Yet more about pointers…

Void Pointers

Function Pointers with Void Pointer

Problem: previous ADT code is not general enough

- As you noticed with A4, we are reliant on #ifdef to implement different types for the different problems
- This also caused us to have to recompile our ADT “library” each time we used it with a different problem

Solution:

- Let’s write a routine that searches for array elements and returns their index in the array.
- But…let’s do this in a way that works *no matter the type of the array elements*!

Function Pointers with Void Pointer

Problem: previous ADT code is not general enough

- As you noticed with A4, we are reliant on #ifdef to implement different types for the different problems
- This also caused us to have to recompile our ADT "library" each time we used it with a different problem

Solution:

- Using void pointers in our function pointers' argument types

Function Pointers with Void Pointer

Problem: previous ADT code is not general enough

- As you noticed with A4, we are reliant on #ifdef to implement different types for the different problems
- This also caused us to have to recompile our ADT "library" each time we used it with a different problem

Solution:

- Using void pointers in our function pointers' argument types

But what does that mean?

Function Pointers with Void Pointer

What are void pointers?

- A void pointer is a “generic” pointer
- Remember, a pointer has two parts:
    1. the memory address (*where the pointer is pointing*)
        - this is always of size int
        - so it doesn’t matter the “type” of the pointer
    2. the increment size
        - how many bytes needed to get to the next element

Function Pointers with Void Pointer

What are void pointers?

- A void pointer is a “generic” pointer
- Remember, a pointer has two parts:
  1. the memory address (*where the pointer is pointing*)
  2. the increment size
- With a void pointer
  1. the memory address is given as usual
  2. however, the increment size is equal to 0

Function Pointers with Void Pointer

What are void pointers?

- A void pointer is a "generic" pointer
- Remember, a pointer has two parts:
    1. the memory address (*where the pointer is pointing*)
    2. the increment size
- When a pointer passed in as an argument to a void * parameter…
    1. the memory address is kept the same
    2. the increment size is set to 0
        - a cast is performed by C on the pointer being passed in
        - during the cast: the address is kept
the increment size remove (set to 0)
        - the resulting ptr is stored in the parameter variable

Function Pointers with Void Pointer

When used inside the function:

- The void point can be **cast** into the pointer type the function "knows" it should be
- Remember the function itself is …
  - going to be a function pointer
  - written by the user of your code (possibly yourself)
  - passed in to the general calling function (i.e. as a callback)
- So the pointer type the void pointer should be cast as … will be coded by the user of the function
- Thus your code *remains general!*, with no need to recompile

Function Pointers with Void Pointer

When used inside the function:

- The cast can be done through assignment (*no cast needed*)

  e.g. let the sum functions signature be

  ```
  void * sum( List * list,
              (*add_fn)(void *x, void *y),
              void * init_value)
  ```

  the user, knowing that "value" is of type double in their application, could then write the following:

  ```
  int start = 0.0;
  int * total = sum(list, add, &start);
  printf("The sum = %d\n", *total);
  ```

So, sum, by using void pointers has be used generally

Function Pointers with Void Pointer

Whe

ion,

Add function using void pointers

```
int add(void * answer_arg, void * x_arg, void * y_arg){
    int * x = x_arg;
    int * y = y_arg;
    int * answer = answer_arg;
    int result = FAILURE
    if (!overflow_condition(x, y)){
        *answer = *x + *y;
        result = SUCCESS;
    }
    return result;
}
```

Note: Since we can only be type general using void pointers,
So we cannot just return answer, as we would with int add(int, int),
as we would have to malloc the space for it.

So, sum, by using void pointers has be used generally

Function Pointers with Void Pointer

When used inside the function:

- The cast can be done through assignment (*no cast needed*)

  e.g. let the sum functions signature be

  ```
  void * sum( List * list,
              (*add_fn)(void *x, void *y),
              void * init_value)
  ```

  the user, knowing that “value” is of type double in their application, could then write the following:

  ```
  int start = 0.0;
  int * total = sum(list, add, &start);
  printf("The sum = %d\n", *total);
  ```

So, sum, by using void pointers has be used generally

Generalizing functions for the ADT

There are many other changes needed to the linked lists functions:

- For example, in push or append, you have to create a node, where you need to copy a value using an "abstract type", not just an int or a char [80], … who knows what
- There are two approaches to solving this problem
  1. Fixed size value approach using void pointers
  2. Passing in a create_node function as a function pointer

Generalizing functions for the ADT

Push example (*previously called add_front*): Original code

```
Node * push(Node ** head, int value){
    Node *new_node = malloc(sizeof(Node));
    if (new_node != NULL) {
        new_node->num = value;
        new_node->next = *head;
        *head = new_node;
    }
    return new_node
}
```

Generalizing functions for the ADT

Push example (*previously called add_front*): New code

- Fixed size value approach using void pointers

```
Node * push(Node ** head, void * value, int size){

    Node *new_node = malloc(sizeof(Node));
    char * value_copy = malloc(size);

    if (new_node != NULL && value_copy != NULL) {

        memcpy(value_copy, value, size);

        new_node->value = value_copy;
        new_node->next = *head;
        *head = new_node;
        return new_node;
    } else {
        return NULL;
    }
}
```

increment size
= 1 byte

copy content
byte-by-byte

copy new pointer
into node

Generalizing functions for the ADT

Push example (*previously called add_front*): New code

- Passing in a create_node function as a function pointer

```
Node * push(Node ** head, void * value,
            void * (* create_node)(void *){

    Node *new_node = malloc(sizeof(Node));
    Void * value_copy = create_node(value);

    if (new_node != NULL && value_copy != NULL) {
        new_node->value = value_copy
        new_node->next = *head;
        *head = new_node;
        return new_node;
    } else {
        return NULL;
    }
}
```

create_node passed in
by the user

Generalizing functions for the ADT

Push example (*previously called add_front*): New code

- Passing in a create_node function as a function pointer

```
Node * push( Node ** head, void * value,
             void * (* create_node)(void *){

    Node *new_node = malloc(sizeof(Node));
    Void * value_copy = create_node(value);

    if (new_node != NULL && value_copy != NULL) {
        new_node->value = value_copy
        new_node->next = *head;
        *head = new_node;
        return new_node;
    } else {
        return NULL;
    }
}
```

will have
malloc'd space

Generalizing functions for the ADT

Push example (*previously called add_front*): New code

- Passing in a create_node function as a function pointer

```
Node

    N
    V

    i
        new_node->value = value_copy;
        new_node->next = *head;
        *head = new_node;
        return new_node;
    } else {
        return NULL;
    }
}
```

In either case:

remember that value in the new node

has been malloc'd and so needs to be freed

when the list is to be destroyed

will have
malloc'd space

Generalizing functions for the ADT

Push example (*previously called add_front*): **New code**

- **Passing in a create_node function as a function pointer**

```
Node * push( Node ** head, void * value
```

Almost all Node functions to be used by the ADT will have to be tweaked in this way

```
    if (new_node != NULL && value_copy != NULL) {
        new_node->value = value_copy
        new_node->next = *head;
        *head = new_node;
        return new_node;
    } else {
        return NULL;
    }
}
```

will have
malloc'd space