Introduction to Programming Systems

CS 217

Course Notes, Fall 2002

Andrew W. Appel

Princeton University

Goals

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

Outline

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

Coursework

- Eight programming assignments (60%)
  - symbol table
  - efficient symbol table using hashing
  - game player
  - game referee
  - "echo" and "wc" in assembly language
  - "sort" in assembly language
  - circuit simulator (interpreted)
  - circuit simulator (compiled)
- Exams (30%)
  - midterm
  - final
- Class participation (10%)
  - Precept attendance October 3--11 is ***mandatory***

Materials

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

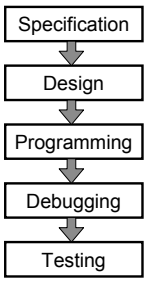
Facilities

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

Logistics

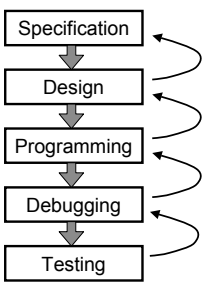
- Lectures
  - T,Th 10AM, CS105(?), 10:00AM
  - introduce concepts
  - work through programming examples
- Precepts
  - T,Th 1:30 Friend 112
  - W,F 10:00 C.S. 105
  - W,F 1:30 Friend 112
  - demonstrate tools (gdb, makefiles, emacs, …)
  - work through programming examples
  - collaborative work (first two weeks of 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
  - Well-designed
  - Consistent
  - Documented

  ← Write code in modules with well-defined interfaces
- Robust
  - Works for any input
  - Tested

  ← Write code in modules and test them separately
- Reusable
  - Components

  ← Write code in modules that can be used elsewhere
- Efficient
  - Only matters for 1%

  ← Write code in modules and optimize the slow ones

Modules

- Programs are made up of many modules
- Each module is small and does one thing
  - string manipulation
  - mathematical functions
  - 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
  - decouple clients from implementation
  - hide implementation details
- An interface specifies…
  - data types and variables
  - functions that may be invoked

```
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
  - different algorithms for different situations
  - 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, Interfaces, Implementations

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

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


Clients

- A client uses a module via its interface
- Clients see only the interface
  - can use module without knowing its implementation
- Client is unaffected if implementation changes
  - 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);
}
```


C Programming Conventions

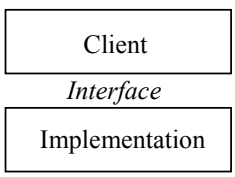
- Interfaces are defined in header files (.h)

stringlist.h

```
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);
```


Clients, Interfaces, Implementations

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



- Examples from real world?

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 **`ctypes.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
  - A program is broken up into meaningful modules
  - An interface defines what a module does
  - An implementation defines how the module does it
  - A client sees only the interfaces, not the implementations

Modules

CS 217

The C Programming Language

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

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

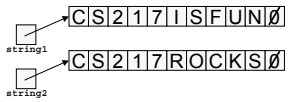
- Decompose execution into modules
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - Easier to test and debug
  - **Easier to reuse code**
  - Easier to make changes

```
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
  - Read strings
  - Sort strings
  - Write strings
- Interfaces hide details
  - Localize effect of changes
- Why is this better?
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - **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;
}
```



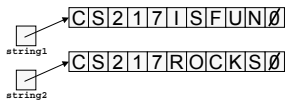
stringarray.c

```
#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]);
}
```


Modularity

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



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;
      }
}
```


Separate Compilation

- Move string array into separate file
  - Declare interface in **`stringarray.h`**
  - Provide implementation in **`stringarray.c`**
  - 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);
```


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.a
        cc -o sort sort.o stringarray.a

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

stringarray.a: stringarray.c
        cc -c stringarray.c
        ar ur stringarray.a stringarray.o

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


Interface with Function Pointers

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 ,
             int (*compare)(char *string1, char *string2));
```


Interface with Function Pointers

stringarray.c

-
-
-

```
void SortStrings(char **strings, int nstrings,
         int (*compare)(char *string1, char *string2))
{
  int i, j;

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


Interface with Function Pointers

main.c

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

#define MAX_STRINGS 128

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

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

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

  return 0;
}
```


Interface with Function Pointers

main.c

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

#define MAX_STRINGS 128

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

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

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

  return 0;
}
```

or, just call strcmp directly!

Abstract Data Type (ADT)

- An ADT module provides:
  - Data type
  - 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
  - Ensure that client cannot possibly access representation directly
  - C++, Java, other object-oriented languages have *private* fields
  - C has opaque pointers
- Example: stack ADT

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

Unchecked opaque pointers: void *

- C language has adequate, not perfect, support for opaque pointers
  - 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)
  - Can't make stack-of-(char*) because char is not a struct
- Solution: **void ***
  - Advantage: more flexible
  - 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

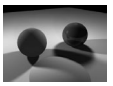
- Modularity is key to good software
  - Decompose program into modules
  - Provide clear and flexible interfaces
- Advantages
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes
  - Separate compilation
- Abstract Data Type (ADT) is an important principle
  - Design interfaces as ADTs
  - Provides more independence between modules

Programming Style & Scope in Programming Languages

CS 217

Programming Style

- Who reads your code?
  - compiler
  - 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=u>0?sqrt(u):1e31,u=b-u>1e-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=1./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,e?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?
  - Bugs are often created due to misunderstanding of programmer
    - What does this variable do?
    - How is this function called?
  - Good code == human readable code
- How can code become easier for humans to read?
  - Structure
  - Conventions
  - Documentation
  - 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
  - use white space freely
    e.g., to separate code into paragraphs
  - use indentation to emphasize structure
    use editor's autoindent facility
  - break long lines at logical places
    e.g., by operator precedence
  - line up parallel structures

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


Structure

- Convey structure with modules
  - separate modules in different files
    e.g., sort.c versus stringarray.c
  - simple, atomic operations in different functions
    e.g., ReadStrings, WriteStrings, SortStrings, etc.
  - 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
  - implement multiway branches with `if` … `else if` … `else`
  - emphasize that only one action is performed
  - avoid empty `then` and `else` actions
  - handle default action, even if can't happen (use `assert(0)`)
  - avoid `continue`; minimize use of `break` and `return`
  - avoid complicated nested structures

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

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


Conventions

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

Documentation

- Documentation
  - comments should add new information

```
i = i + 1;   /* add one to i */
```

  - comments must agree with the code
  - comment procedural interfaces liberally
  - comment sections of code, not lines of code
  - 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;
}
```

Scope

- The scope of an identifier says where it can be used

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);
```

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;
}
```


Local Variables & Parameters

- Functions can declare and define local variables
  - created upon entry to the function
  - destroyed upon return
- Function parameters behave like initialized local variables
  - 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;
}
```


Definitions and Declarations

- A declaration 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 definition declares the identifier and causes storage to be allocated for it

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


Local Variables & Parameters

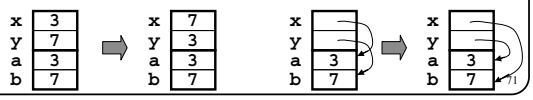
- Function parameters are transmitted by value
  - values copied into "local variables"
  - use pointers to pass variables "by reference"

```
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



Global Variables

- Functions can use global 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
}
```


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

`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"

Scope and Programming Style

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

Summary

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