CS201
Structures and Pointers to Structures (Linked Lists)

In addition to naming places and processes, an identifier can name a collection of places. When a collection of places is given a name, it is called a composite data type and is characterized by how the individual places within variables of the data type can be accessed. In this section we will look at the **struct** and then later we will look at **classes,** an even more powerful mechanism than the struct.

The record is a very versatile data type. It is a collection of components of any data type in which the components are given names. The components of a record are called fields. Each field has its own field identifier and data type. Hence, a record often contains variables of different types.

C++ has its own vocabulary in relation to the general concept of a record. Records are called structures (abbreviated to struct), fields are called members, and field identifiers are called member names. We use record when we are referring to the general concept and struct when we are referring to a specific C++ type or variable.

To define a struct, use the keyword struct followed by the name of the structure. Then use curly braces followed by variable types and names:

```
struct StructName
{
        type1   var1;
        type2   var 2;
        …
        type3   var N;
};
```

Note the need for a semicolon at the end of the right curly brace!

The above defines a structure named "StructName". You can use StructName like it is a new data type. For example, the following will declare a variable to be of type "StructName":

```
StructName myVar;
```

Some older compilers will require you to use **struct StructName** as the type instead of just StructName.

To access the members (variables) within the structure, use the variable name followed by a dot and then the variable within the struct. This is called the member selector:

```
myVar.var1;
```

Here is an example structure:

```
struct  Recording
{
   string    title;
   string    artist;
   float     cost;
   int       quantity;
};

Recording  song;
```

Recording is a pattern for a group of four variables. song is a struct variable with four members: title, artist, cost, and quantity are the member names. The accessing expression for members of a struct variable is the struct variable identifier followed by the member name with a period in between.

song.title is a string variable.
song.artist is a string variable.
song.cost is a float variable.
song.quantity is an int variable.

The only aggregate operation defined on structures is assignment, but structures may be passed as parameters and they may be the return value type of a function. Assignment copies each member variable from the source to the destination struct. For example, the following is valid:

```
Recording song1,song2;

song1.title = "Star";
song1.artist = "Belly";
song1.cost = 10.50;
song1.quantity = 5000;
song2 = song1;
cout << song2.title << endl;
```

This will print out "Star" as the contents of song1 get copied to song2.

Hierarchical Records

A member of a record can be another record. For example, we can define an additional member, date, of type DateType that is a struct containing members month, day, and year.

```
struct  DateType
{
   int  month;
   int  day;
   int  year;
};

struct  Recording
{
   string      title;
   string      artist;
   float       cost;
   int         quantity;
   DateType    date;
};

Recording  song;
```

song.date.month accesses the month member of the date member of song.
song.date.day accesses the day member of the date member of song.

Initialization of Structs

You can initialize records by listing the values for the members within curly braces. The following definition creates struct variable myFavorite, of type Recording (defined earlier), and initializes the members.

```
Recording  myFavorite =
{
   "La Vie En Rose",
   "Edith Piaf",
   5.00,
   1,
   {
      10, 11, 1935
   }
};
```

Note that this requires we know the order that elements are defined in the structure.

Example: Here is the sample code for an array of structures listed on the CS201 website.

```
const int ASIZE = 10;

struct mystruct {
        int num;                // A number
        string name;            // A name
};

int main()
{
 mystruct arr[ASIZE];   // Array of structs
 int i;

 // Set each structures num and name to something
 // '0' + i gives ASCII for 0,1,2...9
 for (i=0; i<ASIZE; i++) {
       arr[i].num = 100 + i;
       arr[i].name = string("Somename") + char('0'+i);
 }

// Print out each structures num and name
 for (i=0; i<ASIZE; i++) {
       cout << arr[i].num << " " << arr[i].name << endl;
 }

 return 0;
}
```

Output:

```
100 Somename0
101 Somename1
102 Somename2
103 Somename3
104 Somename4
105 Somename5
106 Somename6
107 Somename7
108 Somename8
109 Somename9
```

