ECE 264 Advanced C Programming

Contents

1 C Programs using Multiple Files

If a program uses multiple files, these files are usually structured in the following way:

- Pairs of `.h` (header) and `.c` (source) files. Header files **declare** types and functions. Source files **define** (i.e. implement) the functions.
- Source files use `#include` to include header files.
- A source file includes a header file if the source file needs information (type or function declaration) from the header file.
- A file called `main.c` has the main function.
- `Makefile` is used to compile the source files and link the object files.

2 Linked List

So far we have handled data whose sizes are known (1) when the program is written or (2) when the program starts executing. For the latter, we may use `malloc` for arrays whose sizes are not known when the programs are written. However, for these cases, their sizes are known *before* we use the array elements.

```
int a;          /* scalar */
double b;       /* scalar */
int c[100];     /* array of 100 integers */
Person p;       /* one object of programmer-defined type */
Person p2[4];   /* array of four Person objects */
int * ptr;      /* pointer to integer */
ptr = malloc(n * sizeof (int));
```

What happens if we cannot know the size **before** we use the elements? When do we need such capability? A text editor is an example. The editor does not know how many letters the user may enter. If the editor allocates a fixed size, the editor cannot handle a document that is longer than the size. You cannot write a text editor that allows only so many letters; otherwise, users will become angry when they cannot add more letters. You could make the size very large so no user would ever exceed the size. However, this is inefficient. A better solution is to **allocate more memory as needed**. If the user add more pages, the editor allocates more memory. If the user deletes some pages, the editor releases the unused memory. This lecture explains *linked list* to handle data whose sizes are not known even when we start using and processing the initial parts of the data. As more data are added, more memory is allocated to store the data. For simplicity, we consider only integers as the data right now.

A linked list is composed of a group of *list nodes*. Each node is a structure and contains **two attributes**: one for the data, and one to connect (i.e. link to) the next node using a "self-referential" attribute.

```
#ifndef LISTNODE_H
#define LISTNODE_H
typedef struct listnode
{
  struct listnode * ln_next;
  int ln_value;
} Node;

Node * Node_construct(int v);
Node * List_copy(Node * n);
void List_assign(Node * * n1, Node * n2);
void Node_destruct(Node * n);
void List_destruct(Node * n);
Node * List_insert(Node * n, int v);
int Node_getValue(Node * n);
void Node_print(Node * n);
void List_print(Node * n);
#endif
```

When we declare an attribute in a structure, a C compiler has to know the size of the attribute in order to allocate sufficient space. However, `ln_next` is part of `listnode` and we have not finished `listnode` yet. How can the compiler know how much space for the attribute? The trick is declaring it as a **pointer**. When an attribute is a pointer, its size is independent of the type pointed to. The size of a pointer is the same for `char`, `int`, `double`, or `Person`. The size of the pointer is the size of the **addressing space** of the computer, usually 32 or 64 bits. This "next" pointer is the *link* to connect `listnode`. The

structure has another attribute to store an integer value. the rest of the header file declares the functions used to handle the linked list. Some functions are related to individual nodes and the others are related to the whole list. Before we explain how to link the nodes by code, let's see a visual representation of a linked list, on page 448 in ABoC.

When a node is created, its "next" points to zero, or NULL. This means the node is currently not linking to anything. In C, zero is an invalid value for an address. If a node links to another node, the former `ln_next` stores the address of the latter. This linked list is one directional, meaning that a node links to the next node but the next has no link to the previous node. In a *doubly linked list*, each node has two links, one to the next and one to the previous.

How do we implement the functions to handle a node or a list? The first function creates a node that stores an integer and the node does not link to anything.

```
#include "listnode.h"
#include <stdio.h>
#include <stdlib.h>
Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> ln_value = v;
  n -> ln_next = 0;
  return n;
}

Node * List_insert(Node * n, int v)
{
  /* insert at the beginning */
  Node * p = Node_construct(v);
  /* comment */
  p -> ln_next = n;
  return p;
}

Node * List_copy(Node * n2)
{
  Node * next = n2;
  Node * n;
  if (n2 == 0) { return 0; }
  n = Node_construct(n2 -> ln_value);
  next = next -> ln_next;
  while (next != 0)
    {
```

```
      n = List_insert(n, next -> ln_value);
      next = next -> ln_next;
    }
  return n;
}

void List_assign(Node * * n1, Node * n2)
{
  if ((* n1) == n2)
    { return; }
  List_destruct(* n1);
  * n1 = List_copy(n2);
}

void Node_destruct(Node * n)
{
  free (n);
}

void List_destruct(Node * n)
{
  Node * prev = n;
  Node * next;
  while (prev != 0)
    {
      next = prev -> ln_next;
      Node_destruct(prev);
      prev = next;
    }
}

int Node_getValue(Node * n)
{
  return (n -> ln_value);
}

void Node_print(Node * n)
{
  printf("%d ", n -> ln_value);
}

void List_print(Node * n)
```

```
{
  Node * curr = n;
  while (curr != 0)
    {
      Node_print(curr);
      curr = curr -> ln_next;
    }
  printf("\n\n");
}
```

