COMP1927 16x1 Computing 2

C Outside the Style Guide, Linked Lists and Function Pointers

Compiling

- For compiling for normal use

  gcc –Wall –Werror –O –o prog prog.c

- For compiling to run gdb or ddd or valgrind

  gcc –Wall –Werror –gdwarf-2 –o prog prog.c

- Compiling with more than 1 c file, normal use

  gcc –Wall –Werror –O –o prog prog.c f2.c f3.c

COMP1917 Style

- Required use of a restricted subset of C:
  - layout, use of brackets (always)
  - use only **if** and **while**
  - no side-effects in expressions
  - no conditional expressions
  - all functions have one return statement
  - But ... this style used in no texts or real code

COMP1927 Style

- layout: consistent indentation still required
- use of brackets:
  - can omit if control structure owns a single statement
  - put function start bracket on line after function header
- can use all C control structures
  - **if**, **switch**, **while**, **for**, **break**, **continue**

COMP1927 Style (cont)

- can use assignment statements in expressions
  - but you should continue to avoid other kinds of side-effects
- can use conditional expressions
  - but use **x = c ? e1 : e2** with care
- functions may have several return statements, loops can have break, continue
  - can make functions more concise, but possibly more cryptic and make proofs harder, so we still avoid

COMP1927 Style (cont)

- Good: gives you more freedom and power
  - more choice in how you express programs
  - can write code which is more concise (simpler)

- Bad: gives you more freedom and power
  - can write code which is more cryptic
  - can lead to incomprehensible, unmaintainable code

For Loop example

- while version

```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i;
    i++; //i = i+1;
}
```

- for version

```
sum = 0;
for (i = 0; i < 10; i++)
    sum += i; //sum = sum + i;
```

Switch Statements

- If statements such as:

```
if (colour == ‘r’){
    print(“Red”);
}else if(colour == ‘b’){
    printf(“Blue”);
}else if(colour == ‘g’){
    printf(“Green”);
}else{
    printf(“Not valid”);
}
```

Switch Statements (cont)

Can be written as switch statements:

```
switch(colour){
    case ‘r’: printf(“Red”);
              break;
    case ‘b’: printf(“Blue”);
              break;
    case ‘g’: printf(“Green”);
              break;
    default : printf(“Not valid”);
}
```

Note break is critical; if not present, falls through to next case.

Exercise

- Write a function monthName(int) that
  - Accepts a month number 1 = Jan..12=Dec
  - Returns a string containing the month name
  - Assume the string will be read-only
  - Use a switch to decide on the month
- Suggest an alternative approach with an array

Jumping Around

- The **return** statement
  - gives back result to caller of function
  - terminates function (possibly "early")
- The **break** statement
  - allows early termination of a loop
- The **continue** statement
  - allows early termination of one loop iteration
- They all help to avoid deeply nested if statements.

Conditional Expressions

- If statements that compute a value

```
if (y > 0) {
   x = z+1;
} else {
   x = z -1;
}
```

- can be written as a conditional expression:

```
x = (y > 0) ? z+1 : z-1;
```

Exercise Conditionals

- Rewrite each of the following using or a conditional expression or state why it can't be written that way

```
//a
if(x > 0)
 y = x - 1;
else
 y = x + 1;
```

```
//b
if(x > 0)
 y = x - 1;
else
 z = x + 1;
```

```
//c
if(x > 0){
 y = x - 1;
 z = x + 1;
}else{
 y = x + 1;
 Z = x - 1;
}
```

Assignments in Expressions

- C assignment statements are really expressions
- they return a result: the value being assigned
- the return value is generally ignored
- Frequently, assignment is used in loop continuation tests
  - to combine the test with collecting the next value
  - to make the expression of such loops more concise

Assignments in Expressions (cont)

```
nchars = 0;
ch = getchar();
while (ch != EOF) {
   nchars++;
   ch = getchar();
}
```

can be written as

```
nchars = 0;
while ((ch = getchar()) != EOF)
    nchars++;
```

What does this code do?

```
void whatDoesItDo(){
        char ch;
        while ((ch = getchar()) != EOF){
           if(ch == '\n') break;
           if(ch == 'q') return;
           if(!isalpha(ch)) continue;
           printf("%c",ch);
        }
        printf("Thanks!\n");
    }
```

What is a linked list?

- sequential collection of items (i.e. no random access)
  - we can only get to the second by accessing the first, and so on
  - we can't access the *n*th element of a list directly
- easy to re-arrange
  - deleting or inserting is simple
- self-referent structure (may be cyclic)
  - a list element contains a link to another list element

What is a linked list?

- A linked list is a set of items where each item is part of a node that also contains a link to a node. (We also call the list items list elements)

- The final element:

1. contains a null link, pointing to no node, or
2. refers to a dummy node (also called sentinel) containing no item
3. may be the first node (hence the list is circular)

What is a linked list?

