Introduction to Programming Systems

CS 217

Course Notes, Fall 2003

Andrew W. Appel

Princeton University

Goals

- Master the art of programming
  - o learn how to be good programmers
  - o introduction to software engineering
- Learn C and the Unix development tools
  - o C is the systems language of choice
  - o Unix has a rich development environment
- Introduction to computer systems
  - o machine architecture
  - o operating systems
  - o compilers

Outline

- September
  - o C programming
- October
  - o Team programming; computer game algorithm
  - o Unix operating system
- November
  - o Machine architecture
  - o Assembly language
- December
  - o Digital circuits
  - o Assemblers, linkers, simulators

Coursework

- Seven programming assignments (approx. 60%)
  - o string functions
  - o symbol table
  - o game player
  - o game referee
  - o assembly language programming
  - o circuit simulator (interpreted)
  - o circuit simulator (compiled)
- Exams (approx. 30%)
  - o midterm
  - o final
- Class participation (approx. 10%)
  - o Precept attendance October 6--19 is ***mandatory***

Policies

www.cs.princeton.edu/courses/archive/fall03/cs217/policies.html

Programming in an individual creative process much like composition. You must reach your own understanding of the problem and discover a path to its solution. During this time, discussions with friends are encouraged. However, when the time comes to write code that solves the problem, such discussions are no longer appropriate - the program must be your own work. If you have a question about how to use some feature of C, UNIX, etc., you can certainly ask your friends or the teaching assistants, but **do not, under any circumstances, copy another person's program.** Writing code for use by another or using someone else's code in any form is a **violation of academic regulations**. "Using someone else's code" includes using solutions or partial solutions to assignments provided by commercial web sites, instructors, preceptors, teaching assistants, friends, or students from any previous offering of this course or any other course.

Materials

- Required textbooks
  - o C Programming: A Modern Approach, King
  - o SPARC Architecture, etc. Paul
  - o Programming with GNU Software. Loukides & Oram
- These lecture notes
- Recommended textbooks
  - o The Practice of Programming, Kernighan & Pike
- Other textbooks
  - o The C Programming Language, Kernighan & Ritchie
  - o C: A Reference Manual. Harbison & Steele
- Web pages
  - o `www.cs.princeton.edu/courses/cs217/`

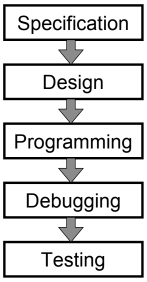
Facilities

- Unix machines
  - o CIT’s **`arizona`** cluster
  - o SPARC lab in Friend 016
- Your own laptop
  - o **`ssh`** access to **`arizona`**
  - o run GNU tools on Windows
  - o run GNU tools on Linux

Logistics

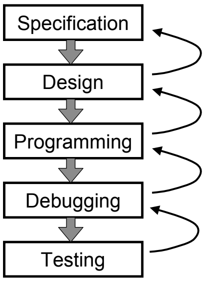
- Lectures
  - o T,Th 10:00 AM, CS105
  - o introduce concepts
  - o work through programming examples
- Precepts
  - – M,W 10:00
  - – M,W 1:30
  - – T,Th 1:30
  - o demonstrate tools (gdb, makefiles, emacs, …)
  - o work through programming examples
  - o collaborative work (two weeks in October)

Software in COS126



1 Person
10^2 Lines of Code
1 Type of Machine
0 Modifications
1 Week

Software in the Real World



Lots of People
10^6 Lines of Code
Lots of Machines
Lots of Modifications
1 Decade or more

Good Software in the Real World

- Understandable
  - o Well-designed
  - o Consistent
  - o Documented

← Write code in modules with well-defined interfaces

- Robust
  - o Works for any input
  - o Tested

← Write code in modules and test them separately

- Reusable
  - o Components

← Write code in modules that can be used elsewhere

- Efficient
  - o Only matters for 1%

← Write code in modules and optimize the slow ones

Modules

- Programs are made up of many <u>modules</u>
- Each module is small and does one thing
  - o string manipulation
  - o mathematical functions
  - o set, stack, queue, list, etc.
- Deciding how to break up a program into modules is a key to good software design

Interfaces

- An interface defines what the module does
  - o decouple clients from implementation
  - o hide implementation details
- An interface specifies…
  - o data types and variables
  - o functions that may be invoked

```
typedef struct stringlist *StringList;
StringList *StringList_new(void);
void StringList_free(StringList *list);
void StringList_insert(StringList *list, char *string);
void StringList_remove(StringList *list, char *string);
void StringList_print(StringList *list);

int StringList_getLength(StringList *list);
```

Implementations

- An implementation defines how the module does it
- Can have many implementations for one interface
  - o different algorithms for different situations
  - o machine dependencies, efficiency, etc.

```
StringList *StringListCreate(void)
{
  StringList *list = malloc(sizeof(StringList));
  list->entries = NULL;
  list->size = 0;
}

void StringListDelete(StringList *list)
{
  free(list);
}

etc.
```

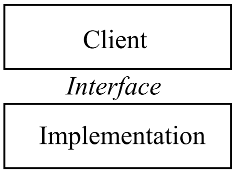
Clients

- A client uses a module via its interface
- Clients see only the interface
  - o can use module without knowing its implementation
- Client is unaffected if implementation changes
  - o as long as interface stays the same

```
int main()
{
  StringList *list = StringListCreate();
  StringListInsert(list, “CS217”);
  StringListInsert(list, “is”);
  StringListInsert(list, “fun”);
  StringListPrint(list);
  StringListDelete(list);
}
```

Clients, Interfaces, Implementations

- Interfaces are contracts between clients and implementations
  - o Clients must use interface correctly
  - o Implementations must do what they advertise



Client

Interface

Implementation

- Examples from real world?

Clients, Interfaces, Implementations

- Advantages of modules with clean interfaces
  - o decouples clients from implementations
  - o localizes impact of change to single module
  - o allows sharing of implementations (re-use)
  - o allows separate compilation
  - o improves readability
  - o simplifies testing
  - o etc.

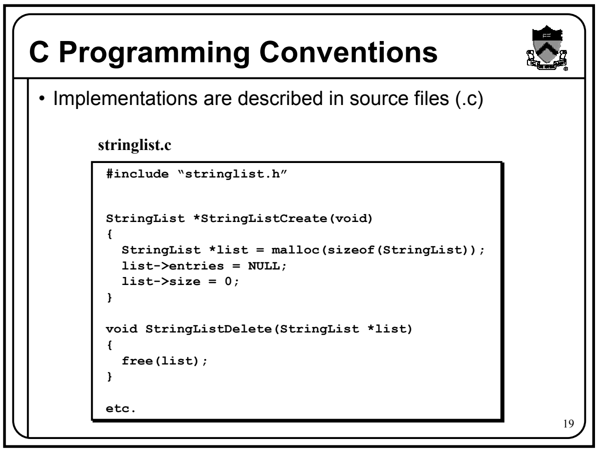
```
int main()
{
  StringList *list = StringListCreate();
  StringListInsert(list, “CS217”);
  StringListInsert(list, “is”);
  StringListInsert(list, “fun”);
  StringListPrint(list);
  StringListDelete(list);
}
```

C Programming Conventions

- Interfaces are defined in header files (.h)

stringlist.h

```
typedef struct stringlist *StringList;
StringList *StringListCreate(void);
void StringListDelete(StringList *list);
void StringListInsert(StringList *list, char *string);
void StringListRemove(StringList *list, char *string);
void StringListPrint(StringList *list);
int StringListGetLength(StringList *list);
```

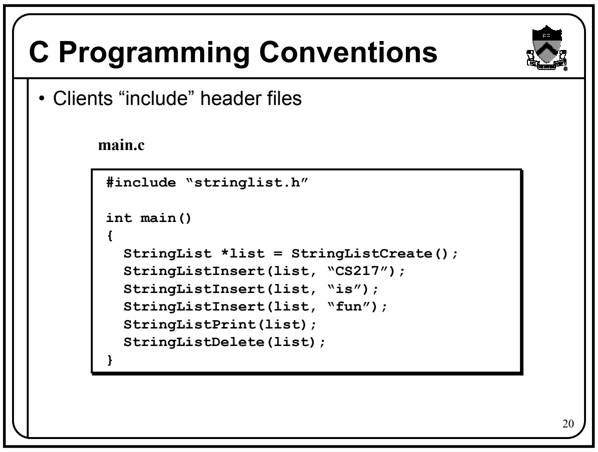


C Programming Conventions

- Implementations are described in source files (.c)

stringlist.c

```
#include "stringlist.h"


StringList *StringListCreate(void)
{
  StringList *list = malloc(sizeof(StringList));
  list->entries = NULL;
  list->size = 0;
}

void StringListDelete(StringList *list)
{
  free(list);
}

etc.
```

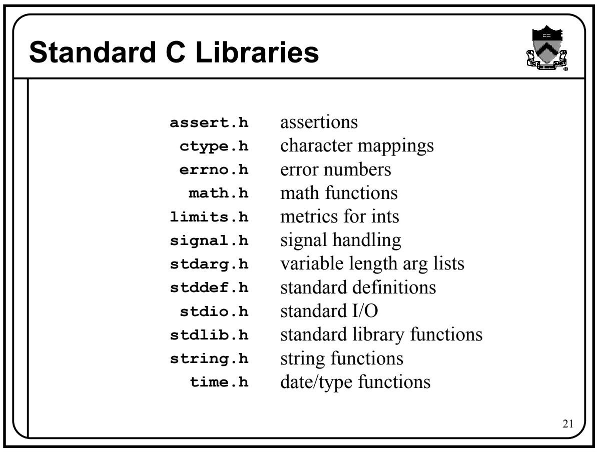


C Programming Conventions

- Clients "include" header files

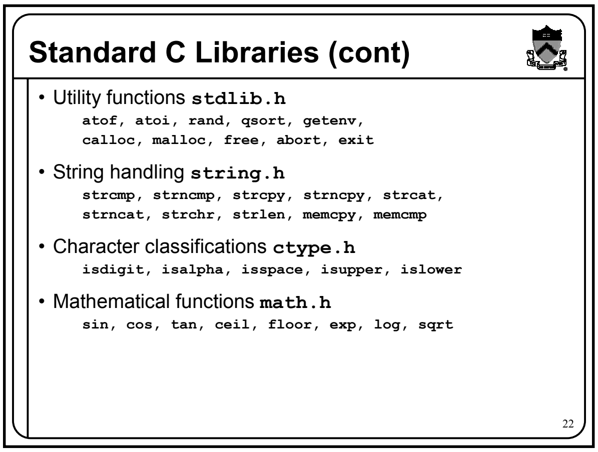
main.c

```
#include "stringlist.h"

int main()
{
  StringList *list = StringListCreate();
  StringListInsert(list, "CS217");
  StringListInsert(list, "is");
  StringListInsert(list, "fun");
  StringListPrint(list);
  StringListDelete(list);
}
```



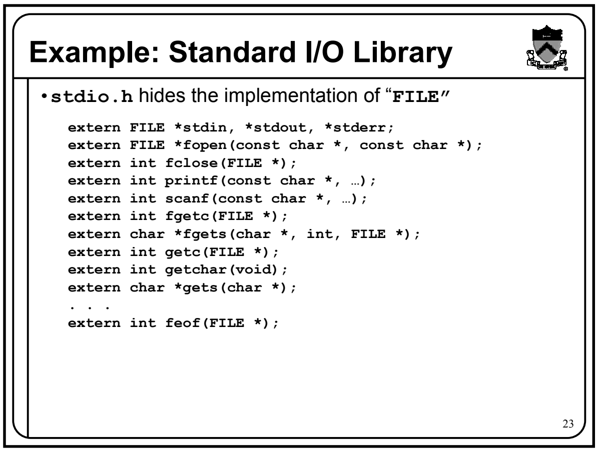
Standard C Libraries

`assert.h`	assertions
`ctype.h`	character mappings
`errno.h`	error numbers
`math.h`	math functions
`limits.h`	metrics for ints
`signal.h`	signal handling
`stdarg.h`	variable length arg lists
`stddef.h`	standard definitions
`stdio.h`	standard I/O
`stdlib.h`	standard library functions
`string.h`	string functions
`time.h`	date/type functions



Standard C Libraries (cont)

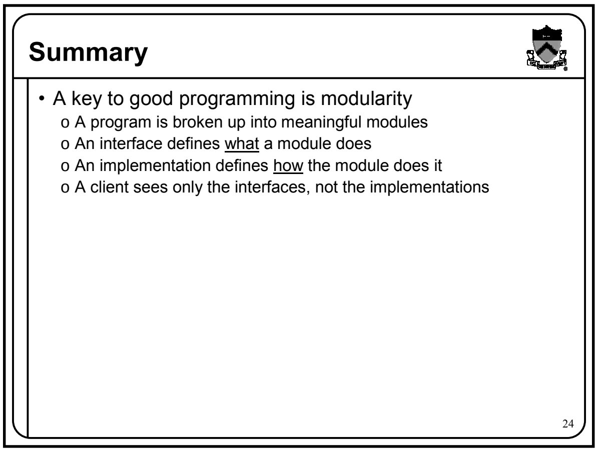
- Utility functions **`stdlib.h`**
  `atof, atoi, rand, qsort, getenv,`
  `calloc, malloc, free, abort, exit`
- String handling **`string.h`**
  `strcmp, strncmp, strcpy, strncpy, strcat,`
  `strncat, strchr, strlen, memcpy, memcmp`
- Character classifications **`ctype.h`**
  `isdigit, isalpha, isspace, isupper, islower`
- Mathematical functions **`math.h`**
  `sin, cos, tan, ceil, floor, exp, log, sqrt`



Example: Standard I/O Library

- `stdio.h` hides the implementation of "`FILE`"

```
extern FILE *stdin, *stdout, *stderr;
extern FILE *fopen(const char *, const char *);
extern int fclose(FILE *);
extern int printf(const char *, …);
extern int scanf(const char *, …);
extern int fgetc(FILE *);
extern char *fgets(char *, int, FILE *);
extern int getc(FILE *);
extern int getchar(void);
extern char *gets(char *);
. . .
extern int feof(FILE *);
```



Summary

- A key to good programming is modularity
  - o A program is broken up into meaningful modules
  - o An interface defines <u>what</u> a module does
  - o An implementation defines <u>how</u> the module does it
  - o A client sees only the interfaces, not the implementations

Modules

CS 217

The C Programming Language

- Systems programming language
  - o originally used to write Unix and Unix tools
  - o data types and control structures close to most machines
  - o now also a popular application programming language
- Notable features
  - o all functions are call-by-value
  - o pointer (address) arithmetic
  - o simple scope structure
  - o I/O and memory mgmt facilities provided by libraries
- History
  - o BCPL → B → C → K&R C → ANSI C
    1960 1970 1972 1978 ↘ 1988 1995 2001
  - o LISP → Smalltalk → C++ → Java → C#

Example Program 1

```
#include <stdio.h>
#include <string.h>

int main()
{
  char *strings[128];
  char string[256];
  char *p1, *p2;
  int nstrings;
  int found;
  int i, j;

  nstrings = 0;
  while (fgets(string, 256, stdin)) {
    for (i = 0; i < nstrings; i++) {
      found = 1;
      for (p1 = string, p2 = strings[i]; *p1 && *p2; p1++, p2++) {
        if (*p1 > *p2) {
          found = 0;
          break;
        }
      }
      if (found) break;
    }
    for (j = nstrings; j > i; j--)
      strings[j] = strings[j-1];
    strings[i] = strdup(string);
    nstrings++;
    if (nstrings >= 128) break;
  }
  for (i = 0; i < nstrings; i++)
    fprintf(stdout, "%s", strings[i]);

  return 0;
}
```

What does this program do?

Example Program 2

```
#include <stdio.h>
#include <string.h>

#define MAX_STRINGS 128
#define MAX_LENGTH 256

void ReadStrings(char **strings, int *nstrings,
                 int maxstrings, FILE *fp)
{ char string[MAX_LENGTH];

  *nstrings = 0;
  while (fgets(string, MAX_LENGTH, fp)) {
    strings[(*nstrings)++] = strdup(string);
    if (*nstrings >= maxstrings) break;
  }
}

void WriteStrings(char **strings, int nstrings,
                  FILE *fp)
{ int i;

  for (i = 0; i < nstrings; i++)
    fprintf(fp, "%s", strings[i]);
}

int CompareStrings(char *string1, char *string2)
{ char *p1 = string1, *p2 = string2;

  while (*p1 && *p2) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;
    p1++;
    p2++;
  }

  return 0;
}
```

```
void SortStrings(char **strings, int nstrings)
{
  int i, j;

  for (i = 0; i < nstrings; i++)
    for (j = i+1; j < nstrings; j++)
      if (CompareStrings(strings[i], strings[j]) > 0) {
        char *swap = strings[i];
        strings[i] = strings[j];
        strings[j] = swap;
      }
}


int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings,
              MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

What does this program do?

Modularity

- Decompose execution into modules
  - o Read strings
  - o Sort strings
  - o Write strings
- Interfaces hide details
  - o Localize effect of changes
- Why is this better?
  - o **Easier to understand**
  - o Easier to test and debug
  - o Easier to reuse code
  - o Easier to make changes

```
int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

Modularity

- Decompose execution into modules
  - o Read strings
  - o Sort strings
  - o Write strings
- Interfaces hide details
  - o Localize effect of changes
- Why is this better?
  - o Easier to understand
  - o **Easier to test and debug**
  - o Easier to reuse code
  - o Easier to make changes

```
int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdout);
  WriteStrings(strings, nstrings, stdout);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

Modularity

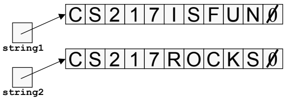
- Decompose execution into modules
  - o Read strings
  - o Sort strings
  - o Write strings
- Interfaces hide details
  - o Localize effect of changes
- Why is this better?
  - o Easier to understand
  - o Easier to test and debug
  - o **Easier to reuse code**
  - o Easier to make changes

```
MergeFiles(FILE *fp1, FILE *fp2)
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, fp1);
  WriteStrings(strings, nstrings, stdout);

  ReadStrings(strings, &nstrings, MAX_STRINGS, fp2);
  WriteStrings(strings, nstrings, stdout);
}
```

Modularity

- Decompose execution into modules
  - o Read strings
  - o Sort strings
  - o Write strings
- Interfaces hide details
  - o Localize effect of changes
- Why is this better?
  - o Easier to understand
  - o Easier to test and debug
  - o Easier to reuse code
  - o **Easier to make changes**

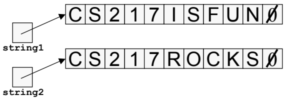
```
int CompareStrings(char *string1, char *string2)
{
  char *p1 = string1;
  char *p2 = string2;

  while (*p1 && *p2) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;
    p1++;
    p2++;
  }

  return 0;
}
```



Modularity

- Decompose execution into modules
  - o Read strings
  - o Sort strings
  - o Write strings
- Interfaces hide details
  - o Localize effect of changes
- Why is this better?
  - o Easier to understand
  - o Easier to test and debug
  - o Easier to reuse code
  - o **Easier to make changes**

```
int StringLength(char *string)
{
  char *p = string;
  while (*p) p++;
  return p - string;
}


int CompareStrings(char *string1, char *string2)
{
  return StringLength(string1) -
         StringLength(string2);
}
```



Separate Compilation

- Move string array into separate file
  - o Declare interface in **`stringarray.h`**
  - o Provide implementation in **`stringarray.c`**
  - o Allows re-use by other programs

`stringarray.h`

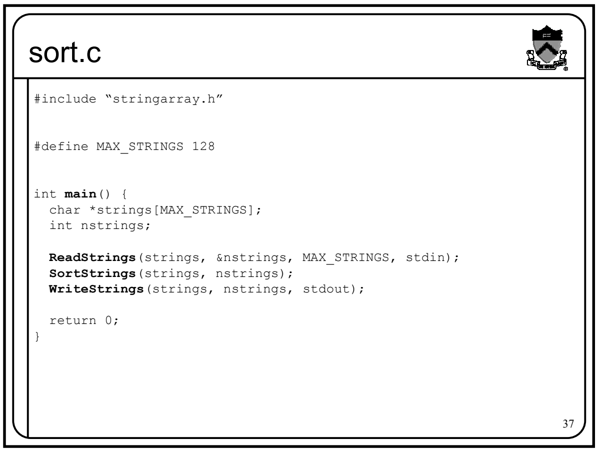
```
extern void ReadStrings(char **strings, int *nstrings,
                        int maxstrings, FILE *fp);

extern void WriteStrings(char **strings, int nstrings,
                         FILE *fp);

extern void SortStrings(char **strings, int nstrings);

extern int CompareStrings(char *string1, char *string2);
```

stringarray.c

```
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 256

void ReadStrings(FILE *fp, char **strings, int *nstrings, int maxstrings)  {

  char string[MAX_LENGTH];

  *nstrings = 0;
  while (fgets(string, MAX_LENGTH, fp)) {
    strings[(*nstrings)++] = strdup(string);
    if (*nstrings >= maxstrings) break;
  }
}


void WriteStrings(FILE *fp, char **strings, int nstrings)  {
  int i;

  for (i = 0; i < nstrings; i++)
    fprintf(fp, "%s", strings[i]);
}
```

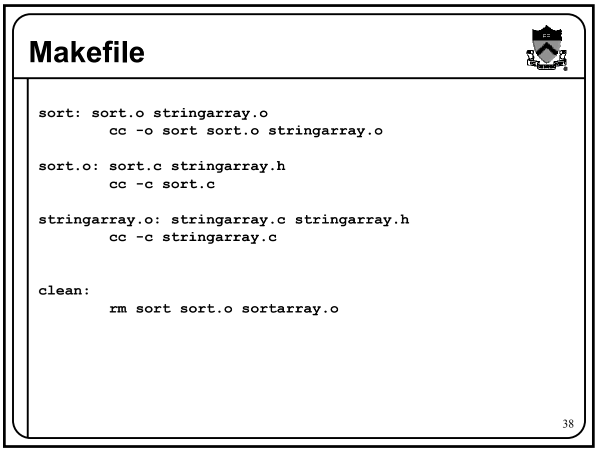
stringarray.c (cont'd)

```
int CompareStrings(char *string1, char *string2)  {
  char *p1, *p2;

  for (p1 = string1, p2 = string2; *p1 && *p2; p1++, p2++)
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2) return 1;

  return 0;
}

void SortStrings(char **strings, int nstrings)  {
  int i, j;

  for (i = 0; i < nstrings; i++)
    for (j = i+1; j < nstrings; j++)
      if (CompareStrings(strings[i], strings[j]) > 0) {
        char *swap = strings[i];
        strings[i] = strings[j];
        strings[j] = swap;
      }
}
```



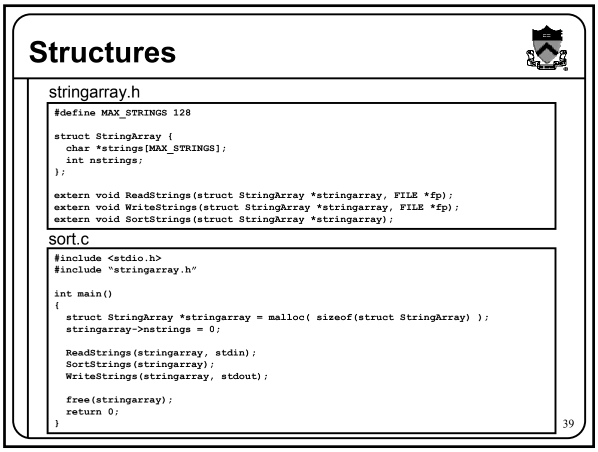
sort.c

```
#include “stringarray.h”


#define MAX_STRINGS 128


int main() {
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```



Makefile

```
sort: sort.o stringarray.o
        cc -o sort sort.o stringarray.o

sort.o: sort.c stringarray.h
        cc -c sort.c

stringarray.o: stringarray.c stringarray.h
        cc -c stringarray.c


clean:
        rm sort sort.o sortarray.o
```



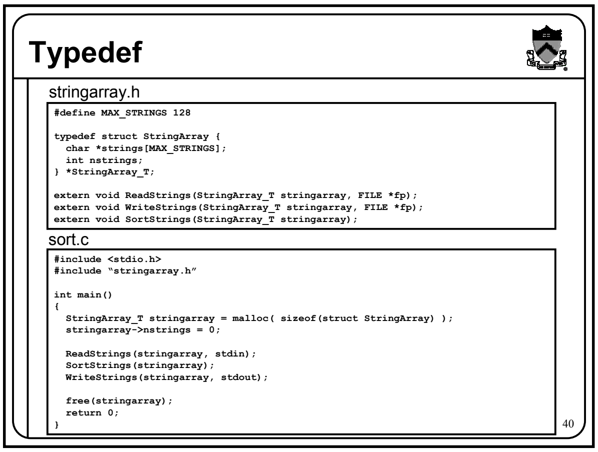
Structures

stringarray.h

```
#define MAX_STRINGS 128

struct StringArray {
  char *strings[MAX_STRINGS];
  int nstrings;
};

extern void ReadStrings(struct StringArray *stringarray, FILE *fp);
extern void WriteStrings(struct StringArray *stringarray, FILE *fp);
extern void SortStrings(struct StringArray *stringarray);
```

sort.c

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  struct StringArray *stringarray = malloc( sizeof(struct StringArray) );
  stringarray->nstrings = 0;

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray);
  WriteStrings(stringarray, stdout);

  free(stringarray);
  return 0;
}
```



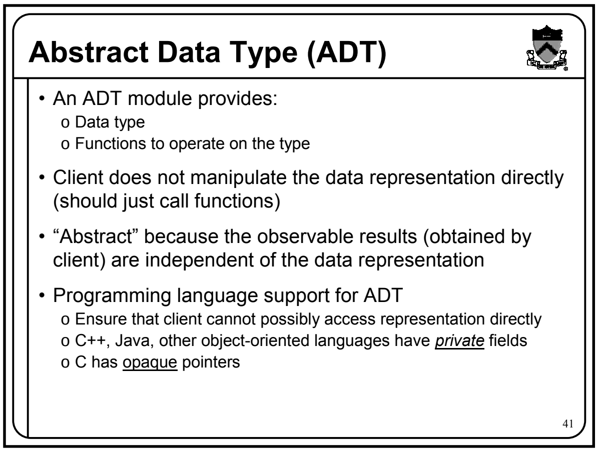
Typedef

stringarray.h

```
#define MAX_STRINGS 128

typedef struct StringArray {
  char *strings[MAX_STRINGS];
  int nstrings;
} *StringArray_T;

extern void ReadStrings(StringArray_T stringarray, FILE *fp);
extern void WriteStrings(StringArray_T stringarray, FILE *fp);
extern void SortStrings(StringArray_T stringarray);
```

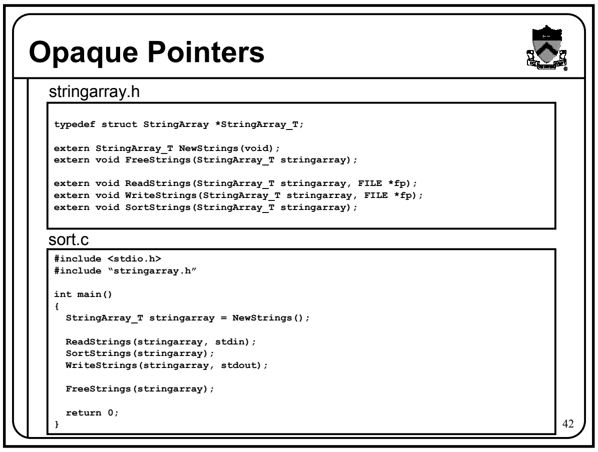
sort.c

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  StringArray_T stringarray = malloc( sizeof(struct StringArray) );
  stringarray->nstrings = 0;

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray);
  WriteStrings(stringarray, stdout);

  free(stringarray);
  return 0;
}
```



Abstract Data Type (ADT)

- An ADT module provides:
  - o Data type
  - o Functions to operate on the type
- Client does not manipulate the data representation directly (should just call functions)
- “Abstract” because the observable results (obtained by client) are independent of the data representation
- Programming language support for ADT
  - o Ensure that client cannot possibly access representation directly
  - o C++, Java, other object-oriented languages have ***private*** fields
  - o C has opaque pointers



Opaque Pointers

stringarray.h

```

typedef struct StringArray *StringArray_T;

extern StringArray_T NewStrings(void);
extern void FreeStrings(StringArray_T stringarray);

extern void ReadStrings(StringArray_T stringarray, FILE *fp);
extern void WriteStrings(StringArray_T stringarray, FILE *fp);
extern void SortStrings(StringArray_T stringarray);
```

sort.c

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  StringArray_T stringarray = NewStrings();

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray);
  WriteStrings(stringarray, stdout);

  FreeStrings(stringarray);

  return 0;
}
```

stringarray.c   (ADT version)

```
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 128
#define MAX_STRINGS 256

typedef struct StringArray {char *strings[MAX_STRINGS];  int nstrings;}
        *StringArray_T;

StringArray_T NewStrings(void) {
  StringArray_T a = malloc(sizeof *a);
  a->nstrings = 0;
  return a;
}

void ReadStrings(StringArray_T a, FILE *fp)  {
  char s[MAX_LENGTH]; int i;

  for(i=0; i < MAX_STRINGS; i++)
      if (fgets(s, MAX_LENGTH, fp))
          strings[i] = strdup(s);
      else break;

  a->nstrings = i;
}
```

Function Pointers

`stringarray.h`

```
extern void ReadStrings(StringArray_T a, FILE *fp);

extern void WriteStrings(StringArray_T a, FILE *fp);

extern void SortStrings(StringArray_T a,
                 int (*compare)(char *string1, char *string2));
```

Calling a function pointer

`stringarray.c`

-
-
-

```
extern void SortStrings(StringArray_T a,
         int (*compare)(char *string1, char *string2))
{
  int i, j;

  for (i = 0; i < a->nstrings; i++)
    for (j = i+1; j < a->nstrings; j++)
      if ((*compare)(a->strings[i], a->strings[j]) > 0) {
        char *swap = a->strings[i];
        a->strings[i] = a->strings[j];
        a->strings[j] = swap;
      }
}
```

Passing a function pointer

`main.c`

```
#include <stdio.h>
#include <string.h>
#include "stringarray.h"

int CompareStrings(char *string1, char *string2) {
  return strcmp(string1, string2);
}

int main()
{
  StringArray_T stringarray = NewStrings();

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray , CompareStrings);
  WriteStrings(stringarray, stdout);

  FreeStrings(stringarray);

  return 0;
}
```

Passing a function pointer

`main.c`

```
#include <stdio.h>
#include <string.h>
#include "stringarray.h"

~~int CompareStrings(char *string1, char *string2) {~~
  ~~return strcmp(string1, string2);~~
~~}~~

int main()
{
  StringArray_T stringarray = NewStrings();

  ReadStrings(stringarray, stdin);
  SortStrings(stringarray , strcmp);
  WriteStrings(stringarray, stdout);

  FreeStrings(stringarray);

  return 0;
}
```

or, just pass strcmp directly!

Generic ADT's with void*

array.h

```
typedef struct Array *Array_T;

extern Array_T Array_New(void);
extern void Array_Free(Array_T a);

extern void Array_Read (Array_T a, void * (*read1)(void));
extern void Array_Write (Array_T a, void * (*write1)(void));
extern void Array_Sort (Array_T a, int (*compare)(void *s1, void *s2)));
```

sort.c

```
#include <stdio.h>
#include "array.h"

int main()
{
  Array_T stringarray = Array_New();

  Array_Read(stringarray, Read_One_String);
  Array_Sort(stringarray, CompareStrings);
  Array_Write(stringarray, Write_One_String);

  Array_Free(stringarray);

  return 0;
}
```

Another ADT example: Stacks

- Like the stack of trays at the cafeteria
- “Push” a tray onto the stack
- “Pop” a tray off the stack
- LIFO: Last-In, First-Out
- Useful in many contexts

stack.h

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

typedef struct Item_t *Item_T;
typedef struct Stack_t *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, Item_T item);
extern Item_T Stack_pop(Stack_T stk);

/* It’s a checked runtime error to pass a NULL Stack_T to any
  routine, or call Stack_pop with an empty stack
*/

#endif
```

Notes on stack.h

- Type **Stack_T** is an <u>opaque pointer</u>
  - o clients can pass **Stack_T** around but can’t look inside
- Type **Item_T** is also an opaque pointer, but define in some other ADT
- **Stack_** is a disambiguating prefix
  - o a convention that helps avoid name collisions
- What does **#ifdef STACK_INCLUDE** do?

Stack implementation module

stack.c

```
#include <assert.h>
#include <stdlib.h>
#include “stack.h”

struct Stack_t {Item_T val; Stack_T next };

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof *stk);
  assert(stk != NULL);
  stk->next = NULL;
  return stk;
}
```

Assert

stack.c

```
#include <assert.h>
#include <stdlib.h>
#include “stack.h”

struct Stack_t {Item_T val; Stack_T next };

Stack_T Stack_new(void) {
  Stack_T stk = malloc(sizeof *stk);
  assert(stk != NULL);
  stk->next = NULL;
  return stk;
}
```

Make sure stk!=NULL, or halt the program!

stack.c, continued

```
int Stack_empty(Stack_T stk) {
   assert(stk);
  return stk->next == NULL;
}

void Stack_push(Stack_T stk, Item_T item) {
   Stack_T t = malloc(sizeof(*t));
  assert(t); assert(stk);
   t->val = item; t->next = stk->next;
  stk->next = t;
}
```

stack.c, continued

```
Item_T Stack_pop(Stack_T stk) {
   Item_T x; Stack_T s;
   assert(stk && stk->next);
   x = stk->next->val;
   s = stk->next;
   stk->next = stk->next->next;
   free(s);
   return x;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#include "stack.h"

int main(int argc, char *argv[]) {
   int i;
   Stack_T s = Stack_new();
   for (i = 1; i < argc; i++)
      Stack_push(s, Item_new(argv[i]));
   while (!Stack_empty(s))
      Item_print(Stack_pop(s));
   return EXIT_SUCCESS;
}
```

Notes on stack.c & user.c

- **user.o** is a client of **stack.h**
  - o change **stack.h** → must re-compile **user.c**
- **user.o** is loaded with **stack.o**
  - o **gcc user.o stack.o**
- **stack.o** is a client of **stack.h**
  - o change **stack.h** → must re-compile **stack.c**

Unchecked opaque pointers: void *

- C language has adequate, not perfect, support for opaque pointers
  - o Can't easily use Stack module to make stack-of-this in one place, stack-of-that in another place (can only have one Item_T in the program)
  - o Can't make stack-of-(char*) because char is not a struct
- Solution: **void ***
  - o Advantage: more flexible
  - o Disadvantage: compiler's type-checker won't help find bugs in your program

stack.h (with void*)

```
#ifndef STACK_INCLUDED
#define STACK_INCLUDED

~~typedef struct Item_t *Item_T;~~
typedef struct Stack_t *Stack_T;

extern Stack_T Stack_new(void);
extern int Stack_empty(Stack_T stk);
extern void Stack_push(Stack_T stk, void *item);
extern void *Stack_pop(Stack_T stk);

/* It's a checked runtime error to pass a NULL Stack_T to any
   routine, or call Stack_pop with an empty stack
*/

#endif
```

Stack implementation (with void*)

stack.c

```
#include <assert.h>
#include <stdlib.h>
#include "stack.h"

struct Stack_t {void *val; Stack_T next };

Stack_T Stack_new(void) {
   Stack_T stk = malloc(sizeof *stk);
   assert(stk);
   stk->next = NULL;
   return stk;
}
```

stack.c (with void*) continued

```
int Stack_empty(Stack_T stk) {
   assert(stk);
   return stk->next == NULL;
}

void Stack_push(Stack_T stk, void *item) {
   Stack_T t = malloc(sizeof(*t));
   assert(t); assert(stk);
   t->val = item; t->next = stk->next;
   stk->next = t;
}
```

stack.c (with void*) continued

```
void *Stack_pop(Stack_T stk) {
   void *x; Stack_T s;
   assert(stk && stk->next);
   x = stk->next->val;
   s = stk->next;
   stk->next = stk->next->next;
   free(s);
   return x;
}
```

client.c (with void*)

```
#include <stdio.h>
#include <stdlib.h>
~~#include "item.h"~~
#include "stack.h"

int main(int argc, char *argv[]) {
   int i;
   Stack_T s = Stack_new();
   for (i = 1; i < argc; i++)
      Stack_push(s, ~~Item_new~~(argv[i]));
   while (!Stack_empty(s))
      printf("%s\n",Stack_pop(s));
   return EXIT_SUCCESS;
}
```

Summary

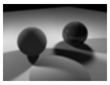
- Modularity is key to good software
  - o Decompose program into modules
  - o Provide clear and flexible interfaces
  - o Easier to understand, test, debug, reuse code
  - o Separate compilation
- Abstract Data Type (ADT) is an important principle
  - o Design interfaces as ADTs
  - o Provides more independence between modules
- Programming techniques
  - o Opaque pointers
  - o Function pointers
  - o Void pointers
  - o Assertions

Programming Style
&
Scope in Programming Languages

CS 217

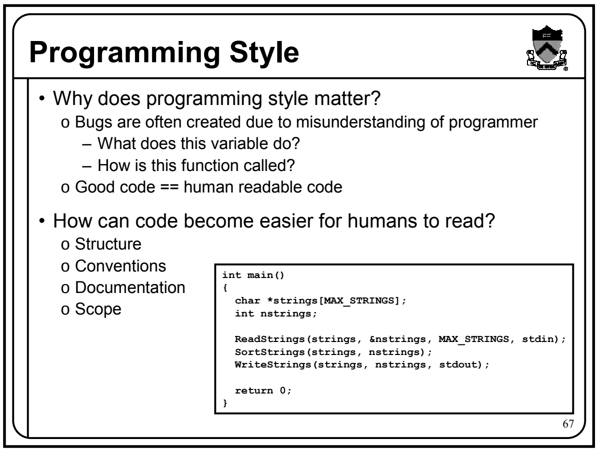
Programming Style

- Who reads your code?
  - o compiler
  - o other programmers
- Which one cares about style?



```
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{
vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,
.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,.5,.2,1.,.7,.3,0.,.05,1.2,1.,8.,-.5,.1,.8,.8,
1.,.3,.7,0.,0.,1.2,3.,-6.,15.,1.,.8,1.,7.,0.,0.,0.,.6,1.5,-3.,-3.,12.,.8,1.,
1.,5.,0.,0.,0.,.5,1.5,};yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A
,B;{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;{B.x+=a*
A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}vec vunit(A)vec A;{return vcomb(1./sqrt(
vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;{best=0;tmin=1e30;s=
sph+5;while(s--sph)b=vdot(D,U=vcomb(-1.,P,s-cen)),u=b*b-vdot(U,U)+s-rad*s -
rad,u=u0?sqrt(u):1e31,u=b-u1e-7?b-u:b+u,tmin=u=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return
amb;color=amb;eta=s-ir;d= -vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s-cen
)));if(d<0)N=vcomb(-1.,N,black),eta=1/eta,d= -d;l=sph+5;while(l--sph)if((e=l -
kl*vdot(N,U=vunit(vcomb(-1.,P,l-cen))))0&&intersect(P,U)==l)color=vcomb(e ,l-
color,color);U=s-color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta* eta*(1-
d*d);return vcomb(s-kt,e0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt
(e),N,black))):black,vcomb(s-ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s-kd,
color,vcomb(s-kl,U,black))));}main(){printf("%d %d\n",32,32);while(yx<32*32)
U.x=yx%32-32/2,U.z=32/2-yx++/32,U.y=32/2/tan(25/114.5915590261),U=vcomb(255.,
trace(3,black,vunit(U)),black),printf("%.0f %.0f %.0f\n",U);}
```