Linked Structures

Dynamic variables combined with structures can be linked together to form dynamic lists. We define a record (called a node) that has at least two members: next (a pointer to the next node in the list) and component (the type of the items on the list). For example, let's assume that our list is a list of integers.

```
struct NodeType
{
   int num;                    // Some numeric value for this node
   NodeType  *next;            // Pointer to a NodeType
};

NodeType  *headPtr;                 // Pointer to the first thing in the list
NodeType  *newNodePtr;              // extra pointer
```

To form dynamic lists, we link variables of NodeType together to form a chain using the next member. We get the first dynamic list node and save its address in the variable headPtr. This pointer is to the first node in the list. We then get another node and have it accessible via newNodePtr:

```
headPtr = new NodeType;
newNodePtr = new NodeType;
```

Next, let's store some data into the node pointers. To access the structure, we have to first de-reference the pointer (using *) and then we need to use the dot notation to access the member of the structure:

```
(*headPtr).num = 51;
(*headPtr).next = NULL;
```

Instead of using the * and the . separately, C++ supports a special operator to do both simultaneously. This operator is the arrow: -> and it is identical to dereferencing a pointer and then accessing a structure:

```
newNodePtr->num = 55;
newNodePtr->next = NULL;
```

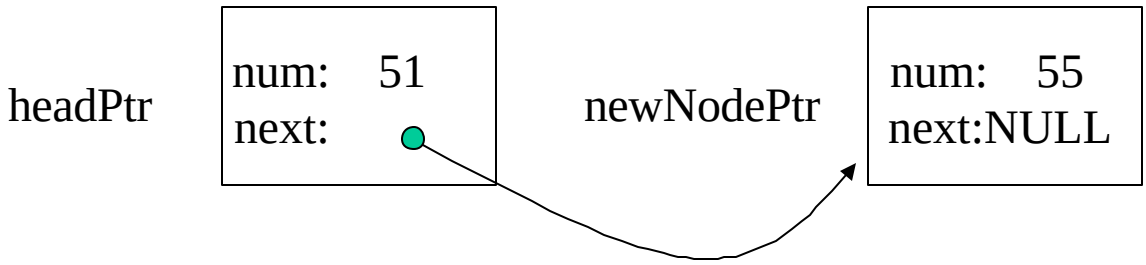
is identical to

```
(*newNodePtr).num = 55;
(*newNodePtr).next = NULL;
```

Right now we have two separate NodeTypes. We can link them together to form a linked list by having the next field of one pointing the address of the next node:

```
headPtr->next = newNodePtr;
```

We now have a picture that looks like:

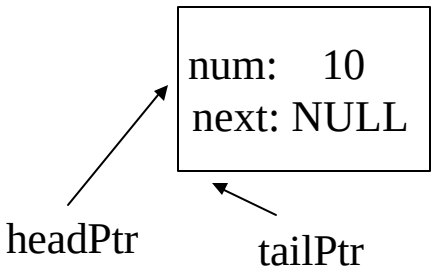


We just built a linked list consisting of two elements! The end of the list is signified by the *next* field holding NULL.

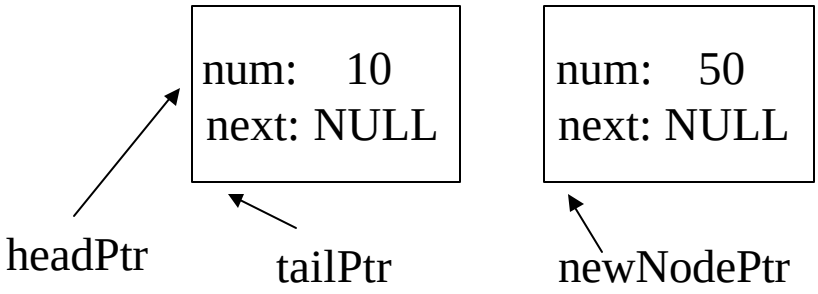
We can get a third node and store its address in the next member of the second node. This process continues until the list is complete. The following code fragment reads and stores integer numbers into a list until the input is –1:

```
struct NodeType
{
 int num;
 NodeType *next;
};

int main()
{
 NodeType *headPtr, *newNodePtr, *tailPtr, *tempPtr;
 int temp;

 headPtr = new NodeType;
 headPtr->next = NULL;
 tailPtr = headPtr;                    // Points to the end of the list
 cout << "Enter value for first node" << endl;
 cin >> headPtr->num;          // Require at least one value

 cout << "Enter values for remaining nodes, with -1 to stop." << endl;
 cin >> temp;
 while (temp!=-1) {
       // First fill in the new node
       newNodePtr = new NodeType;
       newNodePtr->num = temp;
       newNodePtr->next = NULL;
       // Now link it to the end of the list
       tailPtr->next=newNodePtr;
       // Set tail to the new tail
       tailPtr = newNodePtr;
       // Get next value
       cin >> temp;
 }
```

