CS100 Introduction to Programming

Recitation 3

llk89

NO PLAGIARISM!!!

- The most likely cause for failing this course.
- You WILL be caught!
- We WILL punish!
- They WILL know!
  - Parents
  - University
  - School
  - Fellows
- Again we caught several people
- Copy straight from Q1 to Q3? Really think we are just scaring you?

Overview

- structs & unions
- linked lists
- recursion
- ed--, Part 2

structs & unions

structs and functions

- struct as part of function prototype?
  - Well defined part of language
  - Discouraged except in a few situation
  - Involves coping of large amount of data
- Use struct pointers instead

struct casting

- Actually undefined behavior
- In practice it works
  - most of the time
  - somewhat commonly used

```
struct Foo {
        int foo;
};

struct Bar {
        int bar;
};

struct Bar *get_bar(struct Foo *foo) {
        return (struct Bar *) foo;
}
```

- Should we tell this to students at all?

Unions

- Length?
- Wrong type?
- Common usage
  - Same data, different perspective
  - Same location, different data

Same data, different perspective

- Allow access using index
- Allow access using human readable names

```
union vertex{
    struct{
        float x,y,z;
    };
    float data[3];
};
```

Same location, different data

- Allow some degree of flexibility

```
union data_type {
    int val;
    char *name;
};

struct list_node {
    struct list_node *next;
    union data_type data;
};
```

Recursion

Function stacks

```c
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

function call	type of value
main	return address

Function stacks

```
int sum(int i) {          <-------  i = 3
    int ret;              <-------  ret not known
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

i = 2

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame
	return address

Function stacks

```
int sum(int i) {          <-------  i = 2
    int ret;              <-------  ret not known
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame

Function stacks

```
int sum(int i) {          <-------  i = 1
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame
	return address

Function stacks

```
int sum(int i) {          <-------  i = 1
    int ret;              <-------  ret not known
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame
	return address
sum(1)	stack frame

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

i = 0

- Notice how stack continues to grow

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame
	return address
sum(1)	stack frame
	return address

Function stacks

```
int sum(int i) {          <-------  i = 0
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame
	return address
sum(1)	stack frame
	return address

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

i = 1

ret = 1 + 0

got 0

- Notice how stack start to shrink, or popping those coming in later

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame
	return address
sum(1)	stack frame

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

i = 2

ret = 2 + 1

got 1

- Notice how stack start to shrink, or popping those coming in later

function call	type of value
main	return address
sum(3)	stack frame
	return address
sum(2)	stack frame

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));
    return 0;
}
```

i = 3

ret = 3 + 3

got 3

- Notice how stack start to shrink, or popping those coming in later

function call	type of value
main	return address
sum(3)	stack frame

Function stacks

```
int sum(int i) {
    int ret;
    if (i > 0)
        ret = i + sum(i - 1);
    else
        return 0;
    printf("sum at %d now: %d", i, ret);
    return ret;
}

