Modules and Interfaces

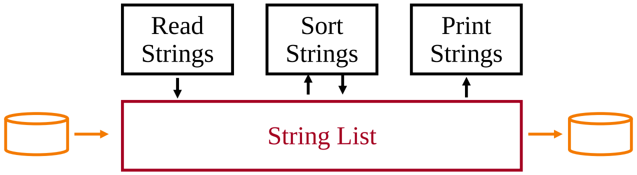
CS 217

Review: Constants

- C has several ways to define a constant
- Use #define
  - `#define MAX_VALUE 10000`
  - Substitution by preprocessing (will talk about this later)
- Use “const”
  - `const double = 1.56;`
  - Qualifier to declare that a variable is a constant
- Declare an enumerate constant type
  - `enum color { WHITE, YELLOW, BLUE, RED };`
  - Offers the chance of checking

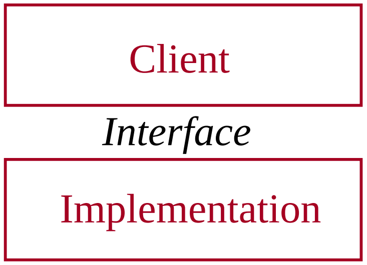
Modules

- Programs are made up of many modules
- Each module is small and does one thing
  - Set, stack, queue, list, etc.
  - String manipulation
  - Mathematical functions
- Deciding how to break up a program into modules is a key to good software design



Clients, Interfaces, Implementations

- Interfaces (Application Programming Interfaces or APIs) are contracts between clients and implementations
  - Clients must use interface correctly
  - Implementations must do what they advertise



- Examples from real world?

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

strlist.h:

```
typedef struct {
    StrList *entries;
    int size;
} StrList;

extern StrList *StrList_create(void);
extern void StrList_delete(StrList *list);
extern void StrList_insert(StrList *list, char *string);
extern void StrList_remove(StrList *list, char *string);
extern int StrList_write(StrList *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.

```
#include “strlist.h”

StrList *StrList_create(void)
{
    StrList *list = malloc(sizeof(StrList));
    list->entries = NULL;
    list->size = 0;
}

void StrList_delete(StrList *list)
{
    free(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

```
#include “strlist.h”

int main()
{
    StrList *list = StrList_create();
    StrList_insert(list, “CS217”);
    StrList_insert(list, “is”);
    StrList_insert(list, “fun”);
    StrList_write(list);
    StrList_delete(list);
}
```

Clients, Interfaces, Implementations

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

```
#include “strlist.h”

int main()
{
    StrList *list = StrList_create();
    StrList_insert(list, “CS217”);
    StrList_insert(list, “is”);
    StrList_insert(list, “fun”);
    StrList_write(list);
    StrList_delete(list);
}
```

C Programming Conventions

- Interfaces are defined in header files (.h)

strlist.h

```
typedef struct {
    StrList *entries;
    int size;
} StrList;
extern StrList *StrList_create(void);
extern void StrList_delete(StrList *list);
extern void StrList_insert(StrList *list, char *string);
extern void StrList_remove(StrList *list, char *string);
extern int StrList_write(StrList *list);
```

C Programming Conventions

- Implementations are described in source files (.c)

strlist.c

```
#include “strlist.h”
StrList *StrList_create(void)
{
    StrList *list = malloc(sizeof(StrList));
    list->entries = NULL;
    list->size = 0;
}

void StrList_delete(StrList *list)
{
    free(list);
}
. . .
```

C Programming Conventions

- Clients “include” header files

main.c

```
#include “strlist.h”

int main()
{
    StrList *list = StrList_create();
    StrList_insert(list, “CS217”);
    StrList_insert(list, “is”);
    StrList_insert(list, “fun”);
    StrList_write(list);
    StrList_delete(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’d

- 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**”

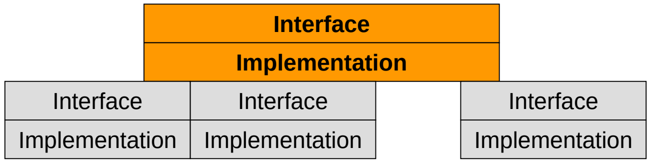
```
extern FILE *stdin, *stdout, *stderr;
extern FILE *fopen(const char *, const char *);
extern int fclose(FILE *);
extern int printf(const char *, …);
extern int scanf(const char *, …);
extern int fgetc(FILE *);
extern char *fgets(char *, int, FILE *);
extern int getc(FILE *);
extern int getchar(void);
extern char *gets(char *);
. . .
extern int feof(FILE *);
```

Goals of Modularity

- Decomposability
  - Divide a problem into sub-problems
- Composability
  - Build a system using building blocks
- Continuity
  - A small spec change affects changes in a small number of modules
- Understandability
  - Readability by people
- Protection
  - Error occurs in a local place

Decomposability

- Divide a problem into sub-problems and work on each
- Use a top-down, layered approach
  - Each layer provides an abstraction (by an interface)
  - Example: networking
    - – Application (FTP, email, etc.)
    - – Transport (TCP)
    - – Network (IP)
    - – Link (device driver and network interface)
- Avoid circular dependency



Composability

- Build software systems with building blocks (modules and interfaces)
- API calls are powerful, expressive and yet simple to use
- Good examples
  - C stdio
  - C string
- Bad example
  - Too many calls in an interface and many never get used

Continuity

- A small change in the specification leads to small changes in a small number of modules
- Good example
  - Add a StrList_Sort call into the interface
- Bad example
  - Change the definition of data type StrList

Understandability

- Understand a module by reading it or a few modules at its neighborhood
- Good example
  - Modules providing good abstractions (top-down layered)
- Bad example
  - An implementation that uses global variables defined and used in multiple modules

Protection

- Effect of an error is limited to one module or a small number of neighbor modules
- Good example
  - An error in StrList_insert()
- Bad example
  - An error occur in a global variable modified by multiple modules

Separate Compilations

- Simple case
  - Compile strlist.c to strlist.o
  - Compile test.c and link with strlist.o
- Typical software product
  - Compile many implementation .c files
  - Link them into a library or build an executable

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 have great advantages
  - Easier to understand
  - Easier to test and debug
  - Easier to reuse code
  - Easier to make changes