CS11001/CS11002
Programming and Data Structures (PDS)

(Theory: 3-1-0)

Dynamic Memory Allocation

- All variables, arrays, structures and unions that we worked with so far are *statically* allocated, meaning that whenever an appropriate scope is entered (e.g. a function is invoked) an amount of memory dependent on the data types and sizes is allocated from the stack area of the memory.
- When the program goes out of the scope (e.g. when a function returns), this memory is returned back to the stack.
- There is ~~an alternative way of allocating memory, more precisely~~, from the heap part of the memory.
- In this case, the user makes specific calls to capture some amount of memory and continues to hold that memory unless it is explicitly (i.e., by distinguished calls) returned back to the heap.
- Such memory is said to be *dynamically* allocated.

Linked Lists with Arrays

- A static array can implement such lists, but has two disadvantages:
  - The size of a static array is fixed during declaration, i.e., a static array can handle lists of a bounded size.
- The linked structure can be incorporated in the framework of an array, but that requires (often awful) calculations to find the locations of the next objects.
- If pointers with dynamically assigned memory are used, accessing objects following the links becomes much easier.

Using malloc for dynamic memory

```
■ #define SIZE1 25
  #define SIZE2 36
  int *p;
  long double *q;
   p = (int *)malloc(SIZE1 * sizeof(int));
   q = (long double *)malloc(SIZE2 * sizeof(long double));
```

The first call of malloc allocates to p a (dynamic) array capable of storing SIZE1 integers.

The second call allocates an array of SIZE2 long double data to the pointer q. In addition to the size of each array, we need to specify the sizeof (size in bytes of) the underlying data type.

- malloc allocates memory in bytes and reads the amount of bytes needed from its sole argument.
- If you demand more memory than is currently available in your system, malloc returns the NULL pointer.
  - So checking the allocated pointer for NULLity is the way how one can check if the allocation request has been successfully processed by the memory management system.
- malloc allocates raw memory from some place in the heap. No attempts are made to initialize that memory.
  - It is the programmer's duty to initialize and then use the values stored at the locations of a dynamic array.

Example

```
foollection initfc ( int type )
 { foollection fc; /* Set type of the collection */
   fc.type = type; /* Allocate memory for the data
   pointer */
  if (type == 1)
      fc.data = (int *)malloc(10*sizeof(int));
 else if (type == 2)
   fc.data = (int *)malloc(10000000*sizeof(int));
 else fc.data = NULL; /* Check for error conditions */
   if (fc.data == NULL)
      fprintf(stderr, "Error: insufficient memory or
   unknown type.\n"); return fc; }
```

Another Example

```
typedef struct _node {
   int data;
   struct _node *next;
} node;
node *head, *p;
int i;
head = (node *)malloc(sizeof(node)); /* Create the first node */
head->data = 3; /* Set data for the first node */
p = head; /* Next p will navigate down the list */
for (i=1; i<=3; ++i)
{ p->next = (node *)malloc(sizeof(node)); /* Allocate the next node */
    p = p->next; /* Advance p by one node */
p->data = 2*i+3; /* Set data */
}
p->next = NULL; /* Terminate the list by NULL */
```

Finer Points

- An important thing to notice here is that we are always allocating memory to p->next and not to p itself. For example, first consider the allocation of head and subsequently an allocation of p assigned to head->next.
- head = (node *)malloc(sizeof(node));
  p = head->next;
  p = (node *)malloc(sizeof(node));

Finer Points

- After the first assignment of p, both this pointer and the next pointer of *head point to the same location.
- However, they continue to remain *different pointers*. Therefore, the subsequent memory allocation of p changes p, whereas head->next remains unaffected.
- For maintaining the list structure we, on the other hand, want head->next to be allocated memory.
- So allocating the running pointer p is an error. One should allocate p->next with p assigned to head (not to head->next).
- Now p and head point to the same node and, therefore, both p->next and head->next refer to the same pointer -- the one to which we like to allocate memory in the subsequent step.
- This example illustrates that the first node is to be treated separately from subsequent nodes.
  - This is the reason why we often maintain a *dummy node* at the head and start the actual data list from the next node.

Realloc

- The realloc call reallocates memory to a pointer. It is essentially used to change the amount of memory allocated to some pointer.
- If the new size s' of the memory is larger than the older size s, then s bytes are copied from the old memory to the new memory. The remaining s'-s bytes are left uninitialized.
- On the contrary, if s'<s, then only s' bytes are copied. If the reallocation request fails, the original pointer remains unchanged and the NULL pointer is returned.
- As an example, suppose that we want to change the size of the dynamic array pointed to by foochain from one million to two millions, but without altering the data currently stored in the array. We can use the following call:
-

```
#define NEW_SIZE 2000000
 foochain = realloc(foochain, NEW_SIZE * sizeof(foollection));
 if (foochain == NULL)
    fprintf(stderr, "Error: unable to reallocate storage.\n");
```

- Memory allocated by malloc, calloc or realloc can be returned to the heap by the free system call. It takes an allocated pointer as argument. For example, the foochain pointer can be deallocated memory by the call:
  - **free(foochain);**
- When a program terminates, all allocated memory (static and dynamic) is returned to the system.
- There is ~~no necessity to free memory explicitly~~.
- However, since memory is a bounded resource, allocating it several times, say, inside a loop, may eventually let the system run out of memory.
- So it is a good programming practice to free memory that will no longer be used in the program.

