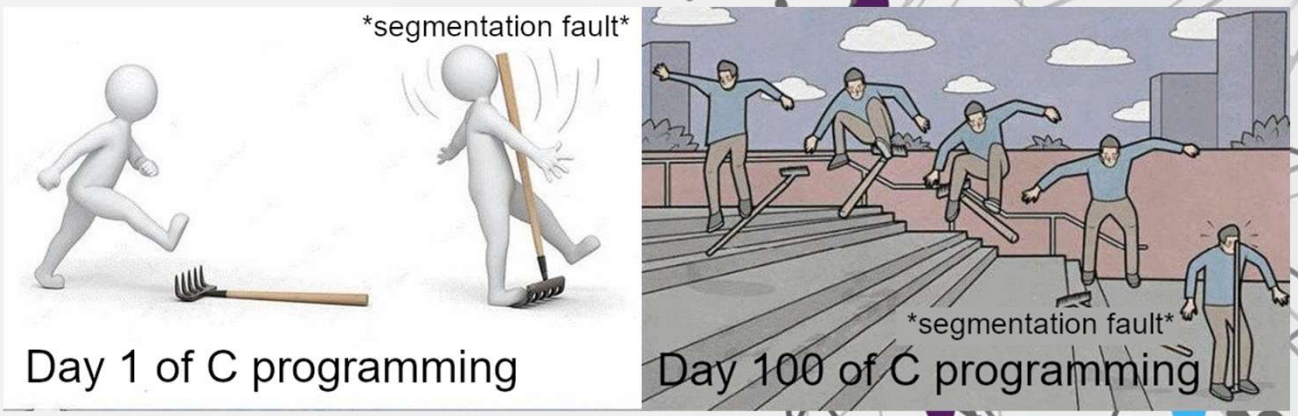


TA 5

Generic programming

GENERIC PROGRAMMING

GENERIC PROGRAMMING

Goal

Write code that is **reusable** and **type-independent**, without **code duplication.**

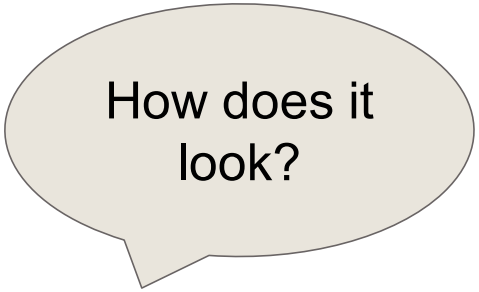
- Since C is **type based**, generic programming is often complicated and imperfect.
- Several ways to achieve:
  - **void* / char***
  - **Function Pointer**
  - macros

GENERIC POINTER VOID*

- void* can point to **any** data type but when we use it:
  - **No** pointer arithmetic.
  - **No** dereferencing.
- No **type safety** – void* generic programming bypass the type system, hence gives up type safety.
- Might cause reduced **efficiency** – often demands extensive use of **casting.**

Function Pointers

- C Allows to declare a pointer to a function.
- Pointer to a function **points the code** of the function.
- Function pointers can be **assigned**, **passed** to and from function, **placed** in arrays, etc...
- Read with the **right-left rule.**
- We often use **typedef** to ease the use of function pointers.



Function Pointers

```
void fun(int a)
{
        printf("The Value of a is %d ",
a);
}
```

```
void main()
{
        void (*fun_ptr)(int) = fun;

        fun_ptr(20);
        return 0;
}
```

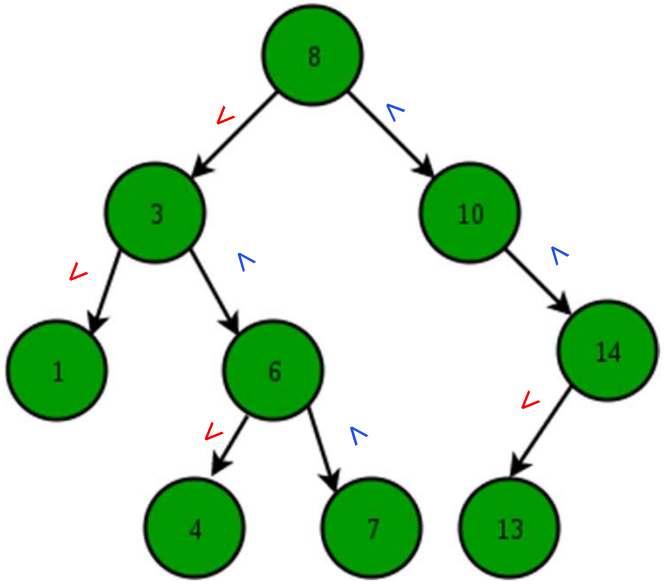
```
typedef void (*int_func)(int);

void main()
{
        int_func fun_ptr = fun;
        fun_ptr(20);
        return 0;
}
```

GENERIC BST WITH VOID*

BST – BINARY SEARCH TREE (With unique values)

- Each Node has up to two subtrees (binary).
- **Left** subtree of each node contains only nodes with **smaller** values.
- **Right** subtree of each node contains only nodes with **greater** values.



