COMP281
Principles of C and memory management

Lecture 8

Dr. Frans Oliehoek
Department of Computer Science
University of Liverpool

Reminder

- Assignment 2 due: **March 1st, 4:30pm**.

Last Time

- Multi-dimensional arrays
  - vs. arrays of arrays…
  - Q: main difference?
- Custom data types:
  - structs
  - unions
  - Q: main difference?

Today

- Wrap-up custom data types
  - typedefs, enums
  - incl. forward declarations
    - needed to get 'nested' structs or functions to work
- Dynamic data structures
  - combining malloc and structs...
  - ...to have data structures that grow as needed
- Abstract Data Types

Wrap up "Custom Data Types"

Refresher: structs

- Useful to group data:
  - Declaration:

```
struct comp281result {
  char assignment;
  int grade;
  int student_number;
}
```

  - Definition:

```
int main(){
  struct comp281result Johns_result = {1,66,42};
  ...
}
```

- Remember: C has **data** and **pointers**...
  - If you say 'data' it will pass/store data
  - Q: where does `Johns_result` live in memory?

Refresher: Unions

- Store **different types of data in the same space**

```
union assignment
{
    int raw_mark ;
    char letter_grade ;
};

main ( )
{
    union assignment assignment_result;
    assignment_result.raw_mark =100;
    assignment_result.letter_grade='A' ;
    printf("%d \n",assignment_result.raw_mark);
}
```

Question: What does this code produce?

`union.c`

  - bug-sensitive...
  - ...might only be worth the trouble under severe memory constraints

typedefs

- Cleaner syntax via a ‘typedef’
  - see also lecture 1
  - The format is:

```
typedef old_type_name name_type_name;
```

  - e.g. `typedef int Integer;`
  - Now you can use the word Integer instead of int; e.g., `Integer i = 10;`

- Similarly, for structs:
  - avoid typing 'struct' again and again...

```
typedef struct { int number, char street[100] } address;
address my_address={108,”Anywhere”};
```

typedefs

these are now 'aliases' for the old type name

- Cleaner syntax via a ‘typedef’
  - see also lecture 1
  - The format is:

```
typedef old_type_name name_type_name;
```

  - e.g. `typedef int Integer;`
  - Now you can use the word Integer instead of int; e.g., `Integer i = 10;`

- Similarly, for structs:
  - avoid typing 'struct' again and again...

```
typedef struct { int number, char street[100] } address;
address my_address={108,"Anywhere"};
```

typedefs – example

```
typedef struct
{
  int number;
  char street[100];
} address;

typedef struct
{
  char name[100];
  address *address;
} record;

address my_address=
  {108,"Anywhere Road"};
record my_record =
  { "Tony",&my_address};
```

```
void nextDoor(
  address* an_address)
{
  an_address->number += 2;
}

main()
{
  nextDoor(&my_address);
  printf("%d %s \n",
    my_record.address->number,
    my_record.name
  );
}
```

struct.c

Enumerations

- What if your program needs to use something other than numbers?
- E.g., for storing the system state?
  - such as {running, sleeping, waiting, finished}
  - They can be 'enumerated'

```
enum day {sunday, monday, tuesday, wednesday,
thursday, friday, saturday}
```

  - sunday == 0, monday == 1, etc.

- Can be stored in a variable of the same 'enum day' type:

```
enum day today;
today = saturday;
```

- You can still treat the variable as an integer:

```
today = today + 1;
printf("%d \n",today);
```

using enums for related integer constants is good practice

- avoid too cryptic arithmetic

Enumerations

- By default, the first element has value 0
  - each subsequent element increases by 1
  - but each can be set manually

- What does this do?

```
enum state {off,warmup=2,starting,on=5};
printf("%d \n",starting+on);
```

enum.c

- You can also remove the identifier, and the values are treated as ints:

```
enum {zero,one,two,three};
int sum = two+three;
```

enum.c

Forward Declarations

- Remember:
  - A variable must be declared before it is used
  - A function must be declared before it is used