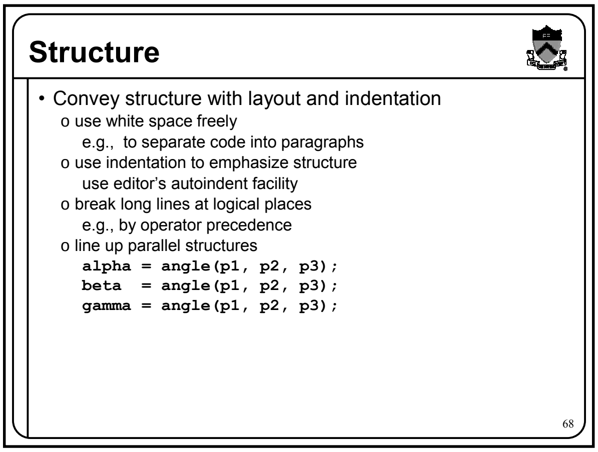
This is a working ray tracer! (courtesy of Paul Heckbert)



Programming Style

- Why does programming style matter?
  - o Bugs are often created due to misunderstanding of programmer
    - – What does this variable do?
    - – How is this function called?
  - o Good code == human readable code
- How can code become easier for humans to read?
  - o Structure
  - o Conventions
  - o Documentation
  - o Scope

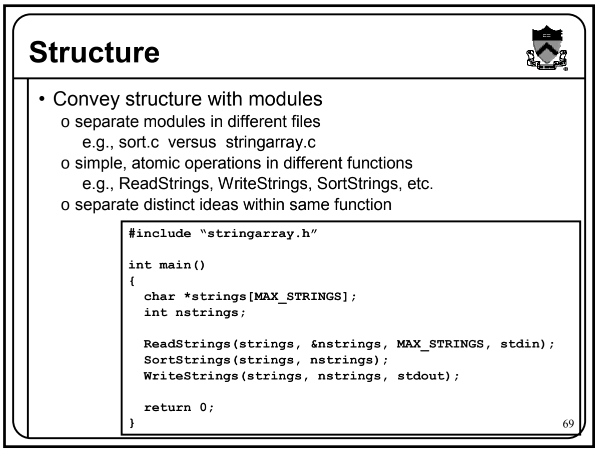
```
int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```



Structure

- Convey structure with layout and indentation
  - o use white space freely
    e.g., to separate code into paragraphs
  - o use indentation to emphasize structure
    use editor’s autoindent facility
  - o break long lines at logical places
    e.g., by operator precedence
  - o line up parallel structures

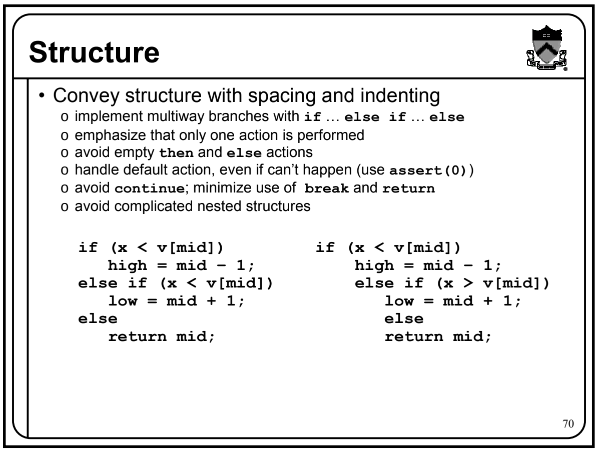
```
alpha = angle(p1, p2, p3);
beta  = angle(p1, p2, p3);
gamma = angle(p1, p2, p3);
```



Structure

- Convey structure with modules
  - o separate modules in different files
    e.g., sort.c versus stringarray.c
  - o simple, atomic operations in different functions
    e.g., ReadStrings, WriteStrings, SortStrings, etc.
  - o separate distinct ideas within same function

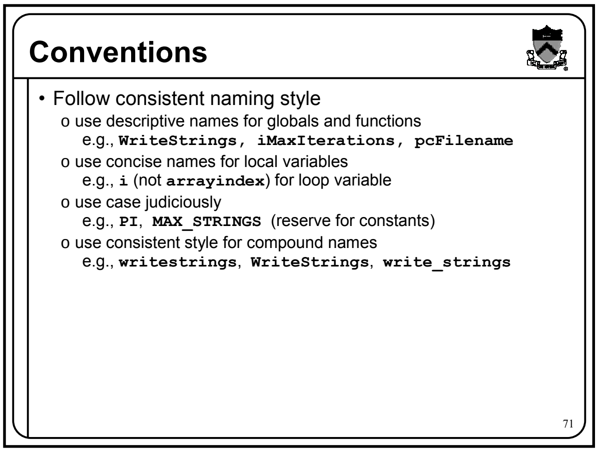
```
#include “stringarray.h”

int main()
{
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```



Structure

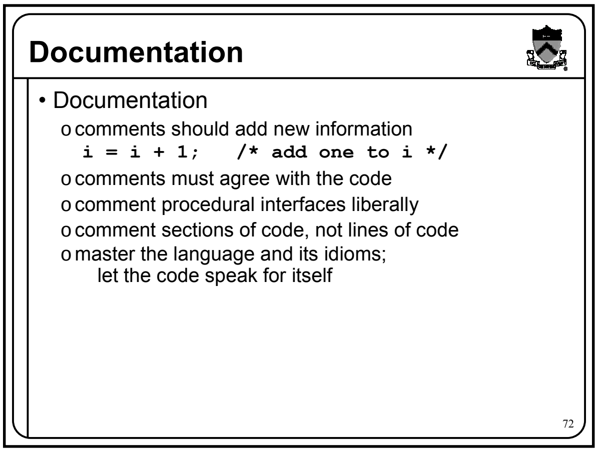
- Convey structure with spacing and indenting
  - o implement multiway branches with **`if`** … **`else if`** … **`else`**
  - o emphasize that only one action is performed
  - o avoid empty **`then`** and **`else`** actions
  - o handle default action, even if can’t happen (use **`assert(0)`**)
  - o avoid **`continue`**; minimize use of **`break`** and **`return`**
  - o avoid complicated nested structures

```
if (x < v[mid])           if (x < v[mid])
   high = mid – 1;            high = mid – 1;
else if (x < v[mid])          else if (x > v[mid])
   low = mid + 1;                low = mid + 1;
else                             else
   return mid;                   return mid;
```



Conventions

- Follow consistent naming style
  - o use descriptive names for globals and functions
    e.g., **`WriteStrings, iMaxIterations, pcFilename`**
  - o use concise names for local variables
    e.g., **`i`** (not **`arrayindex`**) for loop variable
  - o use case judiciously
    e.g., **`PI`**, **`MAX_STRINGS`** (reserve for constants)
  - o use consistent style for compound names
    e.g., **`writestrings`**, **`WriteStrings`**, **`write_strings`**



Documentation

- Documentation
  - o comments should add new information
    `i = i + 1;   /* add one to i */`
  - o comments must agree with the code
  - o comment procedural interfaces liberally
  - o comment sections of code, not lines of code
  - o master the language and its idioms;
    let the code speak for itself

Example: Command Line Parsing

```
/****************************************/
/* Parse command line arguments         */
/* Input is argc and argv from main     */
/* Return 1 for success, 0 for failure */
/****************************************/

int ParseArguments(int argc, char **argv) {
  /* Skip over program name */
  argc--; argv++;

  /* Loop through parsing command line arguments */
  while (argc > 0) {
    if (!strcmp(*argv, "-file")) { argv++; argc--; pcFilename = *argv; }
    else if (!strcmp(*argv, "-int")) { argv++; argc--; iArg = atoi(*argv); }
    else if (!strcmp(*argv, "-double")) { argv++; argc--; dArg = atof(*argv); }
    else if (!strcmp(*argv, "-flag")) { iFlag = 1; }
    else {
      fprintf(stderr, "Unrecognized recognized command line argument: %s\n", *argv);
      Usage();
      return 0;
    }
    argv++; argc--;
  }

  /* Return success */
  return 1;
}
```

Example: Command Line Parsing

```
/****************************************/
/* Parse command line arguments         */
/* Input is argc and argv from main     */
/* Return 1 for success, 0 for failure */
/****************************************/

int ParseArguments(int argc, char **argv) {
   int i;

  /* Loop through parsing command line arguments */
  for (i=1; i<argc; i++)    /* Skip over program name */
    if (!strcmp(argv[i], "-file")) {pcFilename = argv[i+1]; i++; }
    else if (!strcmp(argv[i], "-int")) {iArg = atoi(argv[i+1]); i++; }
    else if (!strcmp(argv[i], "-double")) {dArg = atof(argv[i+1]); i++; }
    else if (!strcmp(argv[i], "-flag")) { iFlag = 1; }
    else {
      fprintf(stderr, "Unrecognized recognized command line argument: %s\n", argv[i]);
      Usage();
      return 0;
    }

   /* Return success */
  return 1;
}
```

Scope

- The <u>scope</u> of an identifier says where it can be used

stringarray.h

```
extern void ReadStrings(char **strings, int *nstrings, int max, FILE *fp);
extern void WriteStrings(char **strings, int nstrings, FILE *fp);
extern void SortStrings(char **strings, int nstrings);
```

sort.c

```
#include "stringarray.h"

#define MAX_STRINGS 128

int main() {
  char *strings[MAX_STRINGS];
  int nstrings;

  ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
  SortStrings(strings, nstrings);
  WriteStrings(strings, nstrings, stdout);

  return 0;
}
```

Definitions and Declarations

- A <u>declaration</u> announces the properties of an identifier and adds it to current scope

```
extern int nstrings;
extern char **strings;
extern void WriteStrings(char **strings, int nstrings);
```

- A <u>definition</u> declares the identifier and causes storage to be allocated for it

```
int nstrings = 0;
char *strings[128];
void WriteStrings(char **strings, int nstrings)
{
  …
}
```

Global Variables

- Functions can use <u>global</u> variables declared outside and above them

```
int stack[100];

int main() {
    . . .           ← stack is in scope
}

int sp;

void push(int x) {
    . . .           ← stack, sp is in scope
}
```

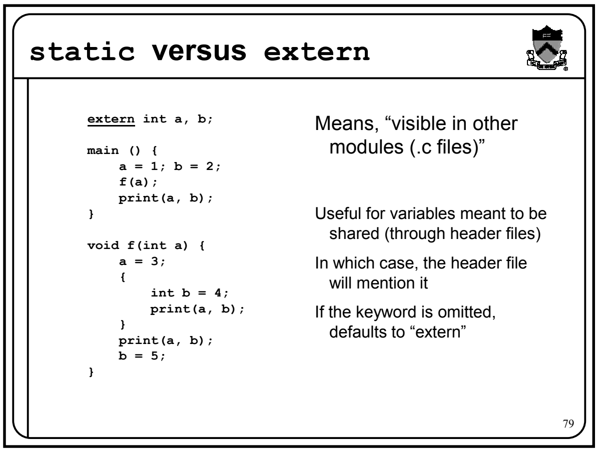
`static` versus `extern`

```
static int a, b;

main () {
    a = 1; b = 2;
    f(a);
    print(a, b);
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        print(a, b);
    }
    print(a, b);
    b = 5;
}
```

Means, "not visible in other modules (.c files)"

Prevents "abuse" of your variables in by "unauthorized" programmers

Prevents inadvertant name clashes



static versus extern

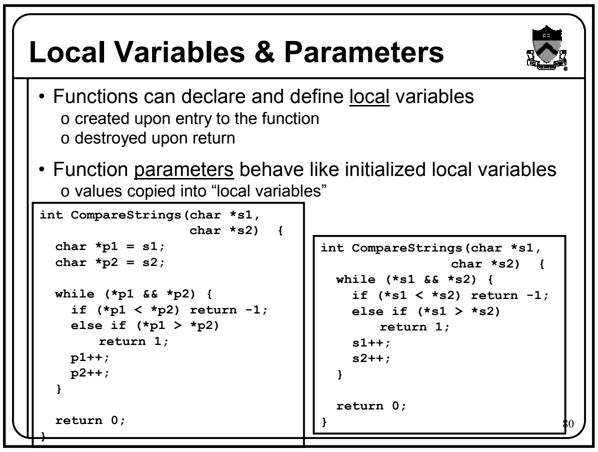
```
extern int a, b;

main () {
    a = 1; b = 2;
    f(a);
    print(a, b);
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        print(a, b);
    }
    print(a, b);
    b = 5;
}
```

Means, “visible in other modules (.c files)”

Useful for variables meant to be shared (through header files)

In which case, the header file will mention it

If the keyword is omitted, defaults to “extern”

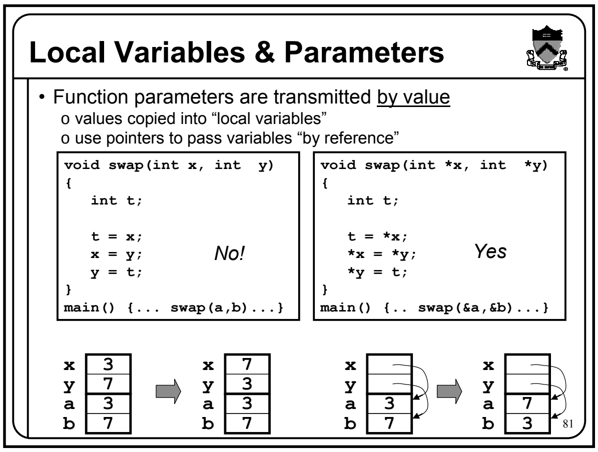


Local Variables & Parameters

- Functions can declare and define local variables
  - o created upon entry to the function
  - o destroyed upon return
- Function parameters behave like initialized local variables
  - o values copied into “local variables”

```
int CompareStrings(char *s1,
                   char *s2)  {
  char *p1 = s1;
  char *p2 = s2;

  while (*p1 && *p2) {
    if (*p1 < *p2) return -1;
    else if (*p1 > *p2)
        return 1;
    p1++;
    p2++;
  }

  return 0;
}
```

```
int CompareStrings(char *s1,
                   char *s2)  {
  while (*s1 && *s2) {
    if (*s1 < *s2) return -1;
    else if (*s1 > *s2)
        return 1;
    s1++;
    s2++;
  }

  return 0;
}
```



Local Variables & Parameters

- Function parameters are transmitted by value
  - o values copied into “local variables”
  - o use pointers to pass variables “by reference”

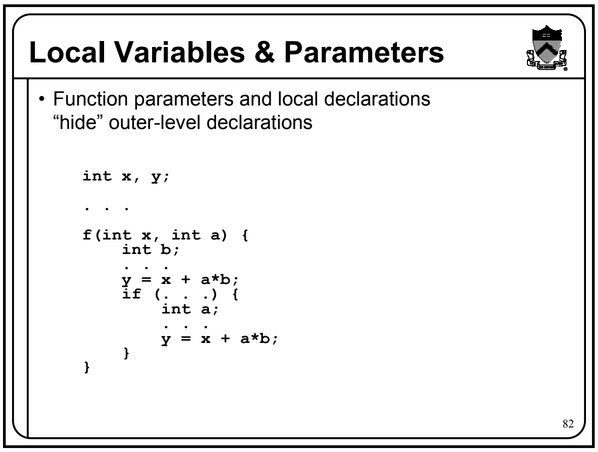
```
void swap(int x, int  y)
{
   int t;

   t = x;
   x = y;
   y = t;
}
main() {... swap(a,b)...}
```

No!

```
void swap(int *x, int  *y)
{
   int t;

   t = *x;
   *x = *y;
   *y = t;
}
main() {.. swap(&a,&b)...}
```

Yes

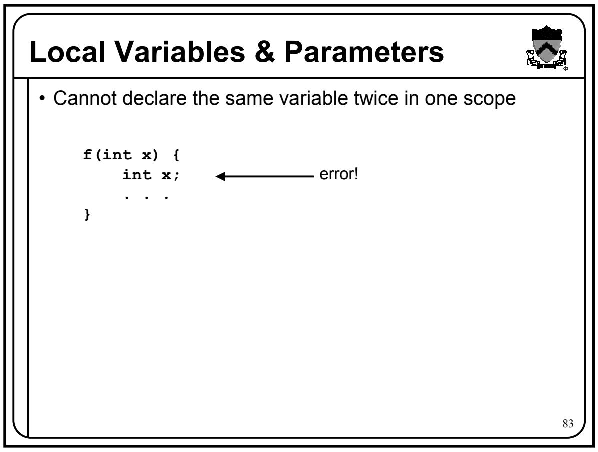
x 3
y 7
a 3
b 7
x 7
y 3
a 3
b 7
x
y
a 3
b 7
x
y
a 7
b 3



Local Variables & Parameters

- Function parameters and local declarations “hide” outer-level declarations

```
int x, y;

. . .

f(int x, int a) {
    int b;
    . . .
    y = x + a*b;
    if (. . .) {
        int a;
        . . .
        y = x + a*b;
    }
}
```

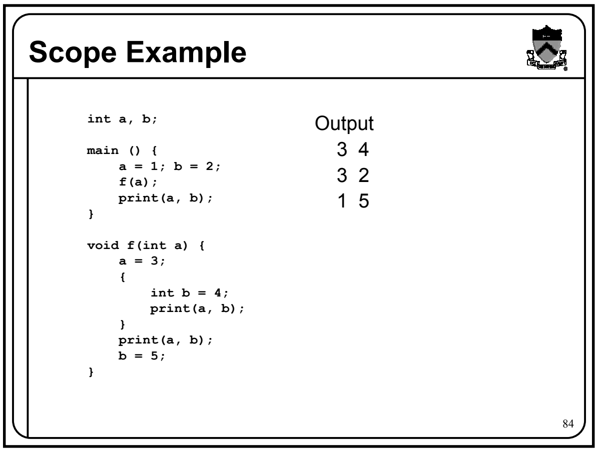


Local Variables & Parameters

- Cannot declare the same variable twice in one scope

```
f(int x) {
    int x;
    . . .
}
```

← error!



Scope Example

```
int a, b;

main () {
    a = 1; b = 2;
    f(a);
    print(a, b);
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        print(a, b);
    }
    print(a, b);
    b = 5;
}
```

Output

```
3 4
3 2
1 5
```

Scope and Programming Style

- Avoid using same names for different purposes
  - o Use different naming conventions for globals and locals
  - o Avoid changing function arguments
- Use function parameters rather than global variables
  - o Avoids misunderstood dependencies
  - o Enables well-documented module interfaces
  - o Allows code to be re-entrant (recursive, parallelizable)
- Declare variables in smallest scope possible
  - o Allows other programmers to find declarations more easily
  - o Minimizes dependencies between different sections of code
  - o Can use `static` to help minimize scope

Summary

- Programming style is important for good code
  - o Structure
  - o Conventions
  - o Documentation
  - o Scope
- Benefits of good programming style
  - o Improves readability
  - o Simplifies debugging
  - o Simplifies maintenance
  - o May improve re-use
  - o etc.

Program Design
&
Hash Tables

CS 217

Program design

1. Problem statement and *requirements*
   What is the problem?
2. *Specification*
   Detailed description of *what* the system should do, not *how*
3. *Design*
   Explore design space, identify algorithms and key *interfaces*
4. *Programming*
   Implement it in the *simplest* possible way; use libraries
5. *Testing*
   Debug and test until the implementation is *correct* and *efficient enough*
6. *Iterate*
   Do the design and implementation conform to the specification?

Design methodologies

- Two important design methodologies
  - o *top-down* design, or stepwise refinement
  - o *bottom-up* design
- Reality: use both
  - o top-down: what functionality do I need?
    Avoids designing and building useless functionality
  - o bottom-up: what functionality do I know how to provide?
    Avoids requiring impossible functionality
- Iterate up and down over the design until everything is both useful and feasible
  - o sometimes overlaps with implementation phase

Stepwise refinement

- Top-down design
  starts with a high-level abstract solution
  refines it by successive transformations to lower-level solutions
  refinement ends at programming-language statements
- Key idea: each refinement or *elaboration*
  must be *small* and *correct*
  must move toward final solution
- Accompany refinements with *assertions*
- Refinements use English & pseudocode, but ultimately result in code

Example: library books

1. Problem statement:
   The circulation file has a line of author,title for each checked out book
   Need a program to find books checked out frequently
2. Specification
   Read a text file; print out one copy of any line that appears 10 or more times
3. Design: how many lines are in a typical circulation file?

```
<findfreq> ≡
    <for each line of input>
        <look up the line in the table (add it if not already there)>
        <increment this line’s count>
    <for each member of the table>
        <if that member’s count ≥ 10>
            <print the line>
```

4. Programming: make forward progress by elaborating chunks

What modules?

- ADT: string table
- Modules:
  - o **main.c** handle command-line arguments (if any) and top-level loops

```
<findfreq> ≡
    <includes>
    <defines>
    int main(int argc, char *argv[]) {
      <locals>
      <for each line of input>
        <look up the line in the table (add it if not already there)>
        <increment this line’s count>
      <for each member of the table>
        <if that member’s count ≥ 10>
          <print the line>
     return EXIT_SUCCESS;
    }
```

  - o **symtable.h** interface for string table
  - o **symtable.c** implementation for string table

Elaboration

- Some elaborations can be done without defining the ADTs

```
<for each line of input> ≡
  while (fgets(line, MAXLINE, stdin))

<defines> ≡
  #define MAXLINE 512

<locals> ≡
  char line[MAXLINE];
```

ADT: string table

symtable.h describes *abstract* operations, *not* implementation; *what*, not *how*

```
typedef struct SymTable *SymTable_T;

SymTable_T SymTable_new(void); /* create a new, empty table. */

int SymTable_put(SymTable_T table, char *key,
                 void *value);
  /* enter (key,value) binding in the table; else return 0 if already there */

void *SymTable_get(SymTable_T table, char *key);
  /* look up key in the table, return value (if present) or else NULL */

void SymTable_map(SymTable_T table,
   void (*f)(char *key, void *value, void *extra),
   void *extra);
  /* apply f to every key in the table ... */
```

This was *top-down* design: specify just those operations necessary for client program

Next step: re-use, if possible

- Avoid some work by searching for an existing module or library that can do the work of SymTable module
- If found, then throw away symtable.h
- Let’s pretend we didn’t find one

A bit of bottom-up design

- Now that we’ve committed to create SymTable ADT, add more operations that make it useful in other applications.
- Don’t get carried away! You’ll end up doing useless work
- This step is optional: you can always do it later as needed.

More of symtable interface

```
void SymTable_free(SymTable_T table);
```
/* Free table */

```
int SymTable_getLength(SymTable_T table);
```
/* Return the number of bindings in table.
It is a checked runtime error for table to be NULL. */

```
int SymTable_remove(SymTable_T table,
                    char *key);
```
/* Remove from table the binding whose key is key. Return 1 if successful, 0 otherwise.

It is a checked runtime error for table or key to be NULL. */

Cleaning up the interface

- Keep ADT interfaces small
  - o If an operation can be performed entirely outside the ADT, remove it from the interface
  - o Example: SymTable_getLength

```
void count_me(char *key, void *value, void *pCnt){
 *((int *)pCnt) += 1;
}
SymTable_getLength(Symtable_T table) {
 int count = 0;
 SymTable_map(table, count_me, &count);
 return count;
}
```

Back to the client

- ADT interface gives enough information to finish the client, main.c

<locals> +≡
```
   SymTable_T table = SymTable_new();
   struct stats *v;
```
<includes> +≡
```
   #include “symtable.h”;
```
<global-defs> ≡
```
   struct stats {int count;};
```
(also must define makeStats...)

<look up the line in the table (add it if not already there)> ≡
```
   v = SymTable_get(table, line);
   if (!v) {
      v = makeStats(0);
      SymTable_put(table, line, v);
   }
```

Finishing the client

<for each member of the table> ≡
```
   SymTable_map(table, maybeprint, NULL);
```

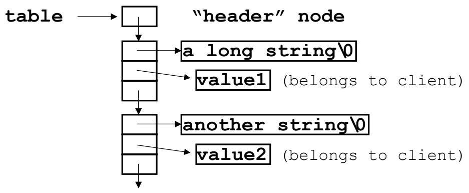
<if that member’s count ≥ 10, print the line> ≡
```
void maybeprint(char *key, void *stats,
                void *extra){
 if (((struct stats*)stats)->count >= 10)
     fputs(key, stdout);
}
```

What the client main looks like

```
int main(int argc, char *argv[]) {
   char line[MAXLINE];
   SymTable_T table = SymTable_new();
   struct stats *v;
   while (fgets(line, MAXLINE, stdin)) {
      v = SymTable_get(table, line);
      if (!v) {
            v = makeStats(0);
            SymTable_put(table, line, v);
      }
      incrementStats(v,1);
   }
   SymTable_map(table, maybeprint, NULL);
   return EXIT_SUCCESS;
 }
```

ADT implementation

- Now, begin to design the ADT implementation
- Start with a simple algorithm / data structure
  - o It’s good for debugging and testing the interface
  - o Maybe it’s good enough for the production system -- that would save the work of implementing a clever algorithm



Next: Implement the ADT module

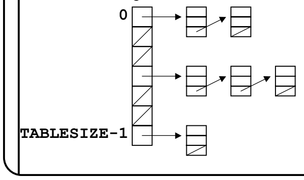
- You've already done this in Programming Assignment 1.
- So I won't explain it here.

Testing

5. **Testing:** findfreq works, but runs too slowly on ***large*** inputs. Why?
   - o Improve symtable's implementation; don't change its interface

- Solution: use a hash table

  A symtable will be a pointer to an array of `TABLESIZE` linked lists
  "Hash" the string into an integer `h`
  let `i = h % TABLESIZE`
  search the `i`th linked list for the string, or
  add the string to the head of the ith list



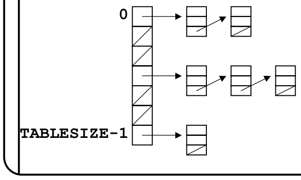
How large an array?

Array should be long enough that average "bucket" size is 1.

If the buckets are short, then lookup is fast.

If there are some very long buckets, then average lookup is slow.

This is OK:



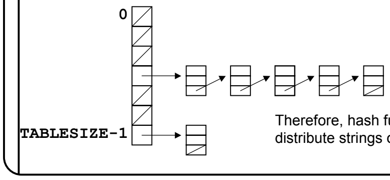
The need for a good hash function

Array should be long enough that average "bucket" size is 1.

If the buckets are short, then lookup is fast.

If there are some very long buckets, then average lookup is slow.

This is not so good:



Therefore, hash function must evenly distribute strings over integers 0..TABLESIZE

A reasonable hash function

How to hash a string into an integer?

Add up all the characters? (won't distribute evenly enough)

How about this: $(\Sigma\, a^i x_i) \bmod c$ (best results if a,c relatively prime)

- Choose a = 65599, c = 2^{32}

```
unsigned hash(char *string) {
  int i; unsigned h = 0;
  for (i=0; string[i]; i++)
      h = h * 65599 + string[i];
  return h;
}
```

- How does this implement $(\Sigma\, a^i x_i) \bmod c$?

Hash table in action

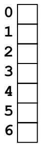
Example: TABLESIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: the. hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found.



Hash table in action

Example: TABLESIZE = 7

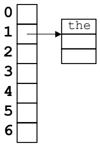
Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string “the”; not found

Now: table[1] = makelink(key, value, table[1])

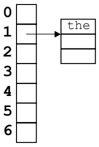


Hash table in action

Second word: “cat”. hash(“cat”) = 3895848756. 3895848756 % 7 = 2.

Search the linked list table[2] for the string “cat”; not found

Now: table[2] = makelink(key, value, table[2])

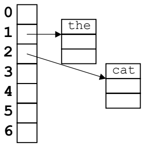


Hash table in action

Third word: “in”. hash(“in”) = 6888005. 6888005% 7 = 5.

Search the linked list table[5] for the string “in”; not found

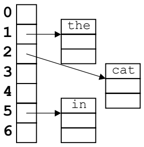
Now: table[5] = makelink(key, value, table[5])



Hash table in action

Fourth word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string “the”; found it!



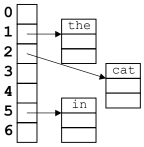
Hash table in action

Fourth word: “hat”. hash(“hat”) = 865559739. 865559739 % 7 = 2.

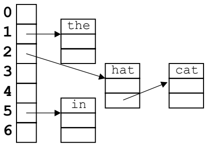
Search the linked list table[2] for the string “hat”; not found.

Now, insert “hat” into the linked list table[2].

At beginning or end? Doesn’t matter.



Hash table in action



Number of buckets

- Average bucket size should be short
- Thus, number of buckets should be (approximately) greater than number of entries in table
- If (approximate) number of entries is known in advance, this is easy to arrange
- If (approximate) number of entries is unpredictable, then one can dynamically grow the hash table
- How to do it; cost analysis; ...

References on hashing

- Hanson, *C Interfaces and Implementations,* §3.2
- Sedgewick, *Algorithms 3rd Edition in C,* §14.2

Memory Allocation

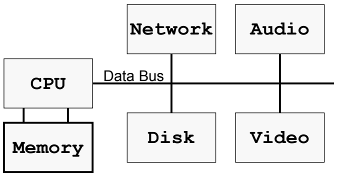
CS 217

Memory Allocation

- Good programmers make efficient use of memory
- Understanding memory allocation is important
  - o Create data structures of arbitrary size
  - o Avoid "memory leaks"
  - o Run-time performance

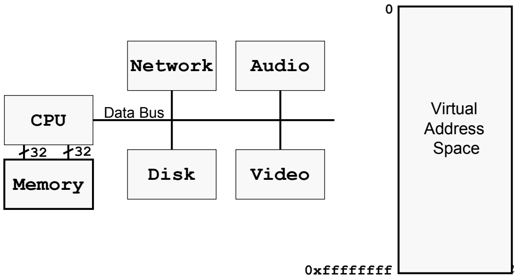
Memory

- What is memory?
  - o Storage for variables, data, code, etc.



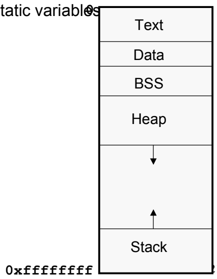
Memory

- What is memory?
  - o Storage for variables, data, code, etc.
  - o Unix provides virtual memory



Memory Layout

- How is memory organized?
  - o Text = code, constant data
  - o Data = initialized global and static variables
  - o BSS = uninitialized (zero) global and static variables
  - o Stack = local variables
  - o Heap = dynamic memory



Memory Layout

data

text (or “read-only data”)

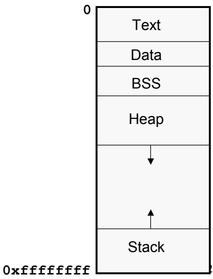
bss

stack

heap

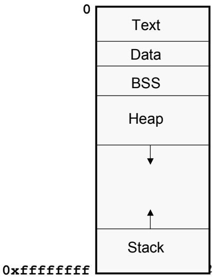
text

```
char string = “hello”
int iSize;

char *f(void)
{
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



Memory Allocation

- How is memory allocated?
  - o Global and static variables = program startup
  - o Local variables = function call
  - o Dynamic memory = malloc()



Memory Allocation

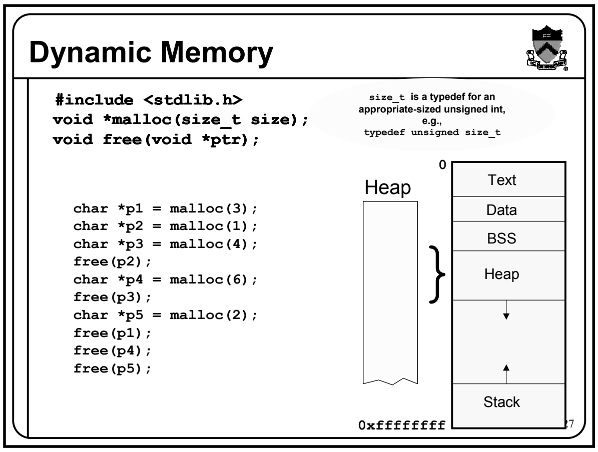
```
int iSize;              ← allocated in BSS, set to zero at startup

char *f(void)
{
    char *p;            ← allocated on stack at start of function f
    iSize = 8;
    p = malloc(iSize);  ← 8 bytes allocated in heap by malloc
    return p;
}
```

Memory Deallocation

- How is memory deallocated?
  - o Global and static variables = program finish
  - o Local variables = function return
  - o Dynamic memory = free()
- All memory is deallocated at program termination
  - o It is good style to free allocated memory anyway

Memory Deallocation

```
int iSize;              ← available until program termination

char *f(void)
{
    char *p;            ← deallocated by return from function f
    iSize = 8;
    p = malloc(iSize);  ← deallocate by calling free(p)
    return p;
}
```

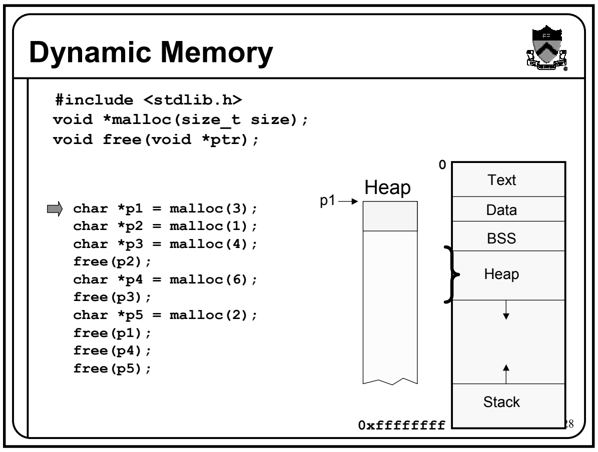


Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

size_t is a typedef for an appropriate-sized unsigned int, e.g., typedef unsigned size_t

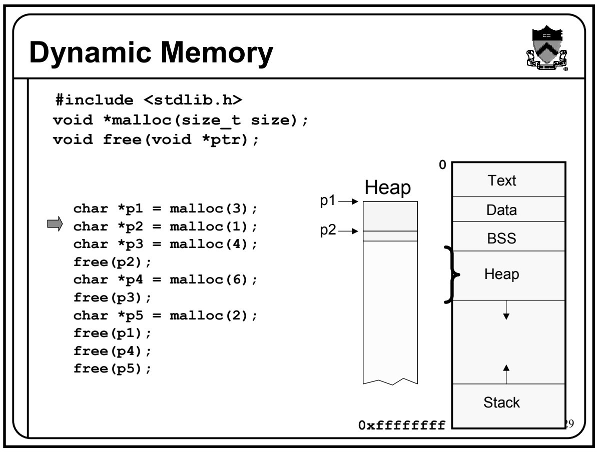
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

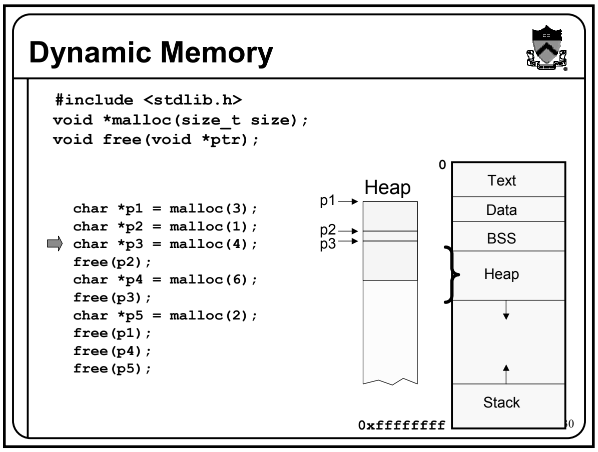
```
⇨ char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

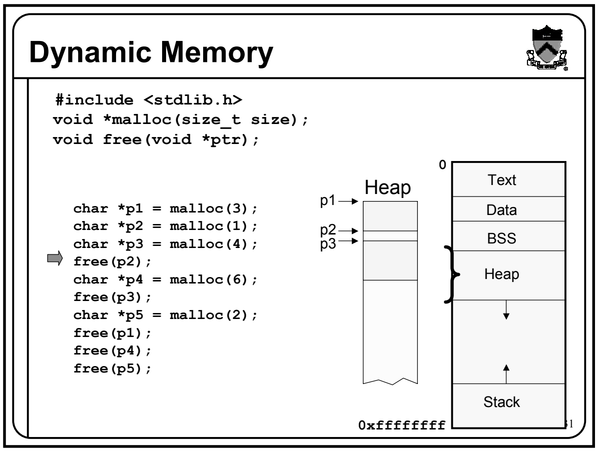
```
  char *p1 = malloc(3);
⇨ char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

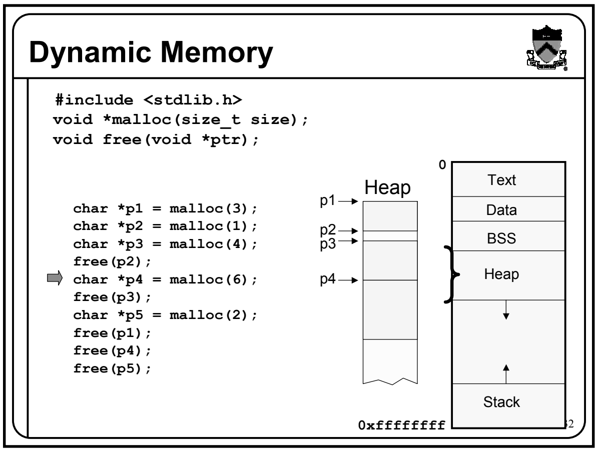
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
⇨ char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
⇨ free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

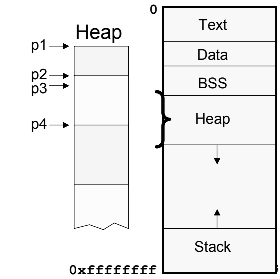
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
⇨ char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```

Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

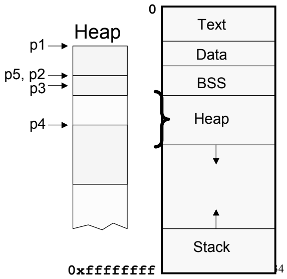
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
⇨ free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

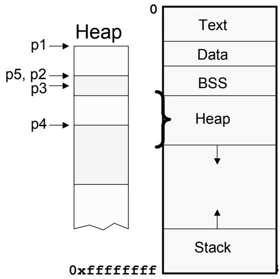
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
⇨ char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

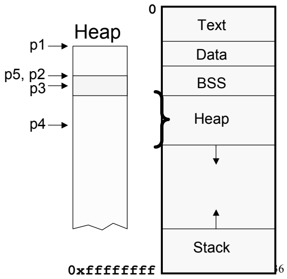
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
⇨ free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

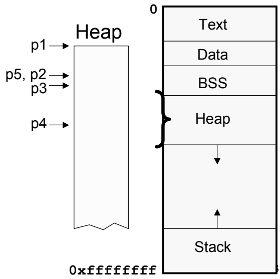
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
⇨ free(p4);
  free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
⇨ free(p5);
```

