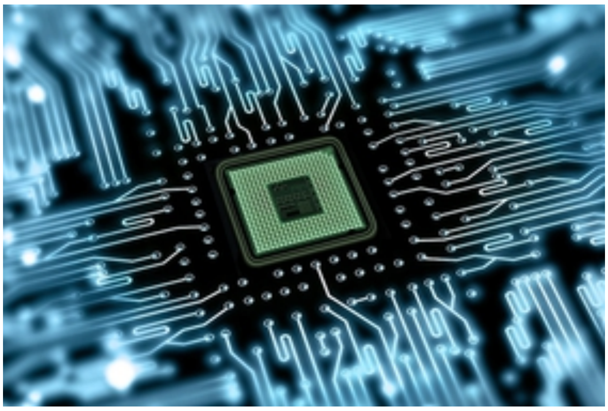
Week 01

Course Introduction

COMP1521 17s2 Computer Systems Fundamentals

2/67



LiC: John Shepherd, jas@cse.unsw.edu.au

Web: http://webcms3.cse.unsw.edu.au/COMP1521/17s2/

Course Staff

3/67

The Cast (of thousands):

- Lecture Stream A: John Shepherd
- Lecture Stream B: Zainab Abaid
- Tutors:
  Adrian Goldwaser, Angus Yuen, Austin Tankiang, Gregory Omelaenko, Jacob Godbout, Jacob Mikkelsen, John Luo, Johnson Shi, Matthew Di Meglio, Matthew French, Michael Manansala, Minjie Shen, Nicola Gibson, Ning Teh, Oliver Scott, Peter Kydd, Stanislav Shkel
- Lab assistants:
  Angela Yang, Aydin Itil, Bing Wen, Caspian Baska, Daniel Li, Darren Zhu, Jeremy Ng, Kevin Ni, Louis Cheung, Nicholas Mulianto, Oliver Shen, Patrick Song, Thomas Kilkelly, William Gilbert, Xia Li, Xiaowen Ma, Shanush Prema Thasarathan

---

Me

4/67

Research:

- information extraction, database systems, online learning

Teaching:

- UG: COMP{1921,1521,2041,3311,4011}
- PG: COMP{9311,9315}, GSOE9010

Admin:

- Deputy Head of School (Education)

Life:

- Craft Beer, AFL, Craft Beer, K-Drama, Craft Beer, Nordic Noir, ...

You

5/67

Students in this course have completed:

- COMP1511 or COMP1917 or COMP1911 (maybe with bridging)

Everyone has learned *fundamental C programming*

COMP1511/1917 have also learned *linked lists*

COMP1511 have also studied *sorting*

For this week ...

- review/strengthen C knowledge
- ensure that everyone knows linked structures in C
- revise the core data structures used in systems (stacks, queues)

Course Goals

6/67

COMP1511/1911/1917 ...

- get you thinking like a *programmer*
- solving problems by developing programs
- expressing your solution in the C language

COMP1521 ...

- investigates the structure of computer systems
- describes how they work at a low-level
- allows you to understand run-time behaviour
- better able to reason about your C programs

Note: these are *not* the same goals as COMP2121

COMP1511/1911/1917 vs COMP1521

7/67

COMP1511/1911/1917 ...