int main() {
    printf("sum: %d", sum(3));   got 6
    return 0;
}
```

- Notice how stack start to shrink, or popping those coming in later

function call	type of value
main	

WARNING

We will be using practices that benefit larger projects more but offers little to smaller projects. Many of things we mention may seem unnecessary for a code base of this scale. This is on purpose to show you these technique exists and how they should be carried out!

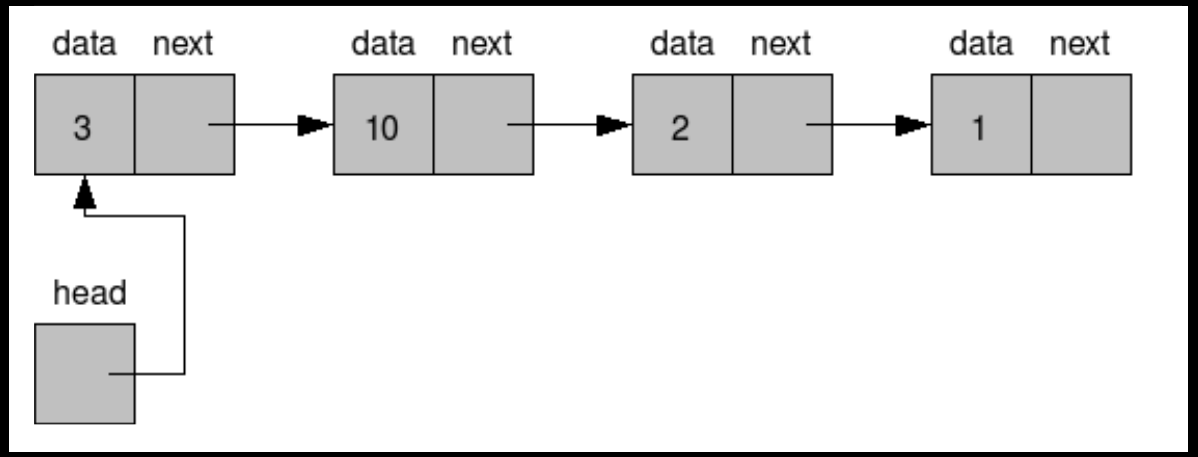
Linked list

What to do now?

- This will be …
  - A brief guide on how to create a linked list
  - A brief guide on how programs are designed in chronical order
- This will NOT be …
  - An explanation of linked list
  - A step by step walk through of professor’s code

Before you start

- Gain a big picture on what you are writing



Start small

- Begin from pieces that every other piece depends on
  - Initialize and clean up
- Document as you go
  - Essential parts
    - Meaning of each parameter
    - Meaning of return value
    - Side effects, e.g. print to console, file opened
    - ……
  - Optional
    - Anticipated problems
    - Example usage

Start small

- Test as you go
  - Initialize and clean up and seldom error, though

- Aftermath
  - Is there better implementation?
    - Better write down if any
  - Could there be some typical misuse?
    - Mention them in document!

Documentation & testing: side effect

- Documentation & testing flushes any uncertainty
- Writing documentation & tests …
  - Help you explore vast unknowns
    - e.g. document the parameters gives you a clear picture what you may handle
  - Help you identify bad design
    - e.g. identify behaviors that seems normal from an implementer's perspective but unusual from the caller’s perspective.

Continue enhancing

- Work out what can to be added next
  - Removal?
    - Crazy. Nothing in the list yet.
  - Addition!
- Insist add removal first?
  - Addition may assume a special layout in the list
  - May work forthis trivial task, not going to work in more difficult ones

Continue enhancing

- The same checklist as for initialization and clean up
- Documentation?
- Testing?
  - This starts to become necessary
  - Be exhaustive
  - Be creative
- Aftermath

A list of all functions implemented

- bool add_list_head_element(LIST*, const LIST_DATA&);
- bool remove_list_head_element(LIST*);

- bool add_list_tail_element(LIST*, const LIST_DATA&);
- bool remove_list_tail_element(LIST*);

- bool add_current_list_element_before(LIST*, const LIST_DATA&);
- bool add_current_list_element_after(LIST*, const LIST_DATA&);
- bool add_list_element_before(LIST*, LIST_NODE*, const LIST_DATA&);
