Data Structures

Lecture Eight

Second Stage

First Course - 2024-2025

ALI MUWAFAQ SHABAN

Master of Computer Engineering

Data Structures

Pointers

As we have seen in the previous section, a pointer is needed to refer to a struct that has been allocated on the heap. In can also be used more generally to refer to an element of arbitrary type that has been allocated on the heap. For example:

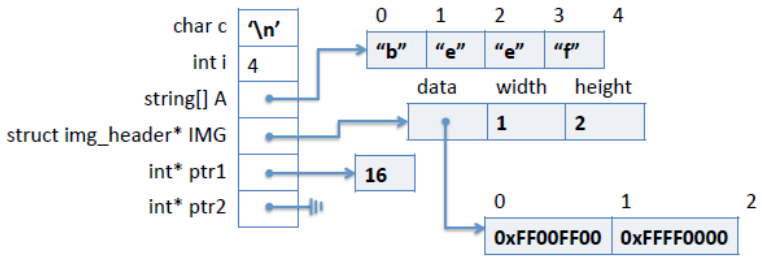
```
--> int* ptr1 = alloc(int);
ptr1 is 0xFFAFC120 (int*)
--> *ptr1 = 16;
```

(ptr1) is 16 (int)
*--> *ptr1;*
16 (int)

In this case we refer to the value using the notation *p, either to read (when we use it inside an expression) or to write (if we use it on the left-hand side of an assignment). So we would be tempted to say that a pointer value is simply an address. But this story, which was correct for arrays, is not quite correct for pointers. There is also a special value NULL. Its main feature is that NULL is not a valid address, so we cannot dereference it to obtain stored data. For example:

--> int ptr2 = NULL;*
p is NULL (int)*
*--> *ptr2;*
Error: null pointer was accessed
Last position: <stdio>:1.1-1.3

Graphically, NULL is sometimes represented with the ground symbol, so we can represent our updated setting like this:



To rephrase, we say that a pointer value is an address, of which there are two kinds. A valid address is one that has been allocated explicitly with alloc, while NULL is an invalid address. In C, there are opportunities to create many other invalid addresses, as we will discuss in another lecture. Attempting to dereference the null pointer is a safety violation in the same class as trying to access an array with an out-of-bounds index. In C0, you will reliably get an error message, but in C the result is undefined and will not necessarily lead to an error. Therefore:

*Whenever you dereference a pointer p, either as *p or p->f, you must have a reason to know that p cannot be NULL.*

In many cases this may require function preconditions or loop invariants, just as for array accesses.

Structs

C++ Struct Initialization

To create a C++ structure, we use the struct keyword, followed by an identifier. The identifier becomes the name of the struct. Here is the syntax for creation of a C++ struct:

```
struct struct_name
{
    // struct members
}
```

In the above syntax, we have used the struct keyword. The struct_name is the name of the structure. The struct members are added within curly braces. These members probably belong to different data types. For example:

```
struct Person
{
   char name[30];
    int citizenship;
    int age;
}
```

In the above example, Person is a structure with three members. The members include name, citizenship, and age. One member is of char data type, while the remaining 2 are integers when a structure is created, memory is not allocated. Memory is only allocated after a variable is added to the struct.

Creating Struct Instances

In the above example, we have created a struct named Person. We can create a struct variable as follows:

```
Person p;
```

The p is a struct variable of type Person. We can use this variable to access the members of the struct.

Accessing Struct Members

To access the struct members, we use the instance of the struct and the dot (.) operator. For example, to access the member age of struct Person:

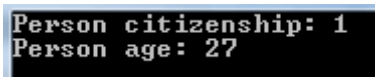
```
p.age = 27;
```

We have accessed the member age of struct Person using the struct's instance, p. We have then set the value of the member age to 27, Example:

```
#include <iostream>
using namespace std;
```

```
struct Person
{
	int citizenship;
	int age;
};
int main(void) {
	struct Person p;
	p.citizenship = 1;
	p.age = 27;
	cout << "Person citizenship: " << p.citizenship << endl;
	cout << "Person age: " << p.age << endl;
	return 0;
}
```

