <file> or #include "file"
- b) #define NAME string
- c) #define NAME(arg,arg) string\_with\_args
- d) #ifdef, #ifndef, ... #endif
- e) #if (expression)
- f) #undef

#### 5. Want to minimize use of the preprocessor

- a) Why?
- b) Readability (macros used far from definition)
- c) Tool accessibility
- d) Semantics : string substitution is not semantic substitution
  - (1) #define VALUE 3 + 3.14159
  - (2) x = VALUE \* 2

## V. Lets write the game of life in C

### A. Start with life\_local.h

1. Definitions for board size
2. Definitions for other modules entry points

### B. Notes

#### 1. Definitions needed by other files

- a) Functions, types, constants,

#### 2. Start with block comment

#### 3. Then check for multiple inclusion

- a) `#ifndef HEADER_FILE_ALREADY_INCLUDED`
- b) `#define HEADER_FILE_ALREADY_INCLUDED`
- c) ...
- d) `#endif`

#### 4. Then include other header files that are needed/used

- a) `#include ...`

#### 5. Inside put the various definitions

- a) Types
- b) Functions
- c) Constants

#### 6. Constants

- a) Traditionally done with `#define`
- b) Today should be done using `const` declarations
  - (1) Can be tricky (don't want to allocate storage)
  - (2) Different compilers treat differently

#### 7. Function definitions

- a) Like Java interface definitions
- b) `extern <type> function(args...)`
- c) Parameters don't need to be named
- d) Extern is required

## VI. Types

### A. In Java types are relatively simple

1. Primitive types
2. Interfaces and classes
3. Arrays
4. C has a more complex (and simpler) type structure

### B. Primitive Types

#### 1. C Primitives

- a) char, short [int], int, long [int], long long
- b) float, double, long double
- c) void
- d) unsigned char, signed char
- e) unsigned short, unsigned int, unsigned long, unsigned long long
- f) Note that sizes aren't specified
  - (1) int is natural integer for the machine
  - (2) long is long enough to hold a pointer
  - (3) char is long enough to hold a character (not unicode)
- g) const X, volatile X

#### 2. What's missing

- a) boolean :: any value != 0 (int most common)
- b) Object, String

#### 3. typedef: user names for types

- a) typedef int Int32;
- b) Look like declarations, but the item being declared is a type
  - (1) Same type as a variable declared in that situation
- c) Very useful construct
  - (1) Provides abstractions for use types
  - (2) Makes type definitions readable

### C. Structs

#### 1. Java classes provide a way of grouping related data

- a) Example: Point has x and y data
- b) Example: List element has reference to next, reference to prior, and a value

## 2. In C these are represented as structs

- a) `struct Point { int x_pos; int y_pos; }`
  - (1) Note naming conventions
- b) Represented as a block of memory
  - (1) Draw picture
- c) Names defined here are in their own namespace
  - (1) Need to use 'struct Point' as the type
- d) Typically combine with typedef:
  - (1) `typedef struct _Point { ... } Point;`
  - (2) Use different names for C++

## 3. LIFE data structures: board

### D. Pointers

#### 1. Java objects are implemented as structs

- a) Stored as blocks of memory
- b) How do you implement a reference to an object?
- c) By the address of its block of memory

#### 2. C does this explicitly by defining pointer types

- a) `Point *` represents a pointer to a Point structure
- b) `typedef struct _ListElt {`
  - `struct _ListElt * next_elt;`
  - `struct _ListElt * prior_elt;`
  - `int elt_value;`
  - `} ListEltInfo, *ListElt;`
  - (1) This defines 2 types, ListEltInfo and ListElt
  - (2) Which is more useful (and why?)
  - (3) Which corresponds to Java class?

#### 3. Forward definitions of structures

- a) `typedef struct _ListElt *ListElt;`
- b) Then use ListElt inside the definition

#### **4. void \* is a generic pointer**

#### **5. Accessing structure elements**

- a) `x . y` used if you have the actual structure
- b) `x -> y` used if you have a pointer

#### **6. No checking for invalid pointers**

### **E. Strings**

#### **1. Pointers used for more than structures in C**

#### **2. A C string is a block of memory containing text**

- a) Ascii characters followed by a null (0) byte
- b) Represented as `char *`
- c) Picture

#### **3. C strings are mutable (unlike Java strings)**

- a) Very different way of thinking of string processing
- b) If you think Java is bad for dealing with strings, wait until you deal with C

### **F. Arrays**

#### **1. What is an array**

- a) A sequence of zero or more entities of the same type
- b) Array of `int`, `char`, structures, pointers