- bool add_list_element_after(LIST*, LIST_NODE*, const LIST_DATA&);

- bool remove_current_list_element(LIST*);
- bool remove_list_element(LIST*, LIST_NODE*);

- bool move_list_pointer_to(LIST*, int);

- bool destroy_list(LIST*);

- int get_list_count(LIST*);

- LIST_NODE* get_list_head(LIST*);
- LIST_NODE* get_list_tail(LIST*);

- LIST_DATA& get_list_element(const LIST_NODE*);
- void set_list_element(LIST_NODE*, const LIST_DATA&);

- void print_list(LIST*);

ed--

Part 2

Recap

- Line based editor
- One buffer per line
- Implemented Replacement and searching

Spec has changed!

- A new functionality: sorting
- The same as your homework 3
- Sort the whole file in ascending or descending alphabet order
- The command for this functionality would be
  - sa: sort in ascending order
  - sd: sort in descending order

Spec has changed!

- Program must be able to handle files with at least 10k lines!

Today’s menu

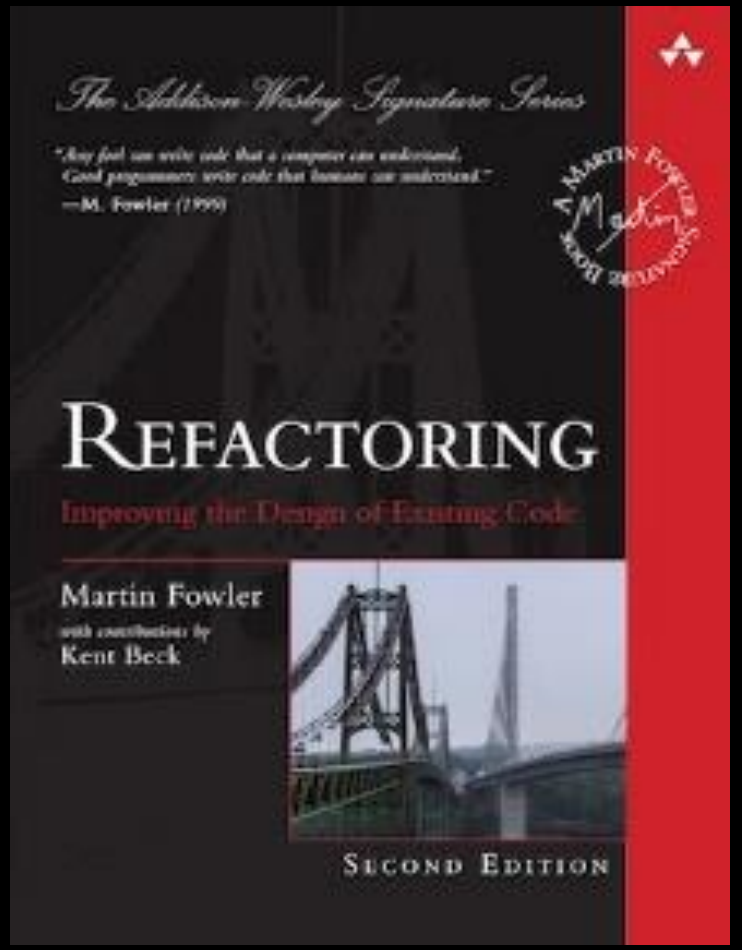
- Implement command `a`, `sa` and `sd`

Problem

- Command “a”
- Append a line means a new line buffer
- This addition could happen in the middle of file!
- We used to hold all pointers to a line buffer in an dynamic array
  - Shift every pointer after inserted line!
  - Super inefficient!
- Solution: replace dynamic array with a doubly linked list!
- How?
  - Refactoring

Refactoring?

- A dramatic change to existing code base is called refactoring
- Refactoring can be time consuming, difficult and error prone
- Refactoring can be and often is necessary
- (Mostly) cheaper than restart from scratch



What refactoring is?

- Replace an obsolete part with an up-to-date one
- Clean up hard coded behaviors
- Decouple functional units within program
- Fixing manageable systematic problems
  - e.g. every minor change requires changing a dozens of source files
- ……

What refactoring is not?

- Rename a local variable
- Fix a trivial bug
- Add comments
- Repair a piece of code that just don’t work
  - e.g. four years old project with no release
- ……

Refactoring considerations: before

- Is this necessary?
  - Rarely change existing working code when alternatives exists
  - *Refactoring can be time consuming, difficult and error prone*
- In our case,
  - dynamic array may work for smaller files
  - definitely not going to work on larger files
    - which we want to handle
- So it is necessary

Refactoring considerations: before

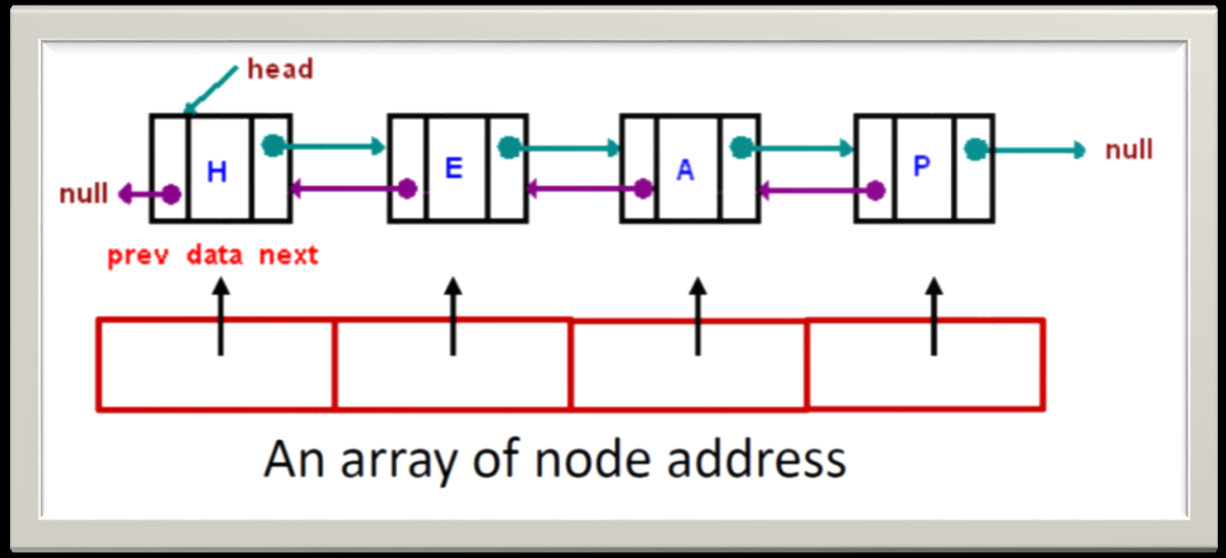
- What we will do?
  - Replace dynamic array with linked list
- What code we have now rely on replaced part?
  - line_replace and line_search
    - We don’t have anything else anyway
- What marks our refactoring is a success?
  - line buffers being managed in linked list
  - efficient line insertion
