Who reads your code?
  - compiler
  - other programmers

Which one cares about style?

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

```c
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

```c
#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

```c
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

```c
/** 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 */
    argv--; 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

```c
#include "stringarray.h"

#define MAX_STRING 128

int main()
{
    char *strings[MAX_STRING];
    int nStrings;

    ReadStrings(strings, &nStrings, MAX_STRING, 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

```c
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

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

**static** VERSUS **extern**

**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"

```c
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;
}
```

Local Variables & Parameters

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

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

```c
void swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```
Local Variables & Parameters

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

```c
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

```c
f(int x) {
  int x;  // error!
  ...
}
```
Scope Example

```c
int a, b;
main (void) {
    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
```

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.