Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 9
15.11.2018

Program Design

Interfaces

A definition of a set of functions that provide a coherent module (or library)

- Data structure (e.g., list, binary tree)
- User interface (e.g., drawing graphics)
- Communication (e.g., device driver)

Interface - modularity

Hide the details of implementing the module from its usage

- Specification – “what”
- Implementation – “how”

Interface – information hiding

Hide “private” information from outside

- The “outside” program should not be able to use internal variables of the module
- Crucial for modularity

Resource management

- Define who controls allocation of memory (and other resources)

Example interface - StrStack

A module that allows to maintain a stack of strings

Operations:

- Create new
- Push string
- Pop string
- IsEmpty
- Free

Example interface - StrStack

```
#ifndef _STRSTACK_H
#define _STRSTACK_H

typedef struct StrStack StrStack;

StrStack* StrStackNew();
void StrStackFree(StrStack** stack);

// This procedure *does not* duplicate s
void StrStackPush(StrStack* stack, char* s);
// return NULL if the stack is empty
char *StrStackPop(StrStack* stack);
// Check if the stack is empty
int StrStackIsEmpty(StrStack const* stack);

#endif // _STRSTACK_H
```

Implementation of StrStack

Decision #1: data structure

- Linked list
- Array (static? dynamic?)
- Linked list of arrays
- …

We choose linked list for simplicity

Implementation of StrStack

Decision #2: Resource allocation

- Duplicate strings on stack or keep pointer to original?
- If duplicate, who is responsible for freeing them?

We choose not to duplicate → leave this choice to user of module

Implementation of StrStack

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "StrStack.h"

typedef struct StrStackLink {
   char* str;
   struct StrStackLink *next;
} StrStackLink;


struct StrStack {
   StrStackLink* top;
};


int StrStackIsEmpty(StrStack const* stack) {
   assert( stack != NULL );
   return stack->top == NULL;
}
```

Implementation of StrStack

```
StrStack* StrStackNew() {
   StrStack* stack = (StrStack*) malloc(sizeof(StrStack));
   if (stack != NULL) {
      stack->top = NULL;
   } else {
      printf("out of memory, cannot create stack\n");
   }
   return stack;
}


void StrStackFree(StrStack** stack) {
   while (!StrStackIsEmpty(*stack))    {
      StrStackPop(*stack);
   }
   free(*stack);
   *stack=NULL;
}
```

Implementation of StrStack

```
void StrStackPush(StrStack* stack, char* s)
{
   assert( stack != NULL );
   StrStackLink *p = (StrStackLink*)
                              malloc(sizeof(StrStackLink));
   if (p == NULL)
   {
      printf("out of memory, cannot push a string to the
              stack\n");
      return;
   }
   p->str = s;
   p->next = stack->top;
   stack->top = p;
}
```

Implementation of StrStack

```
char* StrStackPop(StrStack* stack)
{
   char *s;
   StrStackLink *p;
   assert( stack != NULL );
   if (stack->top == NULL) {
      return NULL;
   }
   s = stack->top->str;
   p = stack->top;
   stack->top = p->next;
   free(p);
   return s;
}
```

Using StrStack

```
#include <stdlib.h>
#include <stdio.h>
#include "StrStack.h"
char * readline() {  ... } //A function to read a line
int main() {
   char *line;
   StrStack *stack = StrStackNew();
   while ((line = readline()) != NULL)
   {
      StrStackPush(stack, line);
   }
   while ((line = StrStackPop(stack)) != NULL)
   {
      printf("%s\n", line);
      free(line);
   }
   StrStackFree(&stack);
   return 0;
}
```

Interface Principles

Hide implementation details

1. Hide data structures
2. Don’t provide access to data structures that might be changed in alternative implementation
3. A “visible” detail cannot be later changed without forcing the user to change the code that used your interface!

Interface Principles

Use small set of “primitive” actions

1. Maximize functionality with minimal set of provided operations
2. Do not provide unneeded functions “just because you can”
3. How much functionality? Two approaches:
   - **Minimal**
     for few users, don't waste your time
   - **Maximal**
     when many users will use it, e.g. OS

Interface Principles

1. Do not use global variables unless you must
2. Don't have unexpected side effects!
3. Use comments if you assume specific order of operations by the user (and force it if you can, e.g., by assertions)

Interface Principles

Consistent Mechanisms:

Do similar things in a similar way

- strcpy(dest, source)
- memcpy(dest, source)

Interface Principles

Resource Management

1. Free resource at the same level it was allocated – the one who allocates the resource is responsible to free it
2. If you have assumptions about resources – specify this clearly

Generic programming in C

The use of void*

- Generic data-structures are data-structures that can hold data of any type (or, at least, of several types)
- The specific type that the instance of the data-structure holds is determined during run-time
- The main tool C provides for generic data-structures implementation is:

`void*`

memcpy

Before we begin to discuss implementation of generic data structures, let us introduce the function `memcpy`

Prototype:

```
void *memcpy(void *destination,
             const void *source,
             size_t num);
```

memcpy

```
void *memcpy(void *destination,
             const void *source,
             size_t num);
```

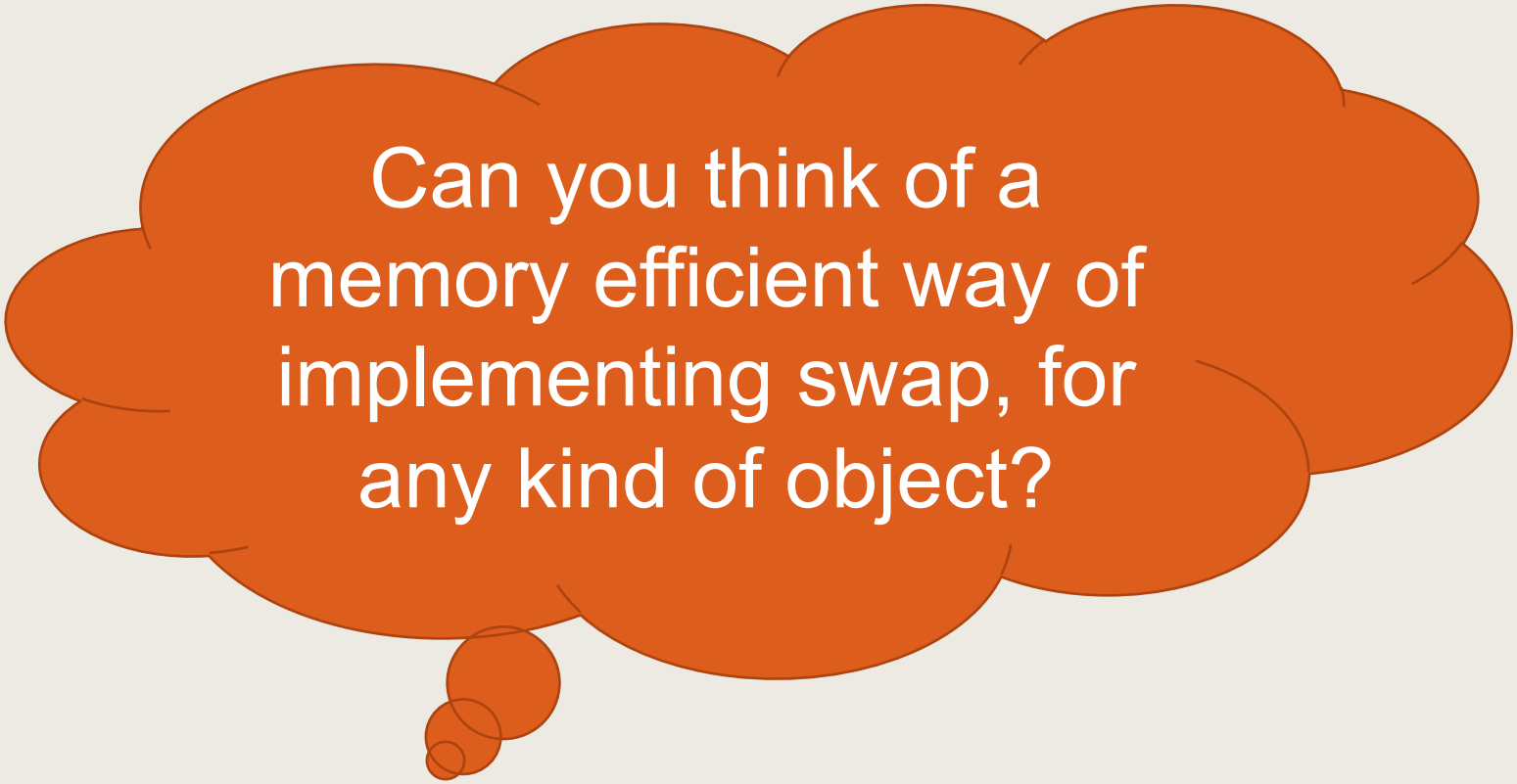
- `memcpy` copies a block of memory of specific size from one address to another address.
- `memcpy` doesn't know the *type* of variable(s) being copied.
- The main challenges:
  - how to iterate void* (no pointer-arithmetics is defined for void*)
  - How to dereference the pointers

Possible implementation of memcpy