BST Interface

```
Bst * tree = new_bst(...);

bst_insert(tree, <value_1>);

bst_insert(tree, <value_2>);

bst_insert(tree, <value_3>);

print_tree(...);

free_tree(...);
```

BST – WHAT DO WE NEED TO IMPLEMENT?

- BST itself
- Tree basic unit - **Node**
  - A **Node** contains:
    - Right Node
    - Left Node
    - Data
- Insertion
- Free (Tree deletion)
- More functions (We won't implement)

int BST – int NODE

- First, let's look at non-generic, int BST: (partial implementation)

How do we generalize?

```
typedef struct Node {
    struct Node *left;
    struct Node *right;
    int data;
} Node;
```

```
typedef struct Bst
{
    Node *root;
} Bst;
```

```
Node *create_new_node(int data)
{
    Node *new_node = malloc(sizeof(Node));
    if (new_node != NULL)
    {
        new_node->left = NULL;
        new_node->right = NULL;
        new_node->data = data;
    }
    return new_node;
}
```

generic BST – generic NODE

- **1.** Implement the struct "*Node*"
- **2.** Try to implement the function "**Node* new_node(void* data)**"

- **NOTE: We want our Node to have strong ownership on a copy of the data!**

What is Copy Function?

- We want to create a new node with a given pointer to “new_data”
- How can we deep copy the given “new_data” into our data?
- Assuming our actual data type is int (which is pointed by int*)

```
void* copy_int(void *src)
{
    //    memcpy(new_data, data, elem_size);
    int* int_src = src;
    int *new_val = malloc(sizeof(int));
    if(new_val != NULL)
    {
        *new_val = *int_src;
    }
    return new_val;
}
```

We cast our void pointer into int pointers

We can dereference our int pointers so copy the value

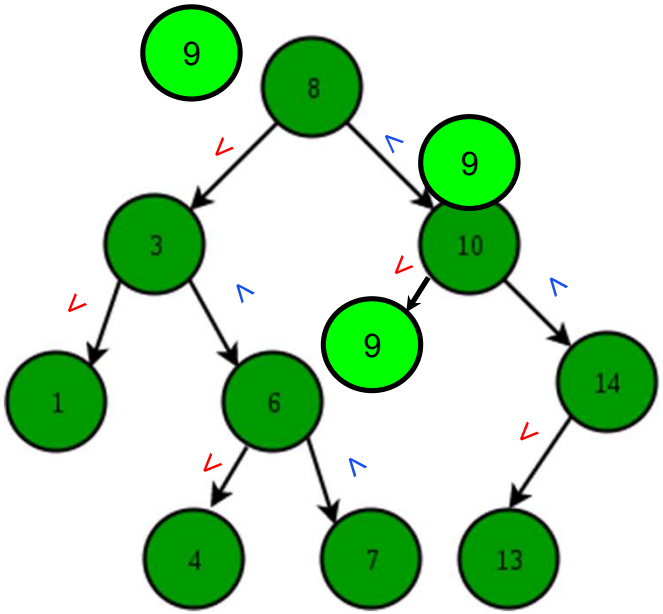
BST – NODE

- **3.** Write typedef for copy function named **copy_func** that:
  - a. receives a **generic** pointer.
  - b. returns a pointer of a hard copy of the original pointer.
- **4.** Add **copy_f** field of type **copy_func** to Bst.
- **5.** Complete implementation of new_node():
  - a. Get **copy_f** variable of type **copy_func** as an argument.
  - b. Use **copy_f** to copy the given data.