Output:



Pointers to Structure

It's possible to create a pointer that points to a structure. It is similar to how pointers pointing to native data types like int, float, double, etc. are created. Note that a pointer in C++ will store a memory location, Example:

```
#include <iostream>
using namespace std;
struct Length
{
	int meters;
	float centimeters;
};
int main()
{
	Length *ptr, l;
	ptr = &l;
	cout << "Enter meters: ";
	cin >> (*ptr).meters;
	cout << "Enter centimeters: ";
	cin >> (*ptr).centimeters;
	cout << "Length = " << (*ptr).meters << " meters " <<
(*ptr).centimeters << " centimeters";
	return 0;
```

```
}
```

Output:

```
Enter meters: 5
Enter centimeters: 17
Length = 5 meters 17 centimeters
```

Struct as Function Argument

You can pass a struct to a function as an argument. This is done in the same way as passing a normal argument. The struct variables can also be passed to a function. A good example is when you need to display the values of struct members. Let's demonstrates this Example:

```
#include<iostream>
using namespace std;
struct Person
{
        int citizenship;
        int age;
};
void func(struct Person p);
int main()
{
        struct Person p;
        p.citizenship = 1;
        p.age = 27;
        func(p);
        return 0;
}
void func(struct Person p)
{
        cout << " Person citizenship: " << p.citizenship<<endl;
        cout << " Person age: " << p.age;
}
```

Output:

```
Person citizenship: 1
Person age: 27
```

Limitation of a C++ structure

The following are the limitations of structures:

- The struct data type cannot be treated like built-in data types.
- Operators like + -, and others cannot be used on structure variables.

- Structures don't support data hiding. The members of a structure can be accessed by any function regardless of its scope.
- Static members cannot be declared inside the structure body.
- Constructors cannot be created inside a structure.

Record is a connected data like arrays, but it can contain different types of data like record in the database. Field contains number of fields that differs in data in other records. In C++ the record is defined as follows:

```
Name Struct
{
fields
}
```

Example: define record “data” contain name, edge:

```
Struct data
{
Char nam[30]; Int age;
};
```

To define var of type record:

```
Struct data
{
Type field1; Type field2;
.....
} var1;
```

Example:

```
#include<iostream>
using namespace std;
struct student{
   char *name;
   int no;
};
int main()
{
   student sdt1;
   sdt1.name="Mohammed";
   cout<<sdt1.name;
```

```
return 0;
}
```

When executing the program the name “Mohammed” is saved in the faild name of the variable “std1” then print it.

•*Array of records:*

The record can be an array:

```
Struct student
{
Char* name;
Int no;
} data  ;
data student[100]; // define array of type data
```

and to use the record contents use the following way:

```
Student[index].name       & student[index].age
```

•*Records and Pointers:*

After defining the record it can be pointer as follows: Example:

```
#include<iostream>
#include<string.h>
using namespace std;
struct student{
   char *name;
   int age;
} data;
int main()
{
   student *s, sdt1;
   s=&sdt1;
   sdt1.name="Talal";
   sdt1.age=20;
   cout<<sdt1.name<<endl<<sdt1.age<<endl;
return 0;
}
```

using new:

```
float *q = new float        //empty
float *q = new float(3.14)        //contain 3.14
```

Ex:

```
double *p=new double;
if (p==0) then abort ( );  //full memory else
*p=33.2
end if
```

To **Delete an record**, that use the code:-

```
float *q =new float(3.14);
delete q;
```

Linked list

Linked lists are a common alternative to arrays in the implementation of data structures. Each item in a linked list contains a data element of some type and a pointer to the next item in the list. It is easy to insert and delete elements in a linked list, which are not natural operations on arrays, since arrays have a fixed size. On the other hand access to an element in the middle of the list is usually O(n), where n is the length of the list. An item in a linked list consists of a struct containing the data element and a pointer to another linked list. In C0 we have to commit to the type of element that is stored in the linked list. We will refer to this data as having type elem, with the expectation that there will be a type definition Elsewhere telling C0 what elem is supposed to be. Keeping this in mind ensures that none of the code actually depends on what type is chosen. These considerations give rise to the following definition:

```
struct list_node {
elem data;
struct list_node* next;
};
typedef struct list_node list;
```