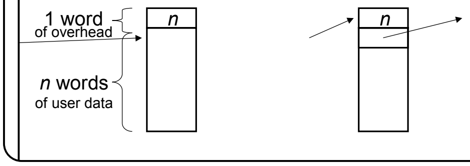


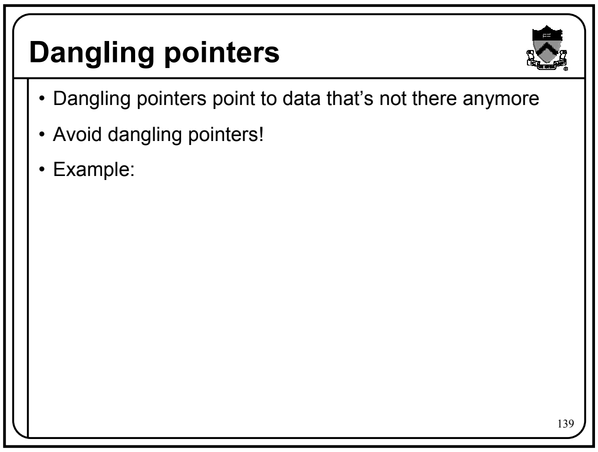
Memory allocator ADT

- Malloc & free are the operations of an ADT
  - o How do they work inside?
- First answer: it's an ADT, you're not supposed to ask!
- Second answer:

malloc(s)
n = ⌈s / sizeof(int)⌉

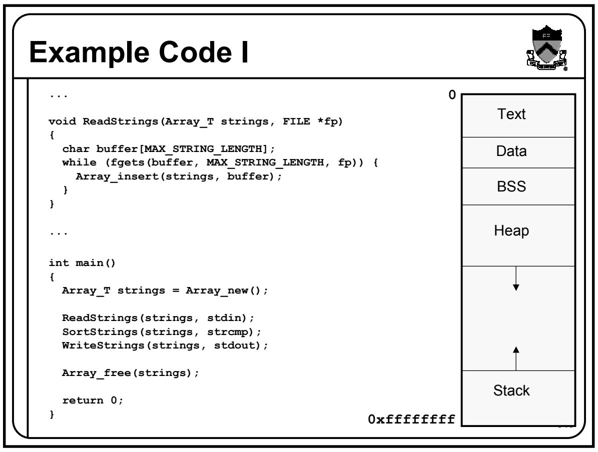
free(p)
put p into linked list of free objects





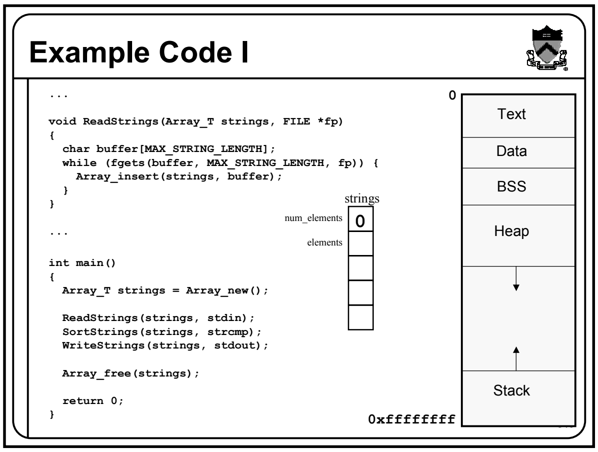
Dangling pointers

- Dangling pointers point to data that's not there anymore
- Avoid dangling pointers!
- Example:



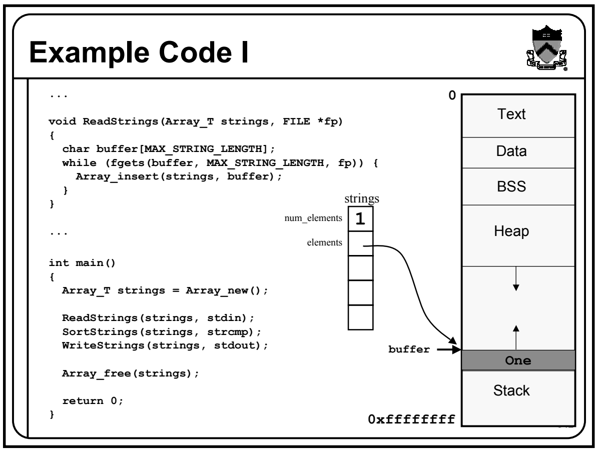
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



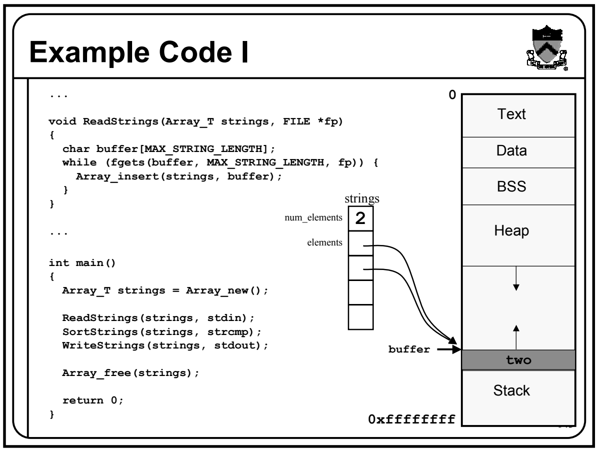
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



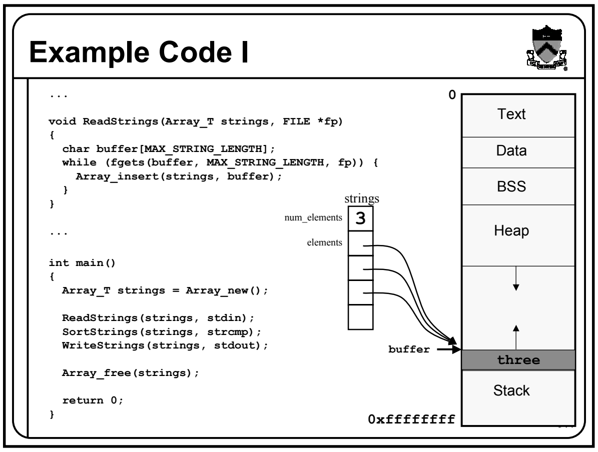
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```

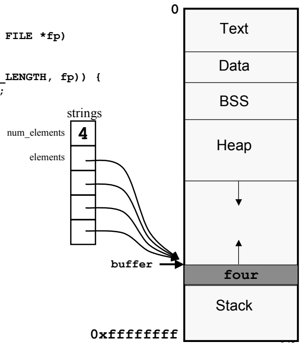


Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```

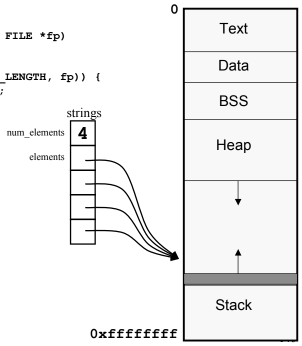
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



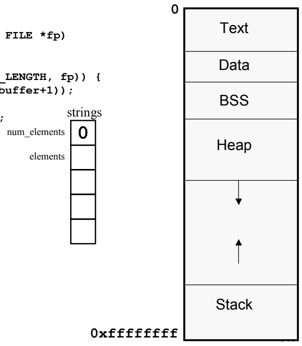
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



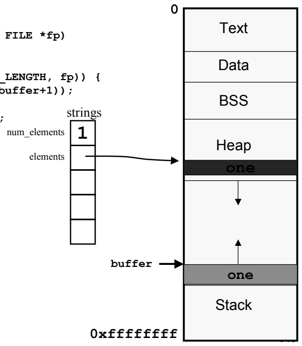
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



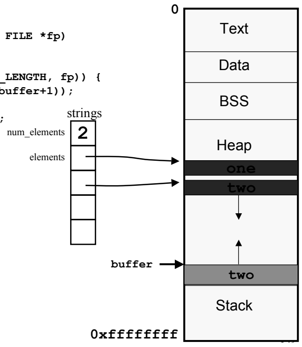
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



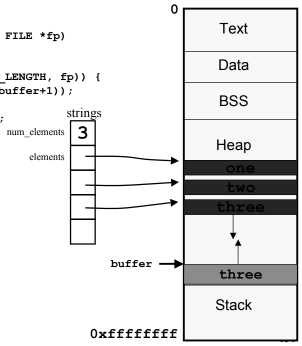
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



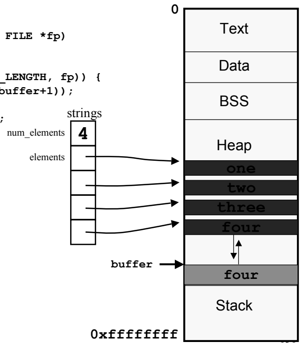
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



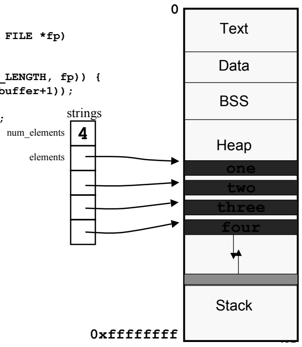
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Example Code II

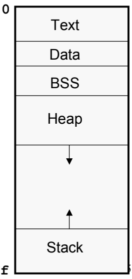
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Static Local Variables

- `static` keyword in declaration of local variable means:
  - o Available (if within scope) throughout entire program execution
  - o Variable is allocated from Data or BSS, not stack
  - o Acts like global variable with limited scope

```
int iSize;

char *f(void)
{
    static int first = 1;
    if (first) {
      iSize = GetSize();
      first = 0;
    }
    ...
}
```



Memory Initialization

- Local variables have undefined values

```
int count;
```

- Memory allocated by malloc has undefined values

```
char *p = malloc(8);
```

- If you need a variable to start with a particular value, use an explicit initializer

```
int count = 0;
p[0] = '\0';
```

- Global and static variables are initialized to 0 by default

```
static int count = 0;
```

is the same as

```
static int count;
```

It is bad style to depend on this

Summary

- Three types of memory
  - o Global and static variables = BSS
  - o Local variables = stack
  - o Dynamic memory = heap
- Three types of allocation/deallocation strategies
  - o Global and static variables (BSS) = program startup/termination
  - o Local variables (stack) = function entry/return
  - o Dynamic memory (heap) = malloc()/free()
- Take the time to understand the differences!

Memory management in program design

CS 217

ADT implementation

- Recall the simple implementation of the symtable ADT:

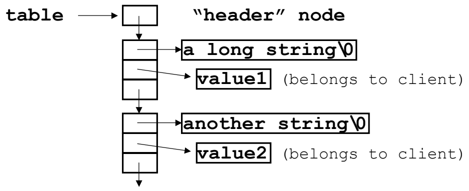
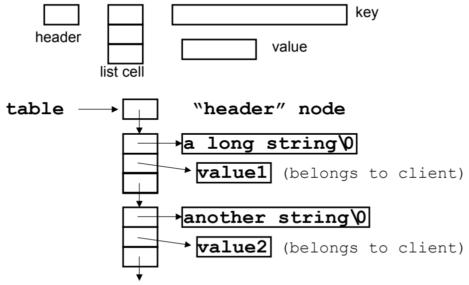


table
"header" node
a long string\0
value1 (belongs to client)
another string\0
value2 (belongs to client)

Memory management issues

- Does ADT or client "own" the data?
  - o Who mallocs/frees each kind of node?


header
list cell
key
value
table
"header" node
a long string\0
value1 (belongs to client)
another string\0
value2 (belongs to client)

Client can't free ADT repr. directly

- What happens if,

```
{struct SymTable_T table; . . . free(table);}
```

then the list cells don't get freed!

- So, ADT must "own" headers and list cells

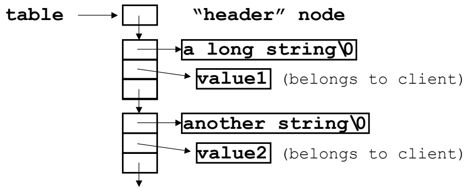


table
"header" node
a long string\0
value1 (belongs to client)
another string\0
value2 (belongs to client)

ADT can't free values directly

- ADT just sees `void *value;`
- Value pointer might be root of big data structure, all the pieces need to be freed.
- Thus, client must "own" the value nodes.

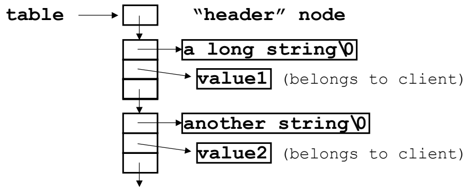


table
"header" node
a long string\0
value1 (belongs to client)
another string\0
value2 (belongs to client)

Who owns the key?

- Both client and ADT "know" about `char *key;`
- Therefore, we are faced with a design choice
- Choice 1: client owns the key.
  - o Consequence: must call SymTable_put only with a string that will last a long time. *(But our client didn't do that!)*

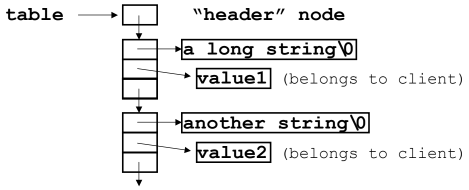


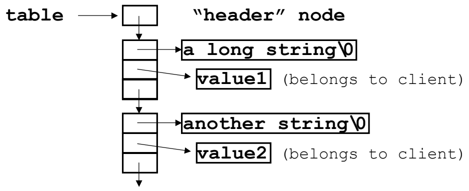
table
"header" node
a long string\0
value1 (belongs to client)
another string\0
value2 (belongs to client)

`line` variable is overwritten each time

```
int main(int argc, char *argv[]) {
 char line[MAXLINE];
 SymTable_T table = SymTable_new();
 struct stats *v;
 while (fgets(line, MAXLINE, stdin)) {
   v = SymTable_get(table, line);
   if (!v) {
         v = makeStats(0);
         SymTable_put(table, line, v);
   }
 SymTable_map(table, maybeprint, NULL);
 return EXIT_SUCCESS;
}
```

Choice 2: ADT owns the key

- Consequence: **`SymTable_put`** must copy its **`key`** argument into a newly malloc'ed string object.



table

"header" node

a long string\0

value1 (belongs to client)

another string\0

value2 (belongs to client)

Put away your toys when you're finished playing . . .

- When client is done with a symbol table, it should give the memory back.
- But client can't call **`free`** directly (as we already demonstrated)
- So there must be an interface function for client to say "I'm done with this"
  `SymTable_free(SymTable_T table);`
- It should free the header, list cells, strings
- Should it free the values?
  - o Can't do it by calling **`free`** directly (as we already demonstrated)
  - o Another design choice!

How to free the values

- Option 1: Client frees all the values before calling `SymTable_free(table)`
  - o Can do this using SymTable_map(table, free_it, NULL);
  - o Minor bother: temporarily leaves dangling pointers in the table
  - o Minor bother: it's clumsy
- Option 2: `SymTable_free` calls client function

```
void SymTable_free(SymTable_T table,
    void (*f)(char *key, void *value, void *extra),
    void *extra);
```

/* Free entire table. During this process, if f is not NULL, apply f to each binding in table. It is a checked runtime error for table to be NULL. */

- We will choose Option 1.

Game-playing programs

CS 217

Why make computers play games?

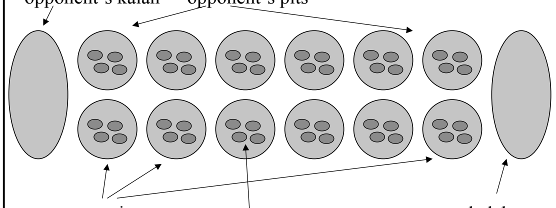
- It's fun
- "Artificial Intelligence" (Can computers do some of the things that people do?)
- An interesting problem in algorithms
- A good software engineering problem

What kind of games

- We will study "deterministic two-player games of perfect information"
  - o Examples: Chess, checkers, tic-tac-toe, othello, kalah . . .
  - o Nonexamples: Gin rummy (random, hidden information), Risk (multiplayer, random), . . .
- For these games, "alpha-beta search" is the basic technique

The game of Kalah

- A traditional African game with many variations, we will use “Egyptian rules”



opponent’s kalah opponent’s pits

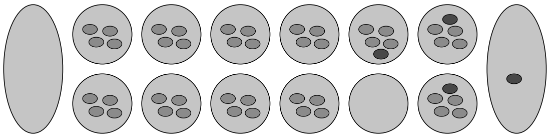
my pits

initially, 4 beans per pit

my kalah

The game of Kalah

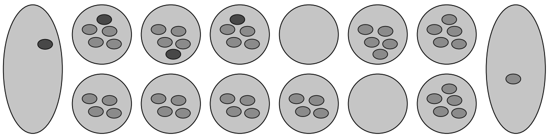
- Move by picking up beans from one of my pits, distributing counter-clockwise
  - o skip opponent’s kalah



- Object of game is to get more beans into my kalah

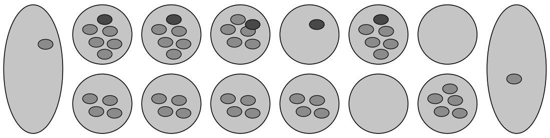
More rules

- If move ends in same player’s kalah, go again



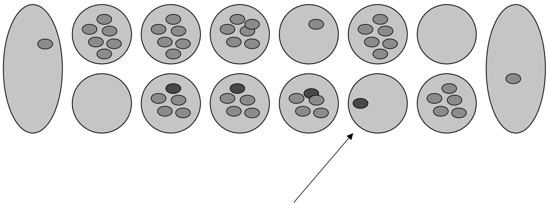
Kalah, continued

- Opponenent has gone again...



Captures

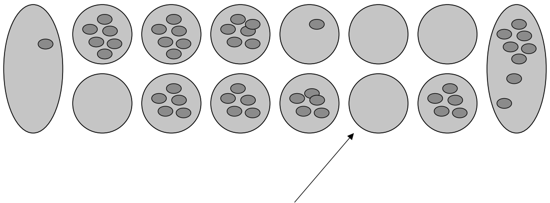
- If last bean ends in empty pit in same player’s side, capture



last bean in empty pit on my side, so . . .

Capture, continued

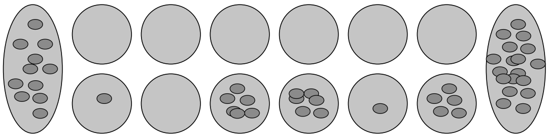
- If last bean ends in empty pit in same player’s side, capture



take this bean and all in opposite pit into my kalah

End of game

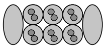
- When one player's pits are all empty, game ends
- Other player keeps all beans in his/her pits



- I win, 35 to 11

Small game to illustrate algorithms

- Only 3 pits each side, only 2 beans per pit.

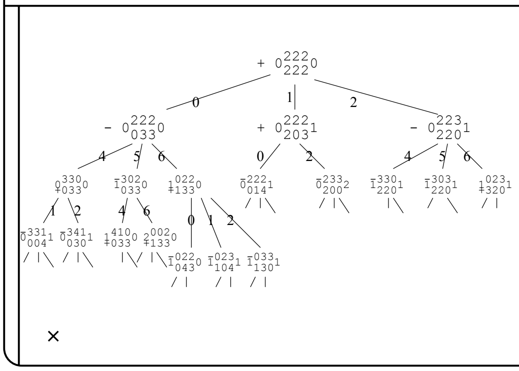


- Sometimes give game state just by numbers:

$+\ 0^{222}_{222}0$

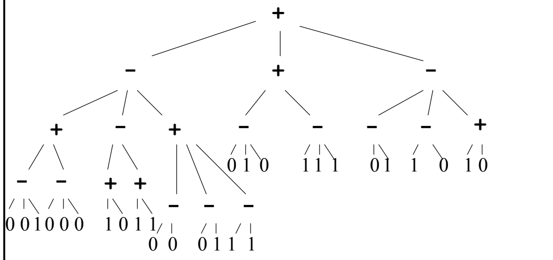
+ means my turn,
− means opponent's

Game tree



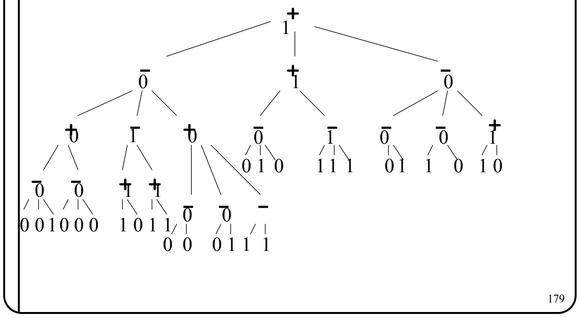
Leaves of game tree: win/loss

[note: these outcomes don't correspond to actual game on previous slide]



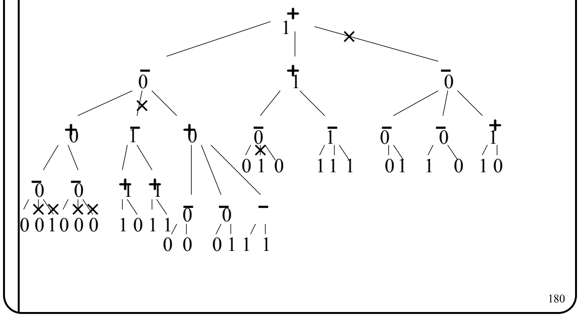
Evaluating a game tree

MAX nodes maximize, MIN nodes minimize



Cutting off the search

Some subtrees can't affect game outcome

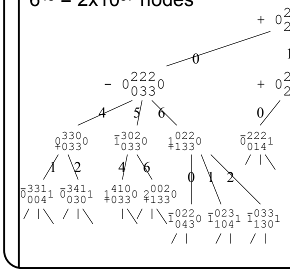


Game tree too deep for complete search

In (3+3)x2 Kalah, approximately 12 levels deep

In (6+6)x4 Kalah, approximately 48 levels deep

$6^{48} = 2x10^{37}$ nodes



Heuristic evaluation function

- "Eyeball" the board and estimate who's winning
- Example for Kalah:

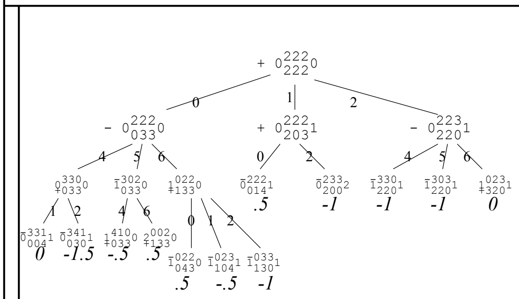
```
If  Game_over  then
   (MyKalah-HisKalah)+(MyPits-HisPits)
else (MyKalah-HisKalah)+½(MyPits-HisPits)
```

Eval($1\,{}^{410}_{033}\,0$) = -.5

Eval($\bar{1}\,{}^{330}_{220}\,1$) = -1

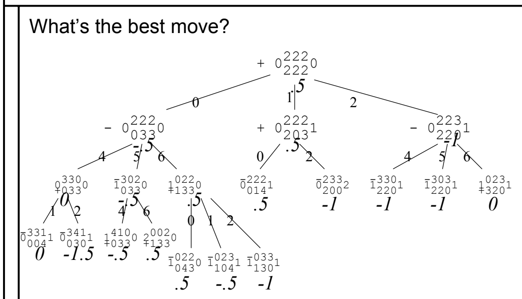
- Important feature of heuristic eval. function: When game over, accurately reports winner!

Game tree with heuristic eval.



Evaluation of game tree

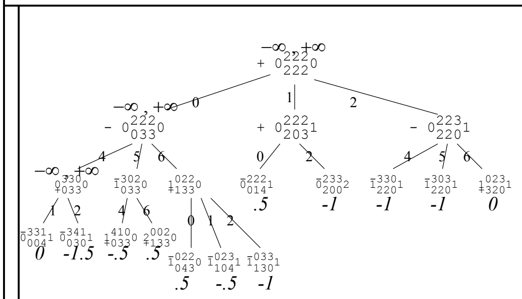
What's the best move?



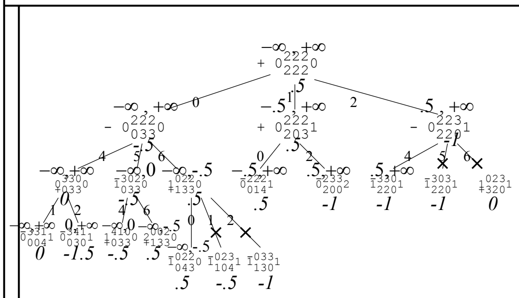
Alpha-Beta search

```
search(N, α, β, depth)  {
    if N is a leaf or depth is too great, then return eval(N)
    if N is a Min node then
        for each successor Ni of N
            β ← Min(β, search(Ni, α, β, depth+1));
            if α >= β then  return α
        return β;
    else (N is a Max node)
        for each successor Ni of N loop
            α ← Max(α, search(Ni, α, β, depth+1));
            if α >= β then  return β
        return α;
}
```

Alpha-beta search of game tree



Alpha-beta search of game tree



What Alpha and Beta mean

"This node can be relevant only if its value is between α and β"

If value < α then Max could do better somewhere else

If value > β then Min could do better somewhere else

If $\alpha > \beta$ then this node can't possibly be relevant!

How to improve α-β search

- Search to greater depth
  - (but search time is exponential in depth, so increasing depth by 1 ply multiplies search time by 3 or 6)
- Improve the heuristic evaluation function
  - (but this might slow down the search!)
- Other tricks beyond the scope of this course
  - opening book
  - endgame tables
  - iterative deepening
  - prove-best or disprove-rest
  - transposition tables

Incremental Evaluation for Alpha-Beta search

CS 217

Incremental evaluation

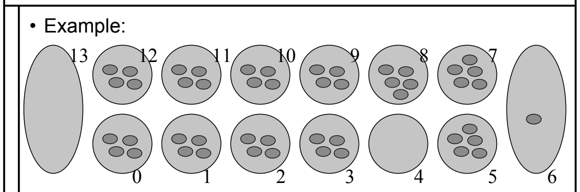
- Heuristic function has to be evaluated at each leaf
- There are exponentially many leaves
- Thus, efficiency is important
- But the heuristic might depend on global properties of the game board -- difficult to evaluate in "an instant"
- Solution: incremental evaluation
  - That is: each game move makes only small, localized changes to the board, so just compute the *change* to the heuristic function

Breaking a move into tiny deltas

- Motivation: computing Δheuristic is easier if the change to board is very simple
- Each game move is one or more simple submoves (deltas)
- For Kalah: a delta is, "put ±*n* beans into pit *p"*
  - *p may be any of the 14 pits, including kalahs*
  - *n may be positive or negative*

Breaking a move into tiny deltas

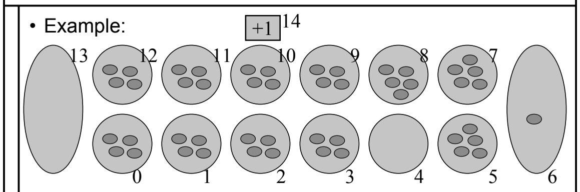
- Example:



- Move 0 expands to (0,-4),(1,1),(2,1),(3,1),(8,-5),(6,6)
- Move 1 expands to (1,-4),(2,1),(3,1),(4,1),(5,1)
- Move 2 expands to (2,-4),(3,1),(4,1),(5,1),(6,1)

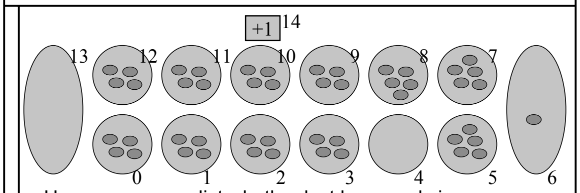
Whose turn is it?

- Example:



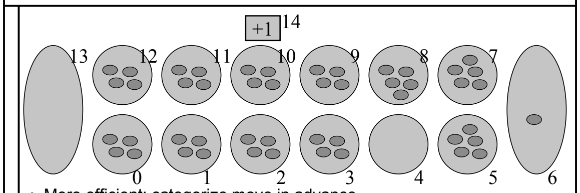
- Move 0 expands to (0,-4),(1,1),(2,1),(3,1),(8,-5),(6,6),(14,-2)
- Move 1 expands to (1,-4),(2,1),(3,1),(4,1),(5,1),(14,-2)
- Move 2 expands to (2,-4),(3,1),(4,1),(5,1),(6,1)

Hints on move expansion



- How can you predict whether last bean ends in
  - o Empty pit on my side
  - o My own kalah
  - o Somewhere else?
- Solution 1:
  - o make a copy of game state, use copy as workspace to see the effect of a move.
  - o Problem: this is slow

Hints on move expansion



- More efficient: categorize move in advance
  - o beans<13 & exactly enough to land in my kalah
  - o beans<13 & lands in empty pit on my side
  - o beans<13 & none of the above
  - o beans=13
  - o beans>13 & exactly enough to land in my kalah
  - o beans>13 & none of the above
- Of course, the number 13 won't appear in your program...

Incremental evaluation

- Example for Kalah:

```
If Game_over then
      (MyKalah-HisKalah)+(MyPits-HisPits)
else (MyKalah-HisKalah)+½(MyPits-HisPits)
```

- Warning: this is not a particularly good function!
- Basic game state has 15 numbers (6+1+6+1+1)
- Now we need two more numbers in game state:

  MyPits, HisPits

Incremental evaluation

```
  If Game_over then
        (MyKalah-HisKalah)+(MyPits-HisPits)
  else  (MyKalah-HisKalah)+½(MyPits-HisPits)
applyDelta(pit,beans) {
  Pits[pit] += beans;
  if (0 ≤ pit ≤ 5)    MyPits += beans;
  if (7 ≤ pit ≤ 12)  HisPits += beans;
}
eval( ) {
  if (MyPits*HisPits==0)
        return Pits[6]-Pits[13]+MyPits-HisPits ;
  else return Pits[6]-Pits[13]+0.5*(MyPits-HisPits) ;
}
```

- *Constant time to evaluate!*
- *Constant time to evaluate!*

Game state

- Game state requires:
  - o Contents of each pit
  - o Whose turn it is
  - o Auxiliary information needed for incremental heuristic evaluator
- Organize game state for speed:
  - o applyDelta() should be constant time, if possible
  - o eval() should be constant time, if possible
  - o testing for end-of-game should be constant time
  - o etc.

Move generation

- In each game state, need to generate all the legal moves
- For Chess, this can be tricky
- For Scrabble, this can be quite intricate
  - o See Appel & Jacobson, "The World's Fastest Scrabble Program", *Communications of the ACM,* 1988.
- For Kalah, it's easy:
  - o Numbers 0-5 (or 7-12), except for empty pits.

Alpha-Beta w/ incremental eval

```
search($\alpha$, $\beta$, depth)  {
    Move moves[]; Delta deltas[];
    if (gameOver() || depth>LIMIT) return eval( )
    if N is a Min node then
        genMoves(moves)
        for each m in moves
            expandMove(m,deltas)
            for each d in deltas  do  applyDelta(d);
            $\beta \leftarrow$ Min($\beta$, search($\alpha$, $\beta$, depth+1));
            for each d in deltas  do  unApplyDelta(d);
            if $\alpha$ >= $\beta$ then  return $\alpha$
        return $\beta$;
    else (N is a Max node)
        do similar stuff...
}
```

The programming assignment

- Implement a modular game player
- Instantiate for Kalah and Tic Tac Toe
- Work in teams of approximately 10 people
- Break the work into modules (during precept)
  - o Typically: 6 modules
- Design header files (during precept)
- Each person implements approximately two modules
- Each module to be implemented approximately 3 times

Schedule

- First precept:
  - o Break problem into modules, design header files
  - o Need "secretaries" to write down header files, type them in
  - o Need e-mail list for all members of team
  - o Team members should choose modules to implement
- Second precept:
  - o Everyone has already started on implementation
  - o Discuss problems with header files, decide on necessary changes
- Third precept
  - o More discussion and adjustment
- Precept attendance is extremely important
  - o Don't let your teammates down!

Rules

- Don't read other people's .c files, even on your own team
  - o Exception: you can help other people debug
- You may discuss general implementation ideas with others (even with people on different teams)
  - o Example: If you're doing expandMove, you could chat with other people doing expandMove on your team or another team
  - o Example: If you're doing applyDelta, you may need to talk to people doing other modules that interact with it.
- You should write your own source code
  - o The usual rules apply

The playoffs!

- We will run a tournament of all the players, one from each team
- The winning team will get extra glory (but not a higher grade in the course)
  - o Thus, you really should feel free to improve your understanding of the problem and solution by discussions with classmates from any team.

Testing, Timing, Profiling, & Instrumentation

CS 217

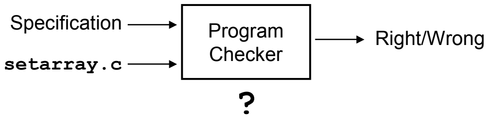
Testing, Profiling, & Instrumentation

- How do you know if your program is correct?
  - o Will it ever core dump?
  - o Does it ever produce the wrong answer?
    - – Testing
- How do you know what your program is doing?
  - o How fast is your program?
  - o Why is it slow for one input but not for another?
  - o Does it have a memory leak?
    - – Timing
    - – Profiling
    - – Instrumentation

See Kernighan & Pike book:
"The Practice of Programming"

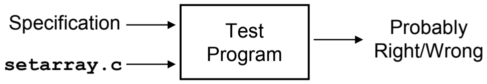
Program Verification

- How do you know if your program is correct?
  - o Can you **prove** that it is correct?
  - o Can you **prove** properties of the code?
    - – e.g., it terminates



Program Testing

- Convince yourself that your program probably works



How do you write a test program?

Test Programs

- Properties of a good test program
  - o Tests boundary conditions
  - o Exercise as much code as possible
  - o Produce output that is known to be right/wrong

How do you achieve all three properties?

Program Testing

- Testing boundary conditions
  - o Almost all bugs occur at boundary conditions
  - o If program works for boundary cases, it probably works for others
- Exercising as much code as possible
  - o For simple programs, can enumerate all paths through code
  - o Otherwise, sample paths through code with random input
  - o Measure test coverage
- Checking whether output is right/wrong?
  - o Match output expected by test programmer (for simple cases)
  - o Match output of another implementation
  - o Verify conservation properties

  - o Note: real programs often have fuzzy specifications

Example Test Program

```
int main(int argc, char *argv[])
{
   Set_T oSet;
   SetIter_T oSetIter;
   const char *pcKey;
   char *pcValue;
   int iLength;

   /* Test Set_new, Set_put, Set_getKey, Set_getValue.  */
   oSet = Set_new(2, myStringCompare);
   Set_put(oSet, "Ruth", "RightField");
   Set_put(oSet, "Gehrig", "FirstBase");
   Set_put(oSet, "Mantle", "CenterField");
   Set_put(oSet, "Jeter", "Shortstop");
   printf("-----------------------------------------------------\n");
   printf("This output should list 4 players and their positions\n");
   printf("-----------------------------------------------------\n");
   pcKey = (const char*)Set_getKey(oSet, "Ruth");
   pcValue = (char*)Set_getValue(oSet, "Ruth");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Gehrig");
   pcValue = (char*)Set_getValue(oSet, "Gehrig");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Mantle");
   pcValue = (char*)Set_getValue(oSet, "Mantle");
   printf("%s\t%s\n", pcKey, pcValue);
   pcKey = (const char*)Set_getKey(oSet, "Jeter");
   pcValue = (char*)Set_getValue(oSet, "Jeter");
   printf("%s\t%s\n", pcKey, pcValue);
```

Systematic Testing

- Incremental testing
  - o Test as write code
  - o Test simple cases first
  - o Test code bottom-up
- Stress testing
  - o Generate test inputs procedurally
  - o Intentionally create error situations for testing

```
void *testmalloc(size_t n)
{
  static int count = 0;
  if (++count > 10) return 0;
  else return malloc(n);
}
```

- Tools!
  - o Test coverage
  - o Regression testing
  - o Automatic testing scripts: run often!

Timing, Profiling, & Instrumentation

- How do you know what your code is doing?
  - o How slow is it?
    - – How long does it take for certain types of inputs?
  - o Where is it slow?
    - – Which code is being executed most?
  - o Why am I running out of memory?
    - – Where is the memory going?
    - – Are there leaks?
  - o Why is it slow?
    - – How imbalanced is my binary tree?

