1

COMP281

Principles of C and memory management

Lecture 9

Dr. Frans Oliehoek
Department of Computer Science
University of Liverpool

Last Time / Today

- Last time: start dynamic data structures…
- …continue with that today

- Other topics:
  - function pointers
  - file handling

Dynamic Data Structures & Abstract Data Types (cont'd)

Dynamic Data

Main idea was the following:

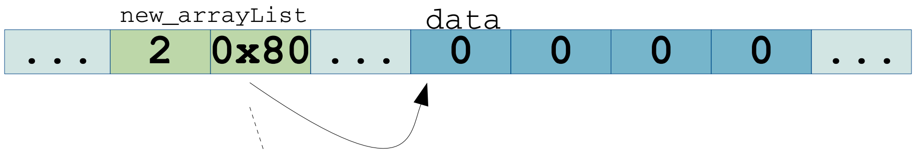
- We can now represent data structures (as structs)
- We can allocate memory dynamically (with malloc)

→ We can now implement dynamic data structures, like trees, linked lists, priority queues etc.

Example: Resizable Array

- Functions to access the array

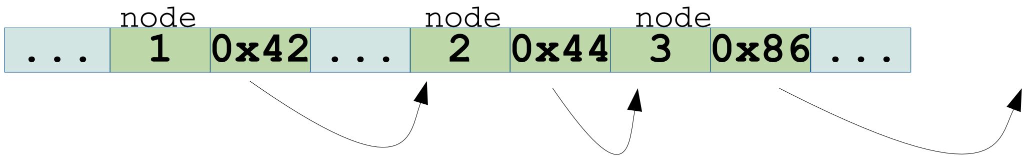
```
int get(arrayList* array, int index)
{   return array->data[index]; }

void set(arrayList* array,
       int index, int value)
{
  if (index >= array->size)
  {
    array->size = index*2;
    array->data = realloc(array->data,
        sizeof(int) * array->size);
  }
  array->data[index] = value;
}
```



data may have been moved!

  - When we try to put more in the array: it can *dynamically* grow

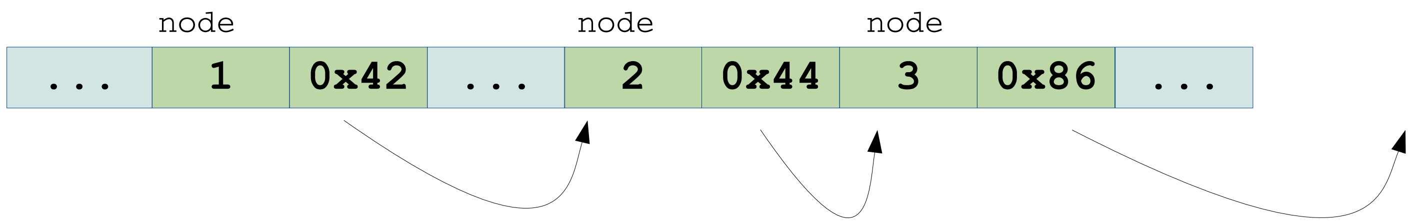
Another Example: Linked Lists



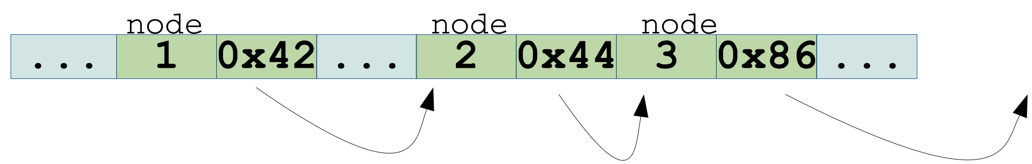
Linked Lists

- Arrays are efficient to access, but…
  - The size is fixed
  - It is difficult to insert into the middle

- A Linked List may be a more suitable data structure
  - Each element points to the next making a 'chain'



Efficiency of Linked Lists



- Ordered reads: efficient
  - just requires dereferencing a pointer for each read
  - but not as efficient as arrays (more cache misses if data spread out)
- Random reads: not efficient
  - Must start from the beginning each time
- Random insertions: much better than arrays!
  - (in an array, you may have to move all of the later elements)
  - in linked list: each insertion requires a small memory allocation
  - but no slow ‘resize’ operations are needed
  - memory space will grow automatically as the list grows

Example: Linked List

- Each element ('node') contains a pointer to another element, e.g.:

```
struct car
{
   char name[30];
   int value;
   struct car * next_car;
}
```

