S.O.L.I.D. in C

Jason Gorman


NEW KIDS ON THE BLOCK
HANGIN' TOUGH


Nintendo GAME BOY
SELECT
START

1989

THE C PROGRAMMING LANGUAGE

Simple Design

- It works ✓
- Clearly communicates intent ✓
- Free of duplication ✓
- Built from simplest parts ✓

"Object Oriented" Design Principles

Tell, Don't Ask ?

Single Responsibility ?

Open-Closed ?

Liskov Substitution ?

Interface Segregation ?

Dependency Inversion ?

carpet_quote.c

```
float quote(struct Room *room, struct Carpet *carpet){
    float area = room->length * room->width;

    if(carpet->roundUp){
        area = ceil(area);
    }

    return area * carpet->pricePerSqrMetre;
}
```

Can we break it down into single responsibilities?

carpet_quote.c

```
float area(const struct Room *room) {
    return room->length * room->width;
}

float price(const struct Carpet *carpet, float area) {
    if(carpet->roundUp){
        area = ceil(area);
    }

    return area * carpet->pricePerSqrMetre;
}

float quote(struct Room *room, struct Carpet *carpet){
    return price(carpet, area(room));
}
```

room.c

```
#include "room.h"

float area(const struct Room *room) {
   return room->length * room->width;
}
```

carpet.c

```
#include <math.h>
#include "carpet.h"

float price(const struct Carpet *carpet, float area) {
   if(carpet->roundUp){
      area = ceil(area);
   }

   return area * carpet->pricePerSqrMetre;
}
```

carpet_quote.c

```
#include "carpet_quote.h"
#include "room.h"
#include "carpet.h"

float quote(struct Room *room, struct Carpet *carpet){
   return price(carpet, area(room));
}
```

Can we hide the data?

room.h

```
#ifndef ENCAPSULATION_ROOM_H
#define ENCAPSULATION_ROOM_H

struct Room {
   float width;
   float length;
};

float area(const struct Room *room);

#endif //ENCAPSULATION_ROOM_H
```

carpet.h

```
#ifndef ENCAPSULATION_CARPET_H
#define ENCAPSULATION_CARPET_H

#include "bool.h"

struct Carpet {
   float pricePerSqrMetre;
   boolean roundUp;
};

float price(const struct Carpet *carpet, float area);

#endif //ENCAPSULATION_CARPET_H
```

```
int main() {
    struct Room room = {2.5, 2.5};
    struct Carpet carpet = {10.0, false};

    float total = quote(&room, &carpet);
```

room.h

```
#ifndef ENCAPSULATION_ROOM_H
#define ENCAPSULATION_ROOM_H

typedef struct Room Room;

Room* new_room(float width, float length);

float area(struct Room *room);

#endif //ENCAPSULATION_ROOM_H
```

room.c

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

struct Room {
   float width;
   float length;
};

Room* new_room(float width, float length){
   Room* room = malloc(sizeof(Room));
   room->width = width;
   room->length = length;
   return room;
}

float area(struct Room *room) {
   return room->length * room->width;
}
```

Can we have polymorphism?

room.h

```
#ifndef ENCAPSULATION_ROOM_H
#define ENCAPSULATION_ROOM_H

typedef struct Room Room;

float area(struct Room *room);

#endif //ENCAPSULATION_ROOM_H
```

rectangular_room.h

```
#ifndef ENCAPSULATION_RECTANGULAR_ROOM_H
#define ENCAPSULATION_RECTANGULAR_ROOM_H

#include "room.h"

Room* new_rectangular_room(float width, float length);
```

rectangular_room.c

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

struct Room {
   float width;
   float length;
};

Room* new_rectangular_room(float width, float length){
   Room* room = malloc(sizeof(Room));
   room->width = width;
   room->length = length;
   return room;
}
```

room.h

```
#ifndef ENCAPSULATION_ROOM_H
#define ENCAPSULATION_ROOM_H

typedef struct Room Room;

float area(struct Room *room);

#endif //ENCAPSULATION_ROOM_H
```

circular_room.h

```
#ifndef ENCAPSULATION_CIRCULAR_ROOM_H
#define ENCAPSULATION_CIRCULAR_ROOM_H

#include "room.h"

Room* new_circular_room(float radius);
```

circular_room.c

```
#include <stdlib.h>
#include “circular_room.h"

struct Room {
   float radius;
};

Room* new_circular_room(float radius){
   Room* room = malloc(sizeof(Room));
   room->radius = radius;
   return room;
}
```

We can’t implement area() twice…

Function pointers to the rescue!

