Programming Style

CS 217

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

Programming Style

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

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
main()
{
  load_film(filename, &frames);
  load_matrices(16, &frames);
  load_matrices(17, &frames);
  load_matrices(18, &frames);
  load_matrices(19, &frames);
  load_matrices(20, &frames);
  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 autodeindent 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);
- e.g., to separate code into paragraphs
- separate distinct ideas within same function
- 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 "Strings.h"
int main()
{
    char *strings[MAX_STRINGS];
    int nstrings;
    ReadStrings(strings, &nstrings, MAX_STRINGS, stdin);
    WriteStrings(strings, nstrings, stdout);
    return 0;
}
```

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

**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 it 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;
```
Conventions

- Follow consistent naming style
  - use descriptive names for globals and functions
    e.g., WriteStrings, iMaxIterations, pFilename
  - use concise names for local variables
    e.g., i (not array index) 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
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

int ParseArguments(int argc, char *argv)
{
    int iArg = 0;
    if (argc < 1) { return 0; }
    if (!strcmp(*argv, "-h")) { argv++; argc--; iArg = 0; }
    if (!strcmp(*argv, "-v")) { argv++; argc--; iArg = 0; }
    if (!strcmp(*argv, "-w")) { argv++; argc--; iArg = 0; }
    if (!strcmp(*argv, "-I")) { argv++; argc--; iArg = 0; }
    if (!strcmp(*argv, "-o")) { argv++; argc--; iArg = 0; }
    iArg = atoi(*argv);
    return iArg;
}
```
**Scope**

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

```c
stringarray.h

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

sort.c

#include "stringarray.h"

#define MAX_STRINGS 128

void push(int x) {
    ...
}

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) {
    ...
}
```

**Global Variables**

- Functions can use **global** variables declared outside and above them.

```c
int stack[100];

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

int ap;

void push(int x) {
    ...
    stack, ap 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 ComparStrings(char* s1, char* s2)
{
    char *p1 = s1;
    char *p2 = s2;
    while(*p1 != 0) {
        if (*p1 == *p2) { p1++; p2++; }
        else 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;
}

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 (x > b) {
        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;

int main (void) {
    a = 3; 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
  - 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.