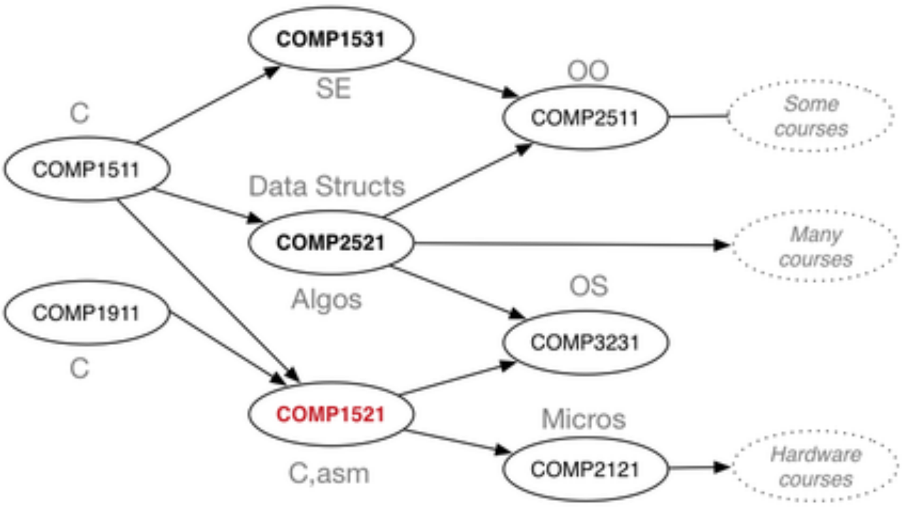
... COMP1511/1911/1917 vs COMP1521

8/67

COMP1521 ...

Course Context

9/67



Themes

10/67

Major themes ...

1. components of modern computer systems
2. how C programs execute (at the machine level)
3. how to write (MIPS) assembly language
4. Unix/Linux system-level programming
5. how operating systems and networks are structured
6. introduction to concurrency, concurrent programming

Goal: you are able to understand execution of software in detail

---

Detailed Topics

11/67

- Processors
  - data representation, instruction set
  - *assembler programming*
- Program execution (mapping C to assembler)
  - memory layout: stack, heap, code
  - control structures, function calls
- Operating system architecture
  - memory, cache, devices, i/o, interrupts
  - virtual memory, processes, file systems, *system calls*
- Concurrency
  - parallelism, synchronisation, coordination
- Network architecture

COMP1521 on the Web

12/67

Primary entry point is WebCMS

http://webcms3.cse.unsw.edu.au/COMP1521/17s2/

Most of the *content* lives under

/home/cs1521/web/17s2/... (e.g.lecs,labs,tutes,...)

Most content is web-accessible via

http://cgi.cse.unsw.edu.au/~cs1521/17s2/index.php

... COMP1521 on the Web

13/67

Most material on WebCMS is publically readable.

Login to WebCMS is via zID/zPass, and is needed for

- quizzes, polls, comments/forums, forming groups, ...

Submit work via `give` (either in WebCMS or on cmd line)

Check marks via `sturec` (either in WebCMS or on cmd line)

Textbook

14/67

There is no textbook.

Material has been drawn from:

- "Introduction to Computing Systems:
  from bits and gates to C and beyond",
  Patt and Patel
- "From NAND to Tetris:
  Building a modern computer system from first principles",
  Nisan and Schocken
- "Computer Systems: A Programmer's Perspective",
  Bryant and O'Halloren
- COMP2121 Course Web Site, Parameswaran and Guo

Note: always give credit to your sources

Systems

15/67

Most work done on *Linux*

- on the *CSE lab machines*
- can use *VLab* to connect to CSE from home
- the *command-line* is a powerful tool on Linux

Compilers: `dcc` (on CSE machines), or `gcc`

Assembly language: MIPS on `QtSpim`

- GUI interpreter ... runs on Linux, MacOS, Windows

Use your own favourite text editor (I use `vim`)

Classes

16/67

Lectures ...

- 3 hours/week for Weeks 1-12,13?
- all lectures will be video'd (Echo360 *and* YouTube)

Tutorials ...

- Weeks 1-12, 1 hour tute, followed by 2-hour lab
- explore lecture material via exercises

Labs ...

- small(ish) implementation tasks, done in pairs
- give skills practice (leading on to assignments/exam)

Note: Monday Week 10 is Public Holiday; will replace lecture by video

---

Assessments

17/67

Lab exercises contribute 10% to overall mark.

Ideally, the lab exercise for Week X must be

- submitted before Sunday at end of week X
- demonstrated to tutor *during* Week X lab
  OR, demonstrated *at the start of* Week X+1 lab
- aim of demo is to get feedback on design and style

Late submissions will get feedback and reduced mark.

Total mark for labs > 10 (scaled to 11, capped to 10).