Input
Program
Output

Timing

- Most shells provide tool to time program execution
  - o e.g., bash "**time**" command

```
bash> tail -1000 /usr/lib/dict/words > input.txt

bash> time sort5.pixie < input.txt > output.txt
real    0m12.977s
user    0m12.860s
sys     0m0.010s
```

Timing

- Most operating systems provide a way to get the time
  - o e.g., UNIX "**gettimeofday**" command

```
#include <sys/time.h>

struct timeval start_time, end_time;

gettimeofday(&start_time, NULL);
  <execute some code here>
gettimeofday(&end_time, NULL);

float seconds = end_time.tv_sec - start_time.tv_sec +
     1.0E-6F * (end_time.tv_usec - start_time.tv_usec);
```

Profiling

- Gather statistics about your program’s execution
  - o e.g., how much time did execution of a function take?
  - o e.g., how many times was a particular function called?
  - o e.g., how many times was a particular line of code executed?
  - o e.g., which lines of code used the most time?
- Most compilers come with profilers
  - o e.g., **`pixie`** and **`prof`**

Profiling Example

```
#include <stdio.h>
#include <string.h>
#include "stringarray.h"


int CompareStrings(void *s1, void *s2)
{
  return strcmp(s1, s2);
}


int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, CompareStrings);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

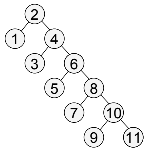
Profiling Example

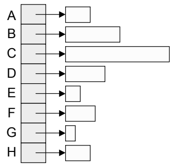
```
bash> cc -o sort5.c etc.
bash> pixie sort5
bash> sort5.pixie < input.txt > output.txt
bash> prof sort5.Counts
```

```
Summary of ideal time data (pixie-counts)--
                  3664181847: Total number of instructions executed
                  3170984513: Total computed cycles
                      16.261: Total computed execution time (secs.)
                       0.865: Average cycles / instruction
-------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
-------------------------------------------------------------------------
excl.secs  excl.%   cum.%      cycles  instructions    calls  function  (dso: file, line)
-------------------------------------------------------------------------
  8.935    54.9%    54.9%  1742355689    1778629217        1  Array_sort (sort5: array.c, 110)
  5.897    36.3%    91.2%  1149885000    1299870000 49995000  CompareStrings (sort5: sort5.c, 7)
  1.386     8.5%    99.7%   270290536     575736340 49995000  strcmp (libc.so.1: strcmp.s, 34)
  0.010     0.1%    99.8%     1879873       2279949    10000  _doprnt (libc.so.1: doprnt.c, 227)
  0.004     0.0%    99.8%      746528        584896    20000  strlen (libc.so.1: strlen.s, 58)
  0.004     0.0%    99.8%      700059        880214    10001  fgets (libc.so.1: fgets.c, 26)
  0.003     0.0%    99.9%      494950        666600    10018  _memccpy (libc.so.1: memccpy.c, 29)
  0.002     0.0%    99.9%      420000        510000    10000  Array_addKth (sort5: array.c, 72)
  0.002     0.0%    99.9%      417401        411003    10000  strcpy (libc.so.1: strcpy.s, 103)
  0.002     0.0%    99.9%      340000        450000    10000  fprintf (libc.so.1: fprintf.c, 23)
  0.002     0.0%    99.9%      310028        250028        1  StringArray_write (sort5: str...c, 22)
  0.001     0.0%    99.9%      267789        296579     2680  resolve_relocations (rld: rld.c, 2636)
  0.001     0.0%    99.9%      264264        345576    10164  cleanfree (libc.so.1: malloc.c, 933)
  0.001     0.0%    99.9%      263196        329639    10038  memcpy (libc.so.1: bcopy.s, 329)
  0.001     0.0%    99.9%      262829        413379    10000  _smalloc (libc.so.1: malloc.c, 127)
```

Instrumentation

- Gather statistics about your data structures
  - o e.g., how many nodes are at each level of my binary tree?
  - o e.g., how many elements are in each bucket of my hash table?
  - o e.g., how much memory is allocated from the heap?

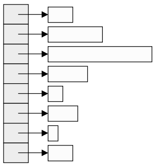




`2,1,4,3,6,5,8,7,10,9,11`

Instrumentation Example

Hash table implemented as array of sets



```
typedef struct Hash *Hash_T;


struct Hash {
  Set_T *buckets;
  int nbuckets;
};


void Hash_PrintBucketCounts(Hash_T oHash, FILE *fp)
{
  int i;

  /* Print number of elements in each bucket */
  for (i = 0; i < oHash->nbuckets; i++)
    fprintf(fp, “%d “, Set_getLength(oHash->buckets[i]), fp);
  fprintf(fp, “\n”);
}
```

Another example: Kalah

- Alpha-beta search programs must go fast (to get more search depth within limited time)
- Do performance tuning of Kalah player for competition
  - o Not necessary for Assignment 3!
- Start with makefile:

$$\texttt{TESTINGFLAGS= }\underbrace{\texttt{-Wall -ansi -pedantic}}_{\text{standard flags for clean code}}\ \underset{\uparrow}{\texttt{-g}}$$

enable symbols for debugger

```
CFLAGS= ${TESTINGFLAGS}
player: player.c minimax.c gamestate.c ...
          gcc ${CFLAGS} player.c minimax.c ...
```

Compiler flags for speed

- Modify compiler settings for performance

```
TESTINGFLAGS= -Wall -ansi -pedantic -g
SPEEDFLAGS= -Wall -ansi -pedantic -O4 -NDEBUG
```

set compiler to highest optimization level

disable assertions

```
CFLAGS= ${SPEEDFLAGS}

test: player testinput
          time player MIN <testinput

player: player.c minimax.c gamestate.c ...
          gcc ${CFLAGS} player.c minimax.c ...
```

Timed execution

```
% make test
gcc -Wall -O4 -DNDEBUG -o player . . .
time player MIN  <testinput

5
     12 11 10  9  8  7
|-------------------------|
|     4  4  4  5  5  5    |
|  0                    1 |
|     4  4  4  4  4  0    |
|-------------------------|
      0  1  2  3  4  5
MIN player reports invalid move by MAX player

real          9.7
user          9.2
sys           0.0

make: *** [test] Error 1
```

wall-clock time

CPU time in user mode

CPU time in operating system

(because testinput stops short of full game)

Profiling with gcc+gprof

- Apparently, **`prof`** doesn't work with **`gcc`**, must use **`gprof`**

```
PROFFLAGS = -Wall -ansi -pedantic -O4 -NDEBUG -pg

CFLAGS= ${PROFFLAGS}

profile: player testinput
          -player MIN <testinput
          gprof player >profile
```

minus sign means "keep going even if errors"

```
player: player.c minimax.c gamestate.c ...
          gcc ${CFLAGS} player.c minimax.c ...
```

Profiled execution

```
% make profile
gcc -Wall -O4 -DNDEBUG -pg -o player . . .
player MIN  <testinput

5
     12 11 10  9  8  7
|-------------------------|
|     4  4  4  5  5  5    |
|  0                    1 |
|     4  4  4  4  4  0    |
|-------------------------|
      0  1  2  3  4  5
MIN player reports invalid move by MAX player

make: *** [profile] Error 1 (ignored)

gprof player >profile
```

Format of gprof profile

```
 called/total      parents
index  %time    self descendents  called+self    name              index
                                  called/total        children

                                                      <spontaneous>
[1]     59.7   12.97        0.00                 internal_mcount [1]
                0.00        0.00       1/3            atexit [35]
-----------------------------------------------
                                                      <spontaneous>
[2]     40.3    0.00        8.75                 _start [2]
                0.00        8.75       1/1            main [3]
                0.00        0.00       2/3            atexit [35]
-----------------------------------------------
                0.00        8.75       1/1            _start [2]
[3]     40.3    0.00        8.75       1          main [3]
                0.00        8.32       1/1            getBestMove [4]
                0.00        0.43       1/1            clock [20]
                0.00        0.00       1/747130       GameState_expandMove [6]
                0.00        0.00       1/1            exit [33]
                0.00        0.00       1/1            Move_read [36]
                0.00        0.00       1/1            GameState_new [37]
                0.00        0.00       6/747135       GameState_getStatus [31]
                0.00        0.00       1/747130       GameState_applyDeltas [25]
                0.00        0.00       1/1            GameState_write [44]
                0.00        0.00       1/1            sscanf [54]
                0.00        0.00       1/1698871      GameState_getPlayer [30]
                0.00        0.00       1/1            _fprintf [58]
                0.00        0.00       1/1            Move_write [59]
                0.00        0.00       3/3            GameState_playerToStr [63]
                0.00        0.00       1/2            strcmp [66]
                0.00        0.00       1/1            GameState_playerFromStr [68]
                0.00        0.00       1/1            Move_isValid [69]
                0.00        0.00       1/1            GameState_getSearchDepth [67]
-----------------------------------------------
                0.00        8.32       1/1            main [3]
[4]     38.3    0.00        8.32       1          getBestMove [4]
                0.27        8.05       6/6            minimax [5]
                0.00        0.00       6/747130       GameState_expandMove [6]
                0.00        0.00      35/4755325      Delta_free [10]
                0.00        0.00       1/204617       GameState_genMoves [17]
                0.00        0.00       5/945027       Move_free [23]
                0.00        0.00       6/747130       GameState_applyDeltas [25]
                0.00        0.00       6/747129       GameState_unApplyDeltas [27]
                0.00        0.00       2/1698871      GameState_getPlayer [30]
-----------------------------------------------
                               747123                 minimax [5]
                0.27        8.05       6/6            getBestMove [4]
[5]     38.3    0.27        8.05       6+747123   minimax [5]
                0.63        3.56  747123/747130       GameState_expandMove [6]
                0.25        1.82 4755290/4755325      Delta_free [10]
                0.07        0.69  204616/204617       GameState_genMoves [17]
                0.02        0.36  945022/945027       Move_free [23]
                0.23        0.00  747123/747130       GameState_applyDeltas [25]
                0.22        0.00  747123/747129       GameState_unApplyDeltas [27]
                0.10        0.00 1698868/1698871      GameState_getPlayer [30]
                0.09        0.00  747129/747135       GameState_getStatus [31]
                0.01        0.00  542509/542509       GameState_getValue [32]
                               747123                 minimax [5]
-----------------------------------------------
                0.00        0.00       1/747130       main [3]
                0.00        0.00       6/747130       getBestMove [4]
                0.63        3.56  747123/747130       minimax [5]
[6]     19.3    0.63        3.56  747130          GameState_expandMove [6]
                0.47        2.99 4755331/5700361      calloc [7]
                0.11        0.00 2360787/2360787      .rem [28]
-----------------------------------------------
                0.00        0.00       1/5700361      Move_read [36]
                0.00        0.00       1/5700361      GameState_new [37]
                0.09        0.59  945028/5700361      GameState_genMoves [17]
                0.47        2.99 4755331/5700361      GameState_expandMove [6]
[7]     19.1    0.56        3.58 5700361          calloc [7]
                0.32        2.09 5700361/5700362      malloc [8]
                0.64        0.00 5700361/5700361      .umul [18]
                0.43        0.00 5700361/5700361      _memset [22]
                0.10        0.00 5700361/5700363      .udiv [29]
-----------------------------------------------
                0.00        0.00       1/5700362      _findbuf [41]
                0.32        2.09 5700361/5700362      calloc [7]
[8]     11.1    0.32        2.09 5700362          malloc [8]
                0.62        0.62 5700362/5700362      _malloc_unlocked <cycle 1> [13]
                0.23        0.20 5700362/11400732     _mutex_unlock [14]
                0.22        0.20 5700362/11400732     mutex_lock [15]
```

First part of gprof profile looks like this; it's for sophisticated users (i.e. more sophisticated than your humble professor) and I will ignore it

Format of gprof profile

```
   %   cumulative    self              self    total
 time    seconds   seconds    calls  ms/call  ms/call name
 57.1      12.97     12.97                            internal_mcount [1]
  4.8      14.05      1.08  5700352     0.00     0.00  _free_unlocked [12]
  4.4      15.04      0.99                            _mcount (693)
  3.5      15.84      0.80 22801464     0.00     0.00  _return_zero [16]
  2.8      16.48      0.64  5700361     0.00     0.00  .umul [18]
  2.8      17.11      0.63   747130     0.00     0.01  GameState_expandMove [6]
  2.5      17.67      0.56  5700361     0.00     0.00  calloc [7]
  2.1      18.14      0.47 11400732     0.00     0.00  _mutex_unlock [14]
  1.9      18.58      0.44 11400732     0.00     0.00  mutex_lock [15]
  1.9      19.01      0.43  5700361     0.00     0.00  _memset [22]
  1.9      19.44      0.43        1   430.00   430.00  .div [21]
  1.8      19.85      0.41  5157853     0.00     0.00  cleanfree [19]
  1.4      20.17      0.32  5700366     0.00     0.00  _malloc_unlocked  [13]
  1.4      20.49      0.32  5700362     0.00     0.00  malloc [8]
  1.3      20.79      0.30  5157847     0.00     0.00  _smalloc         [24]
  1.2      21.06      0.27        6    45.00  1386.66  minimax [5]
  1.1      21.31      0.25  4755325     0.00     0.00  Delta_free [10]
  1.0      21.54      0.23  5700352     0.00     0.00  free [9]
  1.0      21.77      0.23   747130     0.00     0.00  GameState_applyDeltas [25]
  1.0      21.99      0.22  5157845     0.00     0.00  realfree [26]
  1.0      22.21      0.22   747129     0.00     0.00  GameState_unApplyDeltas [27]
  0.5      22.32      0.11  2360787     0.00     0.00  .rem [28]
  0.4      22.42      0.10  5700363     0.00     0.00  .udiv [29]
  0.4      22.52      0.10  1698871     0.00     0.00  GameState_getPlayer [30]
  0.4      22.61      0.09   747135     0.00     0.00  GameState_getStatus [31]
  0.3      22.68      0.07   204617     0.00     0.00  GameState_genMoves [17]
  0.1      22.70      0.02   945027     0.00     0.00  Move_free [23]
  0.0      22.71      0.01   542509     0.00     0.00  GameState_getValue [32]
  0.0      22.71      0.00      104     0.00     0.00  _ferror_unlocked [357]
  0.0      22.71      0.00       64     0.00     0.00  _realbufend [358]
  0.0      22.71      0.00       54     0.00     0.00  nvmatch [60]
  0.0      22.71      0.00       52     0.00     0.00  _doprnt [42]
  0.0      22.71      0.00       51     0.00     0.00  memchr [61]
  0.0      22.71      0.00       51     0.00     0.00  printf [43]
  0.0      22.71      0.00       13     0.00     0.00  _write [359]
  0.0      22.71      0.00       10     0.00     0.00  _xflsbuf [360]
  0.0      22.71      0.00        7     0.00     0.00  _memcpy [361]
  0.0      22.71      0.00        4     0.00     0.00  .mul [62]
  0.0      22.71      0.00        4     0.00     0.00  ___errno [362]
  0.0      22.71      0.00        4     0.00     0.00  _fflush_u [363]
  0.0      22.71      0.00        3     0.00     0.00  GameState_playerToStr [63]
  0.0      22.71      0.00        3     0.00     0.00  _findbuf [41]
```

Second part of profile looks like this; it's the simple (i.e.,useful) part; corresponds to the "prof" tool

Overhead of profiling

```
   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 57.1     12.97    12.97                             internal_mcount
  4.8     14.05     1.08  5700352     0.00     0.00   _free_unlocked
  4.4     15.04     0.99                             _mcount (693)
  3.5     15.84     0.80 22801464     0.00     0.00   _return_zero
  2.8     16.48     0.64  5700361     0.00     0.00   .umul [18]
  2.8     17.11     0.63   747130     0.00     0.01   GameState_expa
  2.5     17.67     0.56  5700361     0.00     0.00   calloc [7]
  2.1     18.14     0.47 11400732     0.00     0.00   _mutex_unlock
  1.9     18.58     0.44 11400732     0.00     0.00   mutex_lock
  1.9     19.01     0.43  5700361     0.00     0.00   _memset [22]
  1.9     19.44     0.43        1   430.00   430.00   .div [21]
  1.8     19.85     0.41  5157853     0.00     0.00   cleanfree [19]
  1.4     20.17     0.32  5700366     0.00     0.00   _malloc_unlo
  1.4     20.49     0.32  5700362     0.00     0.00   malloc [8]
  1.3     20.79     0.30  5157847     0.00     0.00   _smalloc
  1.2     21.06     0.27        6    45.00  1386.66   minimax [5]
  1.1     21.31     0.25  4755325     0.00     0.00   Delta_free [10]
  1.0     21.54     0.23  5700352     0.00     0.00   free [9]
  1.0     21.77     0.23   747130     0.00     0.00   GameState_appl
  1.0     21.99     0.22  5157845     0.00     0.00   realfree [26]
  1.0     22.21     0.22   747129     0.00     0.00   GameState_unAp
  0.5     22.32     0.11  2360787     0.00     0.00   .rem [28]
  0.4     22.42     0.10  5700363     0.00     0.00   .udiv [29]
  0.4     22.52     0.10  1698871     0.00     0.00   GameState_getPl
  0.4     22.61     0.09   747135     0.00     0.00   GameState_getSt
```

Malloc/calloc/free/...

```
   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 57.1     12.97    12.97                             internal_mcount [1]
  4.8     14.05     1.08  5700352     0.00     0.00   _free_unlocked [12]
  4.4     15.04     0.99                             _mcount (693)
  3.5     15.84     0.80 22801464     0.00     0.00   _return_zero [16]
  2.8     16.48     0.64  5700361     0.00     0.00   .umul [18]
  2.8     17.11     0.63   747130     0.00     0.01   GameState_expandMove
  2.5     17.67     0.56  5700361     0.00     0.00   calloc [7]
  2.1     18.14     0.47 11400732     0.00     0.00   _mutex_unlock [14]
  1.9     18.58     0.44 11400732     0.00     0.00   mutex_lock [15]
  1.9     19.01     0.43  5700361     0.00     0.00   _memset [22]
  1.9     19.44     0.43        1   430.00   430.00   .div [21]
  1.8     19.85     0.41  5157853     0.00     0.00   cleanfree [19]
  1.4     20.17     0.32  5700366     0.00     0.00   _malloc_unlocked [13]
  1.4     20.49     0.32  5700362     0.00     0.00   malloc [8]
  1.3     20.79     0.30  5157847     0.00     0.00   _smalloc        [24]
  1.2     21.06     0.27        6    45.00  1386.66   minimax [5]
  1.1     21.31     0.25  4755325     0.00     0.00   Delta_free [10]
  1.0     21.54     0.23  5700352     0.00     0.00   free [9]
  1.0     21.77     0.23   747130     0.00     0.00   GameState_applyDeltas
  1.0     21.99     0.22  5157845     0.00     0.00   realfree [26]
  1.0     22.21     0.22   747129     0.00     0.00   GameState_unApplyDeltas
  0.5     22.32     0.11  2360787     0.00     0.00   .rem [28]
  0.4     22.42     0.10  5700363     0.00     0.00   .udiv [29]
  0.4     22.52     0.10  1698871     0.00     0.00   GameState_getPlayer
  0.4     22.61     0.09   747135     0.00     0.00   GameState_getStatus
  0.3     22.68     0.07   204617     0.00     0.00   GameState_genMoves [17]
```

expandMove

```
   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 57.1     12.97    12.97                             internal_mcount [1]
  4.8     14.05     1.08  5700352     0.00     0.00   _free_unlocked [12]
  4.4     15.04     0.99                             _mcount (693)
  3.5     15.84     0.80 22801464     0.00     0.00   _return_zero [16]
  2.8     16.48     0.64  5700361     0.00     0.00   .umul [18]
  2.8     17.11     0.63   747130     0.00     0.01   GameState_expandMove
  2.5     17.67     0.56  5700361     0.00     0.00   calloc [7]
  2.1     18.14     0.47 11400732     0.00     0.00   _mutex_unlock [14]
  1.9     18.58     0.44 11400732     0.00     0.00   mutex_lock [15]
  1.9     19.01     0.43  5700361     0.00     0.00   _memset [22]
  1.9     19.44     0.43        1   430.00   430.00   .div [21]
  1.8     19.85     0.41  5157853     0.00     0.00   cleanfree [19]
  1.4     20.17     0.32  5700366     0.00     0.00   _malloc_unlocked [13]
  1.4     20.49     0.32  5700362     0.00     0.00   malloc [8]
  1.3     20.79     0.30  5157847     0.00     0.00   _smalloc        [24]
  1.2     21.06     0.27        6    45.00  1386.66   minimax [5]
  1.1     21.31     0.25  4755325     0.00     0.00   Delta_free [10]
  1.0     21.54     0.23  5700352     0.00     0.00   free [9]
  1.0     21.77     0.23   747130     0.00     0.00   GameState_applyDeltas
  1.0     21.99     0.22  5157845     0.00     0.00   realfree [26]
```

- Solution: use the "Hints on move expansion" from the minimax-search lecture; resulting "expandMove" is much faster

Don't even *think* of optimizing these

```
   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 57.1     12.97    12.97                             internal_mcount [1]
  4.8     14.05     1.08  5700352     0.00     0.00   _free_unlocked [12]
  4.4     15.04     0.99                             _mcount (693)
  3.5     15.84     0.80 22801464     0.00     0.00   _return_zero [16]
  2.8     16.48     0.64  5700361     0.00     0.00   .umul [18]
  2.8     17.11     0.63   747130     0.00     0.01   GameState_expandMove [6]
  2.5     17.67     0.56  5700361     0.00     0.00   calloc [7]
  2.1     18.14     0.47 11400732     0.00     0.00   _mutex_unlock [14]
  1.9     18.58     0.44 11400732     0.00     0.00   mutex_lock [15]
  1.9     19.01     0.43  5700361     0.00     0.00   _memset [22]
  1.9     19.44     0.43        1   430.00   430.00   .div [21]
  1.8     19.85     0.41  5157853     0.00     0.00   cleanfree [19]
  1.4     20.17     0.32  5700366     0.00     0.00   _malloc_unlocked  <cycle 1> [13]
  1.4     20.49     0.32  5700362     0.00     0.00   malloc [8]
  1.3     20.79     0.30  5157847     0.00     0.00   _smalloc          <cycle 1> [24]
  1.2     21.06     0.27        6    45.00  1386.66   minimax [5]
  1.1     21.31     0.25  4755325     0.00     0.00   Delta_free [10]
  1.0     21.54     0.23  5700352     0.00     0.00   free [9]
  1.0     21.77     0.23   747130     0.00     0.00   GameState_applyDeltas [25]
  1.0     21.99     0.22  5157845     0.00     0.00   realfree [26]
  1.0     22.21     0.22   747129     0.00     0.00   GameState_unApplyDeltas [27]
  0.5     22.32     0.11  2360787     0.00     0.00   .rem [28]
  0.4     22.42     0.10  5700363     0.00     0.00   .udiv [29]
  0.4     22.52     0.10  1698871     0.00     0.00   GameState_getPlayer [30]
  0.4     22.61     0.09   747135     0.00     0.00   GameState_getStatus [31]
  0.3     22.68     0.07   204617     0.00     0.00   GameState_genMoves [17]
  0.1     22.70     0.02   945027     0.00     0.00   Move_free [23]
  0.0     22.71     0.01   542509     0.00     0.00   GameState_getValue [32]
  0.0     22.71     0.00      104     0.00     0.00   _ferror_unlocked [357]
  0.0     22.71     0.00        4     0.00     0.00   _thr_main [367]
  0.0     22.71     0.00        3     0.00     0.00   GameState_playerToStr [63]
  0.0     22.71     0.00        2     0.00     0.00   strcmp [66]
  0.0     22.71     0.00        1     0.00     0.00   GameState_getSearchDepth [67]
  0.0     22.71     0.00        1     0.00     0.00   GameState_new [37]
  0.0     22.71     0.00        1     0.00     0.00   GameState_playerFromStr [68]
  0.0     22.71     0.00        1     0.00     0.00   GameState_write [44]
  0.0     22.71     0.00        1     0.00     0.00   Move_isValid [69]
  0.0     22.71     0.00        1     0.00     0.00   Move_read [36]
  0.0     22.71     0.00        1     0.00     0.00   Move_write [59]
  0.0     22.71     0.00        1     0.00     0.00   check_nlspath_env [46]
  0.0     22.71     0.00        1     0.00   430.00   clock [20]
  0.0     22.71     0.00        1     0.00     0.00   exit [33]
  0.0     22.71     0.00        1     0.00  8319.99   getBestMove [4]
  0.0     22.71     0.00        1     0.00     0.00   getenv [47]
  0.0     22.71     0.00        1     0.00  8750.00   main [3]
  0.0     22.71     0.00        1     0.00     0.00   mem_init [70]
  0.0     22.71     0.00        1     0.00     0.00   number [71]
  0.0     22.71     0.00        1     0.00     0.00   scanf [53]
```

Interate...

1. Develop program
2. Test; modify program
3. Test again; if bugs, back to step 2
4. Is it fast enough? If not,
5. Profile; modify program; back to step 3

- Typically, reprofile several times until no more performance improvement is justified

Summary & Guidelines

- Test your code as you write it
  - o It is very hard to debug a lot of code all at once
  - o Isolate modules and test them independently
  - o Design your tests to cover boundary conditions
  - o Test modules bottom-up
- Instrument your code as you write it
  - o Include asserts and verify data structure sanity often
  - o Include debugging statements (e.g., #ifdef DEBUG and #endif)
  - o You'll be surprised what your program is really doing!!!
- Time and profile your code **only** when you are done
  - o Don't optimize code unless you have to (you almost never will)
  - o Fixing your algorithm is almost always the solution
  - o Otherwise, running optimizing compiler is usually enough

Robust Programming

CS 217

Program Errors

- Programs encounter errors
  - o Good programmers handle them gracefully
- Types of errors
  - o Compile-time errors
  - o Run-time user errors
  - o Run-time program errors
  - o Run-time exceptions

Compile-Time Errors

- Code does not conform to C specification
  - o Forgetting a semicolon
  - o Forgetting to declare a variable
  - o etc.
- Detected by compiler

```
int a = 0;
int b = 3
int c = 6;

a = b + 3;
d = c + 3;
```

```
cc-1065 cc: ERROR File = foo.c, Line = 2
  A semicolon is expected at this point.

    int c = 6;
    ^

cc-1020 cc: ERROR File = foo.c, Line = 6
  The identifier "d" is undefined.

    d = c + 3;
    ^
```

Link-Time Errors

- Error in linking together the .o files to make an a.out
  - o Symbol referenced (used) in one module, not defined in another
  - o etc.
- Detected by linker
  - o But linker is usally called upon by C compiler

```
extern int not_there;
.
.
.
main() {
 printf("%d",
   not_there);
}
```

```
Undefined       first referenced
 symbol              in file
not_there            foo.o
ld: fatal: Symbol referencing errors.
No output written to a.out
```

Run-Time User Errors

- User provides invalid input
  - o User types in name of file that does not exist
  - o User provides program argument with value outside legal bounds
  - o etc.
- Detected with “if” checks in program
  - o Program should print message and recover gracefully
  - o Possibly ask user for new input
- Your program should anticipate and handle EVERY possible user input!!!

```
int ReadFile(const char *filename)
{
  FILE *fp = fopen(filename, "r");
  if (!fp) {
    fprintf(stderr, "Unable to open file: %s\n", filename);
    return 0;
  }
...
```

Run-Time Program Errors

- Internal error from which recovery is impossible (bug)
  - o Null pointer passed to **`Array_removeLast()`**
  - o Invalid value for array index (k = -7)
  - o Invariant is violated
  - o etc.

?

- Detected with conditional checks in program (assert)
  - o Program should print message and abort

```
#include <assert.h>

void Array_removeLast(Array_T oArray)
{
  assert(oArray);
  oArray->nelements--;
}
```

Exceptions

- Rare error from which recovery may be possible
  - o User hits interrupt key
  - o Arithmetic overflow
  - o etc.
- Detected by machine or operating system
  - o Program can handle them with signal handlers (later)
  - o Not usually possible/practical to detect with conditional checks

```
#include <limits.h>
...
int a = MAX_INT;
int b = MAX_INT;
int c = 6;
int d = 0;
...
a = a + d;
d = a + b;
b = a - c;
...
```

Exceptions (cont.)

- Rare error from which recovery may be possible
  - o User hits interrupt key
  - o Arithmetic overflow
  - o Array access out of bounds
  - o Function argument not in expected domain
- Detected by machine or operating system
- Detected by compiler or explicit program code
  - o Requires programming-language support to work well
  - o C language doesn't have such support
  - o Java, C++, ML, other languages do

Robust Programming

- Your program should never terminate without either ...
  - o Completing successfully, or
  - o Outputting a meaningful error message
- How can a program terminate?
  - o Return from main
  - o Call exit
  - o Call abort

Robust Programming

- Your program should never terminate without either ...
  - o Completing successfully, or
  - o Outputting a meaningful error message
- How can a program terminate?
  - > **Return from main**
  - o Call exit
  - o Call abort

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, strcmp);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

Robust Programming

- Your program should never terminate without either ...
  - o Completing successfully, or
  - o Outputting a meaningful error message
- How can a program terminate?
  - o Return from main
  - > **Call exit**
  - o Call abort

```
...
#include <stdlib.h>

void ParseArguments(int argc, char **argv)
{
  argc--; argv++;

  while (argc > 0) {
    if (!strcmp(*argv, "-filename")) {
       ...
    }
    else if (!strcmp(*argv, "-help")) {
      PrintUsage();
      exit(0);
    }
    else {
      fprintf(stderr, "Unrecognized argument: %s\n", *argv);
      PrintUsage();
      exit(1);
    }
    argv++; argc--;
  }
}
```

Robust Programming

- Your program should never terminate without either ...
  - o Completing successfully, or
  - o Outputting a meaningful error message
- How can a program terminate?
  - o Return from main
  - o Call exit
  - > **Call abort**

```
...
#include <stdlib.h>

void *Array_getKth(Array_T oArray, int k)
{
  if (!oArray) {
    fprintf(stderr, “oArray=NULL in Array_getKth\n”);
    abort();
  }

  if ((k < 0) || (k >= oArray->nelements)) {
    fprintf(stderr, “k=%d in Array_getKth\n”, k);
    abort();
  }

  return oArray->elements[k];
}
```

Error returns from functions

- Functions or modules can detect errors
  - o If a ***bug*** is detected, it's reasonable to terminate the program (with assert, for example)
  - o If an "exceptional condition" is detected, provide feedback to the caller
    - – Use exception-handling construct of programming language (oops, C doesn't have this)
    - – Use special return value

```
FILE *f;
f = fopen(filename, "r");
if (f==NULL) {
   handle the error
} else {
  do the normal thing
}
```

Error returns from functions

- If normal return values cover entire range of data type, need an extra return value!

```
#include <ctype.h>
int  string2int(char *s) {
   int n=0, sign=1;
   if (*s=='-') {sign= -1; s++;}
   if (!isdigit(*s)) ERROR;
   for (; *s; s++) {
      if (isdigit(*s)) n= n*10+s-'0';
      else ERROR;
   }
   return sign*n;
}
```

Error returns from functions

- Use a call-by-reference parameter.

```
#include <ctype.h>
int  string2int(char *s, int *result) {
/* Converts from decimal; puts converted integer value into *result;
  returns 1 for success, else 0 */
   int n=0, sign=1;
   if (*s=='-') {sign= -1; s++;}
   if (!isdigit(*s)) return 0;
   for (; *s; s++) {
      if (isdigit(*s)) n= n*10+s-'0';
      else return 0;
   }
   *result = sign*n;
   return 1;
}
```

Assert

- **`void assert(int expression)`**
  - o Issues a message and aborts the program if ***expression*** is 0
  - o Activated conditionally
    - – While debugging: **`gcc foo.c`**
    - – After release: **`gcc -DNDEBUG foo.c`**
- Typical uses
  - o Check function arguments
  - o Check invariants!!!

`assert.h`

```
#ifdef NDEBUG
#define assert(_e) 0
#else
#define assert(_e) \
  if (_e) { \
    fprintf(stderr, "Assertion failed on line %d of file %s\n", __LINE__, __FILE__); \
    abort(); \
  }
  0
#endif
```

Assert

- **`void assert(int expression)`**
  - o Issues a message and aborts the program if ***expression*** is 0
  - o Activated conditionally
    - – While debugging: **`gcc foo.c`**
    - – After release: **`gcc -DNDEBUG foo.c`**
- Typical uses
  - **> Check function arguments**
  - o Check invariants!!!

```
#include <assert.h>

void *Array_getKth(Array_T oArray, int k)
{
  assert(oArray);
  assert((k >= 0) && (k < oArray->nelements));

  return oArray->elements[k];
}
```

Assert

- **`void assert(int expression)`**
  - o Issues a message and aborts the program if ***expression*** is 0
  - o Activated conditionally
    - – While debugging: **`cc foo.c`**
    - – After release: **`cc -DNDEBUG foo.c`**
- Typical uses
  - o Check function arguments
  - **> Check invariants!!!**

```
#include <assert.h>

void Array_removeKth(Array_T oArray, int k)
{
  int i;

  assert(oArray);
  assert((k >= 0) && (k < oArray->nelements));

  for (i = k+1; i < oArray->nelements; i++)
    oArray->elements[i-1] = oArray->elements[i];

  oArray->nelements--;

  assert(oArray->nelements >= 0);
}
```

What assert is not best for

~~FILE *f;~~

~~f = fopen(filename, "r");~~

~~assert(f);~~

- Assert is meant for *bugs*, conditions that "can't" occur (or if they do, it's the programmer's fault)
- File-not-present happens all the time, *beyond the control of the programmer*
- Instead of an assert, print a nice error message to the user, then exit or retry
- However: **assert** here is much better than not handling the error at all!

N.B. If you've seen assert used this way on lecture slides, then it's a case of "do what I say, not what I do."

Defensive programming

- In real life, many people contribute to software project
- Other people write modules that interact with yours
- Other people will read and modify your program
- Rule of thumb: don't let the other guy's bug crash your module
  - o Generous use of assert
  - o Document preconditions and postconditions
  - o etc.

C Preprocessor

- Invoked automatically by the C compiler
  - o try **gcc –E foo.c**
- C preprocessor manipulates text prior to C compiling
  - o file inclusion
  - o conditional compilation
  - o macros

File Inclusion

- Header files contain declarations for modules
  - o Names of header files should end in **.h**
- User-define header files " ... "

```
#include "mydefs.h"
```

- System header files: **< ... >**

```
#include <stdio.h>
```

Conditional Compilation

- Removing macro definitions

```
#undef plusone
```

- Conditional compilation

```
#ifdef name
#ifndef name
#if expr
#elif expr
#else
#endif
```

- Why use?

```
#ifndef FOO_H
#define FOO_H

#ifdef WINDOWS_OS
#include <windows.h>
#endif

.
.
.
#endif
```

```
gcc -DWINDOWS_OS foo.c
```

Macros

- Provide parameterized text substitution
- Macro definition

```
#define MAXLINE 120
#define lower(c) ((c)-`A'+'a')
```

- Macro replacement

```
char buf[MAXLINE+1];
```

becomes

```
char buf[120+1];
```

```
c = lower(buf[i]);
```

becomes

```
c = ((buf[i])-`A'+'a');
```

Macros (cont)

- Always parenthesize macro parameters in definition

```
#define plusone(x) x+1

i = 3*plusone(2);
  becomes
i = 3*2+1
```

```
#define plusone(x) ((x)+1)

i = 3*plusone(2);
  becomes
i = 3*((2)+1)
```

Macros (cont)

- Always avoid side-effects in parameters passed to macros

```
#define max(a, b) ((a)>(b)?(a):(b))

y = max(i++, j++)
  becomes
y = ((i++)>(j++)?(i++):(j++));
```

Summary

- Programs encounter errors
  - o Good programmers handle them gracefully
- Types of errors
  - o Compile-time errors
  - o Run-time user errors
  - o Run-time program errors
  - o Run-time exceptions
- Robust programming
  - o Complete successfully, or
  - o Output a meaningful error message

Different execution times

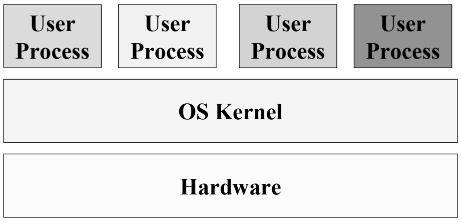
1. Preprocessing time
2. Compile time
3. Run time

Operating Systems

CS 217

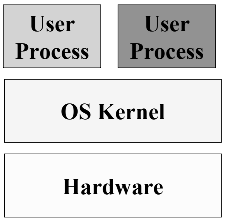
Operating System (OS)

- Provides each process with a virtual machine
  - o Promises each program the illusion of having whole machine to itself



Operating System

- Coordinates access to physical resources
  - o CPU, memory, disk, i/o devices, etc.
- Provides services
  - o Protection
  - o Scheduling
  - o Memory management
  - o File systems
  - o Synchronization
  - o etc.



OS as Government

Randy Wang

- Makes lives easy
  - o Promises everyone whole machine (dedicated CPU, infinite memory, …)
  - o Provides standardized services (standard libraries, window systems, …)
- Makes lives fair
  - o Arbitrates competing resource demands
- Makes lives safe
  - o Prevent accidental or malicious damage by one program to another

OS History

Randy Wang

- Development of OS paradigms:
  - o Phase 0: User at console
  - o Phase 1: Batch processing
  - o Phase 2: Interactive time-sharing
  - o Phase 3: Personal computing
  - o Phase 4: ?

	1981	1999	Factor
MIPS	1	1000	1,000
$/MIPS	$100K	$5	20,000
DRAM Capacity	128KB	256MB	2,000
Disk Capacity	10MB	50GB	5,000
Network B/W	9600b/s	155Mb/s	15,000
Address Bits	16	64	4
Users/Machine	10s	<= 1	< 0.1

Computing price/performance affects OS paradigm

Phase 0: User at Console

Randy Wang

- How things work
  - o One program running at a time
  - o No OS, just a sign-up sheet for reservations
  - o Each user has complete control of machine
- Advantages
  - o Interactive!
  - o No one can hurt anyone else
- Disadvantages
  - o Reservations not accurate, leads to inefficiency
  - o Loading/ unloading tapes and cards takes forever and leaves the machine idle

Phase 1: Batch Processing

Randy Wang

- How things work
  - o Sort jobs and batch those with similar needs to reduce unnecessary setup time
  - o Resident monitor provides "automatic job sequencing": it interprets "control cards" to automatically run a bunch of programs without human intervention
- Advantage
  - o Good utilization of machine
- Disadvantagess
  - o Loss of interactivity (unsolvable)
  - o One job can screw up other jobs, need protection (solvable)

Good for expensive hardware and cheap humans

Phase 2: Interactive Time-Sharing

Randy Wang

- How things work
  - o Multiple users per single machine
  - o OS with multiprogramming and memory protection
- Advantages:
  - o Interactivity
  - o Sharing of resources
- Disadvantages:
  - o Does not always provide reasonable response time

Good for cheap hardware and expensive humans

Phase 3: Personal Computing

Randy Wang

- How things work
  - o One machine per person
  - o OS with multiprogramming and memory protection
- Advantages:
  - o Interactivity
  - o Good response times
- Disadvantages:
  - o Sharing is harder

Good for very cheap hardware and expensive humans

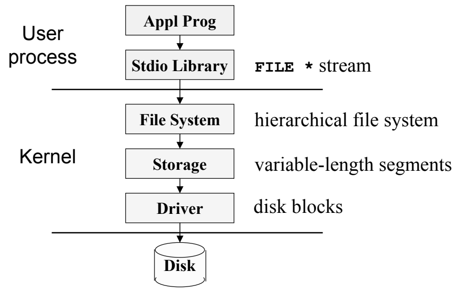
Phase 4: What Next?

- How will things work?
  - o Many machines per person?
  - o Ubiquitous computing?
- What type of OS?

Randy
Wang

Good for
very, very cheap hardware
and expensive humans

Layers of Abstraction



User process

Appl Prog

Stdio Library — `FILE *` stream

Kernel

File System — hierarchical file system

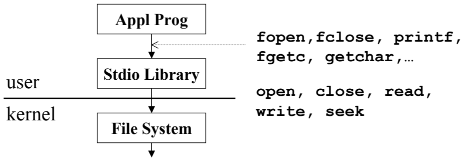
Storage — variable-length segments

Driver — disk blocks

Disk

System Calls

- Method by which user processes invoke kernel services: “protected” procedure call



Appl Prog

Stdio Library

user

kernel

File System

fopen,fclose, printf, fgetc, getchar,…

open, close, read, write, seek

- Unix has ~150 system calls; see
  - o man 2 intro
  - o /usr/include/syscall.h

System Calls

- Processor modes
  - o user mode: can execute normal instructions and access only user memory
  - o supervisor mode: can also execute privileged instructions and access all of memory (e.g., devices)
- System calls
  - o user cannot execute privileged instructions
  - o users must ask OS to execute them - system calls
  - o system calls are often implemented using traps
  - o OS gains control through trap, switches to supervisor model, performs service, switches back to user mode, and gives control back to user

System-call interface = ADTs

ADT
operations

- File input/output
  - o open, close, read, write, dup
- Process control
  - o fork, exit, wait, kill, exec, ...
- Interprocess communication
  - o pipe, socket ...

`open` system call

NAME

`open` - open and possibly create a file or device

SYNOPSIS

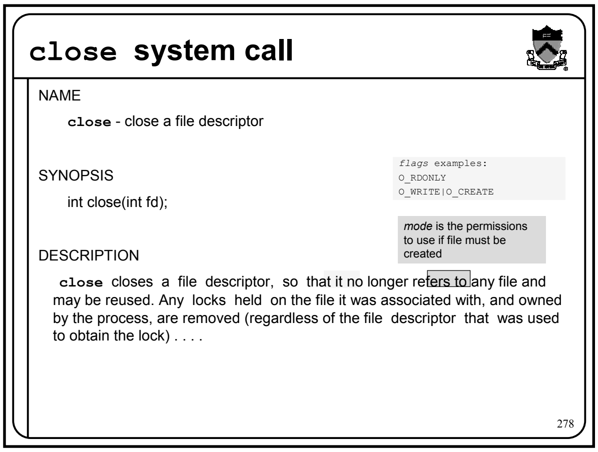
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, mode_t mode);
```

flags examples:
O_RDONLY
O_WRITE|O_CREATE

mode is the permissions to use if file must be created

DESCRIPTION

The open() system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be . . .



close system call

NAME

`close` - close a file descriptor

SYNOPSIS

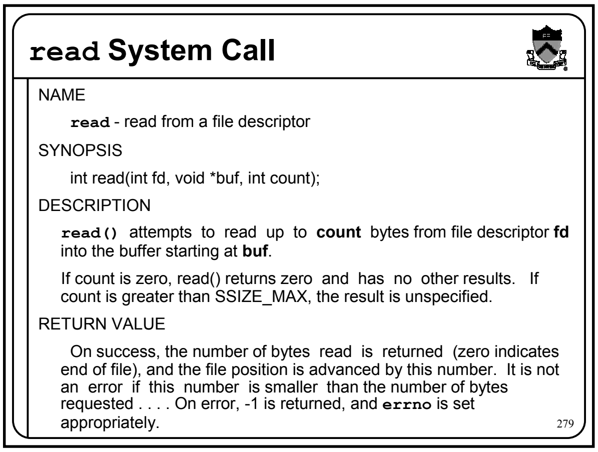
int close(int fd);

flags examples:
O_RDONLY
O_WRITE|O_CREATE

mode is the permissions to use if file must be created

DESCRIPTION

`close` closes a file descriptor, so that it no longer refers to any file and may be reused. Any locks held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock) . . . .



read System Call

NAME

`read` - read from a file descriptor

SYNOPSIS

int read(int fd, void *buf, int count);

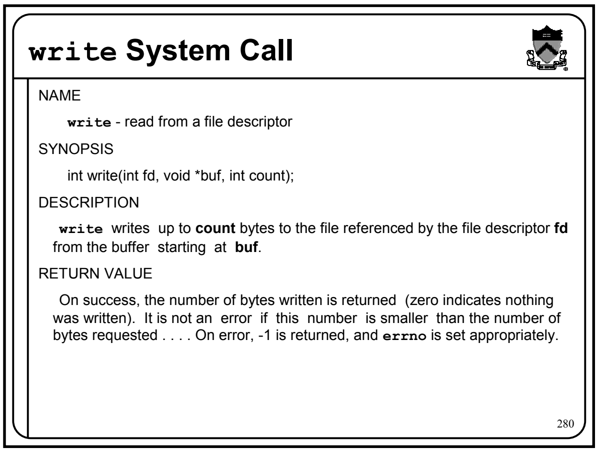
DESCRIPTION

`read()` attempts to read up to **count** bytes from file descriptor **fd** into the buffer starting at **buf**.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested . . . . On error, -1 is returned, and `errno` is set appropriately.



write System Call

NAME

`write` - read from a file descriptor

SYNOPSIS

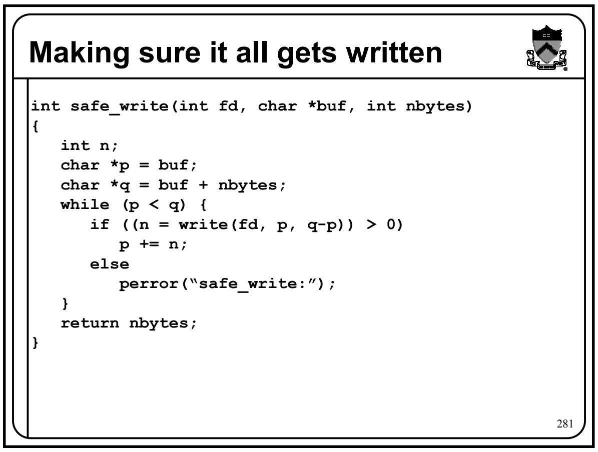
int write(int fd, void *buf, int count);

DESCRIPTION

`write` writes up to **count** bytes to the file referenced by the file descriptor **fd** from the buffer starting at **buf**.

