COP 3331
OBJECT ORIENTED DESIGN
SPRING 2017

WEEK 13: EXCEPTIONS AND LINKED LISTS

SCHINNEL SMALL

USF
UNIVERSITY OF SOUTH FLORIDA

EXCEPTIONS

EXCEPTIONS

- Exceptions are used to signal errors or indicate that something unexpected has occurred or been detected
- They allow a program to deal with the problem in a controlled manner
- An exception can be as simple or complex as the program design requires

ERROR TESTING ORIGINS

- Simple mechanisms for error testing include the if statement (see below)

```
double divide(int numerator, int denominator)
{
     if (denominator == 0)
     {
          cout << "ERROR: Cannot divide by zero.\n";
          return 0;
     }
          else
               return static_cast<double>(numerator) / denominator;
}
```

- This is not reliable because this function returns a predetermined value that could be valid in some cases

EXCEPTIONS

- An *exception* is a value or object that signals an error

- An exception is *thrown* when the error occurs

```
double divide(int numerator, int denominator)
{
     if (denominator == 0)
          throw "ERROR: Cannot divide by zero.\n";
     else
          return static_cast<double>(numerator) / denominator;
}
```

THROWING AN EXCEPTION

- The `throw` keyword is followed by an expression which can be any value
  - Expression can be a variable, constant or object

- When a throw statement is executed, control is passed to another part of the program known as an exception handler
  - The function aborts when an exception is thrown by it

HANDLING AN EXCEPTION

- To handle an exception, the program must have a try/catch construct
- The *try block* – starts with the keyword `try` and is followed by a block of code executing any statements that may (directly or indirectly) cause an exception
- It is immediately followed by one or more *catch blocks* which are the exception handlers
  - It starts with the keyword `catch` followed by a set of parentheses containing the definition of an exception parameter

EXAMPLE: TRY CATCH CONSTRUCT

- (applied to division function)

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
     cout << exceptionString;
}
```

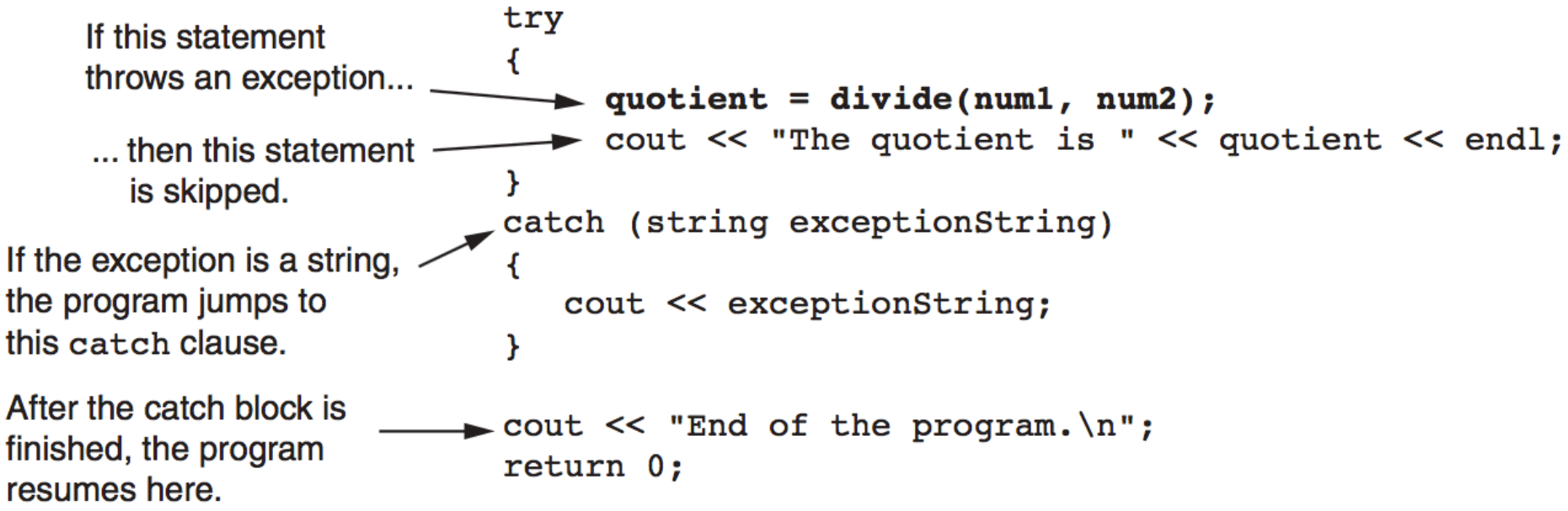
- Note: the parameter in the exception handler is a string, because the function throws a string as an exception

EXCEPTIONS: FLOW OF EXECUTION

- A function that throws an exception is called from within a try block
- If the function throws an exception, the function terminates and the try block is immediately exited
- A catch block to process the exception is searched for in the source code immediately following the try block
- If a catch block is found that matches the exception thrown, it is executed; not none is found, the program terminates

EXCEPTIONS: FLOW OF EXECUTION

- See Week 13 Example 1 on Canvas



EXCEPTIONS: FLOW OF EXECUTION

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (string exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

- A `catch` block can have at most one catch block parameter
- The `catch` block parameter becomes a placeholder for the value thrown

EXCEPTION: ANOTHER VERSION

