Programming without abstract data types

Writing abstract programs when the compiler won't help as much as you might like

Overall strategy

- Figure out what abstraction tools our language does provide
- Adopt a strategy for using what we have to get what we want

Data abstraction in C

- The C language gives us
- well known abstraction facilities…
  - subroutines
  - structures and type definitions
  - header files to define interfaces
- …and a somewhat less well known one
  - incomplete types

An example: the C standard I/O library

- The library provides a type called `FILE`
  - defined in `<stdio.h>`
  - not intended to be used directly
- Most I/O operations are defined in terms of *pointers* to `FILE` objects

```
FILE *f = fopen("foo", "r");
int c = getc(f);
fclose(f);
```

Magic cookies

- A pointer to `FILE` is an example of a magic cookie
  - Term coined by Steve Johnson (author of `yacc`, and of the first portable C compiler)
  - Definition: A small object, of mysterious nature, that one gives to a program to get it to do something
- Key properties: small, mysterious

Using magic cookies for abstraction

- Define a family of functions that knows about a particular kind of cookie
  - One function (or more) allocates a magic cookie and returns (a pointer to) it
  - Others act on the cookie, using it to store and retrieve state as needed
  - One function (or more) deallocates it
- Document the functions, not the cookie

Incomplete types

- C provides a tool that is often useful for keeping magic cookies magical
  - A declaration such as
    ```
    struct Line;
    ```
    says that `Line` is a structure type, but says nothing about how it is put together
  - Such a type is called *incomplete*
  - Programs can use pointers to incomplete types, but cannot dereference the pointers

Using incomplete types

- A header file declares an incomplete type and a family of related functions

```
struct Token;
enum Toktype{WORD, BREAK, END};
struct Token *Token_make(FILE *);
enum Toktype Token_type
    (struct Token *);
char *Token_word(struct Token *);
void Token_destroy(struct Token *);
```

Implementing the functions

- Each implementation file will include the header and also define the structure

```
#include "Token.h"
struct Token {
   char *s;
   enum Toktype t;
};
```

- If there are multiple implementation files, the definition might go into another header

Typical implementation

- C version:

```
enum Toktype Token_type
    (struct Token *tp) {
        return tp->t;
}
```

- C++ version:

```
Toktype Token::type() {
    return t;
}
```

Allocation and deallocation

- Because user code deals with pointers, all objects must be allocated and deallocated inside the implementation

```
struct Token *Token_make(FILE *f) {
    struct Token *tp =
       malloc(sizeof(struct Token));
    /* … */
    return tp;
}
void Token_destroy(struct Token *tp)
{    free(tp->s); free(tp);    }
```

What about strings?

- C offers little help to programs that want to deal with strings
  - pointers to null-terminated character arrays are conventional…
  - but the memory occupied by those arrays is the programmer's responsibility
- Fortunately, we don't need to do much

Two useful C library functions

`strcpy(dest, source)`
`strcat(dest, source)`

- `source` points to the initial character of a null-terminated character array
- in the case of `strcat`, `dest` also is the initial character of a null-terminated array
- in either case, the `dest` array must have enough room for the result

What do we need to do with strings?

- Allocate and deallocate them (`malloc`, `free`)
- Print them (`fprintf`)
- Test whether a string is empty
- Concatenate a string or character on the end of another string

Null strings

- It is convenient to represent a null string by a zero pointer, instead of a pointer to a null-terminated string with no significant characters, because
  `s = NULL;`
  is easier to write than
  `s = malloc(1); *s = '\0';`
- We must cater to that representation

What we would like to say

- We would like to use
  `append_char(s, c)`
  instead of
  `s += c`
  and similarly for `append_str`
- Unfortunately, C gives us no way for a call to append_char(s, c) to change s
- We'll settle for `append_char(&s, c)`

Implementation of `append_char`

```
void append_char(char **s, char c)
{
    int n = 0;
    if (*s) {
        n = strlen(*s);
        *s = realloc(*s, n + 2);
    } else
        *s = malloc(2);
    (*s)[n] = c;
    (*s)[n+1] = '\0';
}
```

Implementation of `append_str`

```
void append_str
        (char **s, const char *t) {
    if (*s) {
        *s = realloc(*s,
           strlen(*s) + strlen(t) + 1);
        strcat (*s, t);
    } else {
        *s = malloc(strlen(t) + 1);
        strcpy(*s, t);
    }
}
```

Observations

- Abstraction is rarely impossible
  - It may require discipline on the part of users
  - There may be a price in convenience or efficiency
- The complete C source code for this example is linked from the course web pages

Objects

- Both the C and C++ versions of this program used objects of particular types to contain the state of the program
- We often think of an operation such as `append` as "acting on" an object
- Such actions are a fundamental part of object-oriented programming

Are objects necessary?

No!!

- Justifying this claim requires a more thorough understanding of what objects are, and how they differ from abstract data types.

What is an object?

- It contains information
- There are operations defined on it
- It has identity
  - Two different objects can contain identical information, and support identical operations, but still be different objects

What is object identity?

- Suppose x and y are names of objects.
- How can we tell if x and y are merely two different names for the same object?
  - Perhaps we can take the address of x and of y
  - Perhaps we can change one of them and see if the change is reflected in the other

Identity and mutable state

- Unless you have a way of checking the identity of an object directly (such as its address), the only way to determine if two objects are the same is to change one and see if the other changes
- Therefore immutable objects have no identity.
- Immutable objects are just values!

Functional programming

- There is a school of thought that says that mutable objects are evil, and all computation should be done with pure values
  - usually called *functional programming*
  - major languages: ML, Haskell
  - very impressive results in the laboratory
  - limited commercial relevance so far

The key issues

- Functional and object-oriented programming are styles of abstraction that address similar problems
  - How do we partition a program?
  - What information flows between the parts of the program?
  - What do authors of one part have to know about other parts?
- They come to very different conclusions

Summary

- Different languages offer different tools for abstraction
  - Every language offers something
  - No language offers everything
- To use a language effectively, you must match its tools to the problem at hand