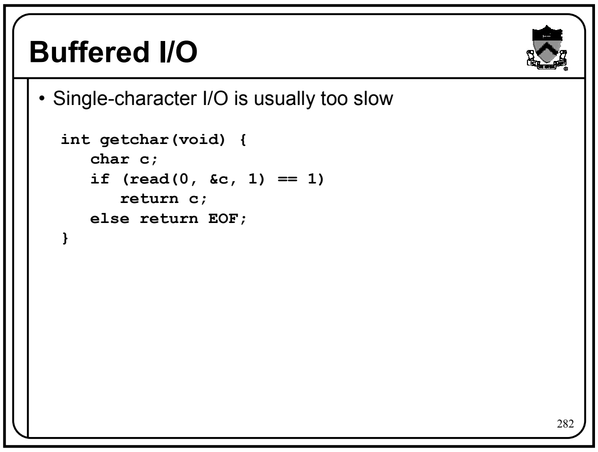
RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested . . . . On error, -1 is returned, and `errno` is set appropriately.



Making sure it all gets written

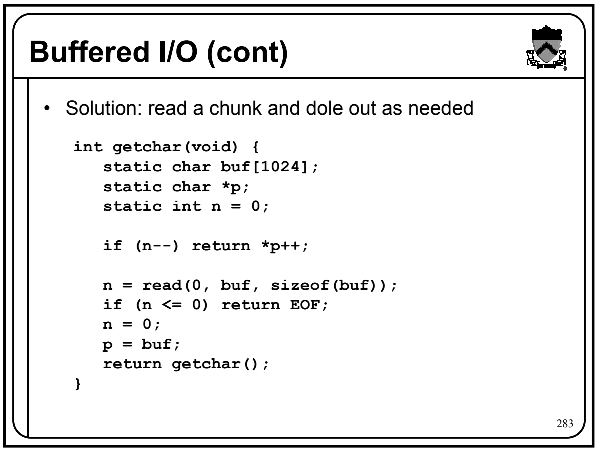
```
int safe_write(int fd, char *buf, int nbytes)
{
   int n;
   char *p = buf;
   char *q = buf + nbytes;
   while (p < q) {
      if ((n = write(fd, p, q-p)) > 0)
         p += n;
      else
         perror(“safe_write:”);
   }
   return nbytes;
}
```



Buffered I/O

- Single-character I/O is usually too slow

```
int getchar(void) {
   char c;
   if (read(0, &c, 1) == 1)
      return c;
   else return EOF;
}
```



Buffered I/O (cont)

- Solution: read a chunk and dole out as needed

```
int getchar(void) {
   static char buf[1024];
   static char *p;
   static int n = 0;

   if (n--) return *p++;

   n = read(0, buf, sizeof(buf));
   if (n <= 0) return EOF;
   n = 0;
   p = buf;
   return getchar();
}
```

Standard I/O Library

```
#define getc(p) (--(p)->_cnt >= 0 ? \
   (int)(*(unsigned char *)(p)->_ptr++) : \
   _filbuf(p))

typedef struct _iobuf {
   int _cnt;    /* num chars left in buffer */
   char *_ptr;  /* ptr to next char in buffer */
   char *_base; /* beginning of buffer */
   int _bufsize;/* size of buffer */
   short _flag; /* open mode flags, etc. */
   char _file;  /* associated file descriptor */
} FILE;

extern FILE *stdin, *stdout, *stderr;
```

Why is getc a macro?

```
#define getc(p) (--(p)->_cnt >= 0 ? \
   (int)(*(unsigned char *)(p)->_ptr++) : \
   _filbuf(p))

#define getchar() getc(stdin)
```

- Invented in ~1975, when
  - o Computers had slow function-call instructions
  - o Compilers couldn't inline-expand very well
- It's not 1975 any more
  - o Moral: don't invent new macros, use functions

fopen

```
FILE *fopen(char *name, char *rw) {
```

Use malloc to create a struct _iobuf

Determine appropriate "flags" from "rw" parameter

Call open to get the file descriptor

Fill in the _iobuf appropriately

```
}
```

Stdio library

- fopen, fclose
- feof, ferror, fileno, fstat
  - o status inquiries
- fflush
  - o make outside world see changes to buffer
- fgetc, fgets, fread
- fputc fputs, fwrite
- printf, fprintf
- scanf, fscanf
- fseek
- *and more ...*

This (large) library interface is not the operating-system interface; much more room for flexibility.

This ADT is implemented in terms of the lower-level "file-descriptor" ADT.

Summary

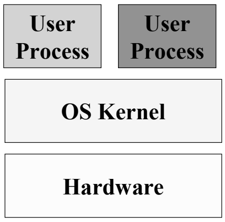
- OS virtualizes machine
  - o Provides each process with illusion of having whole machine to itself
- OS provides services
  - o Protection
  - o Sharing of resources
  - o Memory management
  - o File systems
- Protection achieved through separate kernel
  - o User processes uses system calls to ask kernel to access protected stuff on its behalf
- User level libraries layered on top of kernel interface

Processes

CS 217

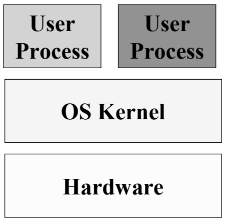
Operating System

- Supports virtual machines
  - o Promises each process the illusion of having whole machine to itself
- Provides services:
  - o Protection
  - o Scheduling
  - o Memory management
  - o File systems
  - o Synchronization
  - o etc.



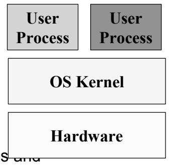
What is a Process?

- A process is a running program with its own …
  - o Processor state
    - – PC, PSR, registers
  - o Address space (memory)
    - – Text, bss, data, heap, stack



Operating System

- Resource allocation
  - o Sharing
  - o Protection
  - o Fairness
  - o Higher-level abstractions
- Common strategies
  - o Chop up resources into small pieces and allocate small pieces at fine-grain level
  - o Introduce level of indirection and provide mapping from virtual resources to physical ones
  - o Use past history to predict future behavior

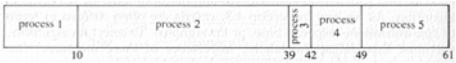


Example: Process Scheduling

- We have a single physical CPU and a whole lot of processes/jobs to run
  - o Which process do we run next?
  - o For how long do we run it?
- Solution 1:
  - o Run each of them to completion in first-come first-served order

Process	Time
1	10
2	29
3	3
4	7
5	12

CPU-bound processes



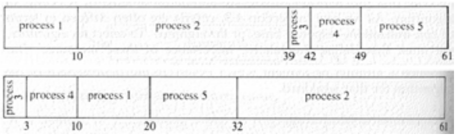
Average wait time of processes: (0+ 10+ 39+ 42+ 49)/ 5 = 28

Example: Process Scheduling

- We have a single physical CPU and a whole lot of processes/jobs to run
  - o Which process do we run next?
  - o For how long do we run it?
- Another solution:
  - o Run them to completion in shortest-first order

Process	Time
1	10
2	29
3	3
4	7
5	12

CPU-bound processes



Average wait time of processes: (0+ 3+ 10+ 20+ 32)/ 5 = 13

CPU-I/O Burst Cycle

- Typical execution of process:

```
load
add
store
read from file

  wait for I/O

store
increment index
write to file

  wait for I/O

load
add
store
read from file

  wait for I/O

...
```

CPU Burst

Wait for I/O

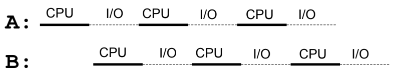
CPU Burst

Wait for I/O

Most processes are not CPU-bound

CPU-I/O Burst Cycle

- Schedule CPU burst for process B while process A is waiting for I/O
  - o Better utilize CPU



Time Slicing

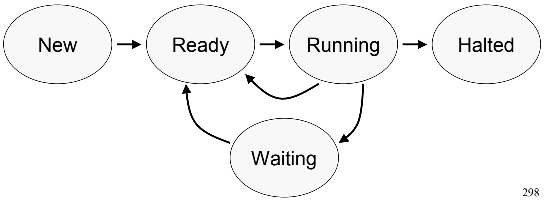
- Divide up time into quantums
  - o Schedule quantums, not complete jobs
  - o Schedule another process if perform I/O
  - o Preempt process at end of quantum
- Motivations
  - o CPU-I/O Burst Cycle
  - o Interactive response

Process	Time
1	10
2	29
3	3
4	7
5	12

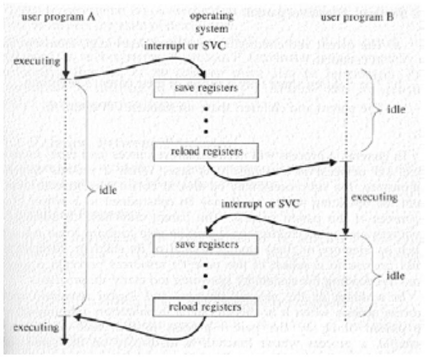
Time Slicing	1	2	3	4	5	1	2	3	4	5	1	2	4	5	1	2	4	5	1	2	5	2	5	2	2	2	2	2	2	2
								15									32		36				44							61

Life Cycle of a Process

- Running: instructions are being executed
- Waiting: waiting for some event (e.g., i/o finish)
- Ready: ready to be assigned to a processor



Context Switch



Process Control Block

- For each process, the kernel keeps track of ...
  - o Process state (new, ready, waiting, halted)
  - o CPU registers (PC, PSR, global, local, ...)
  - o CPU scheduling information (priority, queues, ...)
  - o Memory management information (page tables, ...)
  - o Accounting information (time limits, group ID, ...)
  - o I/O status information (open files, I/O requests, ...)

Fork

- Create a new process (system call)
  - o child process inherits state from parent process
  - o parent and child have separate copies of that state
  - o parent and child share access to any open files

```
pid = fork();
if (pid != 0) {
   /* in parent */
   ...
}
/* in child */
...
```

Fork

- Inherited:
  - o user and group IDs
  - o environment
  - o close-on-exec flag
  - o signal handling settings
  - o supplementary group IDs
  - o set-user-ID mode bit
  - o set-group-ID mode bit
  - o profiling on/off/mode status
  - o debugger tracing status
  - o nice value
  - o stdin
  - o scheduler class
  - o all shared memory segments
  - o all mapped files
  - o file pointers
  - o non-degrading priority
  - o process group ID
  - o session ID
  - o current working directory
  - o root directory
  - o file mode creation mask
  - o resource limits
  - o controlling terminal
  - o all machine register states
  - o control register(s)
- Separate in child
  - o process ID
  - o address space (memory)
  - o file descriptors
  - o active process group ID.
  - o parent process ID
  - o process locks, file locks, page locks, text locks and data locks
  - o pending signals
  - o timer signal reset times
  - o share mask

Exec

- Overlay current process image with a specified image file (system call)
  - o affects process memory and registers
  - o has no affect on file table
- Example:

```
execlp(“ls”, “ls”, “-l”, NULL);
fprintf(stderr, “exec failed\n”);
exit(1);
```

Exec (cont)

- Many variations of **exec**

```
int execlp(const char *file,
           const char *arg, ...)
int execl(const char *path,
           const char *arg, ...)
int execv(const char *path,
           char * const argv[])
int execle(const char *path,
           const char *arg, ...,
           char * const envp[])
```

- Also **execve** and **execvp**

Fork/Exec

- Commonly used together by the shell

```
... parse command line ...
pid = fork()
if (pid == -1)
   fprintf(stderr, “fork failed\n”);
else if (pid == 0) {
   /* in child */
   execvp(file, argv);
   fprintf(stderr, “exec failed\n”);
}
else {
  /* in parent */
  pid = wait(&status);
}
... return to top of loop ...
```

Wait

- Parent waits for a child (system call)
  - o blocks until status of a child changes
  - o returns pid of the child process
  - o returns –1 if no children exist (already exited)

```
pid_t wait(int *status);
```

System

- Convenient way to invoke fork/exec/wait
  - o Forks new process
  - o Execs command
  - o Waits until it is complete

```
int system(const char *cmd);
```

- Example:

```
int main()
{
    system(“echo Hello world”);
}
```

Summary

- Operating systems manage resources
  - o Divide up resources (e.g., quantum time slides)
  - o Allocate them (e.g., process scheduling)
- A processes is a running program with its own …
  - o Processor state
  - o Address space (memory)
- Create and manage processes with ...
  - o **fork**
  - o **exec**
  - o **wait**
  - o **system**

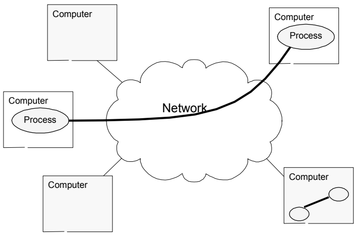
} Used in shell

Interprocess Communication

CS 217

Networks

- Mechanism by which two processes exchange information and coordinate activities

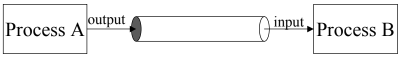


Interprocess Communication

- Sockets
  - o Processes can be on any machine
  - o Processes can be created independently
  - o Used for clients/servers, distributed systems, etc.
- Pipes
  - o Processes must be on same machine
  - o One process spawns the other
  - o Used mostly for filters

Pipes

- Provides an interprocess communication channel



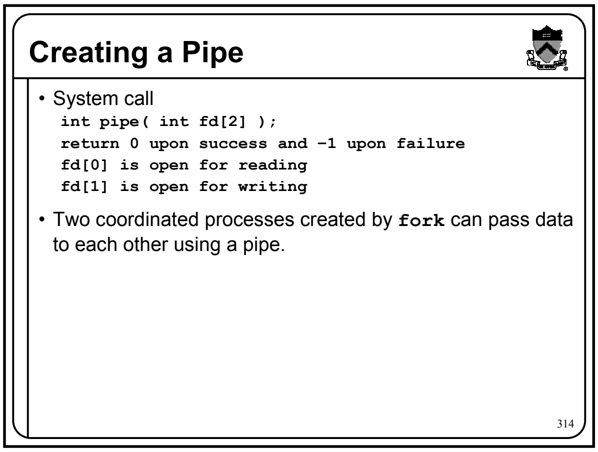
- A filter is a process that reads from `stdin` and writes to `stdout`

stdin
Filter
stdout

Pipes (cont)

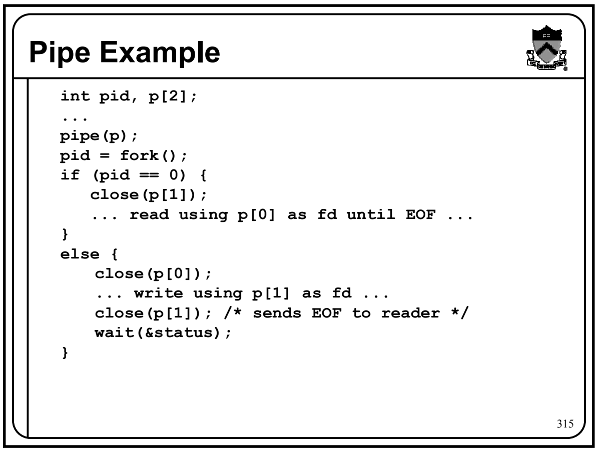
- Many Unix tools are written as filters
  - o **grep, sort, sed, cat, wc, awk ...**
- Shells support pipes

```
ls -l | more
who | grep mary | wc
ls *.[ch] | sort
cat < foo | grep bar | sort > save
```



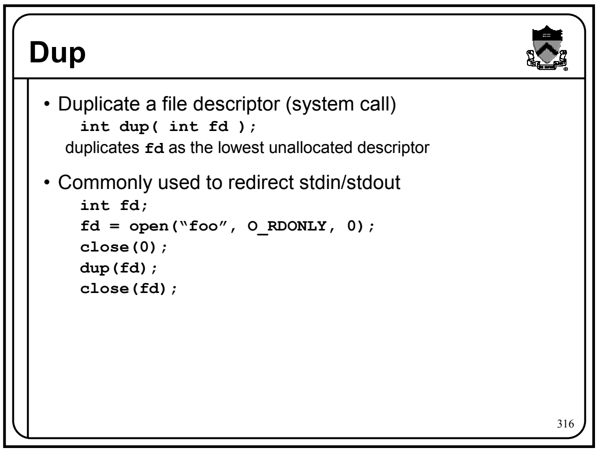
Creating a Pipe

- System call
  ```
  int pipe( int fd[2] );
  return 0 upon success and –1 upon failure
  fd[0] is open for reading
  fd[1] is open for writing
  ```
- Two coordinated processes created by **fork** can pass data to each other using a pipe.



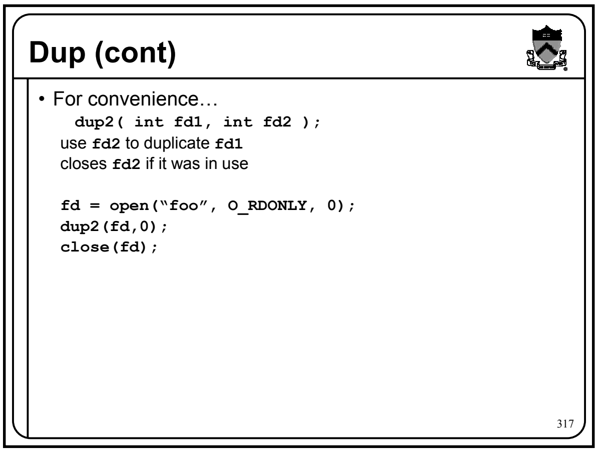
Pipe Example

```
int pid, p[2];
...
pipe(p);
pid = fork();
if (pid == 0) {
   close(p[1]);
   ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```



Dup

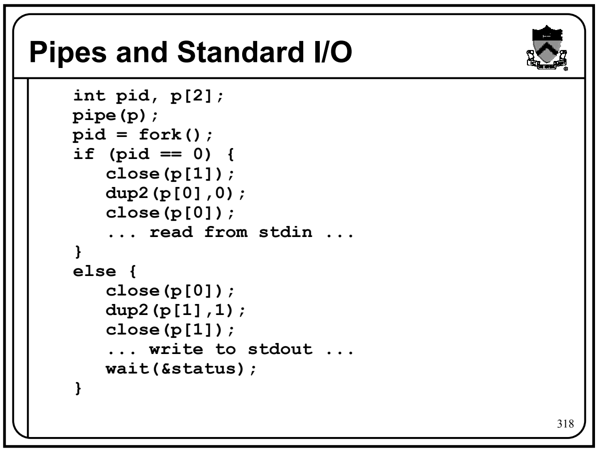
- Duplicate a file descriptor (system call)
  ```
  int dup( int fd );
  ```
  duplicates **fd** as the lowest unallocated descriptor
- Commonly used to redirect stdin/stdout
  ```
  int fd;
  fd = open(“foo”, O_RDONLY, 0);
  close(0);
  dup(fd);
  close(fd);
  ```



Dup (cont)

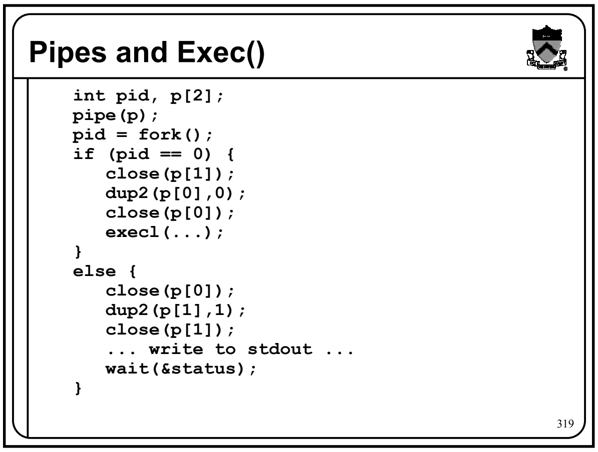
- For convenience…
  ```
  dup2( int fd1, int fd2 );
  ```
  use **fd2** to duplicate **fd1**
  closes **fd2** if it was in use

  ```
  fd = open(“foo”, O_RDONLY, 0);
  dup2(fd,0);
  close(fd);
  ```



Pipes and Standard I/O

```
int pid, p[2];
pipe(p);
pid = fork();
if (pid == 0) {
   close(p[1]);
   dup2(p[0],0);
   close(p[0]);
   ... read from stdin ...
}
else {
   close(p[0]);
   dup2(p[1],1);
   close(p[1]);
   ... write to stdout ...
   wait(&status);
}
```



Pipes and Exec()

```
int pid, p[2];
pipe(p);
pid = fork();
if (pid == 0) {
   close(p[1]);
   dup2(p[0],0);
   close(p[0]);
   execl(...);
}
else {
   close(p[0]);
   dup2(p[1],1);
   close(p[1]);
   ... write to stdout ...
   wait(&status);
}
```

Unix shell (sh, csh, bash, ...)

- Read command line from stdin
- Expand wildcards
- Interpret redirections < > |
- pipe (as necessary), fork, dup, exec, wait
- If `&` then don't wait!
- Start from code on previous slide, edit it until it's a Unix shell!
- Game referee: pipe, fork, dup, fork, dup, exec, exec, read, write . . .

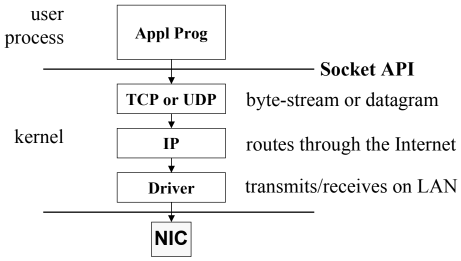
Interprocess Communication

- Pipes
  - o Processes must be on same machine
  - o One process spawns the other
  - o Used mostly for filters
- Messages
  - o Processes can be on any machine
  - o Processes can be created independently
  - o Used for clients/servers, distributed systems, etc.

Messaging Example: Client/Server

- Server: process that provides a service
  - o e.g., file server, web server, mail server
  - o called a passive participant: waits to be contacted
- Client: process that requests a service
  - o e.g., desktop machine, web browser, mail reader
  - o called an active participant: initiates communication

Network Subsystem



Communication Semantics

- Reliable Byte-Stream (like a pipe):
  - o TCP
- Unreliable Datagram:
  - o UDP

Names and Addresses

- Host name
  - o like a post office name; e.g., www.cs.princeton.edu
- Host address
  - o like a zip code; e.g., 128.112.92.191
- Port number
  - o like a mailbox; e.g., 0-64k

Socket API

- Socket Abstraction
  - o end-point of a network connection
  - o treated like a file descriptor
- Creating a socket
  - o `int socket(int domain, int type, int protocol)`
  - o domain = PF_INET, PF_UNIX
  - o type = SOCK_STREAM, SOCK_DGRAM, SOCK_RAW

Sockets (cont)

- Passive Open (on server)

```
int bind(int socket,
            struct sockaddr *addr,
            int addr_len)
int listen(int socket, int backlog)
int accept(int socket,
              struct sockaddr *addr,
              int addr_len)
```

Sockets (cont)

- Active Open (on client)

```
int connect(int socket,
                struct sockaddr *addr,
                int addr_len)
```

- Sending/Receiving Messages

```
int send(int socket, char *buf,
            int blen, int flags)
int recv(int socket, char *buf,
            int blen, int flags)
```

Trivia Question

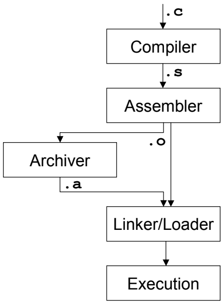
- How many messages traverse the Internet when you click on a link?

Sparc Architecture

CS 217

Compilation Pipeline

- Compiler (`gcc`): `.c` → `.s`
  - – **translates high-level language to assembly language**
- Assembler (`as`): `.s` → `.o`
  - – **translates assembly language to machine language**
- Archiver (`ar`): `.o` → `.a`
  - – **collects object files into a single library**
- Linker (`ld`): `.o` + `.a` → `a.out`
  - – **builds an executable file from a collection of object files**
- Execution (`execlp`)
  - – **loads an executable file into memory and starts it**



Example Compilation

- High-level language

```
x = a + b;
```

- Assembly language

```
ld a, %r1
ld b, %r2
add %r1, %r2, %r3
st %r3, x
```

Symbolic Representation

- Machine language

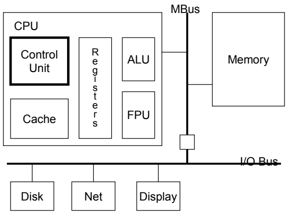
```
110000100000 ...
```

Bit-encoded Representation

Instruction Execution

- CPU's control unit executes a program

```
PC ← memory location of first instruction
while (PC != last_instr_addr)
    execute(MEM[PC]);
```



Instruction Execution

- CPU's control unit executes a program

```
PC ← memory location of first instruction
while (PC != last_instr_addr)
    execute(MEM[PC]);
```

- Multiple phases…
  - fetch: instruction fetch; increment PC
  - execute: arithmetic instructions, compute branch target address, compute memory addresses
  - memory access: read/write memory
  - store: write results to registers

Fetch	**Execute**	**Memory**	**Store**	**Fetch**	**Execute**	**Memory**	**Store**

Instruction Pipelining

- Pipeline

Fetch	**Execute**	**Memory**	**Store**		
	Fetch	**Execute**	**Memory**	**Store**	
		Fetch	**Execute**	**Memory**	**Store**

- PC is incremented by 4 at the Fetch stage to retrieve the next instruction

Instructions

- Each machine instruction is composed of…
  - <u>opcode</u>: operation to be performed
  - <u>operand</u>: data that is operated upon
- Each machine supports a few formats…
  - *opcode*
  - *opcode dst*
  - *opcode src dst*
  - *opcode src1 src2 dst*

Sparc Instruction Set

- Instruction groups
  - o integer arithmetic (**`add`**, **`sub`**, ...)
  - o bitwise logical (**`and`**, **`or`**, **`xor`**, ...)
  - o shift (**`sll`**, **`srl`**, ...)
  - o load/store (**`ld`**, **`st`**, ...)
  - o integer branch (be, bne, **`bl`**, **`bg`**, ...)
  - o Trap (**`ta`**, **`te`**, ...)
  - o control transfer (**`call`**, **`save`**, ...)
  - o floating point (**`ldf`**, **`stf`**, **`fadds`**, **`fsubs`**, ...)
  - o floating point branch (**`fbe`**, **`fb`**ne, **`fbl`**, **`fbg`**, ...)

Sparc Instruction Set

- Instruction formats
  - Format 1 (op = 1) -- e.g., call
  - Format 2 (op = 0): -- e.g., branches
  - Format 3 (op = 2 or 3): -- e.g., add

op	
31	29

Sparc Instruction Set

- Format 3 (op = 2 or 3):

op	rd	op3	rs1	0	ignore	rs2
31	29	24	18	13	12	4

```
add %i1,%i2,%o2
```

Sparc Instruction Set

- Format 3 (op = 2 or 3):

op	rd	op3	rs1	0	ignore	rs2

OR

op	rd	op3	rs1	1	simm13	
31	29	24	18	13	12	4

```
add %i1,360,%o2
```

simm13 is a signed constant within +-4096

Example

- Assembly Language
  - **`add %i1,360,%o2`**
- Machine language

2	10	0	25	1	360	(decimal)
2	12	0	31	1	550	(octal)
31	29	24	18	13	12	

```
10010100000001100110000101101000
```

Other Sparc Instructions

- Format 1 (op = 1) -- e.g., **`call`**

op	disp30
31	29

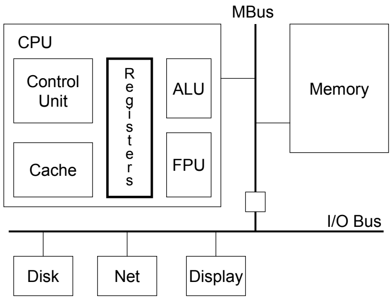
- Format 2 (op = 0) -- e.g., branches

op	rd		op2	imm22
op	a	cond	op2	disp22
31	29	28	24	21

Storage Hierarchy

- Registers
  - ~128, 1-5ns access time (CPU cycle time)
- Cache
  - 1KB – 4MB, 20-100ns (multiple levels)
- Memory
  - 64MB – 2GB, 200ns
- Disk
  - 1GB – 100GB, 10ms
- Long-term Storage
  - 1TB, 1-10s

Machine Architecture



Sparc Registers

- 32 x 32-bit general-purpose registers
  `%r0 … %r31`
- Register map

`%g0 … %g7`	`(%r0 … %r7)`	global
`%o0 … %o7`	`(%r8 … %r15)`	output
`%l0 … %l7`	`(%r16 … %r23)`	local
`%i0 … %i7`	`(%r24 … %r31)`	input

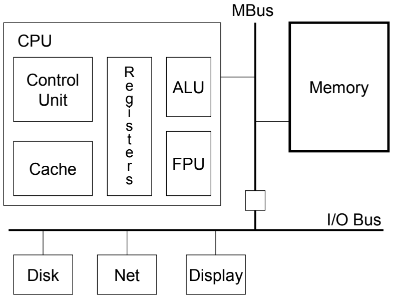
- Some registers have dedicated uses

`%sp (%r14, %o6)`	stack pointer
`%fp (%r30, %i6)`	frame pointer
`%r15 (%o7)`	temporary
`%r31 (%i7)`	return address
`%g0 (%r0)`	always `0`

Sparc Registers (cont)

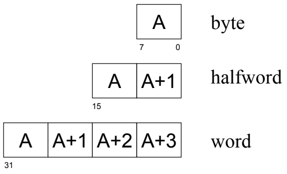
- Special-purpose registers
  - o manipulated by special instructions
- Examples
  - o floating point registers **(%f0 … %f31)**
  - o program counter **(PC)**
  - o next program counter **(NPC)**
  - o integer codition codes **(PSR)**
  - o trap base register **(TBR)**
  - o window **(WIM)**
  - o etc.

Machine Architecture



Addressing Memory

- 8-bit byte is the smallest addressable unit
- 32-bit addresses; thus 32-bit address space
- Can load and store doublewords too
- Sparc is <u>big-endian</u>



Addressing Memory

- Two modes to yield effective address
  - o add contents of two registers

`ld [%o1],%o2`	<u>register indirect</u>
`st %o1,[%o2+%o3]`	<u>register indexed</u>

  - o add contents of register and immediate

`ld [%o1+10],%o2`	<u>base displacement</u>

Upcoming Lectures ...

- Instruction set
- Number systems
- Branching condition codes
- Procedure calls
- Assembler
- Linker
- etc.

SPARC Instruction Set

CS 217

Load Instructions

- Move data from memory to a register
  - ld $\begin{bmatrix}\texttt{u}\\ \\ \texttt{s}\end{bmatrix}\begin{bmatrix}\texttt{b}\\ \texttt{h}\\ \texttt{d}\end{bmatrix}$ {a} [*address*], *reg*
- Examples:
  - `ld [%i1],%g2`
  - `ldud [%i1+%i2],%g3`

11	dst	opcode	src1	0	ignore	src2

OR

11	dst	opcode	src1	1	simm13

31 29 24 18 13 12 4

Load Instructions

- Move data from memory to a register
  - ld $\begin{bmatrix}\texttt{u}\\ \\ \texttt{s}\end{bmatrix}\begin{bmatrix}\texttt{b}\\ \texttt{h}\\ \texttt{d}\end{bmatrix}$ {a} [*address*], *reg*
- Details
  - fetched byte/halfword is right-justified
  - leftmost bits are zero-filled or sign-extended
  - double-word loaded into register pair*;* most significant word in *reg* (must be even); least significant in *reg+1*
  - address must be appropriately aligned

Store Instructions

- Move data from a register to memory
  - st $\begin{bmatrix}\texttt{b}\\ \texttt{h}\\ \texttt{d}\end{bmatrix}$ {a} *reg*, [*address*]
- Examples:
  - `st %g1,[%o2]`
  - `stb %g1,[%o2+o3]`

11	dst	opcode	src1	0	ignore	src2

OR

11	dst	opcode	src1	1	simm13

31 29 24 18 13 12 4

Store Instructions

- Move data from a register to memory
  - st $\begin{bmatrix}\texttt{b}\\ \texttt{h}\\ \texttt{d}\end{bmatrix}$ {a} *reg*, [*address*]
- Details
  - rightmost bits of byte/halfword are stored
  - leftmost bits of byte/halfword are ignored
  - *reg* must be even when storing double words

Arithmetic Instructions

- Arithmetic operations on data in registers
  - o add{x}{cc} ***src1, src2, dst*** dst = src1 + src2
  - o sub{x}{cc} ***src1, src2, dst*** dst = src1 - src2
- Examples:
  - o **`add %o1,%o2,%g3`**
  - o **`sub %i1,2,%g3`**
- Details
  - o *src*1 and *dst* must be registers
  - o *src2* may be a register or a signed 13-bit immediate

10	dst	opcode	src1	0	ignore	src2

OR

10	dst	opcode	src1	1	simm13

31 29 24 18 13 12 4

Bitwise Logical Instructions

- Logical operations on data in registers
  - o and{cc} ***src1, src2, dst*** *dst = src1 & src2*
  - o andn{cc} ***src1, src2, dst*** *dst = src1 & ~src2*
  - o or{cc} ***src1, src2, dst*** *dst = src1 | src2*
  - o orn{cc} ***src1, src2, dst*** *dst = src1 | ~src2*
  - o xor{cc} ***src1, src2, dst*** *dst = src1 ^ src2*
  - o xnor{cc} ***src1, src2, dst*** *dst = src1 ^ ~src2*

10	dst	opcode	src1	0	ignore	rs2

OR

10	dst	opcode	src1	1	simm13

31 29 24 18 13 12 4

Shift Instructions

- Shift bits of data in registers
  - o $s\begin{bmatrix}\mathtt{l}\\\mathtt{r}\end{bmatrix}\begin{bmatrix}\mathtt{l}\\\mathtt{a}\end{bmatrix}$ ***src1,*** $\begin{bmatrix}\textit{src2}\\\textit{0..31}\end{bmatrix}$ ***, dst***

sll: ***dst = src1 << src2;***

slr: ***dst = src1 >> src2;***

- Details
  - o do not modify condition codes
  - o sll and srl fill with 0, sra fills with sign bit
  - o no sla

10	dst	opcode	src1	0	ignore	src2

OR

10	dst	opcode	src1	0	ignore	shift cnt

31 29 24 18 13 12 4

Floating Point Instructions

- Performed by floating point unit (FPU)
- Use 32 floating point registers: **`%f0…%f31`**
- Load and store instructions
  - – ld ***[address],freg***
  - – ldd ***[address],freg***
  - – st ***freg,[address]***
  - – std ***freg,[address]***
- Other instructions are FPU-specific
  - – fmovs,fsqrt,fadd,fsub,fmul,fdiv,…

C Programs

C Code **Assembly code**

```
x = a + 5000;           set a,%i1       (?)
                        ld [%i1],%g1

                        set 5000,%i1    (?)