BST – Insert

- How can we insert new node to BST?
  - Starting from the tree root:
    - if new data is larger than root data: go right
    - Else: go left
  - Keep until reaching to NULL leaf
- Note – we need to be able to **compare** between nodes!
- Let's observe the int BST case.

insert 9:



int BST – insert

```
int insert(Bst *tree, int data)
{
    Node *root = tree->root;
    Node *new_node = create_new_node(data);
    if (new_node == NULL) return 0;
    if (root == NULL)
    {
        tree->root = new_node;
        return 1;
    }

    while (root != NULL)
    {
        if (data > root->data)
        {
            if(root->right == NULL)
            {
                root->right = new_node;
                break;
            }
            root = root->right;
        }
        else
        {
            if(root->left == NULL)
            {
                root->left = new_node;
                break;
            }
            root = root->left;
        }
    }
    return 1;
}
```

A lot of code but actually simple

BST - INSERTION

Let's break it down

```c
int insert(Bst *tree, void *data)
{
    Node *root = tree->root;
    Node *new_node = create_new_node(data, tree->copy_f);
    if (new_node == NULL) return 0;
    if (root == NULL)
    {
        tree->root = new_node;
        return 1;
    }
    while (root != NULL)
    {
        if (root->data > data) {
            if(root->right == NULL)
            {
                root->right = new_node;
                break;
            }
            root = root->right;
        }
        else if(root->data < data)
        {
            if(root->left == NULL)
            {
                root->left = new_node;
                break;
            }
            root = root->left;
        }
    }
    return 1;
}
```

First node of the tree is the root

If our root is bigger - go right. Otherwise - go left

If the current left/right is empty - fill it

Simple right?

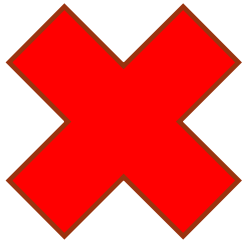
But Wait What does even mean in a generic BST?

Right now we compare addresses, is that correct?

BST - INSERTION

Of course not, we meant to write:

```
if (*(*root)-> data > *givenData)
```



We are trying to dereference void pointer!

How do we solve it??

More functions!

(A Comparator ☺)

BST – INSERTION

- **Comparators** help us to compare between two elements.
- Our current elements are void* variables.
- Assume that our void* pointers are pointing to int variables, this is fine, right?

```
int int_comparator(void *firstElement, void *secondElement)
{
        int intValueOfFirst = *((int *)firstElement);
        int intValueOfSecond = *((int *)secondElement);

        return intValueOfFirst - intValueOfSecond;
}
```

We cast our void pointers into int pointers in order to dereference them
Remember: this would work with any other type just as its working for int

Now we can successfully compare the values

BST – INSERTION - SUMMARY

- We need to be able to **compare** between the **data** of the nodes.
- Compare **function** for **each data type** we'll use with our BST!
- We want **each Node** to use the right **compare function**.
- BST with **int*** data – the code will "**remember**" to use the **int*** version, BST with double* data – the code will "remember" to use the double* version etc.
- This "memory" can be implemented by **function pointers.**

BST – Insert

- **6.** Look at typedef for comparator function **comp_func**.
- **7.** Add **comp_f** field of type **comp_func** to “Bst” struct.
- **8.** Fix **bst_insert** function, start with the argument “void * data” and look forward if there is another argument needed.

Tip: C lion, this button jumps from declaration in .h to implementation .c and vise versa

BST – FREE

- What about memory?

- Our copy function is allocating memory! How can we free it?
    - Maybe just *free(root->data)* ?
  - But…what if our data is more complex than int?
  - We need a free function!

BST – Free Allocations

- **9.** Look at **free_func** typedef.
- **10.** Add **free_f** field of type **free_func** to Bst.
- **11.** Look at free_tree function
- **12.** Fix new_bst function.
- **13.** Run main()

BONUS - STRINGS

SHLEMIEL THE PAINTER'S

An Example by Joel Spolsky*

* ***StackOverflow*** Co-Founder, among other things

STRING FUNCTIONS - Reminder

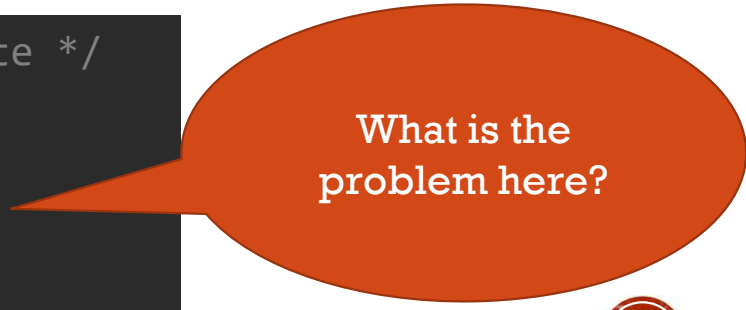
- `size_t strlen(const char *str)`
  - returns the **length** of str
- `char *strcpy(char *dest, const char *src)`
  - **Copies** the string pointed by *src* to *dest*.
  - Returns pointer to the destination string *dest*.
- `char *strcat(char *dest, const char *src)`
  - **Appends** the string pointed by *src* to the end of the string pointed by *dest.*
  - returns a pointer to the resulting string *dest*.