The next function inserts a new value to the list that starts at `Node n`. This function creates a new node whose value is `v`. After this new node is created, the node's `ln_next` points to zero, i.e. no other node. The next statement makes the input node the next of the newly created node. The function then returns the newly created node. Before explaining any of the other functions, let's see how to use the insert function.

```
#include "listnode.h"
#include <stdio.h>
void testFunc1()
{
  Node * list = 0; /* always initialize */
  int inInt = 0;
  do
    {
      printf("Enter a number, 0 to stop ");
      scanf("%d", & inInt);
      if (inInt != 0)
        {
          list = List_insert(list, inInt);
        }
    } while (inInt != 0);
  List_print(list);
  List_destruct(list);
}

/*
  output
  Enter a number, 0 to stop 6
  Enter a number, 0 to stop 7
  Enter a number, 0 to stop 8
  Enter a number, 0 to stop 9
  Enter a number, 0 to stop 0
  9 8 7 6
```

```
*/
void testFunc2()
{
  Node * list1 = 0;
  Node * list2 = 0;
  Node * list3 = 0;
  int cnt;
  for (cnt = 0; cnt < 10; cnt ++)
    {
      list1 = List_insert(list1, cnt + 10);
      list2 = List_insert(list2, cnt + 100);
    }
  List_print(list1);
  List_print(list2);
  list3 = List_copy(list1);
  List_assign(& list2, list2); /* destination = source */
  List_assign(& list1, list2);
  list1 = List_insert(list1, -91);
  list3 = List_insert(list3, 1003);
  List_print(list1);
  List_print(list2);
  List_print(list3);
  List_destruct(list1);
  List_destruct(list2);
  List_destruct(list3);
}
/*
  output
  19 18 17 16 15 14 13 12 11 10
  109 108 107 106 105 104 103 102 101 100
  -91 100 101 102 103 104 105 106 107 108 109
  109 108 107 106 105 104 103 102 101 100
  1003 10 11 12 13 14 15 16 17 18 19
*/

int main(int argc, char * argv[])
{
  testFunc1();
  testFunc2();
  return 0;
}

# Makefile
```

```
listmain: listnode.h listnode.c listmain.c
        gcc -g -Wall -c listnode.c
        gcc -g -Wall -c listmain.c
        gcc -g -Wall listnode.o listmain.o -o listmain
        ./listmain
        valgrind --leak-check=yes ./listmain
clean:
        rm -f *.o listmain
```

In this example `testFunc1`, we first create a node but it points to nothing. It is a good habit to initialize all pointers to zero. **C does not guarantee that a pointer points to zero.** Instead, the pointer may store a random number as the address. You **cannot** check whether the address is zero for determining the pointer contains a valid address. If you do not immediately assign zero to a pointer, you will probably forget later.

The program asks a user to enter a number. If the number is not zero, it is inserted into the list and the program asks the user to enter another number. We call `List_insert` by giving the current list as the first argument and also store the returned pointer at the same list. After the user enters zero, this `while` block ends, the list is printed and then destroyed. You may notice that the output is the reverse order as the input. The value 6 is entered first but it is printed last. This occurs because the way we write the insert function. The function takes the new value and puts it at the **beginning** of the list. The original list is connected through the next attribute. This is called "First-In-Last-Out" (FILO) or "Last-In-First-Out" (LIFO). It is different from what we often see "First-In-First-Out" (FIFO). LIFO/FILO is also called *stack* and it is a very important part of all computer programs. We will discuss that in a later lecture. Since we care about storing data and do not care about the order right now, this reverse order is all right. Later we will consider how to handle the orders, for example, FIFO, or sort the numbers as they are added. `List_print` goes through the list node-by-node. If we have not reached the end of the list (pointing to zero), print the value and go to the next node.

The next test considers the copy and the assign functions. We create two lists first, with values of 0 to 9 and 100 to 109. The function `List_copy` is used to create `list3`. After calling this function, `list1` and `list3` are two **separate** and **independent** lists. We can modify one without affecting the other. Remember to call "copy" for creating the list for the first time. If a list already exists, we need to call the "assign" function. The names are borrowed from C++, corresponding to "copy constructor" and "assignment operator" respectively. We can insert -91 to `list1` and 1003 to `list3`. How do the two functions work? `List_assign` checks whether the two lists are the same. If they are the same, the function returns without doing anything. This prevents destroying a list when the destination and the source are the same. If they are different, destroy the destination to release the memory currently held by the list. The next step copies the list. How does copy work? It is somewhat similar to the print function. To copy the nodes, we go through the nodes one by one.