Bonus up to 1 mark for completing/submitting all labs.

... Assessments

18/67

Two *assignments* ...

- Ass1: Assembly Language, weeks 3-6, 7 marks
- Ass2: Memory Allocator, weeks 7-10, 13 marks
- both assignments are completed individually
- can be completed on your own machine (if you have C compiler)
  - but you *must* test on the CSE machines before you submit

Late penalty: 0.08 off max mark for each hour late

Good time management avoids late penalties

Quizzes

19/67

Five small online quizzes ...

- 3-4 questions, multiple-choice format
- primarily for review of recent topics
- taken in your own time (via WebCMS)

Contribute 10% towards final mark.

Starting this week ... C revision quiz

Then in weeks 3, 5, 7, 9, 11

Each quiz due before Sunday 11:59pm at end of week

Blogs

20/67

Keep a blog about what you're learning

- write it at least weekly
- reflect on and plan your learning

Bonus up to 1 mark if

- you maintain regularly (weekly)
- what you write is interesting

Misconduct

21/67

E.g. plagiarism, contracting, trolling, harassment, ...

Just Don't Do it

Final Exam

22/67

3-hour on-line ~~torture~~ exam during the exam period.

Held in CSE labs (must know lab environment)

On-line documentation available in exam:

- MIPS / QtSpim / C quick reference guides
- Unix Programmers Manual (`man`) (very handy)

Format:

- some programming exercises (Prac)
- some descriptive/analytical questions (Theory)

How to pass? Practice, practice, practice, ...

Course Assessment

23/67

```
CourseWorkMark = QuizMark + LabMark + Ass1Mark + Ass2Mark
                                                   (out of 40)
ExamPracMark    = marks for prac questions on final exam
                                                   (out of 30)
ExamTheoryMark = marks for written questions on final exam
                                                   (out of 30)
ExamMark        = ExamPracMark + ExamTheoryMark    (out of 60)

ExamOK          = ExamMark ≥ 22/60                (true/false)

FinalMark       = CourseWorkMark + ExamMark       (out of 100)

FinalGrade      = UF, if !ExamOK  (regardless of mark)
                = FL, if FinalMark < 50/100
                = PS, if 50/100 ≤ FinalMark < 65/100
                = CR, if 65/100 ≤ FinalMark < 75/100
                = DN, if 75/100 ≤ FinalMark < 85/100
```

```
= HD, if FinalMark ≥ 85/100
```

Supplementary Exams

24/67

Supplementary Exams are available to students who

- do *not* attend the final exam
- have a documented reason for not attending

Sympathy Supp Exams are available to students who

- have `(ExamOK && 47 ≤ FinalMark < 50)`

Passing a Sympathy Supp gives you max 50% overall

Summary

25/67

The goal is for you to become a better programmer

- more confident in your own ability
- with an expanded set of tools to draw on
- and a deeper understanding of "run-time"
- producing better engineered software
- ultimately, enjoying the programming process

Computer Systems

Some History

27/67

A potted history of "computer systems" ...

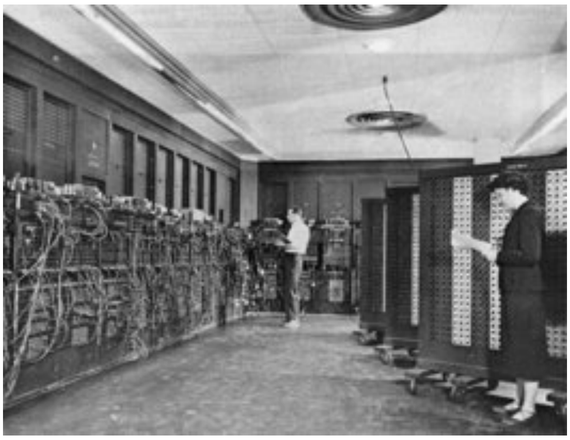
- 1613 ... first use of the word "computer"
  (meaning "a person who performs computations")
- 1800's ... Babbage's mechanical computers
  (Analytical Engine and Difference Engine, with Ada Lovelace)
- 1936 ... Zuse's Z1 electro-mechanical computer
  (first binary programmable computer)
- 1943 ... Eckert/Mauchley's ENIAC
  (first fully functional electric digital computer ... 18000 valves)
- 1949 ... EDSAC and Manchester Mark 1
  (first generation stored program computers ... valves)
- 1955 ... Whirlwhind at MIT
  (first digital computer with magnetic core RAM)
- 1960 ... Digital Equipment Corporation PDP-1
  (first mini-computer ... recognisable as "modern" computer)
- 1971 ... Intel 4004 (first microprocessor)