                        add %g1,%g2,%g1

                        set x,%i1       (?)
                        st %g1,[%i1]
```

Data Movement

- How do we load a constant (e.g., address) into a register
  - o set ***value, dst ?***

Instruction format?

op	*dst*	?

31 29

Data Movement

- Loading a constant (e.g., address) into a register

```
sethi %hi(value),dst
or dst,%lo(value),dst
```

- Details
  - o if `%hi(value) == 0,` omit `sethi`
  - o if `%lo(value) == 0,` omit `or`

sethi instruction format

00	*dst*	100	%hi(*value*)
31	29	24	21

Data Movement (cont)

- Example: direct addressing

```
set a,%g1         sethi %hi(a),%g1
ld [%g1],%g2      or %lo(a),%g1
                  ld [%g1],%g2
```

- Faster alternative

```
sethi%hi(a),%g1
ld [%g1+%lo(a)],%g2
```

Example

C Code	Assembly code

`x = a + 5000;`	`sethi%hi(a),%i1`
	`ld [%i1+%lo(a)],%g1`
	
	`sethi%hi(5000),%i1`
	`add %i1,%lo(5000),%g2`
	
	`add %g1,%g2,%g1`
	
	`sethi%hi(x),%i1`
	`st %g1,[%i1+%lo(x)]`

Synthetic Instructions

- Implemented by assembler with one or more "real" instructions; also called pseudo-instructions

Synthetic	Real
`mov src,dst`	`or %g0,src,dst`
`clr reg`	`add %g0,%g0,reg`
`clr [addr]`	`st %g0,[addr]`
`neg  dst`	`sub %g0,dst,dst`
`neg src,dst`	`sub %g0,src,dst`
`inc dst`	`add dst,1,dst`
`dec dst`	`sub dst,1,dst`

Example Synthetic Instructions

- Complement

```
o neg reg          sub %g0,reg,  reg
o not reg          xnor reg,%g0,reg
```

- Bit operations

```
o btst bits, reg   andcc reg, bits, %g0
o bset bits, reg   or reg, bits, reg
o bclr bits, reg   andn reg, bits, reg
o btog bits, reg   xor reg, bits, reg
```

Summary

- Assembly language
  - o Provides convenient symbolic representation
  - o Translated into machine language by assembler
- Instruction set
  - o Use scarce resources (instruction bits) as effectively as possible
  - o Key to good architecture design

Arithmetic Instructions

CS 217

Arithmetic Instructions

- Arithmetic operations on data in registers
  - o add{x}{cc} ***src1, src2, dst*** dst = src1 + src2
  - o sub{x}{cc} ***src1, src2, dst*** dst = src1 - src2
- Examples:
  - o **`add %o1,%o2,%g3`**
  - o **`sub %i1,2,%g3`**

Number Systems

- General form of a number in ***base*** ***b*** is

$$x = x_n b^n + x_{n-1} b^{n-1} + \ldots + x_1 b^1 + x_0 b^0$$
$$+ x_{-1} b^{-1} + \ldots + x_{-m} b^{-m}$$

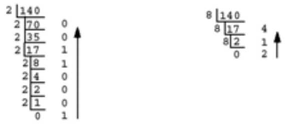
where x_i are the ***positional coefficients***

- Modern computers use binary arithmetic, i.e., base 2

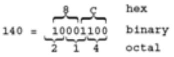
$$\begin{aligned} 140_{10} &= 1 \times 10^2 + 4 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 10001100_2 \\ &= 2 \times 8^2 + 1 \times 8^1 + 4 \times 8^0 = 214_8 \\ &= 8 \times 16^1 + \mathrm{C} \times 16^0 = 8\mathrm{C}_{16} \end{aligned}$$

Conversion

- To convert from decimal to binary, divide by 2 repeatedly, read remainders up.



- Easier to convert to octal, then to binary



Addition

- Addition in base ***b***

$$\begin{array}{r} x_n b^n + x_{n-1} b^{n-1} + x_{n-2} b^{n-2} + \ldots + x_1 b^1 + x_0 b^0 \\ + y_n b^n + y_{n-1} b^{n-1} + y_{n-2} b^{n-2} + \ldots + y_1 b^1 + y_0 b^0 \\ \hline z_{n+1} b^{n+1} + z_n b^n + z_{n-1} b^{n-1} + z_{n-2} b^{n-2} + \ldots + z_1 b^1 + z_0 b^0 \end{array}$$

where $S_i = x_i + y_i + C$, $C = S_{i-1}/b$, and $z_i = S_i \bmod b$ where $S_{-1} = 0$

- Addition in base 2:

```
  00101101
+ 10011001
  --------
  11000110
```

- the sum might have ***one*** more digit than the largest operand

Multiplication

- Multiplication in base 2: `00101101 * 10111001`

```
1 00101101
0  00000000
1   00101101
1    00101101
1     00101101
0      00000000
0       00000000
1        00101101
-------------------
  010000010000101
```

- The product has about as many digits as the two operands combined, i.e.

$$\log(a \times b) = \log(a) + \log(b)$$

Machine Arithmetic

- Computers usually have a fixed number of binary digits ("bits"), e.g., 32 bits
- For example, using 6 bits, numbered 0 to 5 from the right

  largest number $111111_2 = 63_{10} = 2^6 - 1$

  smallest number $000000_2 = 0$
- What is 50 + 20?

```
   110010
 + 010100
   ──────
  1000110
```

- The highest bit doesn't fit, so we get $000110_2 = 6_{10}$
- Spilling over the lefthand side is ***overflow***

Signed Magnitude

- ***Sign-magnitude*** notation:

  bit $n-1$ is the sign; 0 for +, 1 for -

  bits $n-2$ through 0 hold an unsigned number

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $111111_2 = -31_{10} = -(2^{6-1} - 1)$
- Addition and subtraction are complicated when signs differ
- Sign-magnitude is rarely used

One's Complement

- ***One's-complement*** notation: *-k = (2ⁿ-1) - k = 11111...(n bits) - k*

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold ***complement*** of negative numbers

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $100000_2 = -31_{10} = -(2^{6-1} - 1)$

$$-k_{1C} = \hat{}\, k$$

- Addition and subtraction are easy, but there are ***2*** representations for 0

$$a - b = a + (r^n - 1 - b) + 1$$
$$a - b = a + b_{1C} + 1$$

Two's Complement

- ***Two's-complement*** notation: *-k = 2ⁿ - k =* ***(2ⁿ-1) - k*** *+ 1*

  bit $n-1$ is the sign; bits $n-2$ through 0 hold an unsigned number

  bits $n-2$ through 0 hold the ***complement*** of a negative number ***plus 1***

  largest number $011111_2 = 31_{10} = 2^{6-1} - 1$

  smallest number $100000_2 = -32_{10} = -2^{6-1}$; note ***asymmetry***

$$-k_{2C} = \hat{}\, k + 1$$

- To negate a 2's compl. number: first complement all the bits, then add 1

	start with	complement	increment	
+6	000110	111001	111010	-6
-6	111010	000101	000110	+6
+0	000000	111111	000000	-0
+1	000001	111110	111111	-1
+31	011111	100000	100001	-31
-31	100001	011110	011111	+31
-32	100000	011111	100000	-32

Two's Complement (cont)

- Adding 2's-complement numbers: ignore signs, add unsigned bit strings

```
   +20      010100        -20      101100
 + - 7    + 111001      + + 7    + 000111
   ───      ──────        ───      ──────
   +13      001101        -13      110011

   +20      010100        -20      101100
 + + 7    + 000111      + - 7    + 111001
   ───      ──────        ───      ──────
   +27      011011        -27      100101
```

$$a - b = a + (r^n - 1 - b) + 1$$
$$a - b = a + b_{2C}$$

- Signed overflow occurs if

  the carry ***into*** the sign bit differs from the carry ***out*** of the sign bit

```
   +20      010100        -20      101100
 + +17    + 010001      + -17    + 101111
   ───      ──────        ───      ──────
   -27      100101        +27      011011
```

- Same hardware for ***both*** unsigned and signed, but flags ***two*** conditions

  overflow signed overflow

  carry unsigned overflow

Sign Extension

- To convert from a small signed integer to a larger one, copy the sign bit

```
                 +5            -5
4 bits         0101          1011
8 bits     00000101      11111011
```

- To convert a large signed integer to a smaller one: check trunced bits

```
                 +5            -5
8 bits     00000101      11111011
4 bits         0101          1011      OK!

                +20           -20
8 bits     00010100      11101100
4 bits         0100          1100      Bad!
```

- Hardware does extension, but ***may not*** check for truncation; nor does C

```
short small = -50; long big = small;
printf("%d %d\n, small, big);            -50 -50

long big = 40000; short small = big;
printf("%d %d\n", small, big);           -25536 40000

char c = 255;
printf("%d\n", c);                       -1
```

Floating Point Instructions

- Performed by floating point unit (FPU)
- Use 32 floating point registers: `%f0…%f31`
- Load and store instructions
  - ld [*address*],*freg*
  - ldd [*address*],*freg*
  - st *freg*,[*address*]
  - std *freg*,[*address*]
- Other instructions are FPU-specific
  - fmovs,fsqrt,fadd,fsub,fmul,fdiv,…

Floating Point Numbers

- Floating point numbers are like scientific notation

1.386×10^{6}

-3.0083×10^{-14}

4.32×10^{-8}

general form is

$\pm m \times 10^{\pm p}$

exponent

significand

- Significand restricted to range, e.g., $0 \le m < 1$, and fixed number of digits
- Floating point is approx. representation for infinitely many real numbers

$m \times \beta^{k}$	m	is an n-bit ***significand*** or ***fraction***
	β	is the ***base*** (usually 2)
	k	is the ***exponent***

e.g. for base 2

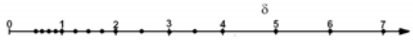
$$0.100011 \times 2^{6} = (1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6}) \times 2^{6}$$

Floating Point Numbers (cont)

- ***Normalized*** floating point numbers make the representation unique
  - most significant digit is nonzero, e.g., $0.00486 \times 10^{1} \Rightarrow 0.486 \times 10^{-1}$
  - for floating point numbers, $\beta^{n-1} \le m < \beta^{n}$ or $1/\beta \le |m| < 1$
  - i.e., when $\beta = 2$, most significant bit of m is 1
- Example: $n = 3, \beta = 2, -1 \le k \le 2$

$m \times \beta^{k}$

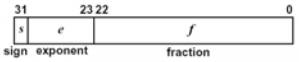
		k			
		-1	0	1	2
m	1.00	.5	1.	2.	4.
	1.01	.625	1.25	2.5	5.
	1.10	.75	1.5	3.	6.
	1.11	.875	1.75	3.5	7.
δ		.125	.25	.5	1.



- What about 0.0? Use reserved values of k, e.g.,
  - $1.00_2 \times 2^{-2}$ for 0.0, $1.11_2 \times 2^{5}$ for ∞

IEEE Floating Point

- IEEE format uses a ***hidden bit*** to increase precision by 1 bit
  - all ***normalized*** floating point numbers have the form $1.f \times 2^{e}$,
  - so ***assume*** the leading 1 and omit it
- Single precision (`float`) format



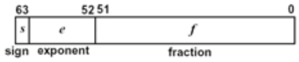
$-126 \le e \le 127$, ***bias*** $= 127, 0 \le f < 2^{23}$

- Values `1.1754943508222875e-38` to `3.4028234663852886000e+38`

$k = e - 127$	f	f. p. number
$-126 \le k \le 127$	$0 \le f < 2^{23}$	$\pm 1.f \times 2^{k}$
128	0	$\pm\infty$
128	$\ne 0$	NaN (signaling/quiet)
-127	0	± 0.0
-127	$\ne 0$	$\pm 0.f \times 2^{-126}$ (denormalized)

IEEE Floating Point (cont)

- Double precision (`double`) format



$-1022 \le e \le 1023$, ***bias*** $= 1023, 0 \le f < 2^{52}$

- Values: `2.2250738585072014e-308` to `1.7976931348623157e+308`

$k = e - 1023$	f	f. p. number
$-1022 \le k \le 1023$	$0 \le f < 2^{52}$	$\pm 1.f \times 2^{k}$
1024	0	$\pm\infty$
1024	$\ne 0$	NaN (signaling/quiet)
-1023	0	± 0.0
-1023	$\ne 0$	$\pm 0.f \times 2^{-1022}$(denormalized)

- Biased exponents in the most-significant bits are useful because
  - integer compare instructions can be used to compare floating point values
  - a bit string of 0's represents the value 0.0

Branching

CS 217

Condition Codes

- Processor State Register (**PSR**)

. . .	icc	. . .
31	23	20

- Integer condition codes (icc)
  - N **set if the last ALU result was negative**
  - Z **set if the last ALU result was zero**
  - V **set if the last ALU result was overflowed**
  - C **set if the last ALU instruction that modified the** icc **caused a carry out of, or a borrow into, bit 31**

Condition Codes (cont)

- `cc` versions of the integer arithmetic instructions set all the codes
- `cc` versions of the logical instructions set only **N** and **Z** bits
- Tests on the condition codes implement conditional branches and loops
- Carry and overflow are used to implement multiple-precision arithmetic

Compare and Test

- Synthetic instructions set condition codes

tst ***reg***	orcc ***reg***,%g0,%g0
cmp ***src1,src2***	subcc ***src1,src2***,%g0
cmp ***src,value***	subcc ***src,value***,%g0

- Using `%g0` as the destination discards the result

Carry and Overflow

- If the carry bit is set
  - **the last addition resulted in a carry, or**
  - **the last subtraction resulted in a borrow**
- Used for multi-word addition

```
addcc %g3,%g5,%g7
addxcc %g2,%g4,%g6
```

(%g6,%g7) = (%g2,%g3) + (%g4,%g5)

o the most significant word is in the even register

- Overflow indicates result of subtraction (or signed-addition) doesn't fit

Branch Instructions

- Transfer control based on **icc**

b{cond}{,a} ***label***

$$\begin{bmatrix}\mathtt{a}\\\mathtt{n}\\..\\\mathtt{z}\end{bmatrix}$$

00	a	cond	010	disp22
31	29	28	24	21

o target is a PC-relative address: PC + 4 x disp22

o where PC is the address of the branch instruction

Branch Instructions (cont)

- Unconditional branches (and synonyms)

ba	jmp	**branch always**
bn	nop	**branch never**

- Raw condition-code branches

bnz	!Z
bz	Z
bpos	!N
bneg	N
bcc	!C
bcs	C
bvc	!V
bvs	V

Branching Instructions (cont)

- Comparisons

instruction	signed	unsigned
be	Z	Z
bne	!Z	!Z
bg bgu	!(Z | (N^V))	!(C | Z))
ble bleu	Z | (N^V)	C | Z
bge bgeu	!(N^V)	!C
bl blu	N^V	C

Control Transfer

- Instructions normally fetched and executed from sequential memory locations
- PC is the address of the current instruction, and nPC is the address of the next instruction (nPC = PC + 4)
- Branches and control transfer instructions change nPC to something else

Control Transfer (cont)

- Control transfer instructions

instruction	type	addressing mode
b*icc*	**conditional branch**	**PC-relataive**
jmpl	**jump and link**	**register indirect**
rett	**return from trap**	**register indirect**
call	**procedure call**	**PC-relative**
t*icc*	**traps**	**register indirect (vectored)**

o PC-relative addressing is like register displacement addressing that uses the PC as the base register

Control Transfer (cont)

- Branch instructions

op	a	cond	op2	disp22

nPC = PC + signextend(disp22) << 2

- Calls

op	disp30

nPC = PC + signextend(disp30) << 2

o position-independent code does not depend on where it's loaded; uses PC-relative addressing

Branching Examples

- if-then-else

```
if (a > b)       #define a %l0
   c = a;        #define b %l1
else             #define c %l2
   c = b;        cmp a,b
                 ble L1; nop
                 mov a,c
                 ba L2; nop
                 L1: mov b,c
                 L2: ...
```

Branching Examples (cont)

- Loops

```
for (i=0; i<n; i++)       #define i %l0
   . . .                  #define n %l1
                          clr i
                      L1: cmp i,n
                          bge L2; nop
                          . . .
                          inc i
                          ba L1; nop
                      L2:
```

Branching Examples (cont)

- Alternative implementation

```
for (i=0; i<n; i++)           #define i %l0
   . . .                      #define n %l1
                              clr i
                              ba L2; nop
                          L1: . . .
                              inc i
                          L2: cmp i,n
                              bl L1; nop
```

Instruction Pipelining

- Pipeline

Fetch	**Decode**	**Oprnd**	**Execute**	**Store**		
	Fetch	**Decode**	**Oprnd**	**Execute**	**Store**	
		Fetch	**Decode**	**Oprnd**	**Execute**	**Store**

. . .

- PC is incremented by 4 at the Fetch stage to retrieve the next instruction

Pipelining (cont)

- A delay slot is caused by a `jmp` instruction

	PC	nPC	instruction					
o	8	12	`add`	`add`	`add`			
o	12	16	`jmp 40`		`jmp`	`jmp`		
o	16	40	*`delay`*			*`delay`*	*`delay`*	
o	. . .							
o	40	44	`sub`				`sub`	`sub`

Delay Slots

- One option: use `nop` in all delay slots
- Optimizing compilers try to <u>fill</u> delay slots

```
if (a>b) c=a; else c=b;
    cmp a,b          cmp a,b
    ble L1;          ble L1
    nop              mov b,c
    mov a,c          mov a,c
    ba L2;       L1: …
    nop
L1: mov b,c
L2: ...
```

Annul Bit

- Controls the execution of the delay-slot instruction
  - bg,a   L1
  - mov   a,c

  o the `,a` causes the `mov` instruction to be executed   if the branch is taken, and not executed if the branch is not taken

- Exception
  - ba,a L **does <u>not</u> execute the delay-slot instruction**

Annul Bit (cont)

- Optimized `for (i=0; i<n; i++) 1;2;…;n`

```
    clr  i              clr   i
    ba   L2             ba,a  L2
L1: 1               L1: 2
    2                   . . .
    . . .               n
    n                   inc   i
    inc  i          L2: cmp   i,n
L2: cmp  i,n            bl,a  L1
    bl   L1             1
    nop
```

- For CS 217, you don't need to bother about the Annul bit, or annulled branches.

While-Loop Example

```
while (...) {
  stmt_1
    :
  stmt_n
}
```

```
test: cmp ...      ┐
      bx done      │ 3 instr
      nop          ┘
      stmt_1
        :
      stmt_n       ┐
      ba test      │ 2 instr
      nop          ┘
done: ...
```

While-Loop (cont)

- Move test to end of loop

```
test: cmp ...
      bx done
      nop
loop: stmt_1
        :
      stmt_n
      cmp ...
      bnx loop
      nop
done: ...
```

- Eliminate first test

```
      ba test
      nop
loop: stmt_1
        :
      stmt_n
test: cmp ...
      bnx loop
      nop
      ...
```

While-Loop (cont)

- Eliminate the **nop** in the loop

```
      ba test
      nop
loop: stmt_2
        :
      stmt_n
test: cmp ...
      bnx,a loop
      stmt_1
      ...
```

now 2 overhead instructions per loop

If-Then-Else Example

```
if (...) {
  t-stmt_1
    :
  t-stmt_n
}
else {
  e-stmt_1
    :
  e-stmt_m
}
```

o How optimize?

```
      cmp ...
      bnx else
      nop
      t-stmt_1
        :
      t-stmt_n
      ba next
      nop
else: e-stmt_1
        :
      e-stmt_m
next: ...
```

Procedure Call

CS 217

Procedure Call

- Involves following actions
  pass arguments
  save a return address
  transfer control to <u>callee</u>
  transfer control back to <u>caller</u>
  return results
- Simplest example: leaf procedure (`a=b*c;`)

```
ld   b,%o0        ld   b,%o0
ld   c,%o1        call .mul
call .mul         ld   c,%o1
nop               st   %o0,a
st   %o0,a
```

Call/Return Instructions

- Procedures are called with either…

  call *label*

01	disp30
31	29

  o leaves PC (location of `call`) in `%o7 (%r15)`

  jmpl *address,reg*

10	reg	111000	rs1	0	0	rs2
10	reg	111000	rs1	1	simm13	
31	29	24	18	13	12	4

  o leaves PC in *reg*

Call/Return (cont)

- Indirect calls

  jmpl *reg*,%r15

  o jumps to the 32-bit address specified in *reg*

  o leaves PC (return address) in `%r15`

  o e.g., for function pointers `a = (*apply)(b,c);`

```
ld   b,%o0
ld   c,%o1
ld   apply,%o3
jmpl %o3,%r15; nop
st   %o0,a
```

Call/Return (cont)

- Procedure call return

  jmpl %r15+8,%g0

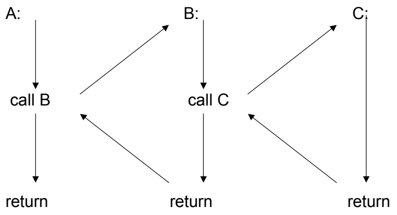
  o transfers control from caller to callee

  o other instructions: `ret` and `retl`

  o why `+8`?

Nested/Recursive Calls

- `A` calls `B`, which calls `C`



A:

B:

C:

call B

call C

return

return

return

  o must work when `B` is `A`

Nested/Recursive Calls (cont)

- Other requirements
  - **pass a variable number of arguments**
  - **pass and return structures**
  - **allocate and deallocate space for local variables**
  - **save and restore caller's registers**
- Entry and exit sequences collaborate to implement these requirements

Stack

- Procedure call information stored on stack
  - **locals, including compiler temporaries**
  - **caller's registers, if necessary**
  - **callee's arguments, if necessary**
- Sparc's stack grows "down" from high to low address
- The stack pointer (`%sp`) points to top word on the stack (must be multiple of 8)

Arguments and Return Values

- By convention
  - caller places arguments in the “out” registers
  - callee finds its arguments in the “in” registers
  - only the first 6 arguments are passed in registers
  - the rest are passed on the stack
- Registers at call time

caller	callee	
%o7	%i7	return address -8
%o6	%i6	stack/frame pointer
%o5	%i5	sixth argument
…	…	
%o0	%i0	first argument

Arguments/Return Value (cont)

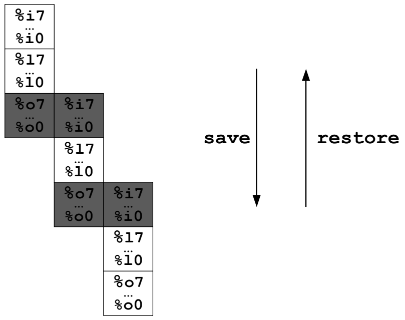
- Registers at return time

callee	caller	
%i5	%o5	sixth return value
%i4	%o4	fifth return value
…	…	
%i0	%o0	first return value

Register Windows

- Each procedure gets 16 “new” registers
- The window “slides” at call time
  - caller’s out registers become synonymous with
  - callee’s in registers
- Instructions
  - save slides the window forward
  - restore slides the window backwards
  - decrement/increments CWP register
- Finite number of windows (usually 8)

Register Windows (cont)



```
%i7
…
%i0
%l7
…
%l0
%o7        %i7
…          …
%o0        %i0
           %l7
           …
           %l0
           %o7        %i7
           …          …
           %o0        %i0
                      %l7
                      …
                      %l0
                      %o7
                      …
                      %o0
```

save

restore

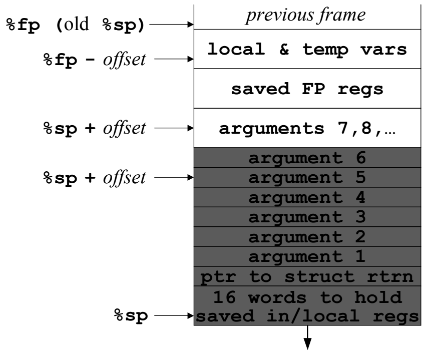
Window Managment

- Call time (`save`)
  - save %sp,*N*,%sp e.g., save %sp,-4*16,%sp
  - current window becomes previous window
  - decrements CWP and checks for <u>overflow</u>
  - adds *N* to the stack pointer (allocates *N* bytes if *N*<0)
  - if overflow occurs, save registers on the stack (must be enough stack space)
- Return time (`restore`)
  - previous window becomes current window
  - increments CWP and checks for <u>underflow</u>

Window Management (cont)

- In both `save` and `restore`
  - <u>source</u> registers refer to <u>current</u> window
  - <u>destination</u> registers refer to <u>new</u> window

Stack Frame

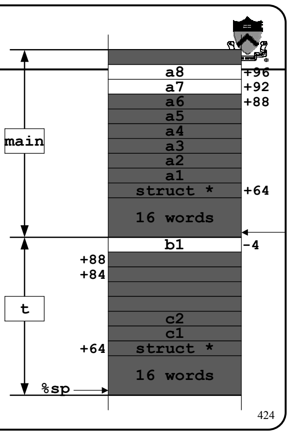


Example Stack Frames

```
main() {
  t(1,2,3,4,5,6,7,8);
}

t(int a1, int a2, int a3, int a4,
  int a5, int a6, int a7, int a8) {
    int b1 = a1;
    return s(b1,a8);
}

s(int c1, int c2) {
  return c1 + c2;
}
```

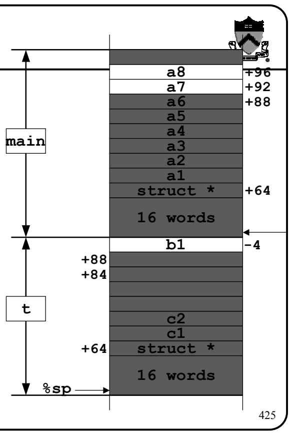
Example (cont)

```
_main: save %sp,-104,%sp
       set 1,%o0
       set 2,%o1
       set 3,%o2
       set 4,%o3
       set 5,%o4
       set 6,%05
       set 7,%i5
       st %i5,[%sp+4*6+68]
       set 8,%i5
       st %i5,[%sp+4*7+68]
       call _t; nop
       ret; restore
```



Example (cont)

```
_t: save %sp,-96,%sp
    st %i0,[%fp-4]
    ld [%fp-4],%o0
    ld [%fp+96],%o1
    call _s; nop
    mov %o0,%i0
    ret; restore

_s: add %o0,%01,%o0
    retl; nop
```



Kernel mode

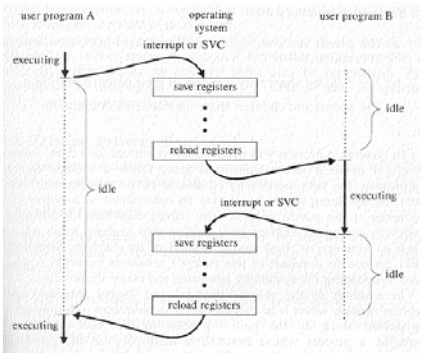
(only a few slides here, more coming)

CS 217

Interrupt-Driven Operation

- Everything OS does is interrupt-driven
  - o System calls use traps to interrupt
- An interrupt stops the execution dead in its tracks, control is transferred to the OS
  - o Saves the current execution context in memory (PC, registers, etc.)
  - o Figures out what caused the interrupt
  - o Executes a piece of code (interrupt handler)
  - o Re-loads execution context when done, and resumes execution

Interrupt Processing



System Calls (cont)

- Parameters passed…
  - o in fixed registers
  - o in fixed memory locations
  - o in an argument block, w/ block's address in a register
  - o on the stack
- Usually invoke system calls with trap instructions
  - o `ta 0`
  - o with parameters in `%g1` (function), `%o0..%o5`, and on the stack
- Mechanism is highly machine-dependent

Read System Call (cont)

- User-side implementation (`libc`)