```
try
{
    if (divisor == 0)
        throw 0;

    quotient = dividend / divisor;

    cout << "Quotient = " << quotient
    << endl;
}
catch (int)
{
    cout << "Error: Division by 0." << endl;
}
```

- In this version:
  - A literal integer is thrown as an exception so the parameter type in the catch block is an integer
  - Note that the parameter has no name
    - We don't use the value '0' in the catch block so no need to pass it via a parameter

WHAT IF THE EXCEPTION IS NOT CAUGHT?

- An exception will not be caught if the try/catch construct contains no catch blocks with exception parameters of the right data type
- An exception that is thrown outside a try block will not be caught as well
- In either case, the exception will cause the entire program to abort execution

MULTIPLE CATCH BLOCKS

- You can create multiple catch blocks to catch exceptions of different types
- A catch block with an ellipsis (. . .) catches any type of exception
  - If used, it should be the last catch block of that sequence
- Be careful about the order in which you list catch blocks

MULTIPLE CATCH BLOCKS EXAMPLE

```
// This program handles negative numbers and input failure exceptions
// It shows how values can be passed from try to catch blocks
#include <iostream>
#include <cmath>
#include <string>
using namespace std;
int main()
{
    double number;
    string message = "The input stream is in the fail state.\n";
    try
    {
        cout << "Enter a number: ";
        cin >> number;
        if (!cin) // if the wrong type of input is entered
            throw message;
        else if (number < 0)
            throw number;

        cout << "The square root of " << number << " is "
             << sqrt (number) << ".\n";
    }
    catch (string s)
    {   cout << s;  }   //Even if it is one statement, it must be in a block
    catch (double n)
    {   cout << n << " is an invalid number.\n";    }
}
```