From: When was the first computer invented?

... Some History

28/67

ENIAC ...



... Some History

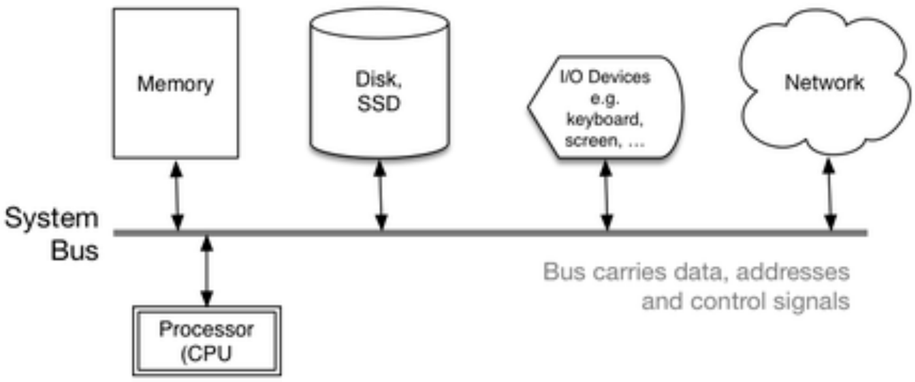
29/67

Whirlwind ...

Computer Systems

30/67

Component view of typical modern computer system



Processor

31/67

Modern processors provide

- control, arithmetic, logic, bit operators
- relatively small set of simple instructions
- small amount of very fast storage (registers)
- small number of control registers (e.g. PC)
- fast fetch-decode-execute cycle (ns)
- access to system bus to communicate with other components
- all integrated on a single chip

We do not consider mutli-core CPUs in this course

Storage

32/67

Memory (main memory) consists of

- very large random-addressable array of bytes
- can fetch single bytes into CPU registers
- can fetch multi-byte chunks into CPU (e.g. 4-byte int)
- typically: access time 70 ns, size 64 GB

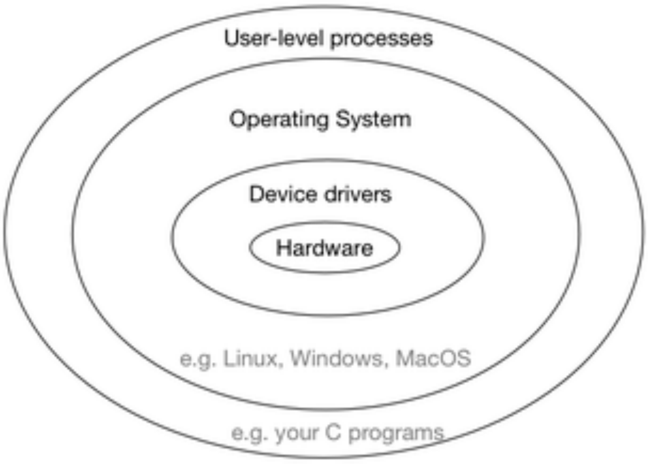
Disk storage consist of

- very very large block-oriented storage
- often on spinning disk, fetching 512B-4KB per request
- typically: access time 30 ms, size 8 TB
- nowadays, SSD: access time 0.8ms, size 1GB

Computer System Layers

33/67

View of software layers in typical computer system



C Program Life-cycle

34/67

Your C programs start as text

- well-designed, readable, maintainable, ...