- Question: how would you do the following?

```
void do_even(int x)
{
    if (x <=0)
        return;
    do_odd(x-1);
}
void do_odd(int x)
{
    do_even(x-1);
}
```

or

```
struct even_entry
{
  char value;
  struct odd_entry* next;
}

struct odd_entry
{
  char value;
  struct even_entry* next;
}
```

Forward Declarations

- You need to include a function declaration ('a prototype', see lecture 4!)
  - called a **forward declaration**
  - enables the compiler understands what you want to do

```
void do_odd(int x);

void do_even(int x)
{
   if (x <=0)
   return;
   do_odd(x+1);
}
void do_odd(int x)
{
   do_even(x+1);
}
```

this is the forward declaration

(the struct version:
left as an exercise)

Dynamic Data Structures

Dynamic Data

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

...so...

- ...can now implement **dynamic data structures**:
  - that **flexibly grows/shrinks memory usage** to store date
  - e.g.,: trees, linked lists, priority queues etc.
- Not included as part of the C standard libraries
  - unlike Java, C++ etc.
- It's fairly easy to write your own

Example: Resizable Array

		array							
...	...	**42**	**43**	**44**	G	A	R	B	...

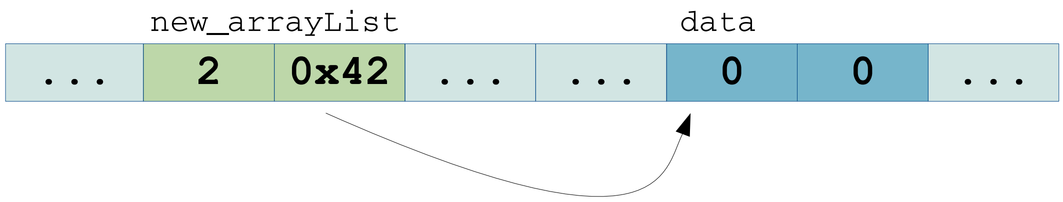
→ array[4] = 3; →

		array							
...	...	**42**	**43**	**44**	**0**	**3**	R	B	...

Example: Resizable Array

- Arrays are efficient to access, but:
  - size is fixed
  - difficult to insert into the middle
  - These issues can be addressed... (Java has an 'ArrayList' ...)

```
typedef struct
{   int size;
    int* data;
} arrayList;
```

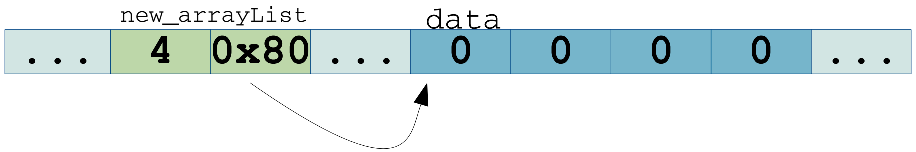



```
/*Create new array of initial_size: */
arrayList* new_array(int initial_size)
{
    arrayList* new_arrayList = malloc(sizeof(arrayList));
    new_arrayList->size = initial_size;
    new_arrayList->data = calloc(sizeof(int),initial_size);
    return new_arrayList;
}
```

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;
}
```



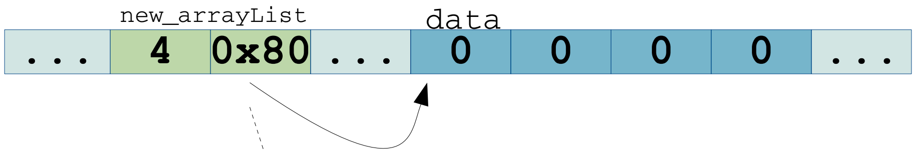
grow if needed
(to 2x index)

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

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;
}
```



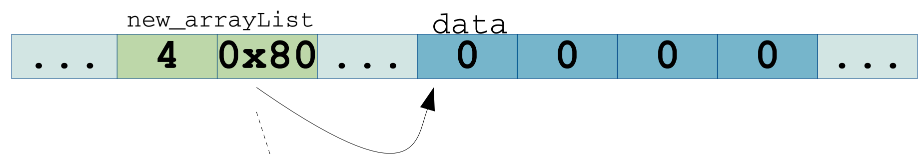
data may have been moved!

