Mapping PL Objects to the Machine – managing the address space

David E. Culler
CS61CL
Feb 9, 2009
Lecture 3
Big Ideas

• Review:
  – Computers manipulate finite representations of things.
  – A bunch of bits can represent anything, it is all a matter of what you do with it.
  – Finite representations have limitations.

• Today
  – Type constructors to compose complex type
  – Mapping of program objects to machine storage
  – An object, its value, its location, its reference

• Pointers are THE most subtle concept in C
  – Very powerful
  – Easy to misuse
  – Completely hidden in Java
C Types - the big picture

• Basic Types
  – “understood” by the machine

• Array
  – sequence of indexed objects of homogeneous type

• Struct
  – collection of named objects of heterogeneous types

• Pointer
  – reference to an object of specified type

• Union
  – an object of one of a specific collection of types
Composing Complex Types in C

• Complex types are really tools for \textit{composing} new types
  – Strings – sequences of characters
  – Vectors – sequences of numbers
  – Matrixes – 2D collections of numbers
  – Records – finite sets of strings and numbers
  – Lists, Tables
  – Sounds, Images, Graphs
  – ...

  – Think induction

• Pointers are fundamentally “understood” by the machine as well
  – address
Where do Objects live and work?
Where do complex objects reside?

- Arrays are stored in memory
- The variable (i.e., name) is associated with the location (i.e., address) of the collection
  - Just like variables of basic type
- Elements are stored consecutively
  - Can locate each of the elements
- Can operate on the indexed object just like an object of that type
  - \( A[2] = x + Y[i] - 3; \)
Where do complex objects reside?

• Struct are stored in memory
• The variable (i.e., name) is associated with the location (i.e., address) of the collection
  – Just like variables of any type
• Elements are stored at fixed offsets
  – Can locate each of the elements
• Can operate on the named member object just like an object of that type
  – S.row = x + S.col – 3;
All objects have a size

• The size of their representation
• The size of static objects is given by sizeof operator

```c
#include <stdio.h>
int main() {
    char c = 'a';
    int x = 34;
    int y[4];
    printf("sizeof(c)=\%d\n", sizeof(c) );
    printf("sizeof(char)=\%d\n",sizeof(char));
    printf("sizeof(x)=\%d\n", sizeof(x) );
    printf("sizeof(int)=\%d\n", sizeof(int) );
    printf("sizeof(y)=\%d\n", sizeof(y) );
    printf("sizeof(7)=\%d\n", sizeof(7) );
}
```
What can be done with a complex object?

- **Access its elements**
  - A[i], S.row

- **Pass it around**
  - Sort(A)
  - x = max(A, n)

- **Copy it**
  - T = S
  - z = munge(S, 3)

- **Note the name of an array behaves as a reference to the object**

- **The name of a struct behaves as the object**
Administration

- HW3 due at start of next week’s lab
  - Try to have it give practice in test tools
- Lab changers and waitlisters must give target TA your prioritized lab request list this week
- Readings are shifting for K&R to P&H
- Project 1 goes out on Tuesday, Due Friday 10/1
An object and its value…

\[
X = X + 1;
\]

The value of variable \( X \)

The storage that holds the value \( X \)
Every object in memory has an address

- That address is a pointer to the object.
- It is a fixed size object itself
- Just like basic type
What can be done with a reference?

- Dereference it
  - Obtain the object that it refers (points) to
  - \( X = *P; \ Y = S->row; \ z = A[0]; \ z = A[i]; \)

- Pass it around, copy it, store it
  - \( Q = P; \)
  - clearfields(S);

- Do type-based arithmetic on it
  - \( P-1 \)
  - \( Q++ \)

- Do both
  - \( S->next = P; \)
  - \( A[i] = 3; \)

- Cast it to an uint and mess with it (!!!)
Array variables are also a reference for the object

- Array name is essentially the address of (pointer to) the zero\textsuperscript{th} object in the array
- There are a few subtle differences
  - Can change what \texttt{c} refers to, but not what \texttt{ac} refers to
What kinds of variables (storage)?

• **Visibility vs Lifetime**

• **Variables declared within a function**
  
  – Arguments and Local Variables
  
  – Visible in remainder of function
  
  – Lifetime = Function Call
  
  – Each call obtains a new set of variables
    
    » Recursive calls too
  
  – C “internals”

• **Variables declared outside any function**

  – Visible in remainder of file (!!!)
    
    » include .h file
    
    » extern vs static
  
  – Lifetime = Whole Program
  
  – C “externals”

• **Malloc’d objects**
Where does the program itself reside?

- In memory, just like the data
- Processor contains a special register – PC
  - Program counter
  - Address of the instruction to execute (i.e. ptr)
- Instruction Execution Cycle
  - Instruction fetch
  - Decode
  - Operand fetch
  - Execute
  - Result Store
  - Update PC
What’s a Process

• Address Space + a thread of control
Logical Structure of an Executing Program

- code
  - printf:
  - main:
  - nextword:

- regs

- PC

- static data

- stack

- heap
Address Space

0000000: <= Local variables
0000000: <= malloc
0000000: <= instructions
0040000: <= externs
1000000: <= OS, etc.
1008000: <= malloc
1008000: <= Local variables
7FFFFFFF: <= stack
7FFFFFFF: <= unused
0000000: <= code
0040000: <= static data
1000000: <= heap
Breaking the Abstraction…

• Attack
  – Cause the OS (or service or application) to do things it should not.
  – Pass unterminated strings, bad length parameters, bad ptrs
  – Corrupting system data may cause it to do other harm

• “Smashing the stack”
  – Send bad msg causing system code to overwrite parts of its stack
    » Local vars and rtn address
    » Bad return
  – OS or app starts executing out of stack as if it were instructions
  – Message contains jump instructions to send it off to attacker code
Summary

• Arrays, Structs, and Pointers allow you define sophisticated data structures
  – Compiler protects you by enforcing type system
  – Avoid dropping beneath the abstraction and munging the bits

• All map into untyped storage, ints, and addresses

• Executing program has a specific structure
  – Code, Static Data, Stack, and Heap
  – Mapped into address space
  – “Holes” allow stack and heap to grow
  – Compiler defines what the bits mean by enforcing type
    » Chooses which operations to perform

• Poor coding practices, bugs, and architecture limitations lead to vulnerabilities