This definition is an example of a recursive type. A struct of this type contains a pointer to another struct of the same type, and so on. We usually use the special element of type t*, namely NULL, to indicate that we have reached the end of the list. Sometimes (as will be the case for our use of linked lists in stacks and queues), we can avoid the explicit use of NULL and obtain more elegant code. The type definition is there to create the type name list, which stands for struct list_node, so that a pointer to a list node will be list*. There are some restriction on recursive types. For example, a declaration such as

```
struct infinite {
int x;
struct infinite next;
}
```

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

Singly Linked List

Doubly Linked List

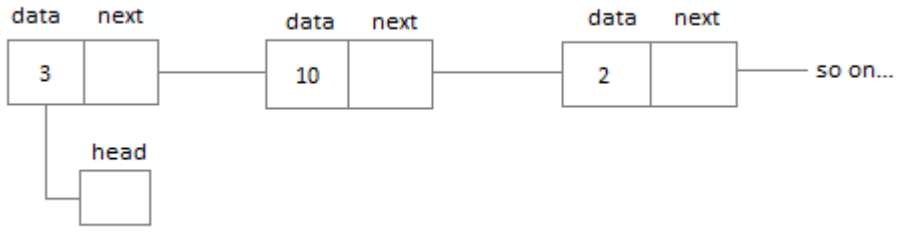
Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

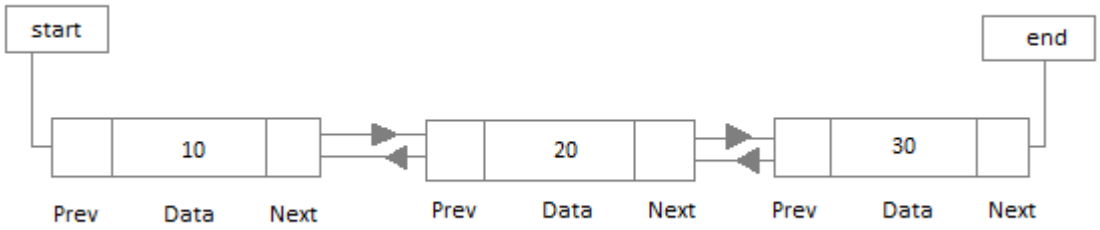
Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are insertion, deletion and traversal.



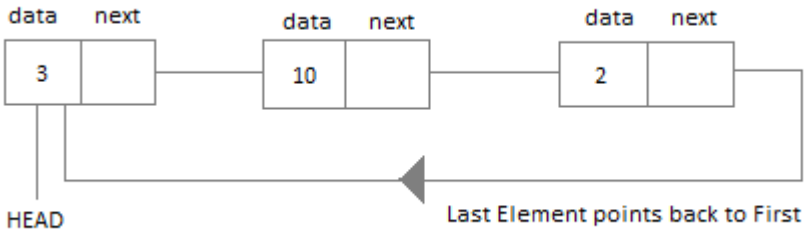
Doubly Linked List

In a doubly linked list, each node contains a data part and two addresses, one for the previous node and one for the next node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.

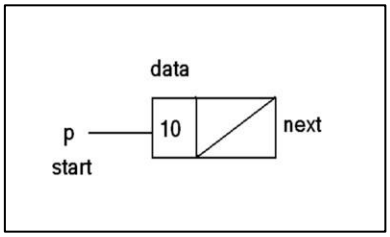


We will learn about all the 3 types of linked list, one by one, in the next tutorials. So click on Next button, let's learn more about linked lists.