- Remember: typedef (so you can use car without the struct)

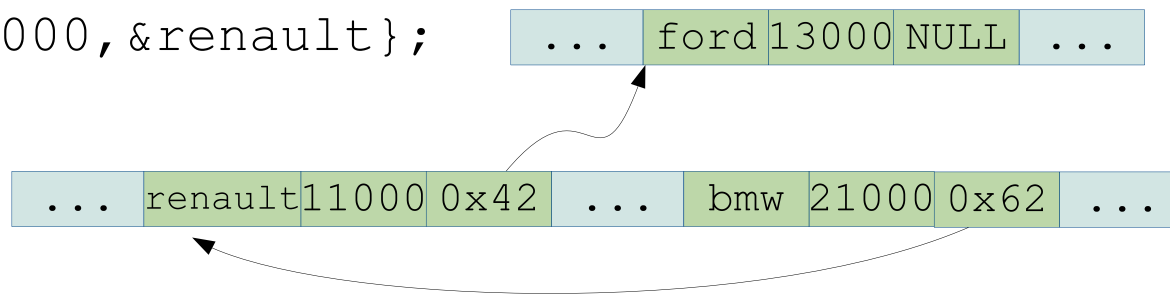
```
typedef struct car car;
struct car
{
    char name[30];
    int value;
    car * next_car;
}
```

or

```
typedef struct car
{
    char name[30];
    int value;
    struct car * next_car;
}  car;
```

Example: Linked List

```
main()
{  car ford ={"ford",13000,NULL};
   car renault ={"renault",11000,&ford};
   car bmw ={"bmw",21000,&renault};
   print_cars(&bmw);
}
```



- NULL is the typical way of saying "it points nowhere"
  - check to see if the value is meaningful **before** you use it
  - uninitialised memory may contain garbage!

```
print_cars(car* thiscar)
{    if (thiscar == NULL)
         return;
     printf("%s\n",thiscar->name);
     print_cars(thiscar->next_car);
}
```

Example: Linked List

- What about allocating memory during runtime? In Java:

```
Car make_car(Car next_car, int price, String name)
{
   Car new_car = new Car();
   new_car.name = name;
   new_car.value = price;
   new_car.next = next_car;
   return new_car;
}
```

- In C, the same idea might look like this…

```
car* new_car(car* next_car , int price, char* name)
{
  car temp_car;
  car* car_pointer = &temp_car;           ← a pointer to a local variable
  car_pointer->value = price;
  car_pointer-> next_car = next_car;
  strcpy(car_pointer->name, name);
  return car_pointer;
}
```

File:pointer_to_local_var.c

Example: Linked List

- **Do never do this! (really, not ever!)**
  - A new struct is created – but **it's on the stack**!
  - This memory will soon be overwritten….!
- Instead, create lasting storage location; explicitly use malloc:

```
car* new_car(car* next_car , int price, char* name)
{
    car* car_pointer = malloc(sizeof(car));
    car_pointer->value  = price;
    car_pointer->next_car = next_car;
    strcpy(car_pointer->name,name);
    return car_pointer;
}
```

- Car contains a char array
  - for which the memory has been allocated in a fixed position in the struct
  - need to use strcpy to copy the desired name into that char []

Example: Linked List

```
void new_car_print(car* next_car , int price, char* name)
{
  car temp_car;
  car* car_pointer = &temp_car;
  car_pointer->value = price;
  car_pointer-> next_car = next_car;
  strcpy(car_pointer->name, get_name);

  print_car(temp_car);
}
```

- temp_car is allocated on the stack
  - it is safe **during this function**,
  - including any function that is called from `new_car_print`.
  - but ***not* after the return**!

Example: Linked List

- Using the new_car function

```
car* new_car(char* name, int price, car* next_car)
{
    car* car_pointer = malloc(sizeof(car));
    car_pointer->value  = price;
    car_pointer->next_car = next_car;
    strcpy(car_pointer->name,name);
    return car_pointer;
}
```

same as before

```
main()
{     car *ford =new_car("ford",13000,NULL);
      car *renault =new_car("renault",11000,ford);
      car *bmw =new_car("bmw",21000,renault);
      print_cars(bmw);
}
```

`File:linked_cars.c`

(Note that memory has not been freed – bad programmer!)