Ultimately they execute on a CPU as machine code

- efficiently producing correct results

- handling error conditions robustly
- utlising services of underlying operating system

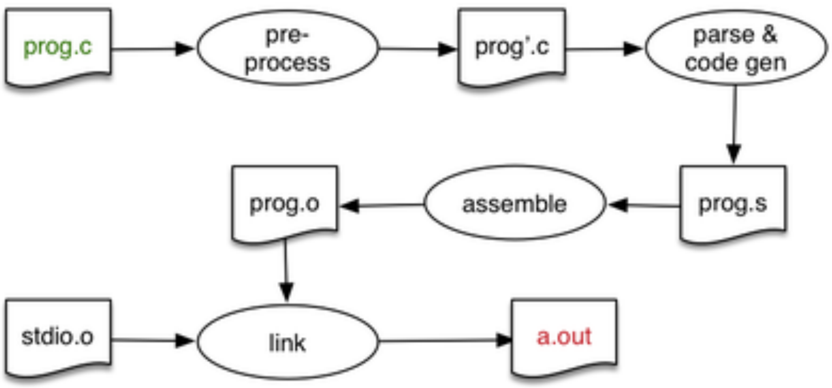
How to map from C source code to machine code?

How to make one C program run on different machine architectures?

... C Program Life-cycle

35/67

From source code to machine code ...



History of C Compilers

36/67

Milestones in C history

- `cc` ... original compiler by Dennis Ricthie (1971)
- `gcc` ... open source compiler by Richard Stallman et al (1987)
- `gcc` 2.0 ... added C++ compilation (1992)
- `clang` ... gcc replacement by Apple et al (2007)
- `dcc` ... Python wrapper on `clang` by Andrew Taylor (2012?)
  - augments error messages to be more helpful to novices
  - incorporates useful run-time checking to help debugging

C Revisited

What (I assume) You Know

38/67

Given a problem specification ...

- design an algorithmic solution
- describe your solution in C code, using ...
  - variables, assignment, tests (==, `!`, <=, `&&`, etc)
  - `if`, `while`, `for`, `break`, `scanf()`, `printf()`
  - functions, `return`, prototypes, `*.h`, `*.c`
  - arrays, files, structs, pointers, `malloc()`, `free()`

I don't assume that you know ...

- recursion, linked data structures, ADTs, bit operations

Abstract Data Types

Abstract Data Types

40/67

A *data type* is ...

- a set of *values* (atomic or structured values)
- a set of *operations* on those values

An *abstract data type* is ...

- an approach to implementing data types
- separates *interface* from *implementation*
- *users/clients* of the ADT see only the interface
- *implementors* of the ADT provide an implementation

E.g. do you know what a (`FILE *`) looks like?

Exercise 1: What's in a `FILE *`?

41/67

You have used `fopen()`, `fgets()`, `fclose()`, `getchar()`

But what kind of data structures lie behind them?

How could we find out?

Collections

42/67

Many of the ADTs we deal with ...

- consist of a *collection* of *items*
- where each item may be a simple type or an ADT
- and items often have a *key* (to identify them)

Collections may be categorised by ...

- *structure*:
  linear (list), branching (tree), cyclic (graph)
- *usage*:
  set, matrix, stack, queue, search-tree, dictionary, ...

... Collections

43/67

Typical operations on collections

- *make* an empty collection
- *insert* one item into the collection
- *remove* one item from the collection
- *find* an item in the collection
- *check* properties of the collection (size,empty?)
- *scan* the collection, item by item
- *show* the collection
- *free* the entire collection

Polymorphism

44/67

Consider a `List` data type.

We have many kinds of Lists ...

- Lists of integers, strings, Students, Processes, ...

Can't call them all `List`, so

- `IntList`, `StringList`, `StudentList`, etc.

Some programming languages allow you to

- define a single *polymorphic* `List` type
- specialise it for different item types e.g. `List<int>`

... Polymorphism

45/67

Similarly with operations on ADTs ...