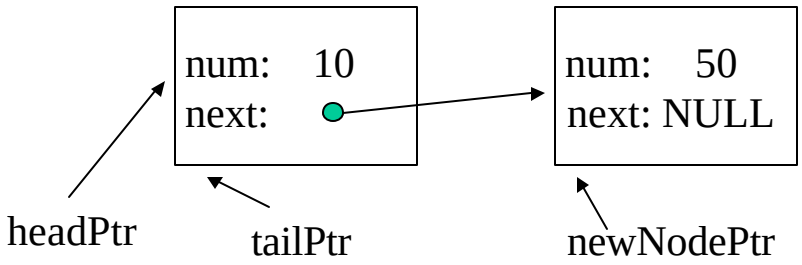
This program (it is incomplete, we'll finish it below) first allocates memory for headPtr and inputs a value into it. It then sets tailPtr equal to headPtr. tailPtr will be used to track the end of the list while headPtr will be used to track the beginning of the list. For example, let's say that initially we enter the value 10:



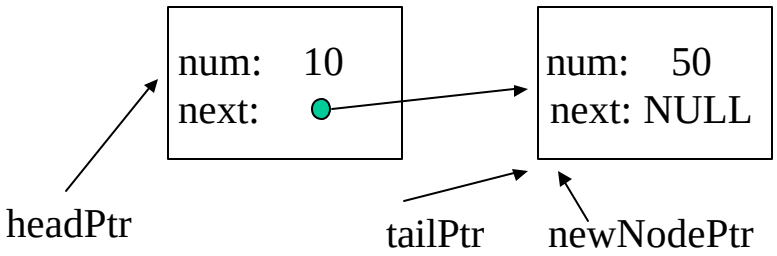
Upon entering the loop, let's say that we enter 50 which is stored into temp. First we create a new node, pointed to by newNodePtr, and store data into it:



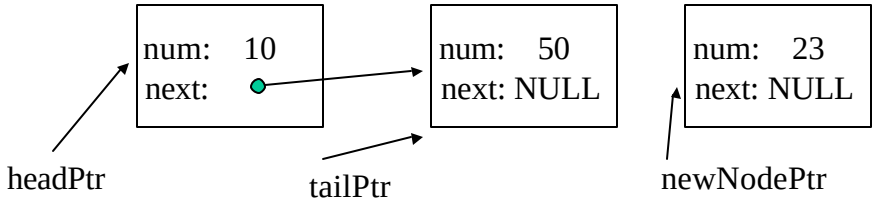
Then we link tailPtr->next to newNodePtr:



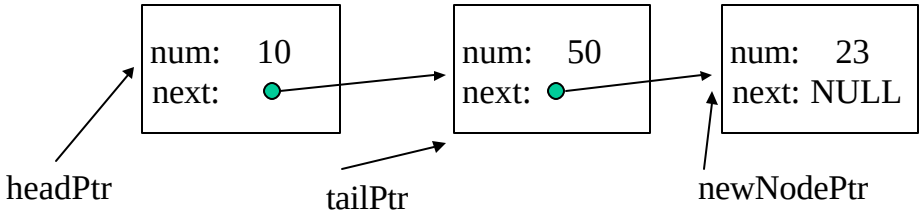
Finally we update tailPtr to point to newNodePtr since this has become the new end of the list:



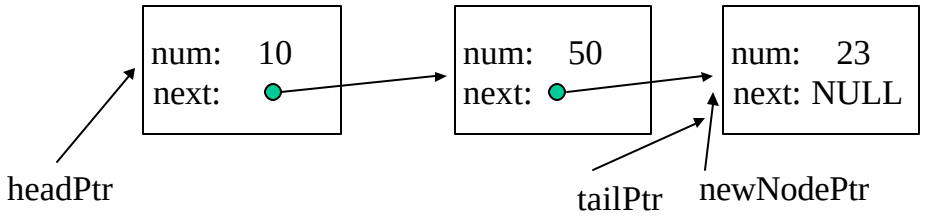
Let's say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by newNodePtr, and filling in its values:



Then we link tailPtr to newNodePtr:



Finally we update tailPtr to point to the new end of the list, newNodePtr:



The process shown above continues until the user enters –1. Note that this allows us to enter an arbitrary number of elements, up until we run out of memory! This overcomes limitations with arrays where we need to pre-allocate a certain amount of memory (that may turn out later to be too small).

Lists of dynamic variables are traversed (nodes accessed one by one) by beginning with the first node and accessing each node until the next member of a node is NULL. The following code fragment prints out the values in the list.

```
cout << "Printing out the list" << endl;
tempPtr = headPtr;
while (tempPtr!=NULL) {
      cout << tempPtr->num << endl;
      tempPtr=tempPtr->next;
}
```

tempPtr is initialized to headPtr, the first node. If tempPtr is NULL, the list is empty and the loop is not entered. If there is at least one node in the list, we enter the loop, print the member component of tempPtr, and update tempPtr to point to the next node in the list. tempPtr is NULL when the last number has been printed, and we exit the loop.

Once we have printed out the data, we're not done! Before we exit the program, we should make certain to free up the memory we allocated to prevent memory leaks. We can do so in a loop similar to the one we used to print out the list:

```
// Now free the dynamically allocated memory to prevent memory leaks
 while (headPtr!=NULL) {
      tempPtr=headPtr;
      headPtr=headPtr->next;
      delete tempPtr;
 }
}        // End program (piecing together all of the above)
```

This loop goes through and frees each node until we reach the end.

Note that we used two pointers above, tempPtr and headPtr. What is wrong with the following?

```
while (headPtr!=NULL) {
     delete headPtr;
     headPtr=headPtr->next;
}
```

Because the types of a list node can be any data type, we can create an infinite variety of lists. Pointers also can be used to create very complex data structures such as stacks, queues, and trees that are the subject of more advanced computer science courses.

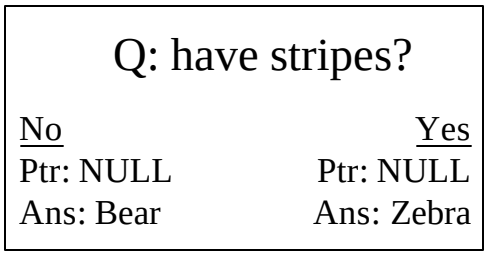
Some sample programs that use pointers to create linked lists and also an array of structs is on the CS201 web page in the Sample Code link.

Pointer and Struct Example: Animal Guessing Game

Let's do a more complex example with pointers and structures. In this example we will write a program to play a guessing game. The player thinks of an animal and the program will ask yes or no questions and try to guess what the player is thinking of.

Initially we'll only have two types of animals that the program knows about. But each time the player is done, if the program is incorrect it will ask the player to input a new question and a new animal which will be incorporated into its knowledge base.

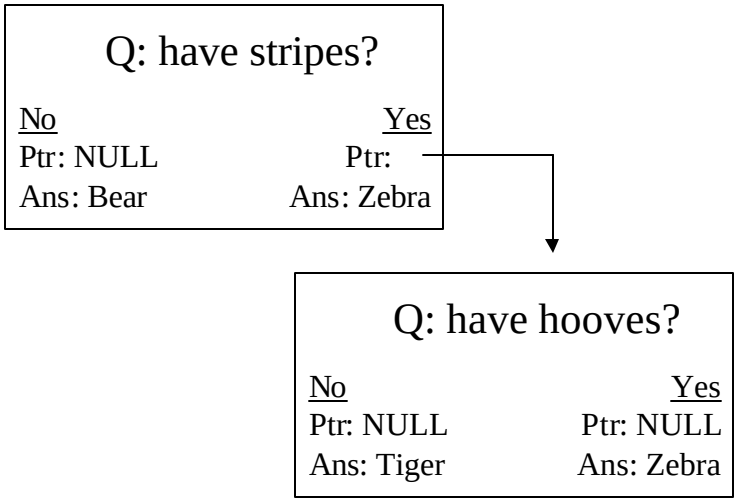
The plan is to start with a data struct as shown below:



This structure contains the following knowledge:

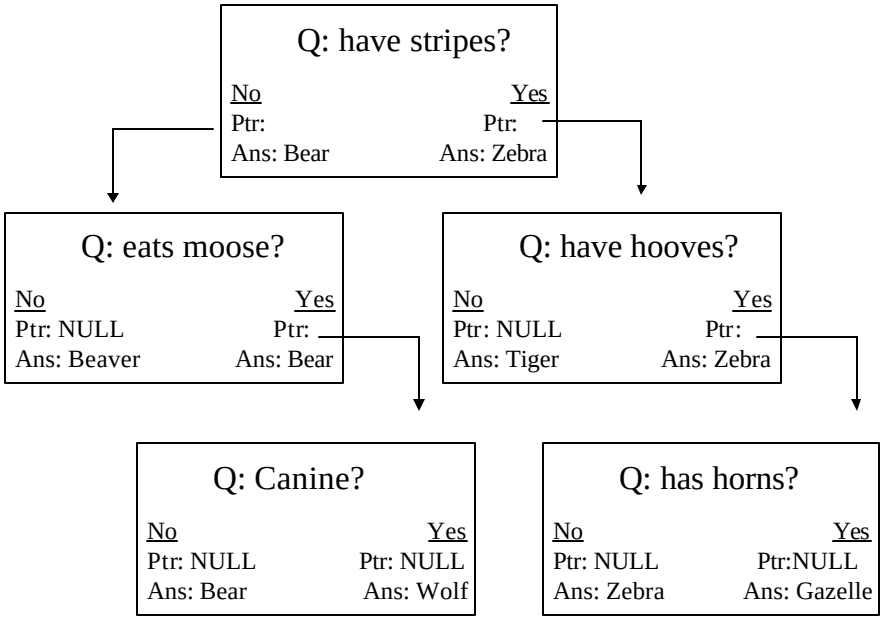
- Ask player if the animal has stripes
- If player says Y, the null pointer means guess "Zebra"
- If player says N, the null pointer means guess "Bear"

Let's say the player is thinking of a tiger. We'll answer Y to "has stripes" but N to the animal being a zebra. The program will then ask the player to enter a new question to identify the tiger. Let's say the question is "has hooves?". We'll now have the knowledge structure shown below:



If we start over, the program will ask "have stripes?" and if the player types "Y" then we go directly to the copy of the node below with "have hooves?". Depending on the answer, the program will guess Tiger or Zebra.

If we continue, each time the game is played and the program is wrong, it gains some new knowledge for later. The structure that is created is called a binary tree because it resembles an upside-down tree with the root at the top. There are at most two branches per node in the tree. A slightly larger tree with additional knowledge is shown below.



We'll represent each node with the following struct:

```
struct animal {
      string question;
      string yesGuess, noGuess;
      animal *noPtr, *yesPtr;
};
```

The "yesGuess" and "noGuess" variables will only be used if the corresponding yesPtr or noPtr variables are equal to NULL.

The complete program follows:

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

struct animal {
      string question;
      string yesGuess, noGuess;
      animal *noPtr, *yesPtr;
};

// Prototypes
void DeleteTree(animal *rootPtr);
bool AskGuess(string sGuess);
void AddNewNode(animal *ptr, char cYesOrNo);
```

```
void DeleteTree(animal *rootPtr)
{
 if (rootPtr==NULL) return;
 DeleteTree(rootPtr->noPtr);
 DeleteTree(rootPtr->yesPtr);
 delete rootPtr;
}

bool AskGuess(string sGuess)
{
 char c;
 cout << "Are you thinking of a " << sGuess << "?" << endl;
 cin >> c;
 cin.ignore();
 if (c=='y') return true;
 return false;
}

