Organizing C Code

Computer Systems No Reference

Problem – Single File Code

- Many different functions
- Hard to find the code you need to work with
- What is the best order for the functions in the file?
  - By category?
    - All allocation functions together
    - All setter functions together
    - …
  - By Call Tree?
    - What if function_x is called by function_1 AND function_2?
  - Alphabetically?

Solution: Organize by Data Type

- For example, project 2
  - warehouse.c:
    - All the functions that deal with the warehouse itself, getting and processing an order
  - slots.c
    - Functions which deal with the slots on the workbench

```
void initSlots();
void getBin(int bin, int slot);
int findSlot(int bin);
void getWidget(int bin);
void printEarnings();
```

Abstraction

- Think of the workbench as an object
- There are certain things you can do with this object
  - initialize
  - get a bin from the warehouse and put it on the workbench
  - Figure out which slot on the workbench a bin is in
  - get a widget out of a bin on the workbench
  - print a report that talks about the cost and earnings

Create Abstraction

- Make a distinction between…
- What is required to USE a group of functions

and

- What is required to IMPLEMENT a group of functions

Organize by File

- File Name: Group Name e.g. “slots” for workbench slots
- Everything needed to USE a group in header (.h) file e.g. slots.h
  - Typically a typedef for any special data types needed
  - Typically declarations of functions that work with that data type
- Everything needed to IMPLEMENT a group in code (.c) file e.g. slots.c
  - #include the header file
  - #include all the header files needed for this code
  - Global declarations for data available to the functions in this code
  - Definitions of functions

Compiling Multi-File Source

- gcc –g –o packem warehouse.c slots.c …
  - Specify all c files
  - Code files will include headers

- In a Makefile…

```
packem : warehouse.c slots.c slots.h
        gcc –g –o packem warehouse.c slots.c
```

recompile if any of these change

but only need code files in gcc cmd

Alternative…. Object Code

```
packem : warehouse.o slots.o
        gcc –g –o packem warehouse.o slots.o

warehouse.o : warehouse.c slots.h
        gcc –g –o warehouse.o warehouse.c

slots.o : slots.c slots.h
        gcc –g –o slots.o slots.c

clean :
        -rm packem warehouse.o slots.o
```

Object Code Pros and Cons

Advantages

- Only need to recompile what has changed and relink

Disadvantages

- Need extra disk space for object code
- Need to recompile lots of code if header file changes

Note on Function Names

- warehouse.c can invoke a function that is defined in slots.c
  - .e.g. getBin(bin,slot)
- warehouse.o must keep the NAME of the function invoked
  - The main function has no idea where getBin instructions are
  - Even if there is no debug (-g) information!
- That's why function names are always available in GDB

"Abstract Data Type"

- Special typedef trick
- Specify a type is a pointer to a struct without defining the struct!
  - A pointer is a pointer is a pointer
  - Violates C's "define before use" rule!
- Put the typedef in your header file (.e.g object.h)
  - User's of this data type can declare and use pointers to the struct
  - User's of this data type CANNOT access fields in that struct!
- Put the definition of the struct in your code file (e.g. object.c)
  - Code file can declare and use pointers to the struct
  - Code file can access fields in that struct!

Example: Linked List Node

node.h

```
typedef struct lnode * node;

node makeNode(int val);
int getVal(node n);
void setVal(node n,int val);
node getNext(node n);
void setNext(node n, node t);
void freeNode(node n);
```

node.c

```
#include "node.h"
#include <stdlib.h>

struct lnode {
        int val;
        node next; };

node makeNode(int val) {
        node n = (node)malloc(…
```

Using an Abstract Data Type

```
#include <node.h>

void push(stack s, int val) {
        node n=makeNode(val);
        setNext(n,s->head);
        s->head=n;
}
…
```

ADT Pros and Cons

Advantages

- Confine code which works on nodes to nodes.c
- Code in other files can't mess with the internals of nodes
- All node handling is encapsulated in nodes.c
- Allows different implementation by changing node.c
- Enables usage of nodes.c in multiple programs

Disadvantages

- Requires "getters" and "setters" for each field
- Complicates function names
- Requires discipline (compiler doesn't do all the checking for you)

Writing Object Oriented C

OO Terminology

- class
- object
- private fields
- creator method
- methods
- Inheritance

C implementation

- structure / typedef
- pointer to structure instance
- structure members
- creation function
- functions w/ ptr argument
- Structure ref. structure

Object Oriented C

- C++ handles method names by mangling class/method name
  - Put class name in all method names in C
- C++ handles “this” object with -> syntax
  - Pass “this” as first parameter
- C++ has fancy private/public/friend stuff
  - Manage this on our own