```
Enter a number: 25
The square root of 25 is 5.
```

```
Enter a number: -8
-8 is an invalid number.
```

```
Enter a number: c
The input stream is in the fail state.
```

TIP FOR USING EXCEPTIONS

- Determine which parts of the code that can cause an exception and place *only* those parts in the try block
  - The statements that should be ignored if an exception is thrown should be in the try block as well

- Exceptions should *not* replace input validation
  - Best used for more abnormal errors (like division by 0 and input failure)

BUILT-IN MECHANISMS FOR EXCEPTION HANDLING

- C++ provides support to handle exceptions via a hierarchy of classes

- `what` function: returns a string containing the exception object thrown by C++'s built-in exception classes
  - May vary by IDE

- `class exception:` base class of the exception classes provided by C++
  - Contained in the header file `exception`

BUILT-IN MECHANISMS FOR EXCEPTION HANDLING

- Two (derived) subclasses of `exception` (both classes defined in `stdexcept`):
  - `logic_error` includes subclasses:
    - `invalid_argument`: for use when illegal arguments are used in a function call
    - `out_of_range`: string subscript out of range error
    - `length_error`: if a length greater than the maximum allowed for a string object is used
  - `runtime_error` includes subclasses:
    - `overflow_error` and `underflow_error`

EXCEPTION HANDLING TECHNIQUES

- When an exception occurs, the programmer usually has three choices:
  - Terminate the program
  - Include code to recover from the exception
  - Log the error and continue

- In some cases, it is best to terminate the program when an exception occurs
  - Example: if an input file does not exist when the program executes
    - There is no point in continuing with the program
    - Program can output an appropriate error message and terminate

EXCEPTION HANDLING TECHNIQUES

- In some cases, you will want to handle the exception and let the program continue
  - Example: a user inputs a letter instead of a number
    - The input stream will enter the fail state
    - Can include input validation to keep prompting the user to input a number until the entry is valid
- In other cases the program must run regardless of the exception
  - Example: if the program is designed to run a nuclear reactor or continuously monitor a satellite
    - It cannot be terminated if an exception occurs
  - When an exception occurs
    - The program should write the exception into a file and continue to run

OBJECT-ORIENTED EXCEPTION HANDLING

CREATING AN EXCEPTION CLASS

- An *exception class* can be defined in a class and thrown as an exception by a member function

- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to `catch` block

- Exception class with member variables typically includes:
  - Constructors
  - The function `what`

- A class can have more than one exception class

CREATING AN EXCEPTION CLASS

- Consider Example 2 (Week 13 docs on Canvas):

```
class Rectangle
{
private:
    double width;      // The rectangle's width
    double length;     // The rectangle's length
public:
    // Exception class for a negative width
    class NegativeWidth
    { };

    // Exception class for a negative length
    class NegativeLength
    { };
```

- Note: The exception class is defined *within* the Rectangle class
  - No members – here, only the name is important (to be used in the exception handling code)

CREATING EXCEPTION HANDLING CODE

- In the implementation file:

```
void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
        throw NegativeWidth();
}
```

- The throw statement causes an instance of the NegativeWidth class to be created and thrown as an exception
  - The NegativeWidth class acts like a public member function

- Any code that uses the Rectangle class must have a catch block to handle the exception

CREATING EXCEPTION HANDLING CODE

- In the Driver Program:

```
// Store the width in the myRectangle object.
while (tryAgain)
{
    try
    {
        myRectangle.setWidth(width);
        // If no exception was thrown, then the
        // next statement will execute.
        tryAgain = false;
    }
    catch (Rectangle::NegativeWidth)
    {
        cout << "Please enter a non-negative value: ";
        cin >> width;
    }
}
```

- The catch statement catches any exception thrown in the try block
  - Since the NegativeWidth class is declared inside the Rectangle Class, we use the :: to qualify the class name
  - No parameter needed as we did not pass a value
  - Note that this program gives the user the chance to recover from the error

CREATING AN EXCEPTION CLASS

- If an exception class has members, those members will be used to pass the cause of the error – i.e. the invalid value – to members of that class

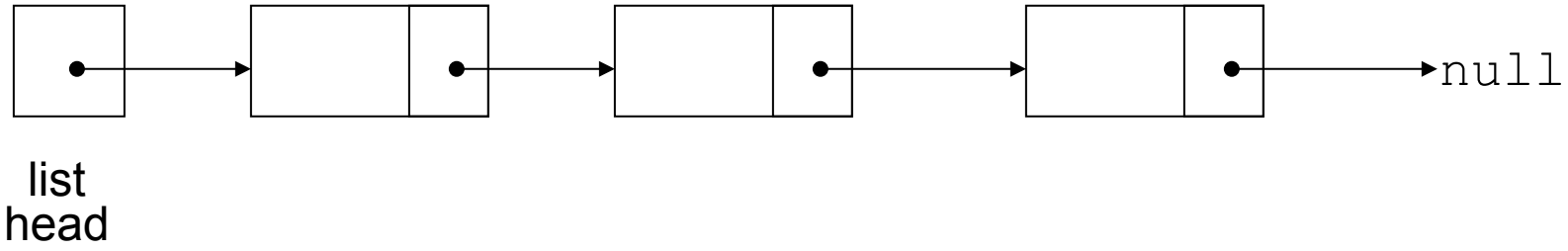
```
// Exception class for a negative width
class NegativeWidth
{
private:
     double value;
public:
     NegativeWidth(double val)
          { value = val; }