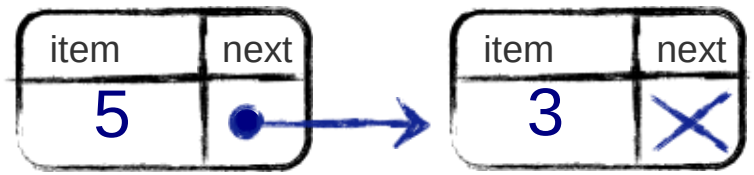
A possible C implementation

insert definition for `Item` here

```
typedef    int    Item;

typedef struct node * link;
 struct node {
       Item item;
       link next;
 };
```

Sedgewick



Common Operations

Memory allocation

```
link x = malloc (sizeof *x);     // RIGHT
link y = malloc(sizeof(struct node)); //RIGHT

link z  = malloc(sizeof(link));  //WRONG
```

Traversing a list

```
for (curr = start; curr != NULL; curr = curr->next) {
   // do something with the node

}
```

Exercise

- Write a function to insert node at the beginning of the list

```
link insertFront(link list, link newNode);
```

- Could we use this prototype instead ?

```
void insertFront(link list, link newNode);
```

- Write a function to insert node at the end of the list

```
link insertEnd(link list, link newNode);
```

Exercise

Implement a function which given a linked list, reverses the order of items

```
link reverse (link list) {




}
```

Deletion on lists

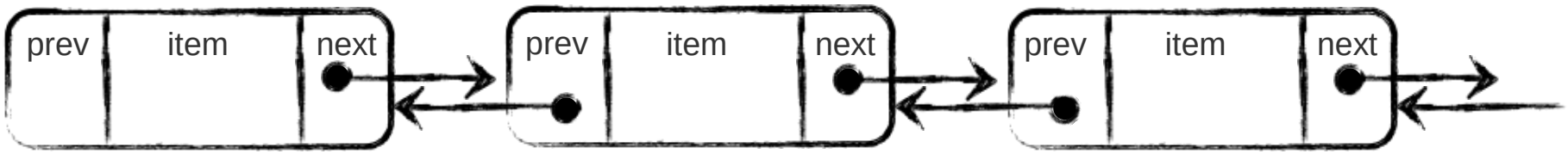
- Delete an item from a linked list (see lecture code for implementation)

```
//Remove a given node from the list
//and return the start of the list
link  deleteItem (link ls, link n);
```

Problem: deletion

- Deletion is awkward, as we always have to keep track of the previous node
- Can we delete a node if we only have the pointer to the node itself?
- We may need to traverse the whole list (if we have a reference to the head) to find the predecessor of `curr`!
- Idea: every node stores a link to the previous node, in addition to the link to the next node

Doubly Linked Lists



- Move forward and backward in such a list
- Delete node in a constant number of steps

```
typedef struct dnode * dlink;
typedef struct dnode {
  Item item;
  dlink next;
  dlink prev;
} ;
```

Doubly linked lists

Deleting nodes

- easier, more efficient

Other basic list operations

- pointer to previous node is necessary in many operations, doesn’t have to be maintained separately for doubly linked lists
- twice the number of pointer manipulations necessary for most list operations
- memory overhead to store additional pointer

Function Pointers

- C can pass functions by passing a pointer to them.
- Function pointers ...
  - are references to memory addresses of functions
  - are pointer values and can be assigned/passed

Function Pointers

- E.g. a pointer to a function mapping

int → int

```
int (*fun)(int)
```

- Function pointer variables/parameters are declared as:

typeOfReturnValue (*fname)(*typeOfArguments*)

Example

```
int square(int x){ return x*x;}

int  timesTwo(int x){return x*2;}

int (*fp)(int);

fp = &square;       //fp points to the square function

int n = (*fp)(10); //call the square function with input 10

fp = timesTwo;      //works without the &

                    //fp points to the timesTwo function

n = (*fp)(2);       //call the timesTwo function with input 2

n = fp(2);          //can also use normal function call

                    //notation
```

Higher-order Functions

- Functions that get other functions as arguments, or return functions as a result
- Example: the function traverse takes a list and a function pointer as argument and applies the function to all nodes in the list

```
void printList(link ls){
    link curr = ls;
    while(curr != NULL){
        printf("%d ",curr->data);  //Process the node
         curr = curr->next;
    }
}
// apply function f to all nodes in ls
void traverse (link ls, void (*f) (link)){
      link curr = ls;
      while(curr != NULL){
              (*f) (curr);
              curr = curr->next;
       }
 }
```

Using Function pointers

```
void printNode(link ls){
    if(ls != NULL){
        printf("%d->",ls->data);
    }
}
```

```
void printGrade(link ls){
    if(ls != NULL){
        if(ls->data>= 85){
            printf("HD ");
        }
        else {
            printf("FL ");
        }
    }
}
```

```
void traverse (link ls, void (*f) (link));

//To call the function
//Function must have matching prototype
traverse(myList,printNode);
traverse(myList,printGrade);
```

Valgrind Demo

- Valgrind is useful for
  - Finding memory leaks
    - Not freeing memory that you malloced
  - Finding memory errors
    - Memory errors
      - Illegally trying access memory

Exercise

- Consider this alternate linked list definition:

```
typedef   int   Item;

typedef struct node * link;
 struct node {
      Item item;
      link next;
 };

typedef struct listImp * List;

struct listImp{
    link first;
    link last;
}
```

Exercise (cont)

- Draw an empty list
- Suppose we insert items 1, 2 and 3 at the end of an empty list

1. Draw the list
2. How many struct nodes would we have?
3. How many struct listImps would we have?
4. Write code to create a new Empty List and to insert an element at the end of the list