Easiest to call the function `make()` for all ADTs

But in C you end up with duplicate symbols, so

- `makeIntSet()`, `makeIntList()`, `makeStringList`, etc.

Some programming languages allow you to *overload*

- same function name for different ADTs
- compiler resolves which function is required

Stacks and Queues

46/67

Stack: Last-in, First-out (LIFO) protocol

- insert operation called `push()`
- remove operation called `pop()`
- has a top (last item added) and a size
- used in implementation of functions in compiled C

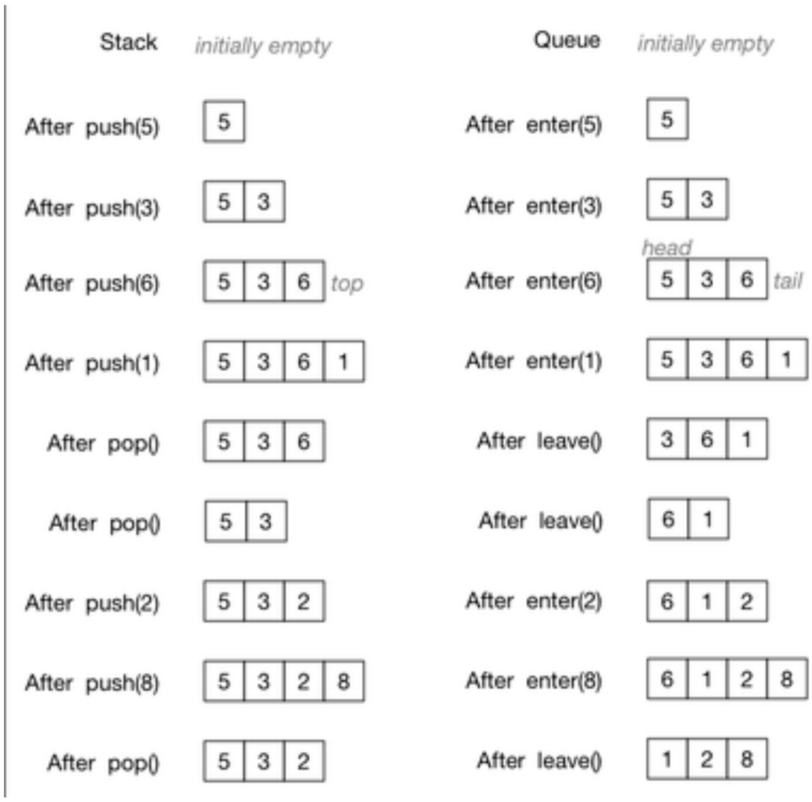
Queue: First-in, First-out (FIFO) protocol

- insert operation called `enter` (or `enqueue()`)
- remove operation called `leave` (or `dequeue()`)
- has a head (first item added), a tail (last item added), and a size
- used in scheduling, managing resource usage

... Stacks and Queues

47/67

Usage of a Stack and Queue of integers ...



Stack Operations

48/67

- **`pushStack(Stack s, Item it)`** ... add item onto stack
- **`Item it = popStack(Stack s)`** ... remove item from stack

Other possible operations:

- **`isEmptyStack(Stack s)`** ... stack contains no items?
- **`itemsInStack(Stack s)`** ... how many items in stack
- **`showStack(Stack s)`** ... display stack on stdout
- **`Stack s = makeStack()`** ... create new empty stack
- **`freeStack(Stack s)`** ... release stack data

Implementing ADTs in C

49/67

To implement e.g. `Stack`, have two files

- `Stack.h` ... signatures of ADT operations, typedef
- `Stack.c` ... implementation of ADT operations

Client programs `#include "Stack.h"`

- then define and create objects of type `Stack`
- and apply `Stack` operations on those objects

`Stack.c` file contains ...