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

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!

tradeoff in new size:

- larger: possibly wasted memory
- smaller: possibly many resizes (slow)

*2 is a reasonable compromise
(at most 2* memory usage, at most log(n) resizes needed)

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

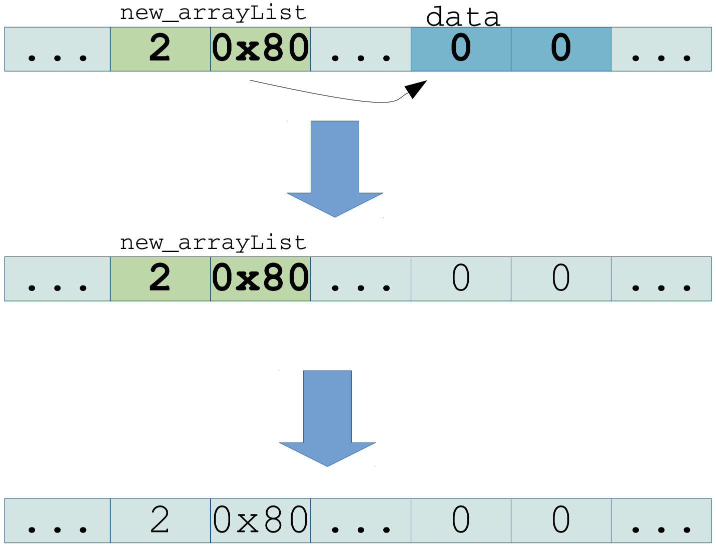
Example: Resizable Array

```
main()
{
  arrayList* array = new_array(10);

  set(array,5,100);
  printf("%d  \n",get(array,5));
  printf("array size = %d \n",array->size);

  set(array,50,1000);
  printf("%d  \n",get(array,50));
  printf("array size = %d \n",array->size);
}

$ ./a.out
100
array size = 10
1000
array size = 100
```

Oh... and don't forget

- And don't forget to have a function to free the memory
  - (including the memory for the arrayList structure)

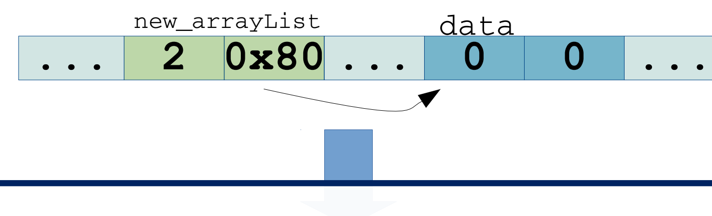
```
void delete_array(arrayList* array)
{
    /* free the memory used to
       store the actual array: */
    free(array->data);

    /* free the memory used for
       the arrayList struct: */
    free(array);
}
```



Oh... and don't forget

- And don't forget to have a function to free the memory
  - (including the memory for the arrayList structure)

```
void delete_array(arrayList* array)
{
    /* free the memory used to
       store the actual array: */
    free(array->data);

    /* free the memory used for
       the arrayList struct: */
    free(array);
}
```



On many slides, I will not show releasing memory...

- (space on slides is limited!)
- ...but you **will** have to do this in your assignments!

arrayList vs array

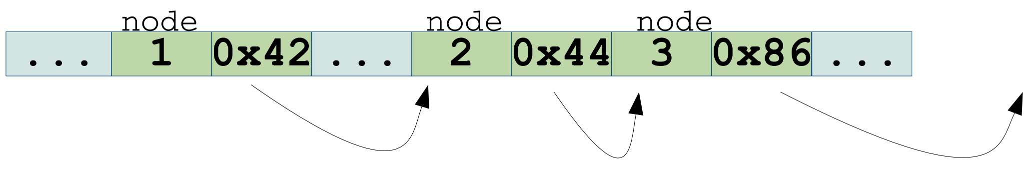
Advantages:

- Error prevention; you can't write over the end of an array
- The programmer can use it with no prior knowledge of the data size
- Is still reasonably efficient for random-accesses

Disadvantages

- Readability (although this is subjective)
- Efficiency
  - Requires a function call for accessing every element the array
  - (Although this may be inlined during compilation)
  - Many resizes could be slow
- Making this generic is difficult
  - Might consider moving to C++ if you need this for many types…

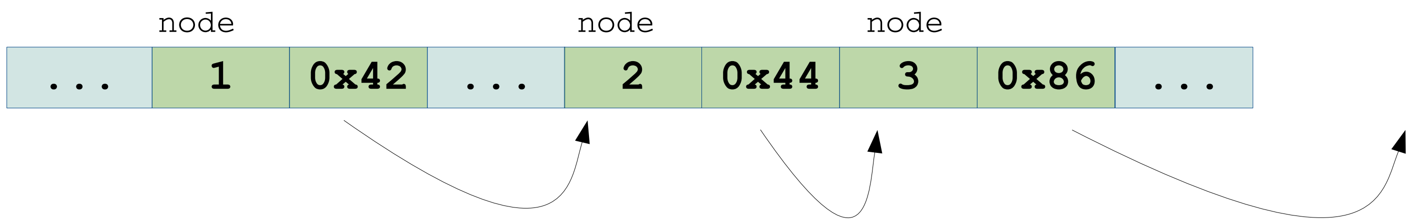
Another Example: Linked Lists



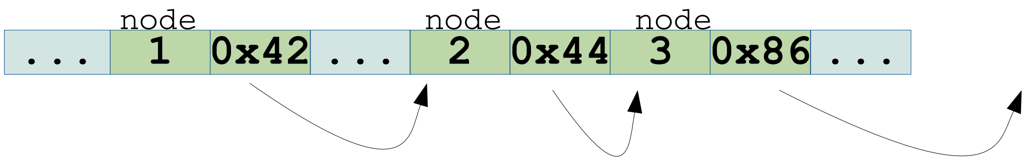
Linked Lists

- 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)

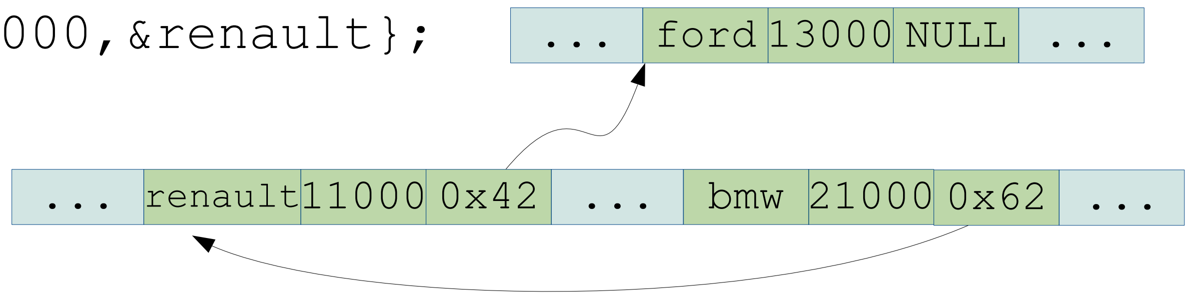
```
typedef struct car car;
struct car
{
     char name[30];
     int value;
     car * next_car;
}
```

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', Generic data structures & Abstract data types

Separating Storage and Data

- Previous car example mixes 'linked list' and 'car data'
- 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;
```

- One step further: **make it generic**

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

Generic Dynamic Data Structures

```
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

```
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;
}
```

Notice:

- node* relates to the data structure
- void* relates to the (generic) data

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

```
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;
  }
}
```

```
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

- Custom data types
  - structs, unions, typedefs, enums, forward declarations
- Structs allow encapsulation of data
  - possibly large amounts of data, it may be best to use a pointer
  - the required memory can be allocated during runtime with malloc
- 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!