```
void *memcpy(void *destination,
             const void *source, size_t num)
{
  char *d = (char*) destination;
  char *s = (char*) source;

  for (int i = 0; i < num; ++i) {
    // pointer arithmetics for char* is done
    // with units of sizeof(char) == 1 byte
    d[i] = s[i];
  }
}
```

Another example – Generic Swap



Generic data-structures using void*

Stack of ints

We would like our stack to:

- hold ints
- allocate its own memory for the data it holds

We would like to support the following operations:

- create new stack
- pop element from the stack head
- push element to the stack head
- check if the stack is empty
- free the stack

Generic stack

We would like our stack to:

- hold ~~int~~ **any type** (Same type to all of the stack nodes)
- allocate its own memory for the data it holds

We would like to support the following operations:

- create new stack
- pop element from the stack head
- push element to the stack head
- check if the stack is empty
- free the stack

Generic stack underlying data structures:

```
typedef struct Node
{
    void * _data; // pointer to anything
    struct Node * _next;
} Node;

typedef struct Stack
{
    Node * _top;
    size_t _elementSize; // we will need
                         // that for memcpy
} Stack;
```

Generic stack alloc

```
Stack* stackAlloc(size_t elementSize)
{
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->_top = NULL;
    stack->_elementSize = elementSize;
    return stack;
}
```

Generic stack push

```
// push the data into stack
void push(Stack* stack, void *data)
{
  //you should check allocation success
  Node* node = (Node*)malloc(sizeof(Node));
  if(node == NULL) { // exit }

  // allocate memory for data
  node->_data = malloc(stack->_elementSize);
  memcpy(node->_data, data, stack->_elementSize);

  // make node the top of the stack
  node->_next = stack->_top;
  stack->_top = node;
}
```

Generic stack pop

```
// remove the top from the stack and copy top's data to headData
void pop(Stack* stack, void *headData) {
  if(stack == NULL) {/*print error message and exit*/}
  if(stack->_top == NULL) {
    printf("stack is empty\n");
    return;
  }
  // get the top Node and copy the data
  Node *node = stack->_top;
  memcpy(headData, node->_data, stack->_elementSize);
  // update the top to point to the next node
  stack->_top = node->_next;
  // free memory
  free(node->_data);
  free(node);
}
```

Generic stack free

```
void freeStack(Stack** stack)
{
    Node* p1;
    Node* p2;
    if (!(*stack == NULL)){
      p1= (*stack)->_top;
      while(p1){
        p2= p1;
        p1= p1->_next;
        free(p2->_data);
        free(p2);
      }
      free(*stack);
      *stack = NULL;
    }
}
```

Using generic stack:

```
int main()
{
  int i, num = 10;
  printf("Generating list with %d ints\n", num);
  Stack *stack = stackAlloc(sizeof(int));
  for(i = 1; i <= num; i++) {
    push(stack,&i);
  }
  for(i = 1; i <= num-2; i++) {
    int headData;
    pop(stack,&headData);
    printf("top value is: %d\n",headData);
  }
  freeStack(&stack);
  return 0;
}
```