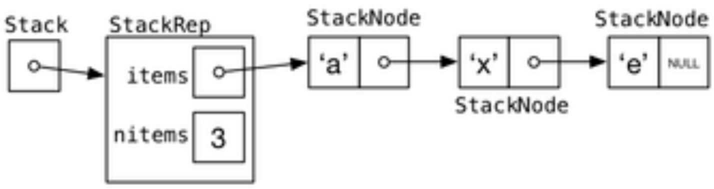
- concrete data structure representing `Stack`s
- implementations of all operations on `Stack`s
- possibly additional private operations and types

Exercise 2: Stack ADT

50/67

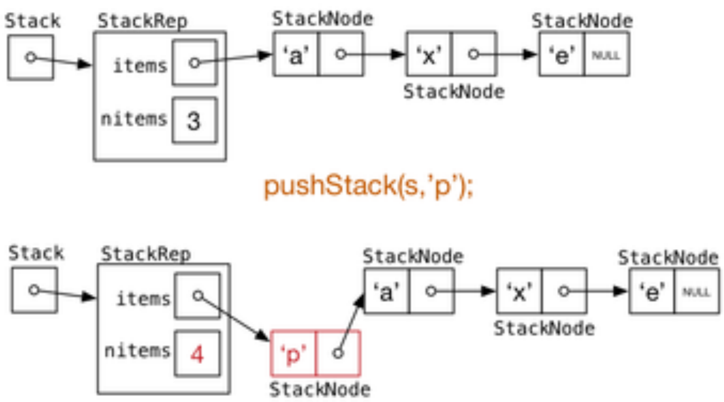
Implement `Stack` ADT for stacks of `char` values

Use a linked-list as the concrete data representation, e.g.

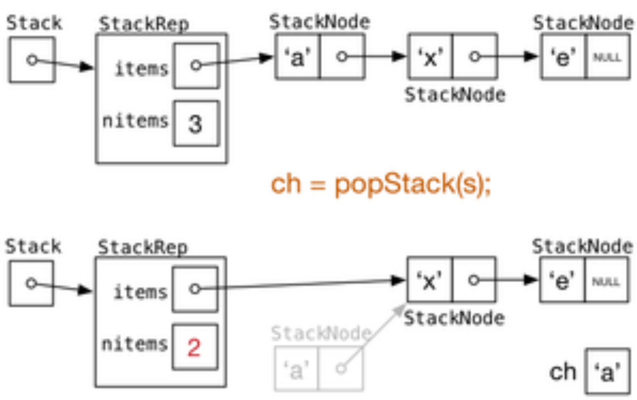


Write a simple driver program to test your ADT

How data structure changes on `push`



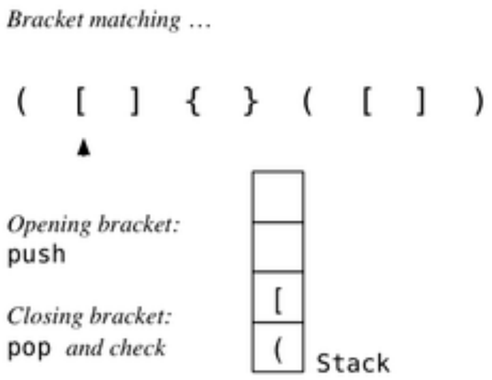
How data structure changes on `pop`



Exercise 3: Bracket Matching

53/67

Write a bracket matching program using the `Stack` ADT



See: CSeLearning video on YouTube which solves precisely this problem.

Bit Manipulation

Bits in Bytes in Words

55/67

Values that we normally treat as atomic, can be viewed as bits, e.g.

- `char` = 1 byte = 8 bits (`'a'` is `01100001`)
- `short` = 2 bytes = 16 bits (`42` is `0000000000101010`)
- `int` = 4 bytes = 32 bits (`42` is `0000000000...0000101010`)
- `double` = 8 bytes = 64 bits

The above are common sizes and don't apply on all hardware
(e.g. `sizeof(int)` might be 16 or 64 bits, `sizeof(double)` might be 32 bits)

C provides a set of operators that act bit-by-bit on pairs of bytes.

E.g. `(10101010 & 11110000)` yields `10100000` (bitwise AND)

C bitwise operators: `&` `|` `^` `~` `<<` `>>`

Binary Constants

56/67

C does not have a way of directly writing binary numbers

Can write numbers in decimal, hexadecimal and octal.