  - existing functions does not drop in performance too significantly

Refactoring considerations: before

- How will this refactoring impact future development?
  - Performance?
  - Sort function?
  - Memory usage?
  - Some super useful function we may add later?
  - Accessing lines?
  - ……

Design considerations

- We need random access
  - User may insert a line after line 3, then want to print the line 2181
  - Buzzword: Random Access (随机访问)
    - Read or write to element with arbitrary index inside collection
- Recall lectures
  - Compromise



Design considerations

- Won’t this bring the performance back down to dynamic array?
- Sometimes, yes
  - e.g. insert and print in alternation
- Sometimes, not
  - insert multiple lines at once
- Solution: Postpone write to node array
- Alternative: LRU cache
  - Better, but beyond our scope
  - Left as an exercise for readers

Next?

- Start coding?
- NO.
- Remember we did implemented a linked list before?
- With good encapsulation you can reuse much of that code

A list of all functions implemented

- `bool add_list_head_element(LIST*, const LIST_DATA&);`
- `bool remove_list_head_element(LIST*);`

- `bool add_list_tail_element(LIST*, const LIST_DATA&);`
- `bool remove_list_tail_element(LIST*);`

- `bool add_current_list_element_before(LIST*, const LIST_DATA&);`
- `bool add_current_list_element_after(LIST*, const LIST_DATA&);`
- `bool add_list_element_before(LIST*, LIST_NODE*, const LIST_DATA&);`
- `bool add_list_element_after(LIST*, LIST_NODE*, const LIST_DATA&);`

- `bool remove_current_list_element(LIST*);`
- `bool remove_list_element(LIST*, LIST_NODE*);`

- `bool move_list_pointer_to(LIST*, int);`

- `bool destroy_list(LIST*);`

- `int get_list_count(LIST*);`

- `LIST_NODE* get_list_head(LIST*);`
- `LIST_NODE* get_list_tail(LIST*);`

- `LIST_DATA& get_list_element(const LIST_NODE*);`
- `void set_list_element(LIST_NODE*, const LIST_DATA&);`

- `void print_list(LIST*);`

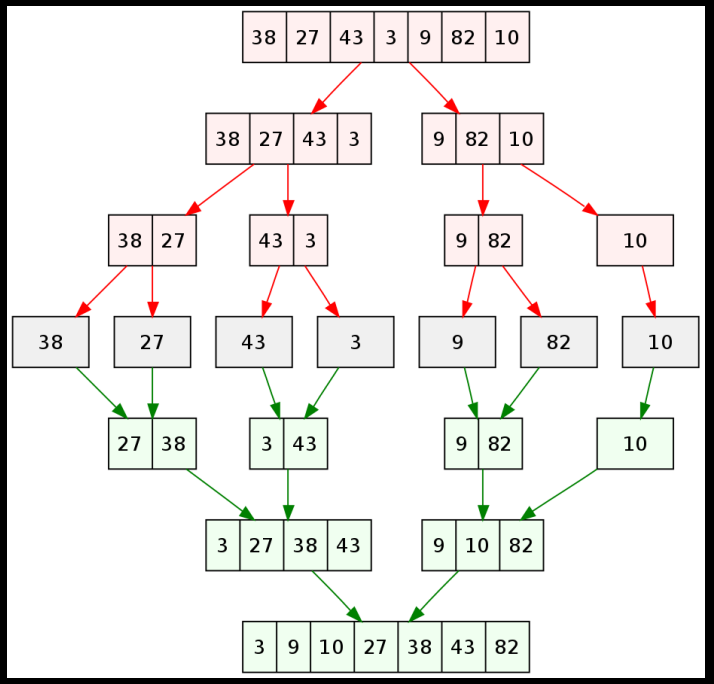
- This looks good
- Add node array and it is finished

Design considerations

- How to sort?
- Bubble sort?
  - Terrible performance
- Quick sort?
  - Viable
  - More work load than random access version
- Merge sort
  - Similar performance.
  - Does not require random access whatsoever

Merge sort

- A recursive sort algorithm
- Still divide and conquer
- Basic idea
  - Cut list in half until undividable
  - Merge pieces together while reordering pieces so that an order is established



Reference implementation

- We will not present a reference implementation
- We are moving away from the rudimentary part of teaching language basics
- We will strength our emphasis on design related topics even more