Freeing of pointers

- The freeing mechanism is different for the four arrays.
  - int i; /* A is a static array and cannot be free'd */
  - /* B is a single pointer */ free(B);
  - /* C is a static **array of pointers** each to be free'd individually */
    for (i=0; i<ROWSIZE; ++i) free(C[i]);
  - /* Free each row */ /* D is a pointer to pointers. Each of these pointers is to be free'd */

  for (i=0; i<ROWSIZE; ++i) free(D[i]);

  /* Free each row */ free(D); /* Free the row top */

Abstract Data Type (ADT)

What is ADT?

- An abstract data type (**ADT**) is an object with a generic description independent of implementation details.
- This description includes a specification of the components from which the object is made and also the behavioral details of the object.

How to implement an abstract data type?

- Specifying only the components of an object does not suffice.
- Depending on the problem you are going to solve, you should also identify the properties and behaviors of the object and perhaps additionally the pattern of interaction of the object with other objects of same and/or different types.
- Thus in order to define an ADT we need to specify:
  - The components of an object of the ADT.
  - A set of procedures that provide the behavioral description of objects belonging to the ADT.

Example: Integer ADT

-
  ```
  typedef struct {
   unsigned long words[313];
   unsigned int wordSize;
  unsigned char sign; }
  bigint;
  ```
  - when such a data type is passed then the entire array needs to be copied. Inefficient!

-
  ```
  #define SIZEIDX 313
  #define SIGNIDX 314
  typedef unsigned long goodbigint[315];
  ```

Using the integer data type

```
#include<stdio.h>
#include<time.h>

main()
{
 typedef unsigned long int goodint[315];
 int i;
 goodint a;
 srand((unsigned int)time(NULL));

 for(i=0;i<315;i++)
   a[i]=1+rand()%99;
 for(i=0;i<315;i++)
   printf("%d ",a[i]);
 printf("\n");
}
```

Integers bigger than big!

- These big integers are big enough, but cannot represent integers bigger than big, for example, integers of bit-size millions to billions.
- Whenever we use static arrays, we have to put an upper limit on the size.
- If we have to deal with integers of arbitrary sizes (as long as memory permits), we have no option other than using dynamic memory and allocate the exact amount of memory needed to store a very big integer.
- But then since the maximum index of the dynamic array is not fixed, we have to store the size and sign information at the beginning of the array.
- Thus the magnitude of the very big integer is stored starting from the second array index. This leads to somewhat clumsy translation between word indices and array indices.
- **#define SIZEIDX 0**
  #define SIGNIDX 1
  typedef unsigned long *verybigint;

A better representation

- A better strategy is to use a structure with a dynamic words pointer.
- **typedef struct**

```
{ unsigned long *words;
  unsigned int size;
  unsigned char sign;
} goodverybigint;
```

- So you have to pay lot of attention, when implementation issues come.
- Good solutions come from experience and innovativeness.

The Complex ADT

```
#include<stdio.h>

  typedef struct{
    double real;
    double imag;
  }complex;

  void cprn(complex z)
  {
    printf("(%lf) + i(%lf)\n", z.real,
   z.imag);
  }
```

```
complex cadd(complex z1, complex z2)
{
  complex z;
  z.real=z1.real+z2.real;
  z.imag=z1.imag+z2.imag;
  return(z);
}
 complex read()
 {
   complex z;
   scanf("%lf",&z.real);
   scanf("%lf",&z.imag);
   return(z);
 }
```

The main function

```
main()
{
  complex a, b;
  a=read();
  b=read();
  cprn(a);
  cprn(b);
  a=cadd(a,b);
  cprn(a);

}
```

The List ADT

- Let us now define a new ADT which has not been encountered earlier in your math courses.
- We call this ADT the ordered list.
  - It is a list of elements, say characters, in which elements are ordered, i.e., there is a zeroth element, a first element, a second element, and so on, and in which repetitions of elements are allowed.

Functions on the List ADT

- L = init();
  - Initialize L to an empty list.
- L = insert(L,ch,pos);
  - Insert the character ch at position pos in the list L and return the modified list. Report error if pos is not a valid position in L.
- delete(L,pos);
  - Delete the character at position pos in the list L. Report error if pos is not a valid position in L.
- isPresent(L,ch);
  - Check if the character ch is present in the list L. If no match is found, return -1, else return the index of the leftmost match.
- getElement(L,pos);
  - Return the character at position pos in the list L. Report error if pos is not a valid position in L.
- print(L);
  - Print the list elements from start to end.

Some functions on the List ADT

```
#include<stdio.h>
#define MAXLEN 100

  typedef struct {
    int len;
    char element[MAXLEN];
  } olist;

olist init()
  {
    olist L;
    L.len = 0;
    return L;
  }

void print(olist L)
  {
    int i;

    for(i = 0; i < L.len; ++i) printf("%c", L.element[i]);
  }
```

```
olist insert(olist L , char ch , int pos)
  {
    int i;

    if ((pos < 0) || (pos > L.len)) {
      fprintf(stderr, "insert: Invalid index %d\n",
     pos);
      return L;
    }
    if (L.len == MAXLEN) {
      fprintf(stderr, "insert: List already full\n");
      return L;
    }
    for (i = L.len; i > pos; --i) L.element[i] =
     L.element[i-1];
    L.element[pos] = ch;
    ++L.len;
    return L;
  }
```

The main function

```
main()
{
 olist L;
 L=init();
 L=insert(L,'a',0);
 print(L);
 printf("\n");
 L=insert(L,'b',0);
 print(L);
 printf("\n");
 L=insert(L,'c',2);
 print(L);
 printf("\n");
}
```

- a
- ba
- bac