CS11001/CS11002
Programming and Data Structures (PDS)

(Theory: 3-1-0)

Structures and Self-Referential Structures

Sizeof Structures

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

  struct stud1 {
  char name[MAXLEN];
  char roll[MAXLEN];
  int height;
  float cgpa;
  };
```

```
struct stud2{
    char *name;
    char *roll;
    int height;
    float cgpa;
   };
main()
{
  printf("%d\n",sizeof(struct
    stud1));
  printf("%d\n",sizeof(struct
    stud2));
}
```

Sizeof the first structure

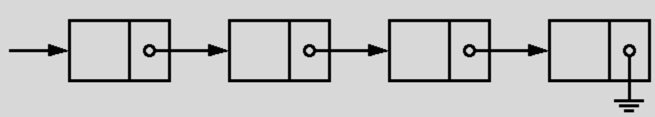
- When a structure is passed to a function, the corresponding sizeof() bytes are copied to the formal argument of the function.
- For example, in my machine sizeof(struct stud) is 208. This includes locations to store the arrays name and roll, the integer height and the floating point number cgpa.
- When a struct stud variable is passed to a function, these 208 bytes are copied to the argument.
- This, in particular, implies that changes in the members of the argument are not visible outside the function.
- This also includes changes in the arrays name and roll.
- When a struct stud value is returned from a function and assigned to a variable in the caller function, 208 bytes are copied from the returned value to the variable.

Sizeof the second structure

- Now sizeof(struct stud2) is 16. This is what is needed to store two pointers, one integer and one floating point number.
- These pointers may point to arrays (or may be allocated memory dynamically), but the memory for these arrays lies outside the structure variable.
- When we pass a struct stud2 variable to a function, only 16 bytes are copied.
  - That includes the pointers name and roll, but not the arrays which they point to.
  - Any change in the arrays pointed to by these pointers is now visible to the caller function.

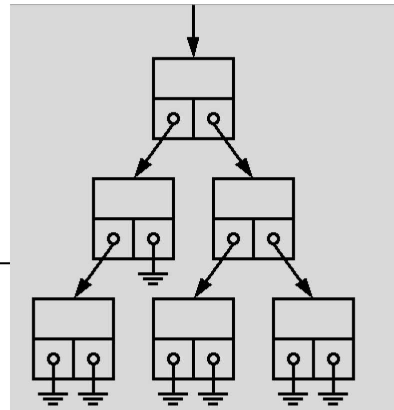
Lists

- A structure with pointer(s) to structure(s) of the same type turns out to be very useful for representing many interesting objects.
- The following figure illustrates how such structures form the basic building block (a node) for representing a list and a tree.



(a)

Tree



Structure Definition for nodes

- We will see later how such objects can be dynamically created and maintained.
- For the time being, let us focus on how a structure representing a node in a list or tree can be defined.

The structure definition

- First consider a node in a list.
- Let us assume that we are dealing with a list of integers.
- In order to create the linked structure of the above figure, we need a node to contain a pointer to another node of the same type.
- In practice, a node may contain data other than an integer and a pointer. For simplicity here we restrict the members of a node to only these two fields.
- **struct _listnode**

```
struct _listnode
  { int data;
    struct _listnode *next;
  };
```

One can also use type definitions:

```
typedef struct _listnode
{
    int data;
    struct _listnode *next;
} listnode;
```

The formal tag after **struct** is needed

- An important thing to note here is that the formal tag after the struct keyword (_listnode in the last example) was absolutely necessary for these declarations, even when the new structure is typedef'd.
- There is nothing other than this formal name that can specify the type of the pointer next.
- It is only after the part inside curly braces can be defined properly, when the typedef makes sense.
- After these definitions we can use individual variables and pointers. The declaration
  - **listnode mynode, *head;**
  - defines a structure mynode of type listnode and a pointer head to a structure of this type.
  - So, mynode has a member called data, and a pointer to a subsequent node, whose definition is also struct _listnode.
  - Thus it is called self-referential structures.

The tree node defined using structures

- A node in a (binary) tree consists of two pointers, the first for pointing to the left child and the second for pointing to the right child.
-
```
typedef struct _treenode
{ int data;
  struct _treenode *left;
  struct _treenode *right;
 } treenode;
```

After this definition one can declare individual nodes like:

- treenode thatNode, leaf[100]; One can declare pointers to nodes in the usual way:
- treenode *root; or by using other type definitions:
- typedef treenode *tnptr;
  tnptr root;

Unions

- Suppose we want to make a list of nodes.
- Each node in the list may be one of two possible types: a data node and a control node.
- Suppose further that a data node stores an int, whereas a control node stores a control information that can be specified by a 16-character string.
- A structure like the following can be used:
  -
```
struct foonode
  { int data;
    char control[16];
  } thisNode, fooArray[1024];
```

Unions is more space efficient for conditional members

- The problem with this is that irrespective of whether a node is a control node or a data node, the structure requires space for both the data and the control string.
- A data node does not use the control string at all, and similarly a control node does not require the data.
- That leads to unnecessary waste of space. In order to reduce the space requirement of each node, we should use a union instead of a struct.
  - **union barnode**

```
{ int data;
  char control[16];
 } thisNode, barArray[1024];
```

Memory Space allocation of Unions

- In this case the compiler reserves the space that is sufficient to store the biggest of the individual members.
- For example, the int member requires 4 bytes, whereas the control string requires 16 bytes.
- For the struct foonode the compiler uses 20 bytes of memory. For the union barnode, on the other hand, a memory of only 16 bytes is allocated.
- That memory (more correctly, a part of it) can be used as an integer variable or as a character string.
- In other words, the members of a union occupy overlapping space.
- When we say thatNode.data or barArray[51].data, the content of the memory is interpreted as an integer, whereas thatNode.control or barArray[51].control refers to a character string.

- This may seem confusing initially, because it is not clear what data is actually stored in the memory.
- Interpreting a character string as an integer need not always make sense, and vice versa.
- The information regarding what kind of data a union stores is to be maintained externally, i.e., outside the union.
- One possibility is to use unions in conjunction with structures.
-
```
#define DATA_NODE 0
#define CONTROL_NODE 1
  struct foobarnode
      { int what; /* can be either DATA_NODE or CONTROL_NODE */
        union
         { int data;
          char control[16];
         } info;
     } thatNode, foobarArray[1024];
```

- This structure stores the type of the node and then the union of an integer and a character string.
- Depending on the value of what, the programmer is to interpret the type of the node. If what is set to DATA_NODE, one should use the union info as an integer data and access this as thatNode.info.data or as foobarArray[131].info.data.
- On the other hand, if what is set to CONTROL_NODE, one should use the union as a character string that can be accessed as thatNode.info.control or as foobarArray[131].info.control.

Example

```
#include <stdio.h>
typedef struct _foostruct
{ int intArray[512]; double dblArray[128];
  char chrArray[1024];
  struct _foostruct *next;
 } foostruct;
typedef struct _barstruct
{ int type;
  union {
    int intArray[512]; double dblArray[128]; char chrArray[1024];
   } data;
struct _barstruct *next; } barstruct;
int main ()
   { printf("sizeof(foostruct) = %d\n", sizeof(foostruct));
     printf("sizeof(barstruct) = %d\n", sizeof(barstruct)); }
```

Size of unions

- **sizeof(foostruct) = 4100**
  - size of int intArray[512]: 512x4=2048.
  - double dblArray[128]: 128x8=1024.
  - char chrArray[1024]: 1024.
  - struct _foostruct *next: 4
  - Total: 4100 bytes
- **sizeof(barstruct) = 2056**
  - size of type: 4
  - size of int intArray[512]: 512x4=2048.
  - struct _foostruct *next: 4
  - Total: 2056 bytes