STRING FUNCTIONS – CAREFUL!

```
char* strcat( char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
}
```

- First, we iterate over the first string **looking for its null-terminator**
- Second, we **copy** one char at a time onto the end of the first string.

```
char big_string[1000];  /* I never know how much to allocate */
big_string[0] = '\0';
strcat(big_string,"John, ");
strcat(big_string,"Paul, ");
strcat(big_string,"George, ");
strcat(big_string,"Joel ");
```



SHLEMIEL THE PAINTER'S ALGORITHM

Shlemiel gets a job as a **street painter**, painting the dotted lines down the middle of the road. **On the first day** he takes a can of paint out to the road and finishes **300 yards** of the road. "That's pretty **good**!" says his **boss**, "you're a fast worker!" and pays him a kopeck*.

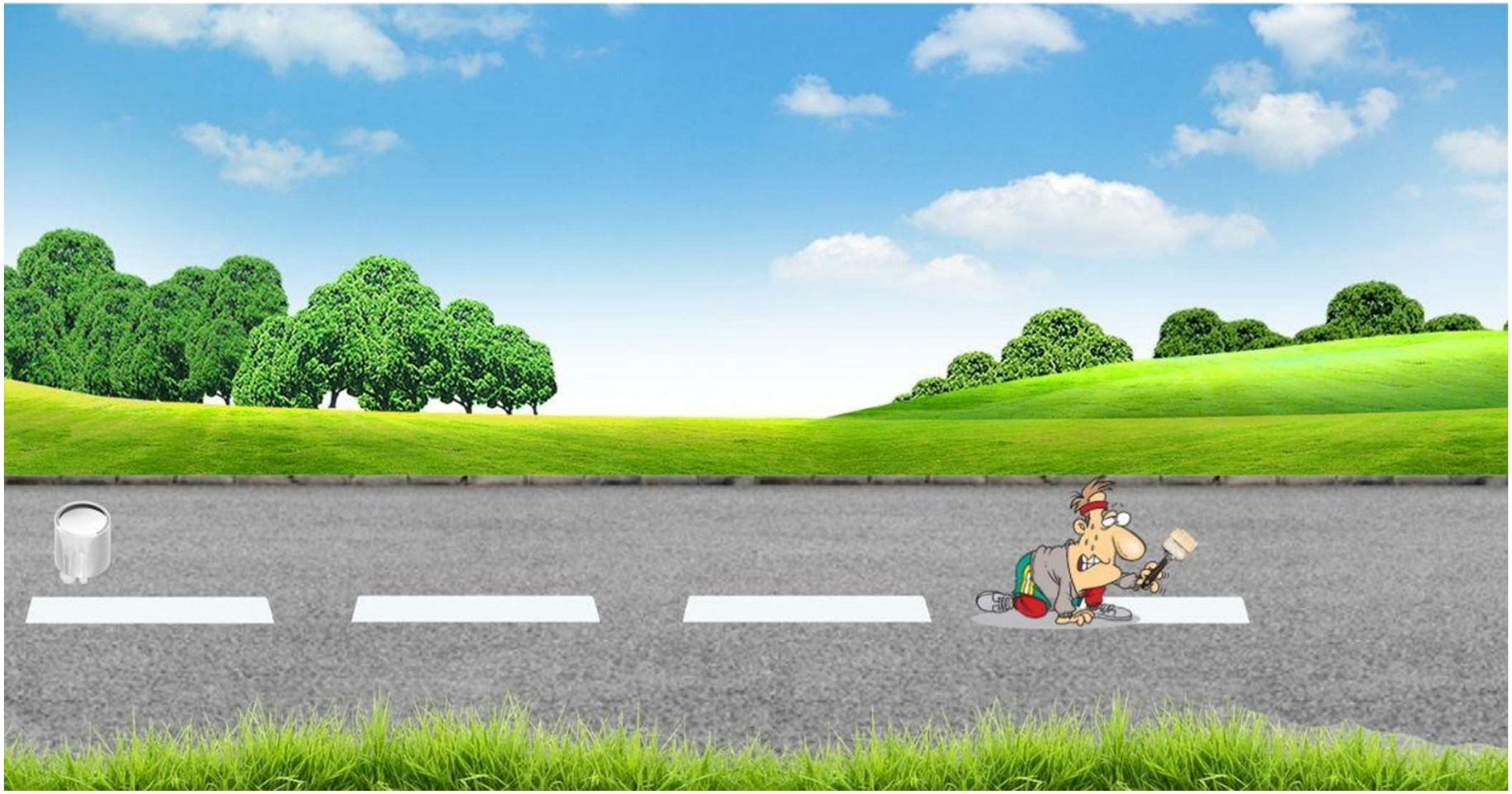
The next day Shlemiel **only gets 150 yards done**. "Well, that's not nearly as good as yesterday, but you're still a fast worker. 150 yards is **respectable**," and pays him a kopeck.