```
read: set 3,%g1
      ta 0
      bcc L1; nop
      set _errno,%g1
      st %o0,[%g1]
      set -1,%o0
  L1: retl; nop
```

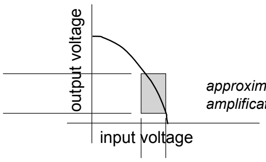
- Kernel-side implementation
  - o sets the C bit if an error occurred
  - o stores an error code in %o0 (see /usr/include/sys/errno.h)

Digital Circuits

CS 217

Analog circuits

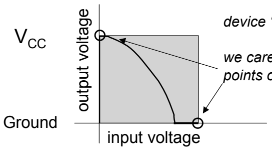
- Components: resistors, inductors, capacitors, transistors ...
- Voltage, current are continuous functions of time
  - o and of d/dt of current, voltage...
- Build: amplifiers, radios ...
- Typical device characteristic:



approximately linear amplification

Digital circuits

- Components: transistors, transistors, transistors ...
  - o (and the occasional capacitor)
- Pick two voltages of interest: "V_{CC}" and "Ground"
- Build: clocks, adders, computers, computers, computers...
  - o "computers" includes: cell phone, Nintendo, cash register, ...
- Typical device characteristic:

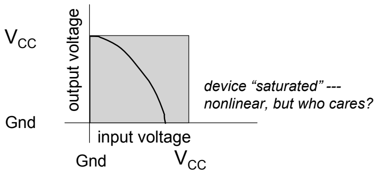


device "saturated" ---

we care only about these two points on the graph

Digital circuits

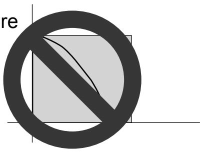
- Call Ground "0" and Vcc "1"
- Look at characteristic of this device:
  - o input = 0 ⇒ output = 1
  - o input = 1 ⇒ output = 0
- It's a "NOT" gate



device "saturated" --- nonlinear, but who cares?

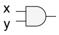
Raise the level of abstraction

- To understand analog circuits, you need physics, real analysis, electrical engineering...
- Let's not go there



- Instead, just think about 0's and 1's.
- *Bonus: voltages won't be on the exam!*

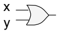
Circuit components



AND gate

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

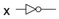
also written: xy



OR gate

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

also written: x+y

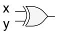


NOT gate

x	~x
0	1
1	0

also written: $\overline{x}$, ¬x

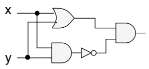
Exclusive-OR circuit



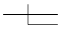
XOR gate

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

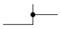
Can synthesize it from the other components



x XOR y = (x+y) & ¬(x&y)



wires crossing
(not connected)

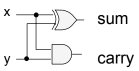


wire junction
(connection)

Adder (for 1-bit binary numbers)

x	y	add(x,y)
0	0	0 0
0	1	0 1
1	0	0 1
1	1	1 0

carry sum



sum = x XOR y

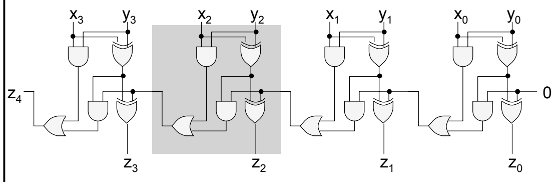
carry = x&y

N-bit binary adder

$$\begin{array}{r} x_3\, x_2\, x_1\, x_0 \\ +\ \ y_3\, y_2\, y_1\, y_0 \\ \hline z_4\, z_3\, z_2\, z_1\, z_0 \end{array}$$

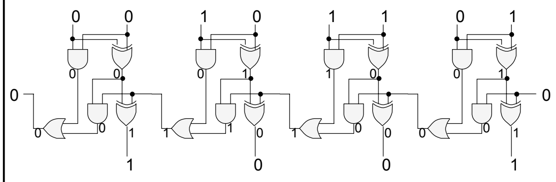
$\text{sum}_i = x_i \text{ XOR } y_i \text{ XOR carry}_{i-1}$

$\text{carry}_i = (x_i \,\&\, y_i) + ((x_i \text{ XOR } y_i) \,\&\, \text{carry}_{i-1})$



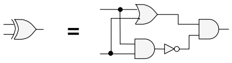
N-bit binary adder

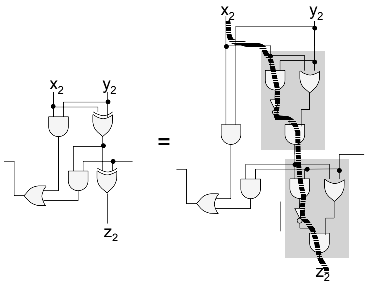
x_3 x_2 x_1 x_0	0 1 1 0	6
+ y_3 y_2 y_1 y_0	+ 0 0 1 1	+ 3
z_4 z_3 z_2 z_1 z_0	0 1 0 0 1	9



"Seat of the pants" design

- You just saw it!
- Can be inefficient:





longest path goes through 6 gates; that's slow

Systematic design

1. State purpose of circuit in words
2. Make truth tables
3. Identify "true" rows
4. Construct sum-of-products expression
5. Construct circuit

Systematic design of adder

1. State purpose of circuit in words
   - Inputs: carry-in, x, y
   - Outputs: z (if odd number of inputs are 1), carry-out (if at least two inputs are 1)
2. Make truth tables

Inputs			*Outputs*	
cin	x	y	z	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Systematic design of adder

3. Identify "true" rows

cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

cin	x	y	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

4. Construct sum-of-products expression (for each output)

$$z = \overline{cin}\,\overline{x}\,y + \overline{cin}\,x\,\overline{y} + cin\,\overline{x}\,\overline{y} + cin\,x\,y$$

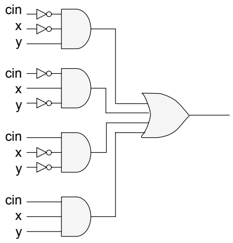
$$cout = \overline{cin}\,x\,y + cin\,\overline{x}\,y + cin\,x\,\overline{y} + cin\,x\,y$$

Systematic design of adder

5. Construct circuit

cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

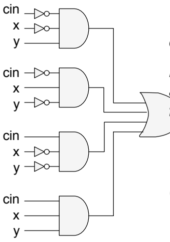
$$z = \overline{cin}\,\overline{x}\,y + \overline{cin}\,x\,\overline{y} + cin\,\overline{x}\,\overline{y} + cin\,x\,y$$



Sum-of-products circuit

cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$z = \overline{cin}\,\overline{x}\,y\ +\ \overline{cin}\,x\,\overline{y}\ +\ cin\,\overline{x}\,\overline{y} + \ cin\,x\,y$



One AND-gate for each 1-output in table

Each AND-gate has as many inputs as truth table

One OR-gate

Constant-depth: 2 (or 3, counting NOTs)

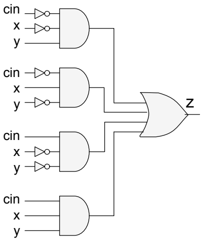
Finishing the adder

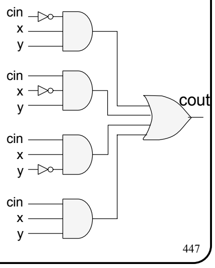
cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

cin	x	y	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$z = \overline{cin}\,\overline{x}\,y\ +\ \overline{cin}\,x\,\overline{y}\ +\ cin\,\overline{x}\,\overline{y} + \ cin\,x\,y$

$cout = \overline{cin}\,x\,y\ +\ cin\,\overline{x}\,y\ +\ cin\,x\,\overline{y} + \ cin\,x\,y$





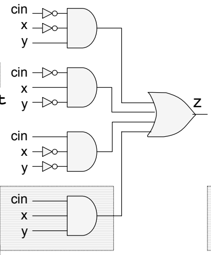
Duplicate terms, duplicate gates

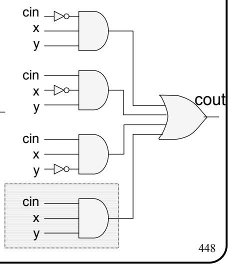
cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

cin	x	y	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$z = \overline{cin}\,\overline{x}\,y\ +\ \overline{cin}\,x\,\overline{y}\ +\ cin\,\overline{x}\,\overline{y} + \boxed{cin\,x\,y}$

$cout = \overline{cin}\,x\,y\ +\ cin\,\overline{x}\,y\ +\ cin\,x\,\overline{y} + \boxed{cin\,x\,y}$





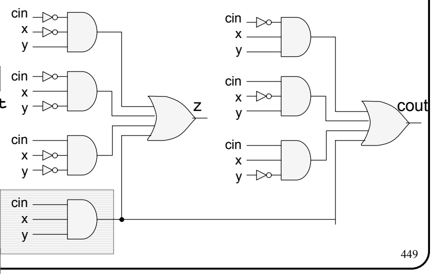
Duplicate terms, duplicate gates

cin	x	y	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

cin	x	y	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$z = \overline{cin}\,\overline{x}\,y\ +\ \overline{cin}\,x\,\overline{y}\ +\ cin\,\overline{x}\,\overline{y} + \boxed{cin\,x\,y}$

$cout = \overline{cin}\,x\,y\ +\ cin\,\overline{x}\,y\ +\ cin\,x\,\overline{y} + \boxed{cin\,x\,y}$



Metrics

- With *N* inputs, *M* outputs in truth table (and in circuit)
  - 2^N rows in table
  - Each AND gate has *N* inputs
  - At most 2^N AND gates total
  - *M* OR gates
  - Each OR gate has at most 2^N inputs

***N* Inputs**			***M* Outputs**	
cin	x	y	z	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

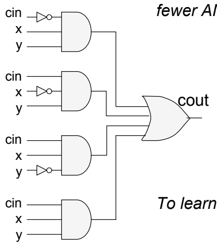
2^N rows

Advanced stuff

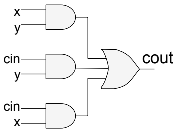
$cout = \overline{cin}\,x\,y\ +\ cin\,\overline{x}\,y\ +\ cin\,x\,\overline{y} + \ cin\,x\,y$

$cout = \ x\,y + cin\,y + cin\,x$

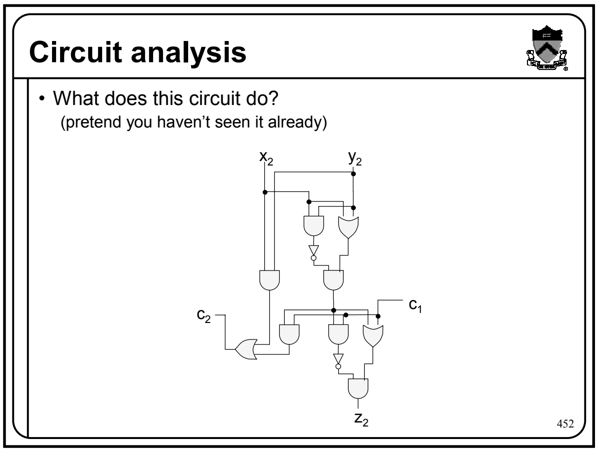
cin	x	y	cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Sometimes you can get by with fewer AND gates

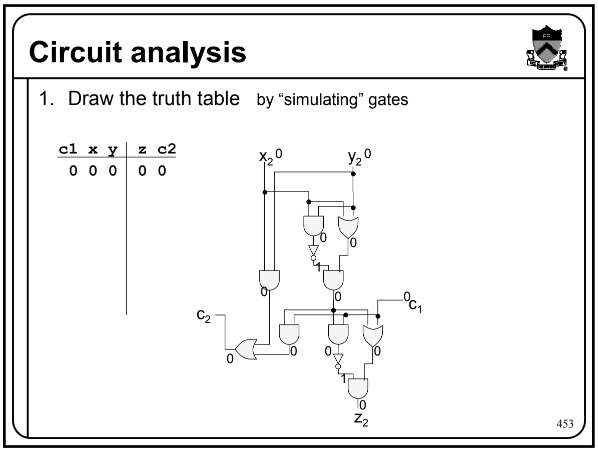


To learn how, take ELE 206!



Circuit analysis

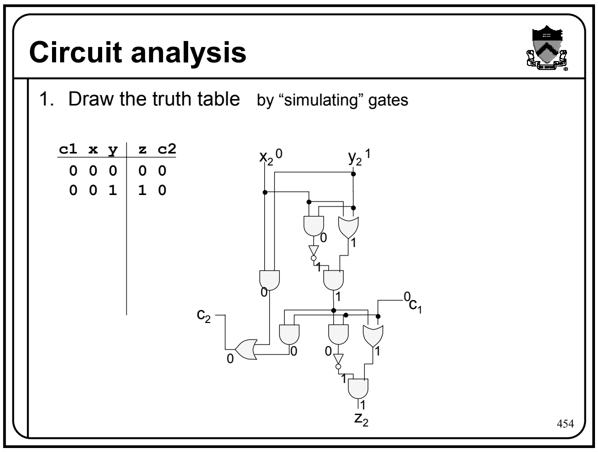
- What does this circuit do?
  (pretend you haven't seen it already)



Circuit analysis

1. Draw the truth table by "simulating" gates

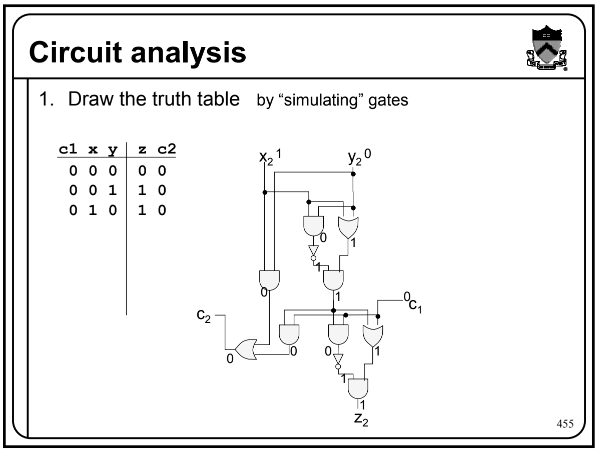
c1	x	y	z	c2
0	0	0	0	0



Circuit analysis

1. Draw the truth table by "simulating" gates

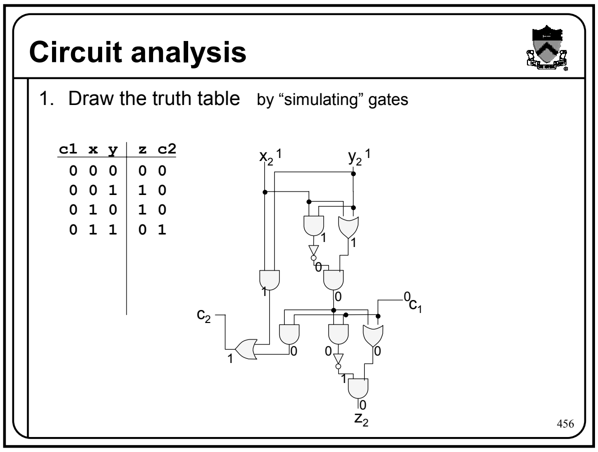
c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0



Circuit analysis

1. Draw the truth table by "simulating" gates

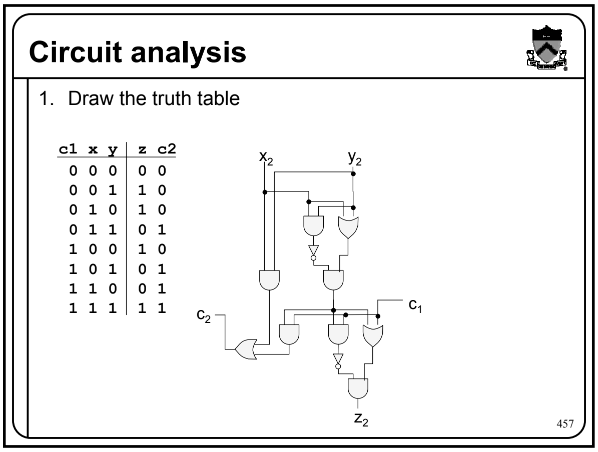
c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0



Circuit analysis

1. Draw the truth table by "simulating" gates

c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1



Circuit analysis

1. Draw the truth table

c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Circuit analysis

2. Say in words what the truth table does

c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

z is 1 if an odd number of inputs are 1

c2 is 1 if at least two inputs are 1

Circuit analysis

3. Apply a flash of insight

c1	x	y	z	c2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

z is 1 if an odd number of inputs are 1

c2 is 1 if at least two inputs are 1

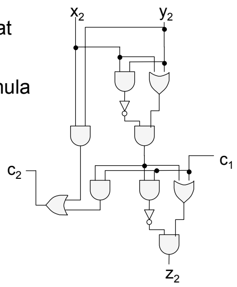
Aha! It's one bit-slice of an adder!

Sequential Circuits

CS 217

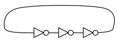
Combinational circuit

- Directed *acyclic* graph (no loops)
- Outputs, at any given time, dependent only on inputs at that time (after *signal propagation*)
- Equivalent to one boolean formula per output



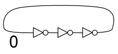
Cycles in the circuit

- What happens if there are cycles?



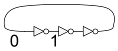
Cycles in the circuit

- Simulate . . .



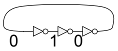
Cycles in the circuit

- Simulate . . .



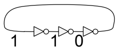
Cycles in the circuit

- Simulate . . .



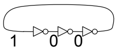
Cycles in the circuit

- Simulate . . .



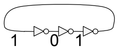
Cycles in the circuit

- Simulate . . .



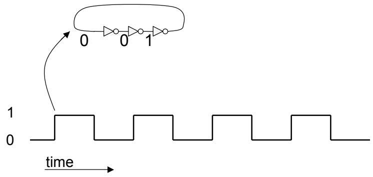
Cycles in the circuit

- Simulate . . .

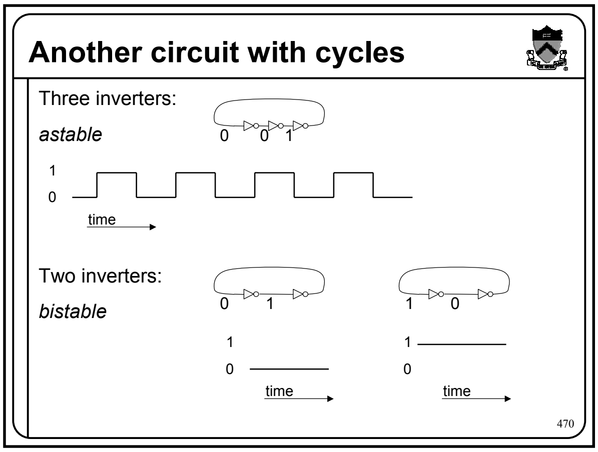


Cycles in the circuit

- Simulate . . .



Outputs, at any given time, dependent ***not*** only on inputs at that time;
also dependent on history. A “sequential” circuit.



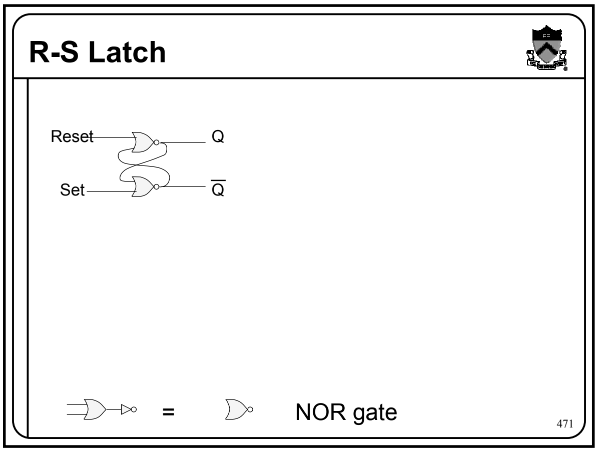
Another circuit with cycles

Three inverters:

astable

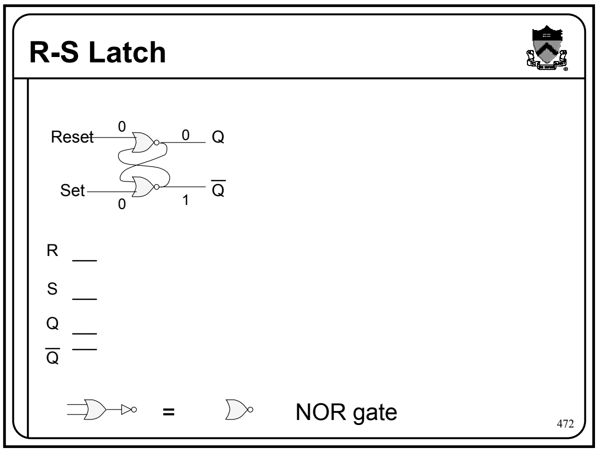
Two inverters:

bistable



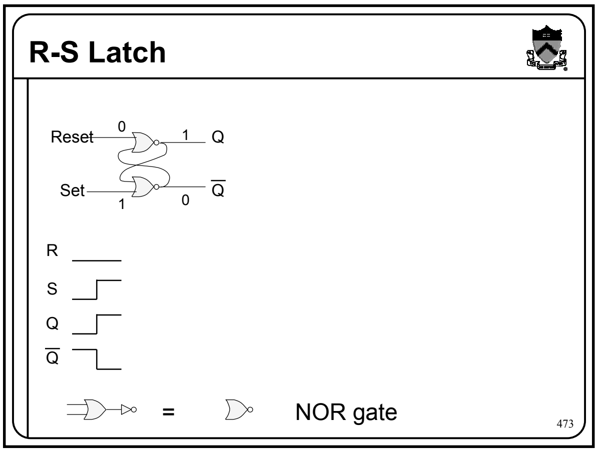
R-S Latch

= NOR gate



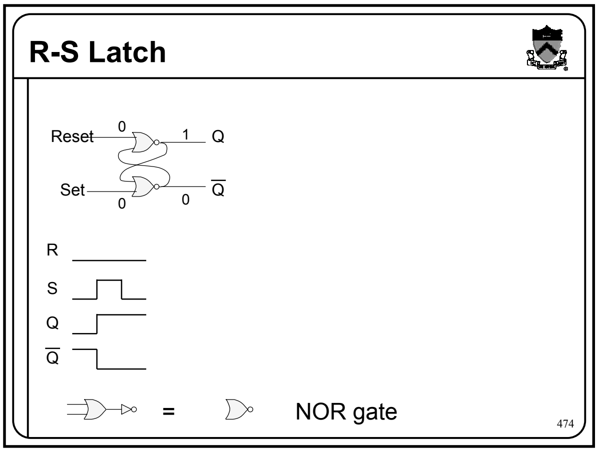
R-S Latch

= NOR gate



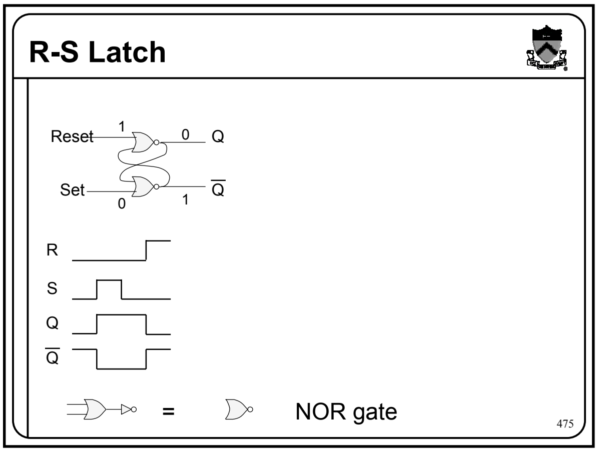
R-S Latch

= NOR gate



R-S Latch

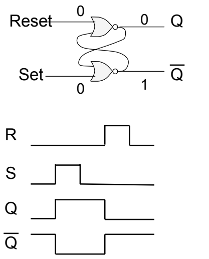
= NOR gate

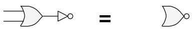


R-S Latch

= NOR gate

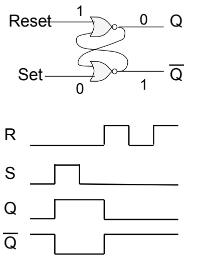
R-S Latch

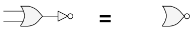




NOR gate

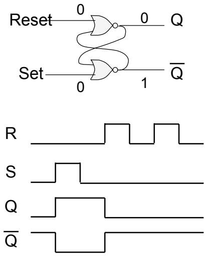
R-S Latch

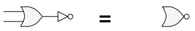




NOR gate

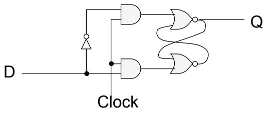
R-S Latch

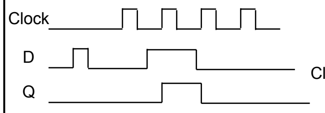




NOR gate

Clocked flipflop





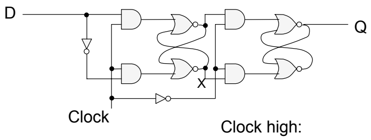
Clock high:

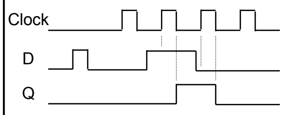
copy D to Q

Clock low:

ignore D, remember Q

Master/slave flipflop





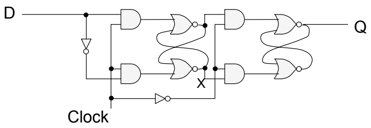
Clock high:

copy D to X; keep Q

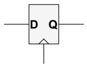
Clock low:

copy X to Q; keep X

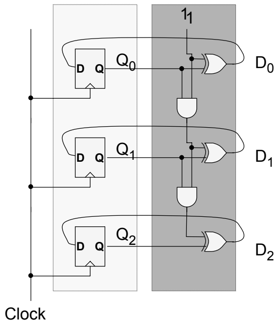
Master/slave flipflop



Circuit symbol:

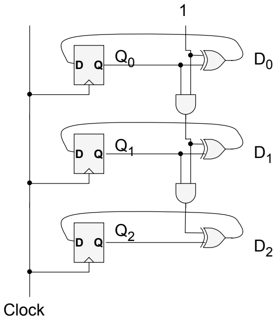


Syncronous sequential circuits



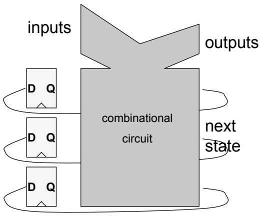
- Flipflops all clocked simultaneously
- Combinational circuit determines next flipflop values (calculates D's from Q's).

Analysis of sequential circuits



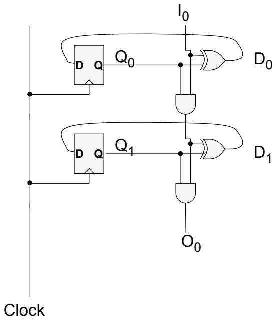
Q_2	Q_1	Q_0	D_2	D_1	D_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Input / Output



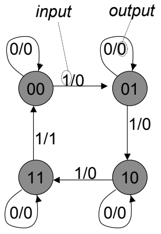
State	*Inputs*	*NextS*	*Outputs*
$Q_2Q_1Q_0$	I_2I_1	$D_2D_1D_0$	O_0
0 0 0	0 0	0 1 0	1
0 0 0	0 1	0 0 0	0
0 0 0	1 0	0 0 0	0
	.		
	.		
	.		
1 1 1	1 0	1 0 1	0
1 1 1	1 1	1 0 1	0

Circuit with I/O



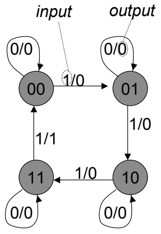
Q_1	Q_0	I_0	D_1	D_0	O_0
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	0	0	1

State Machine



Q_1	Q_0	I_0	D_1	D_0	O_0
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	0	0	1

What does it do?



Counts up if input=1;
stays stationary if input=0.

Output is "carry" when counter wraps around.

Synthesis procedure

1. State purpose of circuit in words
2. Make state machine
3. Make truth tables
4. Apply combinational-circuit synthesis procedure:
   - Identify "true" rows
   - Construct sum-of-products expression
   - Construct circuit

Vending machine

1. State purpose of circuit in words

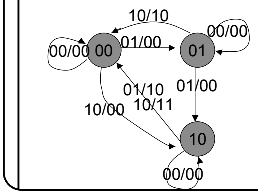
   Accept nickels and dimes

   Candy costs 15¢

   Dispense candy bar and appropriate change

   Inputs: D (dime), N (nickel)   Outputs: C (candy), O (nickel change)

2. Make state machine



States:

00: 0¢ credit

01: 5¢ credit

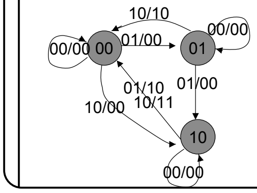
10: 10¢ credit

Vending machine

3. Make truth table

   Assume: D&N impossible

   "**x**" means don't-care



Q_1	Q_0	D	N	D_1	D_0	C	O
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	x	x	x	x
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	0
0	1	1	1	x	x	x	x
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Vending machine

4. Make sum-of-products expressions

$D_1 = \overline{Q_1}\,\overline{Q_0}D\overline{N} + \overline{Q_1}Q_0\overline{D}N + Q_1\overline{Q_0}\,\overline{D}\,\overline{N}$

Q_1	Q_0	D	N	D_1	D_0	C	O
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	x	x	x	x
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	0
0	1	1	1	x	x	x	x
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Vending machine

4. Make sum-of-products expressions

$D_1 = \overline{Q_1}\,\overline{Q_0}D\overline{N} + \overline{Q_1}Q_0\overline{D}N + Q_1\overline{Q_0}\,\overline{D}\,\overline{N}$

$D_0 = \overline{Q_1}\,\overline{Q_0}\,\overline{D}N + \overline{Q_1}Q_0\overline{D}\,\overline{N}$

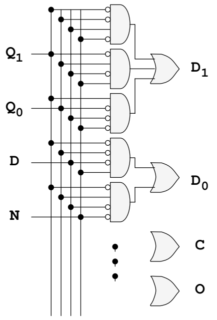
$C = \overline{Q_1}Q_0D\overline{N} + Q_1\overline{Q_0}\,\overline{D}N + Q_1\overline{Q_0}D\overline{N}$

$O = Q_1\overline{Q_0}D\overline{N}$

Q_1	Q_0	D	N	D_1	D_0	C	O
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	x	x	x	x
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	0
0	1	1	1	x	x	x	x
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Vending machine

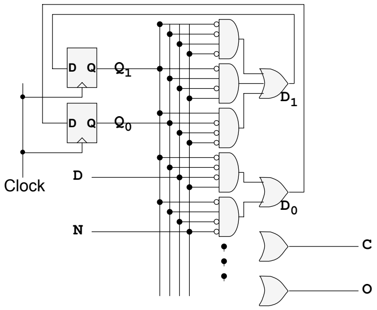
4. Make gates



Q_1	Q_0	D	N	D_1	D_0	C	O
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	x	x	x	x
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	1	0
0	1	1	1	x	x	x	x
1	0	0	0	1	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	1
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Vending machine

5. Hook up flipflops, clocks, inputs, outputs



Circuit Simulator

- Next programming assignment
- Inputs to simulator:
  - o description of sequential circuit
  - o boolean input values for each clock tick
- Simulate execution of gates and flipflops

SPECIFICATION LANGUAGE:

```
INPUT    d  n;                  names of input wires
FLIPFLOP  q1  q0;               names of state variables (D flipflops)
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;      transition rules
```

Representation of vending machine

$D_1 = \overline{Q_1}\,\overline{Q_0}D\overline{N} + \overline{Q_1}Q_0\overline{D}N + Q_1\overline{Q_0}\,\overline{D}\,\overline{N}$

$D_0 = \overline{Q_1}\,\overline{Q_0}\,\overline{D}N + \overline{Q_1}Q_0\overline{D}\,\overline{N}$

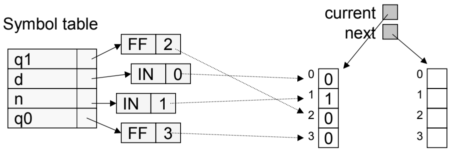
$C = \overline{Q_1}Q_0D\overline{N} + Q_1\overline{Q_0}\,\overline{D}N + Q_1\overline{Q_0}D\overline{N}$

$O = Q_1\overline{Q_0}D\overline{N}$

```
INPUT    d  n;                  names of input wires
FLIPFLOP  q1  q0;               names of state variables (D flipflops)
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;      transition rules
```

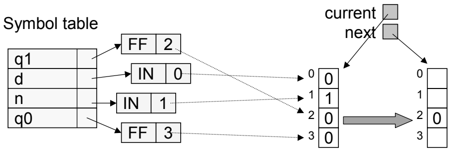
No way to represent output wires; that means less work for you, the implementor of the simulator.

Circuit Simulator



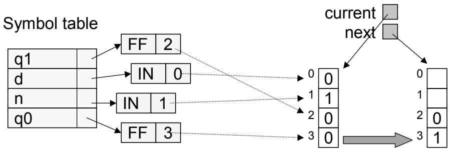
```
INPUT    d  n;                  names of input wires
FLIPFLOP  q1  q0;               names of state variables (D flipflops)
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;      transition rules
```

Evaluate NEXT expressions



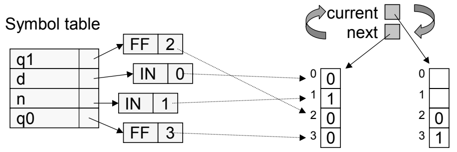
```
INPUT    d  n;
FLIPFLOP  q1  q0;
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;
```

Evaluate NEXT expressions



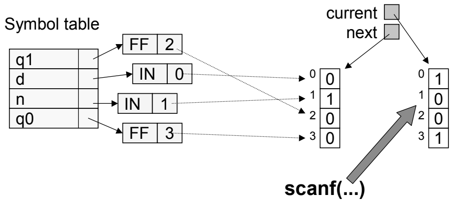
```
INPUT    d  n;
FLIPFLOP  q1  q0;
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;
```

Flip current, next



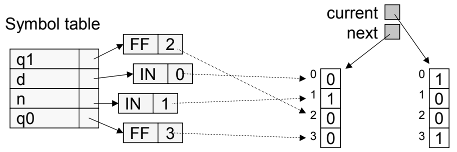
```
INPUT   d  n;
FLIPFLOP  q1  q0;
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;
```

Read more input data



```
INPUT   d  n;
FLIPFLOP  q1  q0;
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;
```

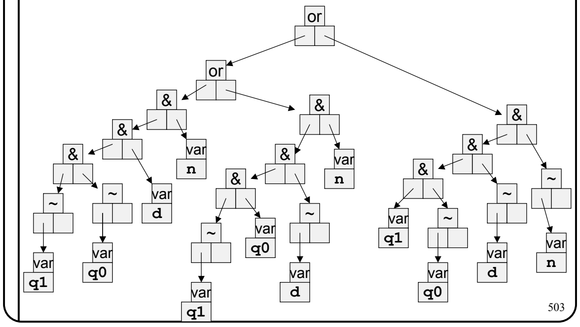
Ready for the next cycle!



```
INPUT   d  n;
FLIPFLOP  q1  q0;
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
NEXT q0 = ~q1&~q0&~d&n | ~q1&q0&~d&~n;
```

Representation of expressions

```
NEXT q1 = ~q1&~q0&d&n | ~q1&q0&~d&n | q1&~q0&~d&~n;
```



Components of simulator

- Parser
  - o translates circuit specification into trees, etc.
  - o We’ll give this to you
- Symbol table
  - o maps name (of input wires and flipflops) to binding (type and index)
  - o You should use your program from assignment #2
- Interpreter
  - o traverses tree, evaluates boolean expression
- Input reader
  - o Reads the time-dependent input data (not the circuit spec.)
- Output printer
- Control
  - o (flips the “current” and “next” pointers; calls the other modules)

Parsing

CS 217

Reading input

- Simplest case: use scanf
- Anything more complicated:
  parse using a grammar
- Separate lexical analysis from parsing
  - o Lexical analysis does “words”
  - o Parsing does “sentences”
- Build an Abstract Syntax Tree
  - o Separates parsing from semantic analysis

Example: Circuit Simulator

- Sample input:

```
INPUT x ;
FLIPFLOP A B ;
NEXT A   (~A & ~B & ~x)
        | (~A & B & x)
        | (A & ~B & x)
        | (A & B & ~x) ;
NEXT B (~A & ~B & ~x)
        | (~A & ~B & x)
        | (A & ~B & ~x)
        | (A & ~B & x) ;
```

- Grammar:

program ::= {stmt}*

stmt ::= INPUT ID+ ';'
stmt ::= FLIPFLOP ID+ ';'
stmt ::= NEXT ID expr ';'

expr ::= term { '|' term}*

term ::= factor { '&' factor}*

factor ::= '~' factor
factor ::= element

element ::= NUMBER
element ::= ID
element ::= '(' expr ')'

Input specification

- Lexical tokens

NUMBER_TOKEN	‘0’ or ‘1’
ID_TOKEN	An alphabetic character, followed by 0 or more alphanumeric characters. Underscore (‘_’) counts as alphabetic.
LPAREN_TOKEN	‘(‘
RPAREN_TOKEN	‘)’
AND_TOKEN	‘&’
OR_TOKEN	‘|’
NOT_TOKEN	‘~’
SEMI_TOKEN	‘;’
INPUT_TOKEN	‘INPUT’
NEXT_TOKEN	‘NEXT’
FLIPFLOP_TOKEN	‘FLIPFLOP’

- Grammar:

program ::= {stmt}*

stmt ::= INPUT ID^{+} ';'
stmt ::= FLIPFLOP ID^{+} ';'
stmt ::= NEXT ID expr ';'

expr ::= term { '|' term}*

term ::= factor { '&' factor}*

factor ::= '~' factor
factor ::= element

element ::= NUMBER
element ::= ID
element ::= '(' expr ')'

Parser

```
term() {
 factor();
 while (curTok==AND_TOKEN) {
   getToken();
   factor();
 }
}

factor() {
 if (curTok==NOT_TOKEN) {
   getToken();
   factor();
 } else
     element();
}

element() {
 switch (curTok) {
 case NUMBER_TOKEN:
   getToken(); break;
 case ID_TOKEN:
   getToken(); break;
 case LPAREN_TOKEN:
   getToken(); expr(); ...
}}
```

- Grammar:

program ::= {stmt}*

stmt ::= INPUT ID^{+} ';'
stmt ::= FLIPFLOP ID^{+} ';'
stmt ::= NEXT ID expr ';'

expr ::= term { '|' term}*

term ::= factor { '&' factor}*

factor ::= '~' factor
factor ::= element

element ::= NUMBER
element ::= ID
element ::= '(' expr ')'

Interface to Lexer

```
term() {
 factor();
 while (curTok==AND_TOKEN) {
   getToken();
   factor();
 }
}

factor() {
 if (curTok==NOT_TOKEN) {
   getToken();
   factor();
 } else
     element();
}

element() {
 switch (curTok) {
 case NUMBER_TOKEN:
   getToken(); break;
 case ID_TOKEN:
   getToken(); break;
 case LPAREN_TOKEN:
   getToken(); expr(); ...
}}
```

lexer.h

```
enum TokenType
{  NUMBER_TOKEN,
   IDENTIFIER_TOKEN,
   INPUT_TOKEN,
   . . .  };
enum TokenType curTok;
void getToken(void);
char *IDvalue;
int NUMvalue;
void lex_init(FILE *);
```

Implementation of lexer

- Parser has:
  - o curToken
  - o getToken()
- Lexer has:
  - o curChar
  - o getChr()

curToken is the “lookahead token”

curChr is the “lookahead character”

```
static FILE *infile;
static int curChar;
void getChr() {
  curChar = fgetc(infile);
}

void getToken() {
 skip white space;
 switch (curChar) {
  case ‘&’:
    getChr(); curTok=AND_TOKEN; return;
  case ‘|’:
   getChr(); curTok=OR_TOKEN; return;
  case ‘0’: case ‘1’: case ‘2’:...
  case ‘8’: case ‘9’:
  read digits;
  NUMvalue = the value that was read;
  curTok=NUM_TOKEN; return;
 . . .
}
```

Syntax tree grammar

- Abstract Syntax

program ::= program stmt
program ::= *(empty)*

stmt ::= INPUT ID+
stmt ::= FLIPFLOP ID+
stmt ::= NEXT ID expr

expr ::= expr AND expr
expr ::= expr OR expr
expr ::= NOT expr
expr ::= NUMBER
expr ::= ID

- Simpler than concrete syntax
  - Not useful for parsing
  - Very useful for trees

- Concrete Syntax

program ::= {stmt}*

stmt ::= INPUT ID+ ';'
stmt ::= FLIPFLOP ID+ ';'
stmt ::= NEXT ID expr ';'

expr ::= term { '|' term}*

term ::= factor { '&' factor}*

factor ::= '~' factor
factor ::= element

element ::= NUMBER
element ::= ID
element ::= '(' expr ')'

Syntax tree data structure

- Abstract Syntax

program ::= program stmt
program ::= *(empty)*

stmt ::= INPUT ID+
stmt ::= FLIPFLOP ID+
stmt ::= NEXT ID expr

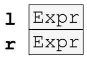
expr ::= expr AND expr
expr ::= expr OR expr
expr ::= NOT expr
expr ::= NUMBER
expr ::= ID

- Simpler than concrete syntax
  - Not useful for parsing
  - Very useful for trees

```
enum expr_kind {      AND_expr,
  OR_expr, NOT_expr, NUM_expr,
  ID_expr };

typedef struct expr *Expr;

struct expr {
 enum expr_kind kind;
 union {
  struct {Expr l, r;} and;
  struct {Expr l, r;} or;
  struct {Expr r;} not;
  struct {int value;} num;
  struct {char *name} id;
 u;
};
```

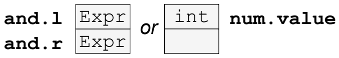
‘union’ feature of C

struct {Expr l, r;}



struct {int value;};

union { struct {Expr l,r;} and;
struct {int value;} num;};

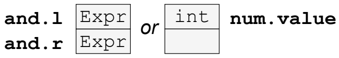


‘union’ access is unchecked

```
union {struct {Expr l,r;} and;
          struct {int value;} num;} u;

Expr ptr;
int i;

u.num.value = i;
ptr = u.and.l;
```

Stores an integer, fetches a pointer!

Undefined (and dangerous) behavior.



“kind” field tags the union

```
Expr p;
if (p->kind==AND_expr) {
  ... p->u.and.l ...
    ... p->u.and.r ...
} else
if (p->kind==NUM_expr) {
  ... p->u.num.value ...
} ...
```

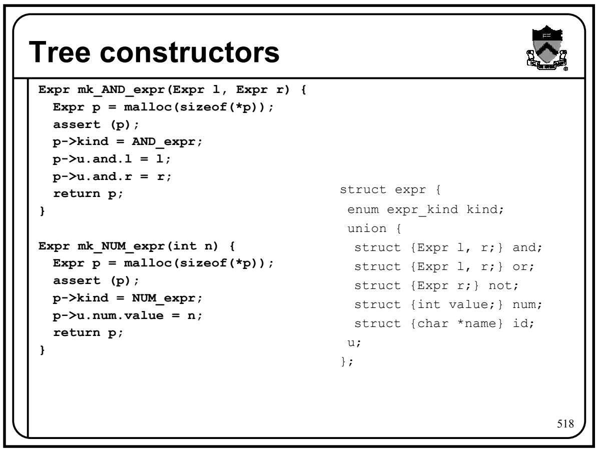
***switch** statement is useful here!*

```
enum expr_kind {      AND_expr,
  OR_expr, NOT_expr, NUM_expr,
  ID_expr };

typedef struct expr *Expr;

struct expr {
 enum expr_kind kind;
 union {
  struct {Expr l, r;} and;
  struct {Expr l, r;} or;
  struct {Expr r;} not;
  struct {int value;} num;
  struct {char *name} id;
 u;
};
```

Tree traversal

```
void visit(Expr p) {
 switch (p->kind) {
  case AND_expr:
   visit(p->u.and.l);
   visit(p->u.and.r);
  break;
 case OR_expr:
   visit(p->u.or.l);
   visit(p->u.or.r);
  break;
 case NOT_expr:
   visit(p->u.not.r);
  break;
 case NUM_expr:
   printNum(p->u.num.value);
  break;
 case ID_expr:
   puts(p->u.id.name);
  break;
 }
}
```

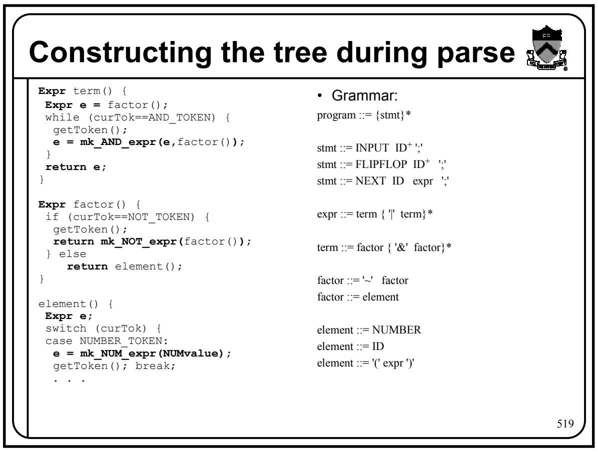
```
enum expr_kind {      AND_expr,
  OR_expr, NOT_expr, NUM_expr,
  ID_expr };

typedef struct expr *Expr;

struct expr {
 enum expr_kind kind;
 union {
  struct {Expr l, r;} and;
  struct {Expr l, r;} or;
  struct {Expr r;} not;
  struct {int value;} num;
  struct {char *name} id;
 u;
};
```



Tree constructors

```
Expr mk_AND_expr(Expr l, Expr r) {
  Expr p = malloc(sizeof(*p));
  assert (p);
  p->kind = AND_expr;
  p->u.and.l = l;
  p->u.and.r = r;
  return p;
}

Expr mk_NUM_expr(int n) {
  Expr p = malloc(sizeof(*p));
  assert (p);
  p->kind = NUM_expr;
  p->u.num.value = n;
  return p;
}
```

```
struct expr {
 enum expr_kind kind;
 union {
  struct {Expr l, r;} and;
  struct {Expr l, r;} or;
  struct {Expr r;} not;
  struct {int value;} num;
  struct {char *name} id;
 u;
};
```



Constructing the tree during parse

```
Expr term() {
 Expr e = factor();
 while (curTok==AND_TOKEN) {
  getToken();
  e = mk_AND_expr(e,factor());
 }
 return e;
}

Expr factor() {
 if (curTok==NOT_TOKEN) {
  getToken();
  return mk_NOT_expr(factor());
 } else
    return element();
}

element() {
 Expr e;
 switch (curTok) {
 case NUMBER_TOKEN:
  e = mk_NUM_expr(NUMvalue);
  getToken(); break;
  . . .
```

- Grammar:

program ::= {stmt}*

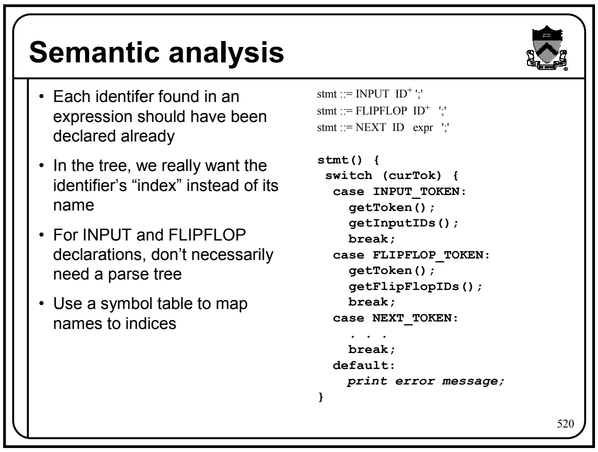
stmt ::= INPUT ID$^+$ ';'
stmt ::= FLIPFLOP ID$^+$ ';'
stmt ::= NEXT ID expr ';'

expr ::= term { '|' term}*

term ::= factor { '&' factor}*

factor ::= '~' factor
factor ::= element

element ::= NUMBER
element ::= ID
element ::= '(' expr ')'

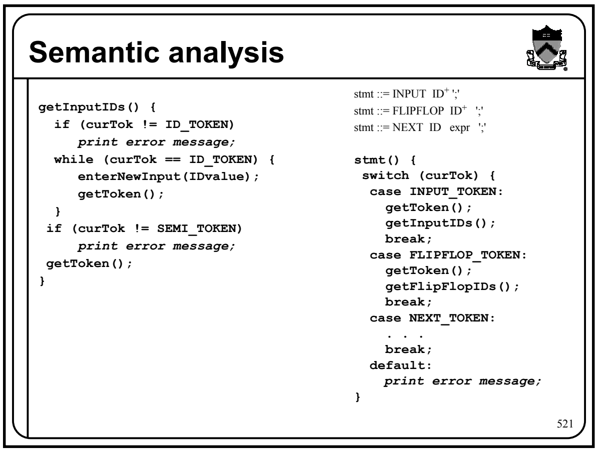


Semantic analysis

- Each identifer found in an expression should have been declared already
- In the tree, we really want the identifier's "index" instead of its name
- For INPUT and FLIPFLOP declarations, don't necessarily need a parse tree
- Use a symbol table to map names to indices

stmt ::= INPUT ID$^+$ ';'
stmt ::= FLIPFLOP ID$^+$ ';'
stmt ::= NEXT ID expr ';'

```
stmt() {
 switch (curTok) {
  case INPUT_TOKEN:
    getToken();
    getInputIDs();
    break;
  case FLIPFLOP_TOKEN:
    getToken();
    getFlipFlopIDs();
    break;
  case NEXT_TOKEN:
    . . .
    break;
  default:
    print error message;
}
```

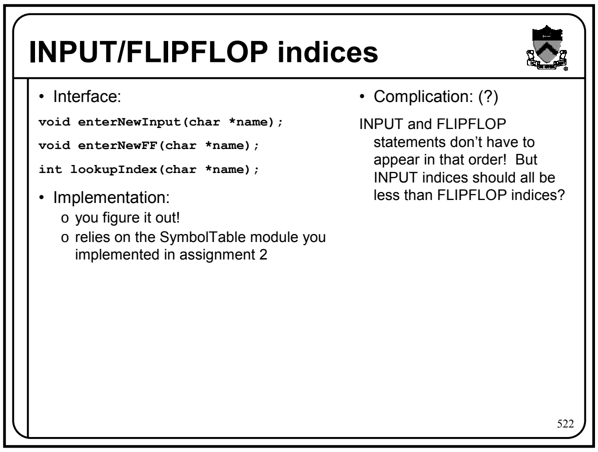


Semantic analysis

```
getInputIDs() {
  if (curTok != ID_TOKEN)
     print error message;
  while (curTok == ID_TOKEN) {
     enterNewInput(IDvalue);
     getToken();
  }
 if (curTok != SEMI_TOKEN)
     print error message;
 getToken();
}
```

stmt ::= INPUT ID$^+$ ';'
stmt ::= FLIPFLOP ID$^+$ ';'
stmt ::= NEXT ID expr ';'

```
stmt() {
 switch (curTok) {
  case INPUT_TOKEN:
    getToken();
    getInputIDs();
    break;
  case FLIPFLOP_TOKEN:
    getToken();
    getFlipFlopIDs();
    break;
  case NEXT_TOKEN:
    . . .
    break;
  default:
    print error message;
}
```



INPUT/FLIPFLOP indices

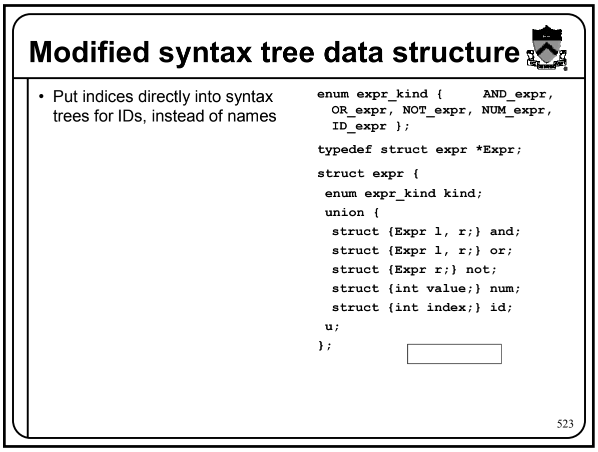
- Interface:

```
void enterNewInput(char *name);
void enterNewFF(char *name);
int lookupIndex(char *name);
```

- Implementation:
  - o you figure it out!
  - o relies on the SymbolTable module you implemented in assignment 2

- Complication: (?)

INPUT and FLIPFLOP statements don't have to appear in that order! But INPUT indices should all be less than FLIPFLOP indices?



Modified syntax tree data structure

- Put indices directly into syntax trees for IDs, instead of names