Separating Storage and Data

- Previous car example mixes 'linked list' and 'car data'
  - This is not usually a good idea!
- Usually better: separate the ‘collection’ from the contained data
  - Aids reuse of code
  - Better encapsulates data
  - So more easily allows changes to the underlying structure

```
struct node{car* data; struct node*next;};
struct node *head=NULL;
```

- Or to make it more generic:

```
struct node{void* data; struct node*next;};
struct node *head=NULL;
```

Generic Dynamic Data Structures

```c
typedef struct node
{
    struct node* next;
    void* data;
} node;

node* head;

void add_node(void* pointer)
{
    node* new_node = malloc(sizeof(node));
    new_node->data = pointer;
    new_node->next = head;
    head = new_node;
}
```

File:linked_generic.c

Generic Dynamic Data Structures

```
print_cars(node* start_node)
{   if (start_node == NULL)
    return;
    car* thiscar = start_node->data; //note the implicit cast here!
    printf("%s\n",thiscar->name);
    print_cars(start_node->next);
}

main()
{   car *renault =new_car("renault",11000);
    car *ford =new_car("ford",13000);
    car *bmw =new_car("bmw",21000);

    add_node(renault);
    add_node(ford);
    add_node(bmw);

    print_cars(head);
}
```

File:linked_generic.c

Generic DDS & different data types

- Previous example uses a void* pointer
  - i.e., a `node` does not know what type of data it points to
  - if you get data from a node, **you** will need to do the casting
  - **you** are **responsible** for knowing what a node stores!

- What if you want to store **different data types** in the same data structure?

```
car *bmw = new_car("bmw",21000);
bus *optare = new_bus("optare",80000);
....
add_node(renault);
add_node(optare);
```

- See above! You will need to code that yourself!
  - You could consider C++ (and go full-blown object oriented)
  - but not needed per se...

Example: Storing different types

- Define identifying types

```
enum  vehicle_type {car_type,bus_type} vehicle_type;
```

- And store it in the ‘node’

```
typedef struct node
{   enum vehicle_type type;
    struct node* next;
    void* data;
} node;

void add_node(void* pointer, enum vehicle_type type)
{   node* new_node = malloc(sizeof(node));
    new_node->type = type;
    new_node->data = pointer;
    new_node->next = head;
    head = new_node;
}
```

Abstract Data Types

- It is common, in modern languages, to separate the idea of an 'Abstract Data Type' from the implementation
- Typical Abstract data types:
  - Container
  - Deque
  - List
  - Map
  - Multimap
  - Multiset
  - Priority queue
  - Queue
  - Set
  - Stack
  - Tree
  - Graph

http://en.wikipedia.org/wiki/Abstract_data_type

Abstract Data Types (ADT)

- An ADT specifies the **user interface...**
  - i.e., the functions via which the user interacts with it
- ...it does not specify the **implementation**
  - implementation should be invisible to the calling functions
  - hence the implementation can be easily changed

- For instance Java has an abstract class `List`
  - There are several methods defined for List: add, remove, get, etc.
  - It may typically be implemented as an ArrayList or LinkedList

- Techniques we have seen allow you to create your own implementation of ADTs!

Implementing an ADT

```c
struct node{int data;struct node*next;};
struct node *head=NULL;
struct node *tail=NULL;

void push(int data)
{
  struct node* new_node=(struct node*)
  malloc(sizeof(struct node));
  new_node->data=data;
  new_node->next=NULL;
  if(tail==NULL)
    head=tail=new_node;
  else
  {
    tail->next=new_node;
    tail=new_node;
  }
}
```

```c
int pop()
{
  int temp;
  struct node *remove_node=head;
  temp=head->data;
  if(head==tail)
    head=tail=NULL;
  else
    head=head->next;
  free(remove_node);
  return temp;
}

main()
{
  push(0);
  push(1);
  push(2);
  printf("%d\n",pop());
  printf("%d\n",pop());
  printf("%d\n",pop());
}
```

Question: What abstract data type is this?

Review Dynamic Data Structures

- Dynamic data structures
  - enable storage of your data that grows with requirements
  - 2 worked out examples: ‘ArrayList’ and ‘LinkedList’
  - can be made ‘generic’ with the use of a void* pointer
    - but care should be taken to use the correct data type

- There are a number abstract data types that may be useful to implement
  - e.g., List, Queue, Stack
  - get to understand when which type is useful!

