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 1960, B 1970, C 1972, K&R C 1978, ANSI C 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_STRING_LENGTH 256


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

  *nstrings = 0;
  while (fgets(string, MAX_STRING_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;
  char *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);
}
```

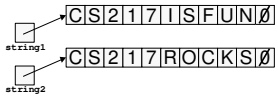
Modularity

- 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;
}
```

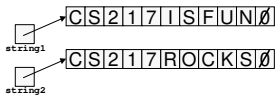


Modularity

- 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);
}
```



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);
```

Separate Compilation (2)

`stringarray.c`

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


#define MAX_STRING_LENGTH 256


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

  *nstrings = 0;
  while (fgets(string, MAX_STRING_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]);
}
```

```
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 (3)

`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;
}
```

Separate Compilation (4)

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));
```

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;
      }
    }
  }
}

.
.
.
```

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;
}
```

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