```
enum expr_kind {     AND_expr,
  OR_expr, NOT_expr, NUM_expr,
  ID_expr };
typedef struct expr *Expr;
struct expr {
 enum expr_kind kind;
 union {
  struct {Expr l, r;} and;
  struct {Expr l, r;} or;
  struct {Expr r;} not;
  struct {int value;} num;
  struct {int index;} id;
 u;
};
```

Summary

- Separate parsing from lexical analysis
- Use concrete syntax for parsing, abstract syntax for tree-building
- Use tagged-union datatype for abstract syntax trees
- Make "constructor functions" to malloc and initialize
- Call constructor functions from parser
- Parsing *declarations* adds information to symbol table
- Parsing *expressions* uses symbol-table information
- Further reading: any good compiler textbook
  - o e.g., *Modern Compiler Implementation* by Andrew Appel.
- Further study: COS 320, "Compiling Techniques"

Building computers from digital circuits

CS 217

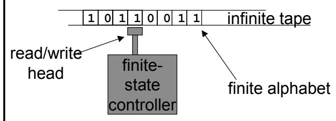
Let's build a computer

- Need:
  - o Memory to hold program
  - o Memory to hold data
  - o Control circuitry
  - o Input mechanism
  - o Output mechanism
- Other requirements:
  - o Must be extremely simple, so it fits on these slides
  - o Must be expressed using limited set of circuit components, so it can run in your simulator
  - o Must be powerful enough to (in principle) do any computation

Turing Machine

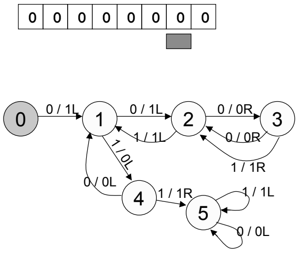
- Named after Alan M. Turing (1912-1954)
  - o First "computer scientist" in the world
  - o Ph.D. Princeton University 1938
- Invented in 1936

  for purpose of explaining what computers could and couldn't do
  (the computers themselves weren't invented until 1940's!)

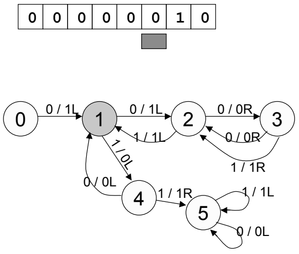


- Still useful in computer science theory today

An example Turing machine

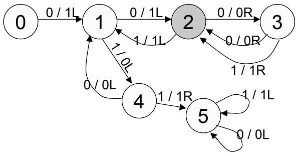


An example Turing machine



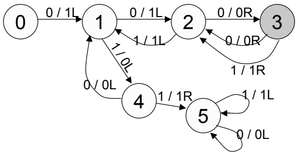
An example Turing machine

0	0	0	0	0	1	1	0



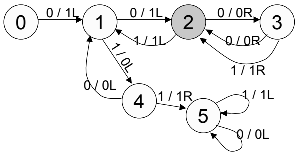
An example Turing machine

0	0	0	0	0	1	1	0



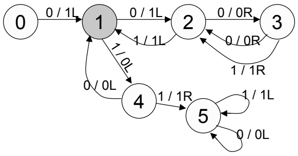
An example Turing machine

0	0	0	0	0	1	1	0



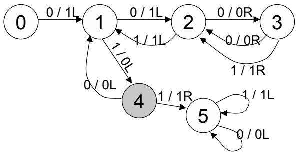
An example Turing machine

0	0	0	0	0	1	1	0



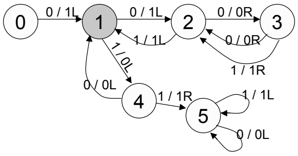
An example Turing machine

0	0	0	0	0	0	1	0

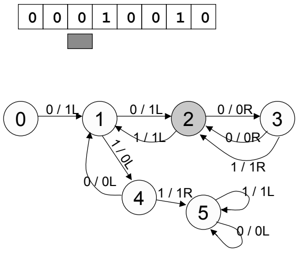


An example Turing machine

0	0	0	0	0	0	1	0

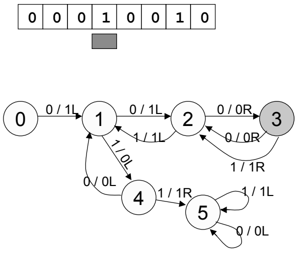


An example Turing machine



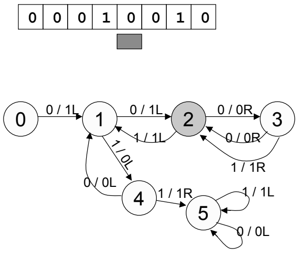
0	0	0	1	0	0	1	0
		▨					

An example Turing machine



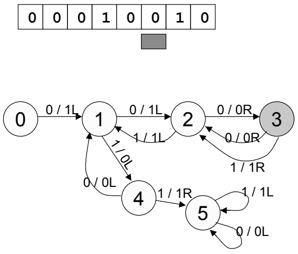
0	0	0	1	0	0	1	0
			▨				

An example Turing machine



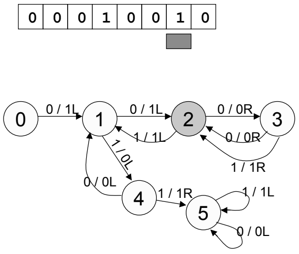
0	0	0	1	0	0	1	0
				▨			

An example Turing machine



0	0	0	1	0	0	1	0
					▨		

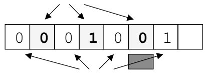
An example Turing machine



0	0	0	1	0	0	1	0
						▨	

What does it do?

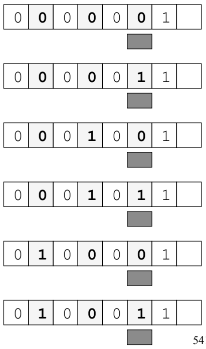
the blue cells are a binary number



0	**0**	0	**1**	0	**0**	1	
					▨		

the white cells say
where the number ends

Look at the tape every time the head is in this position...



0	**0**	0	**0**	0	**0**	1	
					▨		

0	**0**	0	**0**	0	**1**	1	
					▨		

0	**0**	0	**1**	0	**0**	1	
					▨		

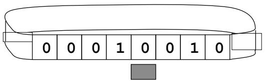
0	**0**	0	**1**	0	**1**	1	
					▨		

0	**1**	0	**0**	0	**0**	1	
					▨		

0	**1**	0	**0**	0	**1**	1	
					▨		

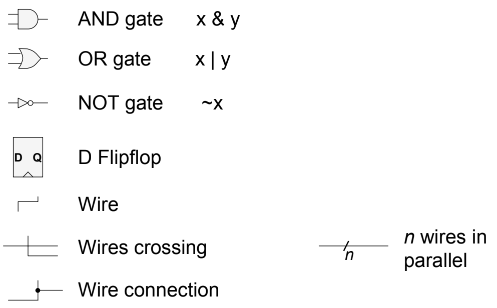
A Turing Machine circuit

- A tape of 1's and 0's is not hard to implement
  - o However, an *infinite* tape is a challenge!
  - o Solution: a circular tape of *N* cells



- Instead of moving the head, move the tape
  - o (That's what your cassette player does anyway!)
- Don't move the tape, move the data
  - o Result: a *shift register*

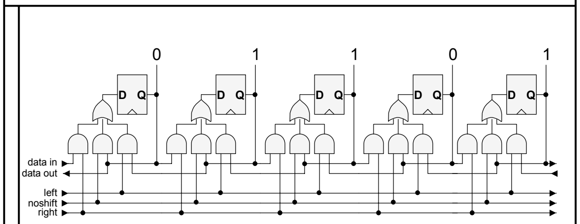
Circuit components



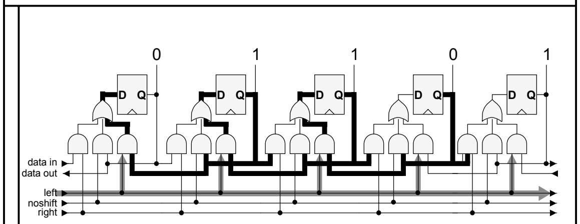
AND gate	x & y
OR gate	x | y
NOT gate	~x
D Flipflop	
Wire	
Wires crossing	
Wire connection	

n wires in parallel

Shift register

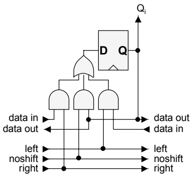


Shift left

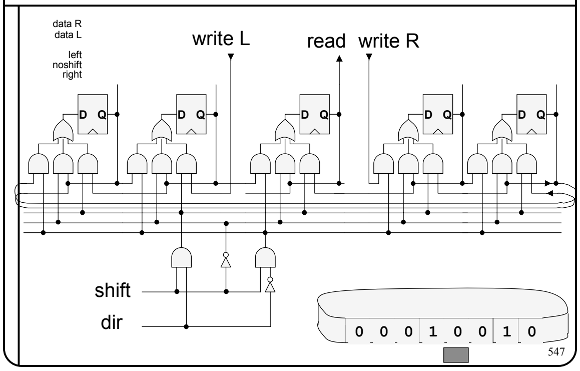


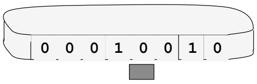
Shift register cell

- Need one copy of this for each cell on the tape



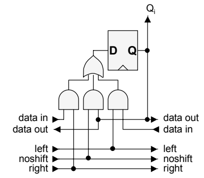
Circular "tape" with head



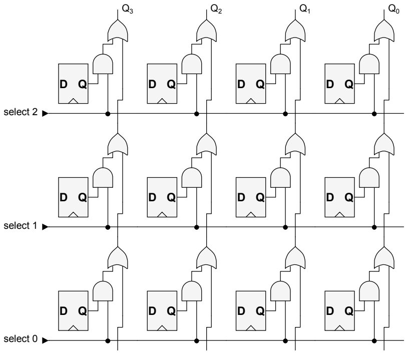


Shift register cell

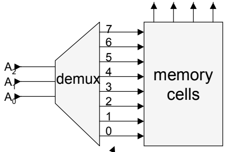
- Need one copy of this for each cell on the tape



Read-only memory cells



Memory addressing

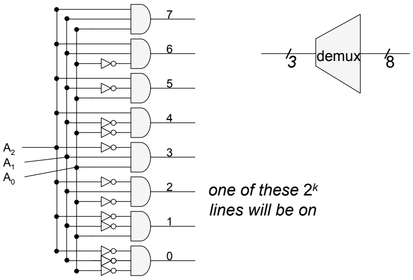


n-bit word output

k-bit binary number

one of these 2^k lines will be on

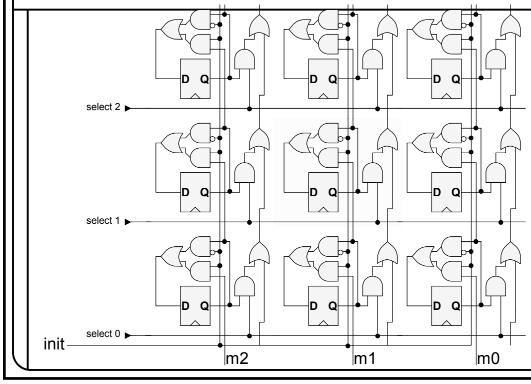
Demultiplexer (demux)



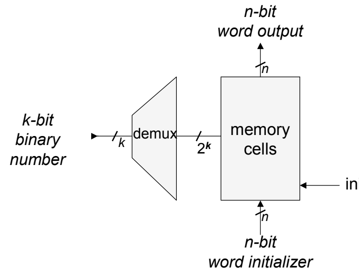
k-bit binary number

one of these 2^k lines will be on

Shifting in the ROM data



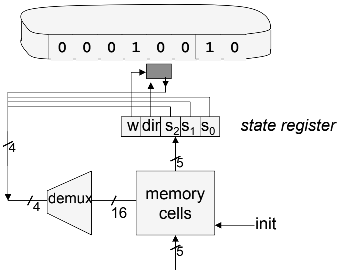
shift-PROM memory



Note: output is random-access, input is sequential.

An ordinary RAM memory has random-access input and output.

Turing machine



Circuit specification for T.M.

```
INPUT init data  wr lr n2 n1 n0 ;

FLIPFLOP c1 c0 w dir
s2 s1 s0                                (three state bits)
t8 t7 t6 t5 t4 t3 t2 t1 t0               (nine tape cells)
d15 d14 d13 d12 d11 d10 d9 d8 d7 d6 d5 d4 d3 d2 d1 d0  (demux)
m0x4 m0x3 m0x2 m0x1 m0x0
m1x4 m1x3 m1x2 m1x1 m1x0
m2x4 m2x3 m2x2 m2x1 m2x0                (16x5 memory cells)
m3x4 m3x3 m3x2 m3x1 m3x0
      .
      .
      .
m13x4 m13x3 m13x2 m13x1 m13x0
m14x4 m14x3 m14x2 m14x1 m14x0
m15x4 m15x3 m15x2 m15x1 m15x0
;
```

Transitions for memory cells

```
NEXT m0x0   init & n0 | ~init & m0x0;
NEXT m0x1   init & n1 | ~init & m0x1;
NEXT m0x2   init & n2 | ~init & m0x2;
NEXT m0x3   init & lr | ~init & m0x3;
NEXT m0x4   init & wr | ~init & m0x4;
NEXT m1x0   init & m0x0 | ~init & m1x0;
NEXT m1x1   init & m0x1 | ~init & m1x1;
NEXT m1x2   init & m0x2 | ~init & m1x2;
NEXT m1x3   init & m0x3 | ~init & m1x3;
NEXT m1x4   init & m0x4 | ~init & m1x4;
NEXT m2x0   init & m1x0 | ~init & m2x0;
NEXT m2x1   init & m1x1 | ~init & m2x1;
NEXT m2x2   init & m1x2 | ~init & m2x2;
NEXT m2x3   init & m1x3 | ~init & m2x3;
NEXT m2x4   init & m1x4 | ~init & m2x4;
```

Transitions for demux

```
NEXT d0   ~t0 & ~s0 & ~s1 & ~s2;
NEXT d1    t0 & ~s0 & ~s1 & ~s2;
NEXT d2   ~t0 &  s0 & ~s1 & ~s2;
NEXT d3    t0 &  s0 & ~s1 & ~s2;
NEXT d4   ~t0 & ~s0 &  s1 & ~s2;
NEXT d5    t0 & ~s0 &  s1 & ~s2;
NEXT d6   ~t0 &  s0 &  s1 & ~s2;
NEXT d7    t0 &  s0 &  s1 & ~s2;
NEXT d8   ~t0 & ~s0 & ~s1 &  s2;
NEXT d9    t0 & ~s0 & ~s1 &  s2;
NEXT d10   ~t0 &  s0 & ~s1 &  s2;
NEXT d11    t0 &  s0 & ~s1 &  s2;
NEXT d12   ~t0 & ~s0 &  s1 &  s2;
NEXT d13    t0 & ~s0 &  s1 &  s2;
NEXT d14   ~t0 &  s0 &  s1 &  s2;
NEXT d15    t0 &  s0 &  s1 &  s2;
```

Why does demux have flipflops in it?

Because this circuit-specification language is too barebones (stripped down) to be able to specify common subexpressions without flipflops.

Transitions for tape

```
NEXT t0  (c0 & c1) & (~dir & t8
  | dir & t1) | init & data |
  ~init & ~(c0 & c1) & t0;
NEXT t1  (c0 & c1) & (~dir & w |
  dir & t2) | ~(c0 & c1) & t1;
NEXT t2  (c0 & c1) & (~dir & t1
  | dir & t3) | ~(c0 & c1) & t2;
NEXT t3  (c0 & c1) & (~dir & t2
  | dir & t4) | ~(c0 & c1) & t3;
NEXT t4  (c0 & c1) & (~dir & t3
  | dir & t5) | ~(c0 & c1) & t4;
NEXT t5  (c0 & c1) & (~dir & t4
  | dir & t6) | ~(c0 & c1) & t5;
NEXT t6  (c0 & c1) & (~dir & t5
  | dir & t7) | ~(c0 & c1) & t6;
NEXT t7  (c0 & c1) & (~dir & t6
  | dir & t8) | ~(c0 & c1) & t7;
NEXT t8  (c0 & c1) & (~dir & t7
  | dir & w) | ~(c0 & c1) & t8;
```

Q: What is **`(c0 & c1)`** ?

Answer: we want tape-shift to happen only at controlled times.

Transitions for state register

```
NEXT s0
   ~(c0 & c1) & s0
 | (c0 & c1)
   & (d0 & m0x0
     | d1 & m1x0
     | d2 & m2x0
     | d3 & m3x0
     | d4 & m4x0
     | d5 & m5x0
     | d6 & m6x0
       . . .
     | d15 & m15x0 );
```

This is really the “guts” of the ROM memory readout.

s1, s2, dir, w are similar.

Transitions for controller

```
NEXT c0   ~init & ~c1;
NEXT c1   ~init & ~c1 & c0;
```

What does this do?

Transitions for controller

```
NEXT c1   ~init & ~c1 & c0;
NEXT c0   ~init & ~c1;
```

To analyze circuit,

First, build transition table

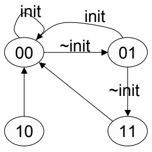
			NEXT	
init	c1	c0	c1	c0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

Transitions for controller

```
NEXT c1   ~init & ~c1 & c0;
NEXT c0   ~init & ~c1;
```

Next, diagram the state machine

			NEXT	
init	c1	c0	c1	c0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

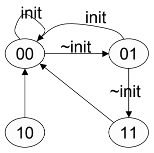


Transitions for controller

```
NEXT c1   ~init & ~c1 & c0;
NEXT c0   ~init & ~c1;
```

Finally, explain it in words

			NEXT	
init	c1	c0	c1	c0
0	0	0	0	1
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

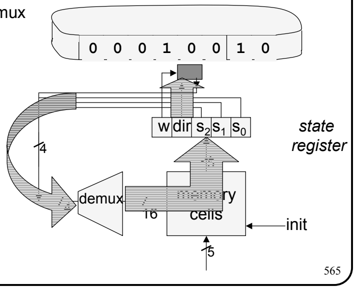


“Resets to zero if init;

counts mod 3 (sort of) if not init.”

Why count by threes?

- It takes three clock ticks for data to propagate
  - o memory into state-register
  - o state register (dir) into tape
  - o tape (read t0) into demux



Assemblers

CS 217

Compilation Pipeline

- Compiler (`gcc`): `.c` → `.s`
  - **translates high-level language to assembly language**
- Assembler (`as`): `.s` → `.o`
  - **translates assembly language to machine language**
- Archiver (`ar`): `.o` → `.a`
  - **collects object files into a single library**
- Linker (`ld`): `.o` + `.a` → `a.out`
  - **builds an executable file from a collection of object files**
- Execution (`execlp`)
  - **loads an executable file into memory and starts it**

Assembly Language

- A symbolic representation of machine instructions
- Assemblers translate assembly language into object code
- Object code contains everything needed to link, load, and execute the program

Translating to machine code

- Assembly language: **`addcc %r3, %r7, %r2`**
  `addcc %r3,  1000, %r2`
- Format of arithmetic instructions:

10	rd	op3	rs1	0	unused(0)	rs2
31	29	24	18	13	12	4 0

10	rd	op3	rs1	1	simm13
31	29	24	18	13	12 0

10	00010	010000	00011	0	00000000	00111
31	29	24	18	13	12	4 0

10	00010	010000	00011	1	0001111101000
31	29	24	18	13	12 0

Packing fields using C

- Assembly language: **`addcc %r3, %r7, %r2`**
- Format of arithmetic instructions:

10	rd	op3	rs1	0	unused(0)	rs2
31	29	24	18	13	12	4 0

rd = 2; op3 = 16; rs1 = 3; rs2 = 7;

w = (2<<29) | (rd<<24) | (op3<<18) | (0<<13) | (0<<4) | (rs2<<0) ;

10	00010	010000	00011	0	00000000	00111
31	29	24	18	13	12	4 0

In C language, you can also use the “bit field” feature.

Assembly Language (cont)

- Assembly language statements…
  - **imperative statements specify instructions; typically map 1 imperative statement to 1 machine instruction**
  - **some assemblers provide synthetic instructions that are mapped to one or more machine instructions**
  - **declarative statements specify *assembly time* actions; e.g., reserve space, define symbols, identify segments, and initialize data (they do not yield machine instructions but they may add information to the object file that is used by the linker)**

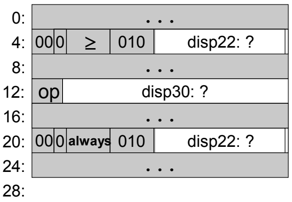
Assembler

- Most important function: symbol manipulation
  - create labels and remember their addresses
- Forward reference problem

```
loop: cmp i,n          .seg   "text"
      bge done                set count,%l0
      nop                     ...
      ...              .seg   "data"
      inc i            count: .long 0
done:
```

Example assembly

```
      .extern f (?)
      .global loop
loop: cmp %r16,%r24
      bge done
      nop
      call f
      nop
      ba loop
      inc %r16
done:
```



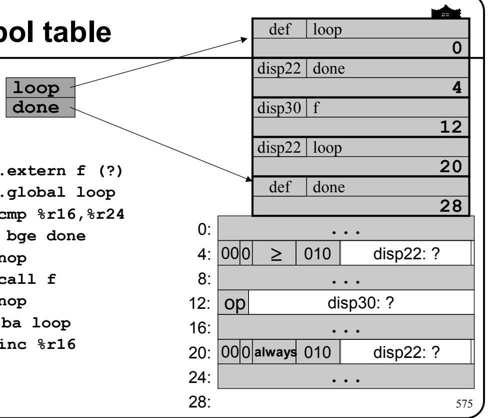
Dealing with forward references

- Most assemblers have two passes
  - Pass 1: symbol definition
  - Pass 2: instruction assembly
- Or, alternatively,
  - Pass 1: instruction assembly
  - Pass 2: patch the cross-references

o I will illustrate this technique

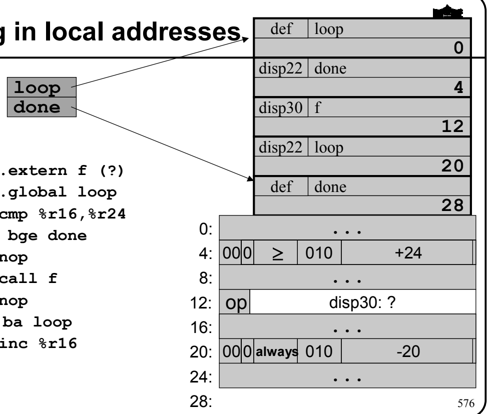
Symbol table

```
      .extern f (?)
      .global loop
loop: cmp %r16,%r24
       bge done
      nop
      call f
      nop
       ba loop
      inc %r16
done:
```



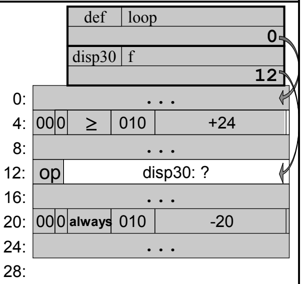
Filling in local addresses

```
      .extern f (?)
      .global loop
loop: cmp %r16,%r24
       bge done
      nop
      call f
      nop
       ba loop
      inc %r16
done:
```



Relocation records

```
      .extern f (?)
      .global loop
loop: cmp %r16,%r24
      bge done
      nop
      call f
      nop
      ba loop
      inc %r16
done:
```



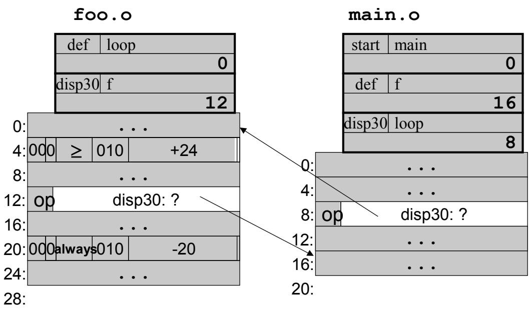
Assembler directives

- Delineate segments
  - .section
  - **may need multiple location counters (one per segment)**
- Allocate/initialize data and bss segments

  .word .half .byte
  .ascii .asciz
  .align .skip
- Make symbols in text externally visible

  .global

Linker



Invoking the linker

- **ld foo.o main.o** -l libc.a -o a.out

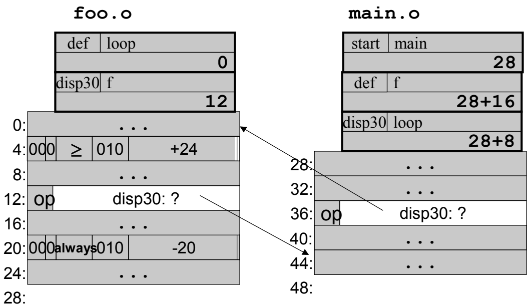
  compiled program modules (**ld foo.o main.o**)

  library (contains more .o files) (-l libc.a)

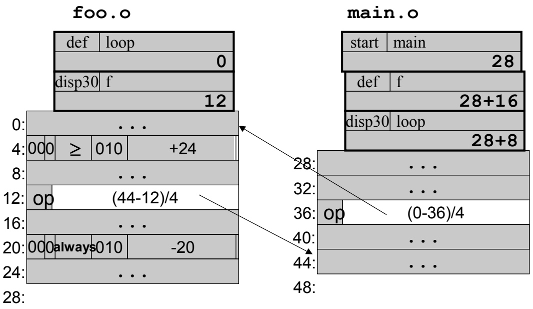
  output (also in ".o" format, but no undefined symbols) (-o a.out)

- Invoked automatically by gcc,
- but you can call it directly if you like.

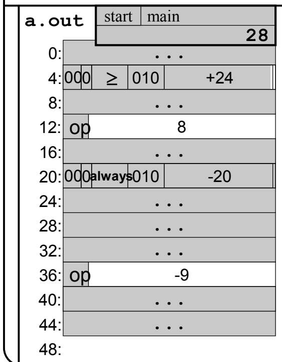
Step 1: pick an order



Step 2: patch



Step 3: concatenate



Can delete most relocation information, or save it for use in debugging.

How to Cheat

CS 217

A contest problem

COS 333 PROGRAMMING CONTEST #1: QUICKROOT

From: Dr. Guy Jacobson
Due: 12:01 A.M. 13 February 1995

Your boss at Yoyodyne laboratories has discovered that a critical slowdown in their mission software is due to a routine that calculates integer cube roots:

```
double cbrt (double);
int quickroot (int i)      {
      return (int) cbrt ((double) i);
}
```

You must rewrite this routine so that it is faster. Much faster. Your boss insists that you give him the following by 13 February:

1. A single file "quickroot.c" that implements the function quickroot.
2. A short (1-2 page) description of how your function works.

A contest problem (cont'd)

Furthermore, he insists that:

a. Your function return, for any non-negative integer $0 \le n < 2^{31}$, the greatest integer not greater than the cube root of n, just like the old slow quickroot().

b. Your function must be in a single .c file of ≤ 5000 characters.

c. Your function have no other externally-visible side effects, except perhaps for allocating memory.

Other than that, all he cares about is speed. Raw, blinding speed. He says that he's going to test your program on arizona, compiling with gcc and using `time(1)` to measure user time, by linking with the following driver program:

A contest problem (cont'd)

```
#include <stdio.h>
main (int ac, char *av[])  {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
           j = quickroot (random ());
   }
```

He won't tell you what number he's going to use as a random seed to srandom,

You are in competition with all the other engineers here at Yoyodyne. Your grade will depend primarily on the speed of your function as measured above (and, of course, its correctness), ranked against to the speed of all the other entries.. The fastest entries get special recognition as well.

No excuses, and good luck.

How to solve it?

- A quick hack:

```
 int quickroot (int i)      {
          return 0;
}
```

- Blindingly fast when used with driver.c:

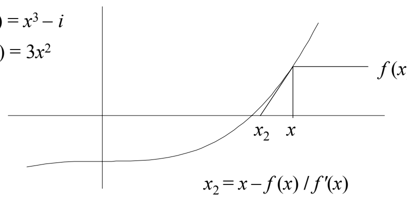
```
#include <stdio.h>
main (int ac, char *av[])   {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
           j = quickroot (random ());
}
```

- Unfortunately, violates rule (a), that `quickroot` must be correct in any context, not just this driver.
- However, quickroot need not be *fast* in all contexts...

Newton's method

$f(x) = x^3 - i$

$f'(x) = 3x^2$



$x_2 = x - f(x) / f'(x)$

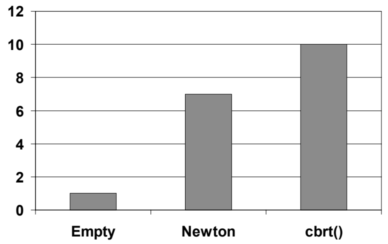
$x_2 = x - (x^3 - i) / (3x^2) = {}^1/_3(2x + i/x^2)$

```
for (n=0;n<7;n++) {
   x = (1/3.0)*(x+x+i/(x*x));
 }
```

Picking a good start point

```
  if (i > 1<<15)
    if (i > 1<<24)
      if (i > 1<<27)
           x = 1<<9;
      else x = 1<<8;
    else if (i > 1<<18)
      if (i > 1<<21)
           x = 1<<7;
      else x = 1<<6;
    else x = 1<<5;
  else if (i > 1<<9)
    if (i > 1<<12)
         x = 1<<4;
    else x = 1<<3;
  else x = 1<<2;

for (n=0;n<7;n++) {
    x = (1/3.0)*(x+x+i/(x*x));
  }
```

Results:



Reduce time from 10 seconds down to 7 seconds.

Question: cbrt( ) uses Newton’s method too; why the improvement?

Winning at all costs

- Let’s use programming-languages theory
  - o A *continuation* is a theoretical representation of “the rest of the execution of the program”
  - o Use continuation transform to write quickroot as,
    quickroot(i, k) = k($\sqrt[3]{i}$)
  - o Invent all-new, special purpose equality-test operator for continuations: ;=)
    k1 ;=) k2   means, continuation k1 has same structure as k2

```
quickroot(i,k) =
  if  k  ;=)  driver_main
     then $k_0(0)$
     else k($\sqrt[3]{i}$)
```

Claim: driver_main($\sqrt{i}$) = $k_0(0)$

My quickroot.c

```
#include <stdio.h>
mainX (int ac, char *av[])   {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
           j = quickroot (random ());
   }
endMain(){}

double cbrt (double);
extern main();
unsigned mycaller[] ={0x81c3e008,0x9010001f};
```

My quickroot.c

```
int quickroot(int i) {
 static x=0;
 if (x) return (int) cbrt ((double) i);
 x=1;


   here is the special hack . . .
}
```

The special hack in quickroot.c

```
unsigned *p, *q, caller;
 union {unsigned *z; unsigned (*f)();} u;
 u.z=mycaller; caller = u.f();
 if (caller <= (unsigned)main ||
     caller >= (unsigned)main+(unsigned)endMain-(unsigned)mainX)
   return quickroot(i);
 for(p=(unsigned*)mainX, q=(unsigned*)main;
       p<(unsigned*)endMain; p++,q++) {
    unsigned px = *p, qx = *q;
    if ((px&0xf0000000) == 0x40000000 &&
        (qx&0xf0000000) == 0x40000000)
        {px += ((unsigned) p)>>2; qx += ((unsigned) q)>>2;}
    if (px != qx) return quickroot(i);
  }
 exit(1);
```

Step 1: find out who called you

```
unsigned mycaller[] ={0x81c3e008,  0x9010001f};
                          retl ;    mov %i7,%o0
```

```
unsigned *p, *q, caller;

union {unsigned *z; unsigned (*f)();} u;

u.z=mycaller;

caller = u.f();
```

Now `caller` *is the return address of quickroot,*

i.e. points somewhere into the middle of main()

Step 2: Is the caller ;=) driver-main?

```
#include <stdio.h>
mainX (int ac, char *av[])  {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
          j = quickroot (random ());
   }
endMain(){}
```

```
if (caller <= (unsigned)main ||
    caller >= (unsigned)main +
              (unsigned)endMain-(unsigned)mainX )
   return quickroot(i);
```

If return address doesn't point within a certain number of bytes from the beginning of the main( ) function, then we're not being called from the test driver. In that case, slowly and carefully compute the actual cube root.

Step 2, continued

```
#include <stdio.h>
mainX (int ac, char *av[])  {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
          j = quickroot (random ());
   }
endMain(){}
```

```
for(p=(unsigned*)mainX, q=(unsigned*)main;
      p<(unsigned*)endMain; p++,q++) {
    unsigned px = *p, qx = *q;
    if (px != qx) return quickroot(i);
  }
```

If any of the instructions in the caller don't match the instructions of mainX, then give up: slowly and carefully compute cube root.

Disassembly of mainX

```
#include <stdio.h>

mainX (int ac, char *av[])  {

      int i, j;

      srandom (atoi (av[1]));

      for (i = 0; i < 10000000; i++)

          j = quickroot (random ());

   }

endMain(){}
```

```
save  %sp, -112, %sp
call  0x209cc <atoi>
ld  [ %i1 + 4 ], %o0
call  0x209d8 <srandom>
nop
sethi  %hi(0x989400), %o1
or  %o1, 0x27f, %o1
add  %o1, 1, %i1
call  0x209e4 <random>
nop
call  0x107ac <quickroot>
nop
addcc  %i1, -1, %i1
bne  0x10780 <mainX+32>
nop
call  0x2099c <exit>
clr  %o0        ! 0x0
```

main   mainX

```
0x9de3bf90   0x9de3bf90   save  %sp, -112, %sp
0x400040ab   0x4000409a   call  0x209cc <atoi>
0xd0066004   0xd0066004   ld  [ %i1 + 4 ], %o0
0x400040ac   0x4000409b   call  0x209d8 <srandom>
0x01000000   0x01000000   nop
0x13002625   0x13002625   sethi  %hi(0x989400), %o1
0x9212627f   0x9212627f   or  %o1, 0x27f, %o1
0xb2026001   0xb2026001   add  %o1, 1, %i1
0x400040aa   0x40004099   call  0x209e4 <random>
0x01000000   0x01000000   nop
0x4000001a   0x40000009   call  0x107ac <quickroot>
0x01000000   0x01000000   nop
0xb2867fff   0xb2867fff   addcc  %i1, -1, %i1
0x12bffffb   0x12bffffb   bne  0x10780 <mainX+32>
0x01000000   0x01000000   nop
0x40004091   0x40004080   call  0x2099c <exit>
0x90102000   0x90102000   clr  %o0        ! 0x0
```

Control Transfer

- Branch instructions

op	a	cond	op2	disp22

nPC = PC + signextend(disp22) << 2

- Calls

op	disp30

nPC = PC + signextend(disp30) << 2

o position-independent code does not depend on where it's loaded; uses PC-relative addressing

My quickroot.c

```
unsigned *p, *q, caller;
 union {unsigned *z; unsigned (*f)();} u;
 u.z=mycaller; caller = u.f();
 if (caller <= (unsigned)main ||
     caller >= (unsigned)main+(unsigned)endMain-(unsigned)mainX)
   return quickroot(i);
 for(p=(unsigned*)mainX, q=(unsigned*)main;
      p<(unsigned*)endMain; p++,q++) {
    unsigned px = *p, qx = *q;
    if ((px&0xf0000000) == 0x40000000 &&
        (qx&0xf0000000) == 0x40000000)
        {px += ((unsigned) p)>>2; qx += ((unsigned) q)>>2;}
    if (px != qx) return quickroot(i);
  }
 exit(1);
```

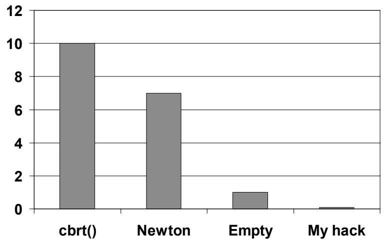
Results

- Correct in all contexts!
  - o In any test that actually measures whether it computes cube roots correctly, quickroot() just calls cbrt()
- Very fast in the contest-driver context!
  - o Just tests whether called from the contest driver, and if so,...

```
#include <stdio.h>
main (int ac, char *av[])  {
      int i, j;
      srandom (atoi (av[1]));
      for (i = 0; i < 10000000; i++)
          j = quickroot (random ());
   }
```

o calls exit() at the very first call to quickroot; doesn't execute the loop 10000000 times

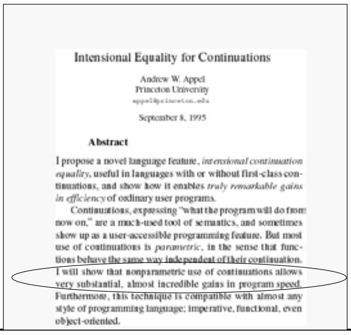
Results:



Reduce time from 10 seconds down to 0.0 seconds (measured to the nearest tenth of a second)

This is even faster than the driver running by itself!

Publish or perish!



Intensional Equality for Continuations

Andrew W. Appel
Princeton University
appel@princeton.edu

September 8, 1995

Abstract

I propose a novel language feature, *intensional continuation equality*, useful in languages with or without first-class continuations, and show how it enables *truly remarkable gains in efficiency* of ordinary user programs.

Continuations, expressing "what the program will do from now on," are a much-used tool of semantics, and sometimes show up as a user-accessible programming feature. But most use of continuations is *parametric*, in the sense that functions behave the same way independent of their continuation. I will show that nonparametric use of continuations allows very substantial, almost incredible gains in program speed. Furthermore, this technique is compatible with almost any style of programming language; imperative, functional, even object-oriented.

Spoof or serious?

From: Andrew W. Appel

To: Simon Peyton Jones, Editor, Journal of Functional Programming

Dear Simon: I enclose a short paper for consideration for publication in J. Functional Programming. It's not exactly a research article...

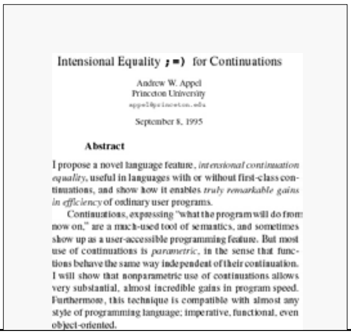
From: Simon Peyton Jones

To: Andrew W. Appel

Dear Andrew:

. . . I don't know what to make of it. (Spoof or serious? If it were dated April 1st I'd know.) Apart from anything else, it patently doesn't work in general (you'd have to compare the stacks too). And it's far from clear that it has applications beyond fooling inadequate test programs.

Revised title



Intensional Equality ;=) for Continuations

Andrew W. Appel
Princeton University
appel@princeton.edu

September 8, 1995

Abstract

I propose a novel language feature, *intensional continuation equality*, useful in languages with or without first-class continuations, and show how it enables *truly remarkable gains in efficiency* of ordinary user programs.

Continuations, expressing "what the program will do from now on," are a much-used tool of semantics, and sometimes show up as a user-accessible programming feature. But most use of continuations is *parametric*, in the sense that functions behave the same way independent of their continuation. I will show that nonparametric use of continuations allows very substantial, almost incredible gains in program speed. Furthermore, this technique is compatible with almost any style of programming language; imperative, functional, even object-oriented.

Try again

From: Andrew W. Appel

To: Richard Wexelblat, Editor, SIGPLAN Notices

Dear Dr. Wexelblat: I hereby submit the enclosed short paper, "Intensional Equality ;=) for Continuations", for publication in ACM SIGPLAN Notices.

From: Richard L. Wexelblat

To: Andrew W. Appel

Dear Andrew:

. . . will apper in February (or possibly March) . . . . Having read it carefully three times, I'm not sure but that it ought to appear in the April first issue,... but that would be unfair to so obviously dedicated a person as yourself.

Warning

- When you have your fun and games, avoid coming too close to academic fraud.
  - o (This applies to professors just as much as students)
- It’s always possible to tune your program to the particular benchmark test; excessive tuning constitutes fraud.