Operation on Singly Linked List :-

- Creating linked list of One node :-

```
struct node {
int data;
struct node *next;
};
main( )
{
struct node *p;
struct node *start;
 p.data =10;
p.next= 0;
start=p;
```



- Creating linked list of N nodes :-

```
void creat(tnod &f) // f is the start pointer of link list
{
        int d;
        cout<<"Enter no. node: ";
        cin>>d;
        nod *p,*q;
        cout<<"Data nodes: \n";
        f=new nod;
        cin>>f->info;
        f->next=0; p=f;
        for(int i=1;i<d;i++)
                { q=new nod;
                cin>>q->info;
                q->next=0;
                p->next=q;
                p=q;
        }
        p=f;
}
```

- Print data for linked list of N nodes :-

```
void print(tnod &f) // f is the start pointer of link list
{
        nod *p;
        p=f;
```

```
        cout<<"\nresult: ";
        while(p!=0)
        {
            cout<<" "<<p->info;
            p=p->next;}
        cout<<"\n ";
}
```

- Search data in linked list of N nodes :-

```
void search(nod *f, int x)
// f is the start pointer of link list, x is an data to search
 {
tnod *p, *q;
   q=p=f;
   while(p->info!=x)
   {
    q=p;
    p=p->next;
    }
 }
```

- Inserted data in linked list after data node:-

```
void insert(tnod &f, int y, int x)
// f is the start pointer of link list, x is an data in link list
// y is data to new node adding
 {nod *t;
 nod *p,*q;
 p=f;
 t=new nod;
 t->info=y;
 t->next=0;
 search(f,x); // call search function
 t->next=p->next;
 p->next=t;
 }
```

- Deleted node from linked list based on data node:-

```
void delet(tnod &f)
```

```
{int x;
nod *p,*q;
cout<<"number you want delete it: "; cin>>x;
search(f,x);   // call search function
if(p==f)
{
  f=f->next;
  p->next=0;
}
else
 {
  q->next=p->next;
  p->next=0;
  }
delete p;
}
```

Full Example for linked list :

```
#include<iostream.h>
typedef struct nod *tnod;
struct nod
{
int info;
nod *next;
};
void creat(tnod &);
void insert(tnod &,int,int);
void delet(tnod &);
void print(tnod &);
void search(nod *,int,tnod &,tnod &);
main ()
{ int w,x,y;
nod *f,*p,*q;
do
{
cout<<"1_creat\n";
cout<<"2_insert\n";
cout<<"3_delete\n";
cout<<"4_print\n";
```

```
cout<<"5_exit\n";
cout<<"enter your chioce\n";
 cin>>w;
 switch(w)
 {
 case 1:creat(f);break;
 case 2: cout<<"no.searching: ";
 cin>>x;cout<<"no.incert: ";cin>>y;
 insert(f,y,x);break;
 case 3:delet(f);break;
 case 4:print(f);break;
 case 5:break;
}}while(w!=5);
 }        //end the main

 void creat(tnod &f)
{ nod *p,*q;
 cout<<"numbers: ";
f=new nod;
 cin>>f->info;
f->next=0;  p=f;
int d;
cout<<"Enter no. node: ";
cin>>d;
 for(int i=1;i<=d;i++)
 { q=new nod;
 cin>>q->info;
 q->next=0;
 p->next=q;
 p=q;
}
 p=f;
 }

 void insert(tnod &f,int y, int x)
 {nod *t;
 nod *p,*q;
 p=f;
 t=new nod;
 t->info=y;
```

```
t->next=0;
search(f,x,p,q);
t->next=p->next;
p->next=t;
}
void search(nod *f,int x,tnod &p,tnod &q)
{
  q=p=f;
  while(p->info!=x)
  {
   q=p;
   p=p->next;
   }
}
void delet(tnod &f)
{int x;
nod *p,*q;
cout<<"number you want delete it: "; cin>>x;
search(f,x,p,q);
if(p==f)
{
  f=f->next;
  p->next=0;
}
else
 {
  q->next=p->next;
  p->next=0;
  }
delete p;
}
void print(tnod &f)
{nod *p;
p=f;
        cout<<"\nresult: ";
while(p!=0)
{ cout<<" "<<p->info;
p=p->next;}
cout<<"\n "; }
```

References:

- Frank Carrano, D.J. Henry: Data Abstraction and Solving with C++, 2012, 6th edition, Pearson Education, Inc.
- Mark Allen Weiss: Data Structures and Algorithm Analysis in C++, 2014, 4th edition, Pearson Education, Inc.