rectangular_room.h

```
float rectangular_area(struct Room *room) {
   return room->length * room->width;
}
```

circular_room.c

```
float circular_area(struct Room *room) {
    return (2 * room->radius) * (2 * room->radius);
}
```

```
int main() {
   Carpet* carpet = new_carpet(10.0, false);

   Room* rectangular_room = new_rectangular_room(2.5, 2.5);

   float total = quote(rectangular_room, &rectangular_area , carpet);

   printf("Price of rectangular carpet with no rounding up is %g\n", total);

   Room* circular_room = new_circular_room(2.5);

   total = quote(circular_room, &circular_area, carpet);

   printf("Price of circular carpet with no rounding up is %g\n", total);

   return 0;
}
```

Type Safety: How do we discourage passing the wrong implementation of Room into area()?

room.h

```
typedef struct RoomData RoomData;
typedef struct Room Room;

struct Room {
   RoomData *this;
   float (*area)(RoomData*);
};
```

rectangular_room.h

```
struct RoomData {
   float width;
   float length;
};

Room* new_rectangular_room(float width, float length){
   RoomData* this = malloc(sizeof(RoomData));
   Room* room = malloc(sizeof(Room));
   this->width = width;
   this->length = length;
   room->this = this;
   room->area = &rectangular_area;
   return room;
}

float rectangular_area(RoomData *this) {
   return this->length * this->width;
}
```

carpet_quote.c

```
float quote(Room *room, Carpet *carpet){
   return price(carpet, room->area(room->this));
}
```

Can we have our modules present client-specific interfaces?

room_area.h

```
#include "room.h"

float area(Room *room);
```

floor_level.h

```
#include "room.h"

int flightsOfStairs(Room *room);
```

room.c

```
#include "room_area.h"
#include "floor_level.h"

struct Room {
   char level;
   float width;
   float length;
};

Room* new_room(float width, float length, char level){
   … etc
}

float area(Room *room) {
   return room->length * room->width;
}

int flightsOfStairs(Room *room){
   if(room->level == 'G'){
      return 0;
   }
   if(room->level == 'B'){
      return -1;
   }
   return room->level - '0';
}
```

carpet_quote.c

```
#include "carpet_quote.h"
#include "carpet.h“
#include "room_area.h"

float quote(Room *room, Carpet *carpet){
   return price(carpet, area(room));
}
```

main.c

```
#include "carpet_quote.h"
#include "room.h"
#include "floor_level.h"
#include "bool.h"

int main() {
   Room* room = new_room(2.5, 2.5, 'G');
   int flights = flightsOfStairs(room);
   printf("Ground floor room involves %d flights of stairs\n", flights);

   return 0;
}
```

Ah, but what about Interface Segregation + Polymorphism?

room.h

```
typedef struct RoomData RoomData;

typedef struct {
   RoomData *this;
   float (*area)(RoomData*);
   int (*flightsOfStairs)(RoomData*);
} IRoom;
```

room_area.h

```
typedef struct {
   RoomData *this;
   float (*area)(RoomData*);
} IRoomArea;

IRoomArea* asArea(IRoom *room);
```

floor_level.h

```
typedef struct {
   RoomData *this;
   int (*flightsOfStairs)(RoomData*);
} IFloorLevel;

IFloorLevel* asFloorLevel(IRoom *room);
```

room_area.c

```
IRoomArea* asArea(IRoom *room){
   IRoomArea *area = malloc(sizeof(IRoomArea));
   area->this = room->this;
   area->area = room->area;
   return area;
}
```

floor_level.c

```
IFloorLevel* asFloorLevel(IRoom *room){
   IFloorLevel *level = malloc(sizeof(IFloorLevel));
   level->this = room->this;
   level->flightsOfStairs = room->flightsOfStairs;
   return level;
}
```

```
void testRoomArea(Area *area, float expected){
   assert(area->area(area->this) ==  expected);
}

void testFloorLevel(FloorLevel *level, int flights){
   assert(level->flightsOfStairs(level->this) == flights);
}

int main() {
   Room *rectangularRoom = new_rectangular_room('G', 2.5, 2.5);
   Room *circularRoom = new_circular_room('B', 2.5);
   testRoomArea(asArea(rectangularRoom), 6.25);
   testRoomArea(asArea(circularRoom), 25);
   testFloorLevel(asFloorLevel(rectangularRoom), 0);
   testFloorLevel(asFloorLevel(circularRoom), -1);
```

"Object Oriented" Design Principles

Tell, Don't Ask	✓
Single Responsibility	✓
Open-Closed	✓
Liskov Substitution	✓
Interface Segregation	✓
Dependency Inversion	✓