// Adds a new node below 'ptr' to the tree of knowledge
// cYesOrNo indicates if we should add to the Yes or No branch

void AddNewNode(animal *ptr, char cYesOrNo)
{
 string sAnimal;
 char c;
 animal *pNewAnimal;

 pNewAnimal = new animal;
 pNewAnimal->noPtr=NULL;
 pNewAnimal->yesPtr=NULL;

 if (cYesOrNo=='y')  {
      pNewAnimal->yesGuess = ptr->yesGuess; // Set guesses to old guess
      pNewAnimal->noGuess = ptr->yesGuess;
      ptr->yesPtr = pNewAnimal;
 }
 else {
      pNewAnimal->yesGuess = ptr->noGuess; // Set guesses to old guess
      pNewAnimal->noGuess = ptr->noGuess;
      ptr->noPtr = pNewAnimal;
 }

 cout << "What is the correct answer?" << endl;
 getline(cin,sAnimal);
 cout << "Please enter a question to identify your animal." << endl;
 getline(cin,pNewAnimal->question);
 cout << "Is the answer 'y' or 'n'?" << endl;
 cin >> c;
 cin.ignore();
 if (c=='y') {
      pNewAnimal->yesGuess = sAnimal;
 }
 else {
      pNewAnimal->noGuess = sAnimal;
 }
 return;
}
```

```
int main()
{
 animal *rootPtr=NULL, *curPtr=NULL;
 char c;
 string sGuess, sQuestion;
 bool playgame = true, madeguess = false;

 // Give some initial knowledge to the program
 rootPtr = new animal;
 rootPtr->noPtr = NULL; rootPtr->yesPtr=NULL;
 rootPtr->question = "Does it have black stripes?";
 rootPtr->yesGuess = "zebra";
 rootPtr->noGuess = "otter";

 cout << "Guess the animal!" << endl;
 curPtr = rootPtr;
 while (playgame) {
      // Travel down tree until we have a guess
      madeguess = false;
      while (madeguess == false) {
            cout << curPtr->question << endl;
            cin >> c;
            cin.ignore();
            if (c=='y') {
                  // Check if we reached bottom of the tree
                  // If so, guess an animal
                  if (curPtr->yesPtr==NULL) {
                     if (AskGuess(curPtr->yesGuess)==false) {
                        // If we're wrong, ask for the answer
                        AddNewNode(curPtr, 'y');
                     }
                     madeguess=true;
                  }
                  else curPtr=curPtr->yesPtr;
            }
            else {
                  // Check if we reached bottom of the tree
                  if (curPtr->noPtr==NULL) {
                     if (AskGuess(curPtr->noGuess)==false) {
                        AddNewNode(curPtr, 'n');
                     }
                     madeguess=true;
                  }
                  else curPtr=curPtr->noPtr;
            }
      }
      cout << "Play again? (y/n)" << endl;
      cin >> c;
      cin.ignore();
      if (c=='n') playgame=false;
      curPtr = rootPtr;       // Reset cur to root node
 }

 DeleteTree(rootPtr);  // Free up memory allocated
```

```
 return 0;
}
```

This program is split up into a couple of functions. The DeleteTree function recursively calls itself on the Yes and No branches so that we delete all of the nodes that we created Note that the delete call is last in the list of recursive calls – this makes the actual deletion the last thing that gets done, deleting the "leaves" of the tree before any "nodes" are deleted. The tree is then deleted from the bottom-up.

AskGuess prompts the user with an animal and returns true or false if that is the animal the player is thinking of.

AddNewNode takes the current pointer, which should be pointing to a leaf in the tree. It takes another value that indicates if we are to add to the "yes" or "no" links of the tree. A new node is created and filled with values input by the user.

The main function controls the action, looping continually until a leaf node is reached (the pointer is NULL). When this happens, the player is asked to guess and if the program is wrong, it asks the player to enter new information via AddNewNode.

Since the program mixes the getline and cin modes of input, the cin.ignore() is used after we read a character to skip over the remaining newline character.

This complete program is available in the Code Samples directory if you would like to run it.