#### **2. C deals with memory**

- a) An array is stored as one entity after another
- b) A `char` array is stored as ...
  - (1) This is the same as a string

#### **3. Arrays are basically pointers**

- a) `char *` is equivalent to `char []`
- b) `int []` is equivalent to `int *`
- c) No bounds checking done in C (buffer overflow)
- d) Quite different from Java (arrays are objects)

#### **4. Arrays can specify bounds**

- a) `int x[10]` (compare to Java)
- b) Dimension must be defined
  - (1) Inside structure, must be constant

c) What does this do in C versus what it does in Java

**5. Can have multidimensional arrays `x[10][20]`**

- a) All but the last dimension must be specified explicitly
  - (1) With constants!
- b) How to allocate a dynamic array?

## **G. Enums**

**1. C provides enumerations to define constants**

- a) Better than `#define`, often better than `const`
- b) Type checked at least in part

**2. `enum _Name { A, B = 5, C };`**

**3. Different from Java enums**

- a) Integers, not objects
- b) Automatic cast back and forth to integer

**4. Typically done using a typedef**

## **H. Unions**

**1. A structure is a block of memory**

**2. What do you do if you sometimes need an integer, sometimes a `Point`, sometimes a pointer?**

- a) In Java you create separate subclasses
- b) Don't have subclasses in C

**3. C provides union types**

- a) Look like structures
- b) Except that the memory for each field overlaps

**4. Again, no checking for safe usage**

**5. Generally you want to avoid unions**

- a) Memory is cheap: use a structure instead

## **I. Functions**

**1. In Java we don't often think of the type of functions**

- a) The compiler does to check argument types
- b) The compiler does to handle overloading
- c) Function types often viewed as object types (`Comparator`)

**2. How does Java handle virtual calls?**

- a) Essentially there is a table of methods for each object
- b) When you make a virtual call, Java finds the corresponding entry in the table
- c) That entry tells where in the code the corresponding virtual routine exists
- d) But this where is just an address: a pointer

### **3. C lets you define this type of pointer**

- a) Pointer to a function
- b) Want to tell C the argument types
  - (1) So compiler can do checking
  - (2) Not required

### **4. Usually done with a typedef**

- a) `typedef int (*IntFunction)(int);`
- b) Not this can get quite confusing

## **VII.Code Files**

### **A. Start with block comment**

- 1. And comment throughout

### **B. Start with #include for each header file required**

### **C. Then variable definitions**

- 1. `static <type> var [= value]` for local variables
- 2. `<type> var [= value]` for global variables
  - a) Even though a global is defined in an extern in a .h file
  - b) Should be defined once in a code file
  - c) Not always required (depends on loader)

### **D. Local types should probably be avoided**

- 1. Define a local (directory or project) .h file instead

### **E. Forward function definitions**

- 1. `static <type> fct(<params>);`
- 2. Needed so compiler can work correctly

### **F. Then the actual function codes**

- 1. Pretty much like Java

2. **Use static if the function is local**
3. **Don't use extern if it is global**

## **VIII.C Code**

### **A. Declarations first in the code**

1. **Actually declarations must be the first thing in a block**
2. **Can't have declarations intermixed with code**
3. **Can't declare inside for statement**

### **B. Storage model**

#### **1. Java objects appear on the heap**

- a) Local variables on the stack
  - (1) Only references to objects and primitives

#### **2. C has a different storage model**

- a) Local variables: stack and registers
  - (1) register declaration pretty much ignored today
- b) Static/external storage in fixed data area memory
- c) Heap supported, not part of language per se

#### **3. Declarations**

- a) At top level:
  - (1) extern :: defined elsewhere
  - (2) static :: defined in fixed area, accessible locally
  - (3) <none> :: defined in fixed area, accessible globally
- b) Inside code
  - (1) extern :: defined elsewhere
  - (2) static :: defined in fixed area, accessible in code
  - (3) <none> :: defined on stack or register
  - (4) register :: defined on stack or register

#### **4. Other language differences**

- a) No exceptions
- b) Casts are not checked
- c) No automatic initialization
- d) No new statement for allocation

- e) short + short is int
- f) Pointer arithmetic
- g) Builtin functions :: sizeof
- h) No garbage collection
- i) Goto statement
- j) if (x = 5) ... is valid code

## 5. Pointer arithmetic

- a) Pointer is an address into memory
- b) Adding to the pointer is still an address in memory
- c) Pointer + int = pointer
  - (1) What does +int mean :: depends on type
  - (2) a[i] == a+i
  - (3) pointer++
- d) Note what this means for strings
- e) Generally avoid pointer arithmetic
- f) &name == address of name
- g) \*ptr == contents of pointer