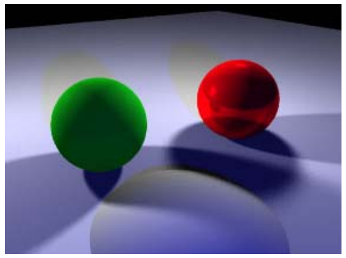
Programming Style

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=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?
  - 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
    - To separate code into paragraphs
  - Use indentation to emphasize structure
    - Use editor’s auto-indent facility
  - Break long lines at logical places
    - 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
    - **`sort.c`** versus **`stringarray.c`**
  - Simple, atomic operations in different functions
    - **`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 multi-way 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;
```

- Use idioms

```
while ( (c = getchar()) != EOF )
    putchar(c);
for (i = 0; i < n; i++ )
    ...
```

Conventions

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

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;
}
```

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)
{
  ...
}
```

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;
}
```

static means:

"not visible in other 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;
}
```

Extern means,
"visible in other 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"

Global Variables

- Functions can use global variables declared outside and above them within same file

```
int stack[100];

int main() {
   . . .     <————————  stack is in scope
}

int sp;

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

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, *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 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

```
#include <stdio.h>

int a, b;

main (void) {
    a = 1; b = 2;
    f(a);
    printf( “%d %d\n”, a, b);
}

void f(int a) {
    a = 3;
    {
        int b = 4;
        printf( “%d %d\n”, a, b);
    }
    printf( “%d %d\n”, a, b);
    b = 5;
}
```

Output

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

Programming Style and Scope

- 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

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.