Function Pointers

function pointers

- A pointer holds an *address*
- An address is a *location* in memory
- Memory locations can contain either program *data* or program *code*
- We have seen pointers to data
- You can also have pointers to code, in the form of *function pointers*

```
org 100h
main:
    mov dx,get_message
    call dx
    mov ah, 9
    int 21h
;this is a BIOS call to print out text
    mov ah, 0
    int 16h;  wait for a key

    mov ax, 4c00h
    int 21h
get_message:
    mov dx,msg
    ret
    resb 6
msg:db "Testing the program"
    db 10
    db  "Press any key$"
; the $ marks the end of the text
```

function pointers

- A **function pointer** is defined like this

```
return_type (* variable_name) (function_parameters)
```

- contains a pointer to a function that accepts function_parameters and returns return_type
- called with the same syntax:

```
result = (* variable_name) (function_parameters);
```

- Example:

```
int add_numbers(int x, int y){ return x+y };

int ( * function_pointer ) (int, int);
function_pointer = &add_numbers;
int result = (*function_pointer)(2,4);
```

File:e32.c

Arrays of Function Pointers

- You can also have arrays of function pointers
  - **declare:** `return_type (* variable_name[4]) (function_parameters)`
  - **call:** `result = (* variable_name[i]) (function_parameters);`

- Example:

```
int add_numbers(int x, int y){ return x+y};
int multiply_numbers(int x, int y){ return x*y};
main()
{
  int ( * function_pointers[2] ) (int, int);
  function_pointers[0] = &add_numbers;
  function_pointers[1] = &multiply_numbers;
  int result = (*function_pointers[0])(2,4);
  result = (*function_pointers[1])(2,4);
}
```

File:e33.c

Arrays of Function Pointers – 2

- An array of function pointers can be quite useful if you want to call a different function for each system state
  - neat alternative to large block of switch statements

- As with everything, you can have pointers to blocks of function pointers….

```
int  ( **function_pointers ) (int, int);
function_pointers =
    malloc(sizeof (int (*) ( int, int)) * 4);
```

Callbacks

- Function pointers are also often used as **callbacks**
  - https://en.wikipedia.org/wiki/Callback_%28computer_programming%29
- E.g.,:
  - Call a function `task` to perform some general task
  - You pass a function pointer as a parameter: `task(&my_func)`
  - if `task` needs some more information, it can 'callback' to the function you specified
  - `my_func` takes care of some of the details

- This is often used by device drivers
  - e.g., call some function every time the screen does a vertical sync
- ...and standard library functions

(In Java, you would typically program similar behavior by implementing an Interface.)

Example: qsort

- C Standard library <stdlib.h> qsort
  - sorts an array

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*)  );
```

- The parameters are:
  - A (void) base pointer to the start of the array to sort
  - The number of elements in the array
  - The size of each element (in bytes)
  - A function pointer for a function that can compare any two elements

`size_t` is a special data type to refer to the size of an object – at least a 16-bit integer

- So, to call qsort, you need a function that returns an int and takes two const void pointers as arguments.
  - Remember a void pointer points to an unknown type of data
  - Why does this use void pointers?

Example: qsort

The C reference material will tell you what the compar function should do

- man qsort
- http://www.cplusplus.com/reference/cstdlib/qsort/

return value	meaning
<0	The element pointed by *p1* goes before the element pointed by *p2*
0	The element pointed by *p1* is equivalent to the element pointed by *p2*
>0	The element pointed by *p1* goes after the element pointed by *p2*

Example: qsort

- Example:

```
int integer_compare_ascending(const void *p1,const void *p2)
{
    return(*(int*)p1)-(*(int*)p2);
}

int string_compare_ascending(const void *p1,const void *p2)
{
    return strcmp(p1,p2);
}
```

Example: qsort

```
int integer_compare_ascending(const void *p1,const void *p2)
{
    return(*(int*)p1)-(*(int*)p2);
}

int string_compare_ascending(const void *p1,const void *p2)
{
    return strcmp(p1,p2);
}
main()
{
    int i =0;
    int num[5]={33,123,11,-3,9};
    char strings[][20]={"aaa","AAA","abc","dddz","bbbb"};
    qsort(num,5,sizeof(int),&integer_compare_ascending);
    qsort(strings,5,sizeof(char)*20,&string_compare_ascending);
    for (i=0; i < 5; i++)
        printf("%d\t%s\t\n",num[i],strings[i]);
}
```

Review

- Dynamic data structures
  - enable storage of your data that grows with requirements
  - separate data and container
  - can be made 'generic' with the use of a void* pointer
    - but care should be taken to use the correct data type
- Abstract data types that may be useful to implement
  - e.g., List, Queue, Stack
  - get to understand when which type is useful!
- Function pointers
  - Provide a method of storing which function to call
  - Can be stored in arrays and accessed by index
  - Can be used for 'callbacks', for use with standard library functions