In hexadecimal, each digit represents 4 bits

```
   0100 1000 1111 1010 1011 1100 1001 0111
0x   4    8    F    A    B    C    9    7
```

In octal, each digit represents 3 bits

```
   01 001 000 111 110 101 011 110 010 010 111
0   1  1   0   7   6   5   3   6   2   2   7
```

Bitwise AND

57/67

The `&` operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical AND on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
  00100111            AND | 0   1
& 11100011            ----|-----
  --------              0 | 0   0
  00100011              1 | 0   1
```

Used for e.g. checking whether a bit is set

Exercise 4: Checking for odd numbers

58/67

One obvious way to check for odd numbers in C

```
int isOdd(int n) { return (n%2) == 1; }
```

Could we use `&` to achieve the same thing? How?

Aside: an alternative to the above

```
#define isOdd(n)  ((n)%2) == 1)
```

What's the difference between the *function* and the *macro*?

Bitwise OR

59/67

The | operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical OR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
  00100111             OR | 0   1
& 11100011            ----|-----
  --------              0 | 0   1
  11100111              1 | 1   1
```

Used for e.g. ensuring that a bit is set

Flag-bits

60/67

Consider file permissions in the Unix file system

Each file has three sets of "flags" defining it permissions

```
$ ls -l odd.c
-rwxrw-r--  1 jas  group  486 26 Jul 23:22 odd.c
```

- `rwx` gives permissions for the owner of the file

- rw- gives permissions for group members
- r-- gives permissions for everyone else

How to represent these? Efficiently?

... Flag-bits

61/67

One possible representation:

```
typedef char Permissions[10];       // e.g. "rwxrw-r--"
```

Another possible representation:

```
typedef int Permissions[9];   //e.g. {1,1,1,1,1,0,1,0,0}
```

Compact representation:

```
typedef unsigned short Permissions;  //e.g. 0764 or 0x1F4
```

Last representation uses 1-bit per permission "flag"

Exercise 5: File Permissions

62/67

Implement file permissions as a set of bits

- a program to read rwx triples and set/show permisions
- a program to read an octal value and set/show permisions

Use the data type

```
typedef unsigned short Permissions;
```

This "wastes" 7 bits ... what else could we do with them?

Bitwise XOR

63/67

The ^ operator

- takes two values (1,2,4,8 bytes), treats as sequence of bits
- performs logical XOR on each corresponding pair of bits
- result contains same number of bits as inputs

Example:

```
  00100111           XOR | 0  1
& 11100011           ----|-----
  --------             0 | 0  1
  11100111             1 | 1  0
```

Used for e.g. in generating random numbers

Bitwise NEG

64/67

The ~ operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- performs logical negation of each bit
- result contains same number of bits as input

Example:

```
~ 00100111          NEG | 0   1
  --------          ----|-----
  11011000              | 1   0
```

Used for e.g. creating useful bit patterns

Left Shift

65/67

The << operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- and a small positive integer *x*
- moves (shifts) each bit *x* positions to the left
- left-end bit vanishes; right-end bit replaced by zero
- result contains same number of bits as input

Example:

```
  00100111 << 2       00100111 << 8
  --------            --------
  10011100            00000000
```

Exercise 6: File Permissions

66/67

Implement file permissions as before

- create masks by shifting rather than hex constants

You can use OR to set a particular bit

How do you ensure that a given bit is cleared?

Right Shift

67/67

The >> operator

- takes a single value (1,2,4,8 bytes), treats as sequence of bits
- and a small positive integer *x*
- moves (shifts) each bit *x* positions to the right
- right-end bit vanishes; left-end bit replaced by zero**
- result contains same number of bits as input

Example:

```
  00100111 >> 2       00100111 >> 8
```

```
--------            --------
00001001            00000000
```

If `signed` quantity, sign bit replaces left-end bit

Produced: 1 Aug 2017