The next day Shlemiel paints **30 yards of the road**. "Only 30!" shouts his boss. "That's **unacceptable!** On the first day you did ten times that much work! **What's going on?"**

"I can't help it," says Shlemiel. "**Every day I get farther and farther away from the paint can!**

* **Kopek** - an Eastern-European currency, one-hundredth of a Ruble

SHLEMIEL THE PAINTER'S ALGORITHM



STRING FUNCTIONS – CAREFUL!

```
char* strcat( char* dest, char* src )
{
    while (*dest) dest++;
    while (*dest++ = *src++);
}
```

```
char big_string[1000];  /* I never know how much to allocate */
big_string[0] = '\0';
strcat(big_string,"John, ");
strcat(big_string,"Paul, ");
strcat(big_string,"George, ");
strcat(big_string,"Joel ");
```

- We are iterating over the whole string every time, and it keeps getting longer!

STRING FUNCTIONS – POSSIBLE SOLUTION

```
char* my_strcat( char* dest, char* src )
{
    while (*dest) dest++;
    while ((*dest++ = *src++));
    return --dest;
}
```

- **Suggestion** – return **pointer to the end** of the string
- **Implication** – in the next call, pass the pointer to the end of the string so that the first 'while' will be skipped.
- **Be Carful –** C's strings are full of hidden "Shlemiels".
- To read more about this interesting phenomenon:
- https://www.joelonsoftware.com/2001/12/11/back-to-basics/

WHAT ABOUT MEMORY?

```
int main()
{
    char *big_string = malloc(INIT_SIZE);
    char *cur_string_ptr = big_string;
    char buff[BUFF_SIZE], cur_str[INIT_SIZE];
    unsigned long cur_size = 0, capacity = INIT_SIZE;
    while(fgets(buff, INIT_SIZE, stdin) != NULL)
    {
        sscanf(buff, "%s", cur_str);
        if(cur_size + strlen(cur_str) >= capacity)
        {
            capacity = cur_size + strlen(curStr) + 1;
            bigString = realloc(bigString, capacity);
            cur_string_ptr = big_string + cur_size;
        }
        cur_string_ptr = my_strcat(cur_string_ptr, cur_str);
        cur_str = cur_string_ptr - big_string;
    }
. . .
. . .
```

- What if we get input from user / file with unknown length?

We reallocate memory in each entrance to the while loop – Inefficient!

WHAT ABOUT MEMORY?

```
int main()
{
    char *big_string = malloc(INIT_SIZE);
    char *cur_string_ptr = big_string;
    char buff[BUFF_SIZE], cur_str[INIT_SIZE];
    unsigned long cur_size = 0, capacity = INIT_SIZE;
    while(fgets(buff, INIT_SIZE, stdin) != NULL)
    {
        sscanf(buff, "%s", curStr);
        if(cur_size + strlen(curStr) >= capacity)
        {
            capacity *= 2;
            big_string = realloc(big_string, capacity);
            cur_string_ptr = big_string + cur_size;
        }
        cur_string_ptr = my_strcat(cur_string_ptr, cur_str);
        cur_size = cur_string_ptr - bigString;
    }
}
```

- Multiply the capacity by 2 or by 1.5 to do maximum *log(n)* reallocations

STRING FUNCTIONS – CAREFUL!

STRING FUNCTIONS

- GTA online is known by its long loading time.
- One blogger discovered why:
- GTA starts with parsing a 10MB, 63K items file with **sscanf()**
- Turns out that **sscanf()** calls **strlen()**
- 63K words – 63K calls to **strlen()** that pass on the whole file each call.
- Fix cutting loading time by half!.
- **Be Careful –** C's strings are full of hidden "Shlemiels".