     double getValue() const
          { return value; }
};
```

INTRODUCTION TO LINKED LISTS

LINKED LISTS

- A linked list is a series of connected *nodes*, where each node is a data structure

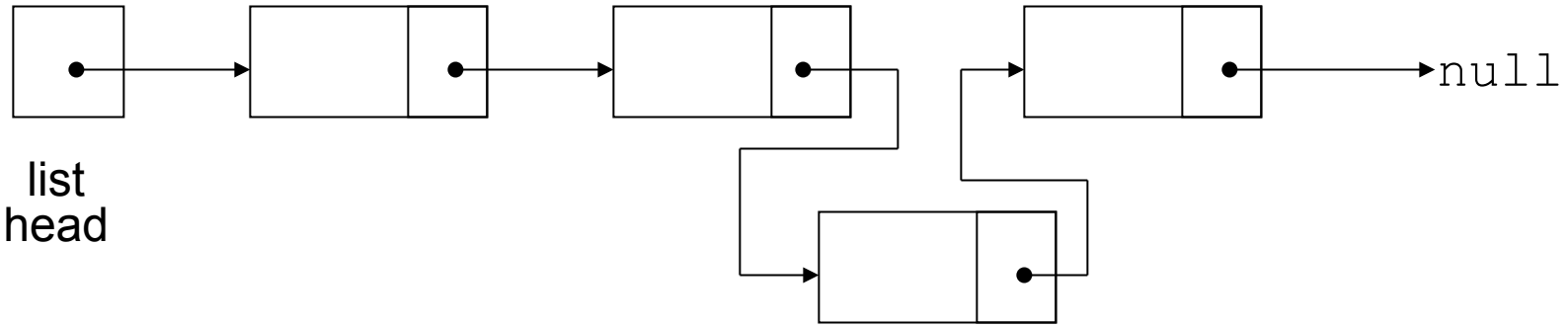


- The nodes are dynamically allocated
  - This allows the data structures to be added to or removed from the linked list during execution

LINKED LISTS VS. ARRAYS AND VECTORS

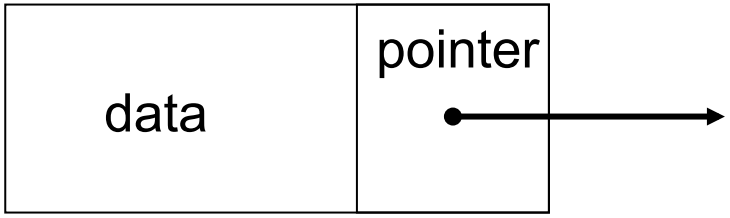
- Unlike arrays, which have a fixed size , Linked lists can grow and shrink as needed

- Linked lists can insert a node between other nodes easily, unlike vectors, which require all the elements below an insertion point to be moved to accommodate it



LINKED LISTS – NODE ORGANIZATION

- A node contains:
  - One or more data fields for data storage
  - A pointer that can point to another node



- A linked list contains 0 or more modes
  - The list head points to the first node
  - The last node points to null (address 0)
    - If the list head points to null the list is empty

LINKED LISTS – DECLARATIONS

- To declare a node:

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

  - No memory is allocated at this time

- Define a pointer for the head of the list:

```
ListNode *head = nullptr;
```

BASIC LINKED LIST OPERATIONS

BASIC LINKED LIST OPERATIONS

- The basic operations of a linked list include:
  - Appending a node
  - Traversing the list
  - Inserting a node
  - Deleting a node
  - Destroying the list

- Refer to Example 3 in Week 13 Docs for the full source code

CREATING A NEW NODE

- Given the node definition:

```
struct ListNode
{
    double value;            // The value in this node
    struct ListNode *next;   // To point to the next node
};
ListNode *head;             // List head pointer
```

  - We create a new node: `newNode = new ListNode;`
  - Initialize the contents: `newNode->value = num;`
  - Set the pointer to nullptr: `newNode->next = nullptr;`

APPENDING A NODE

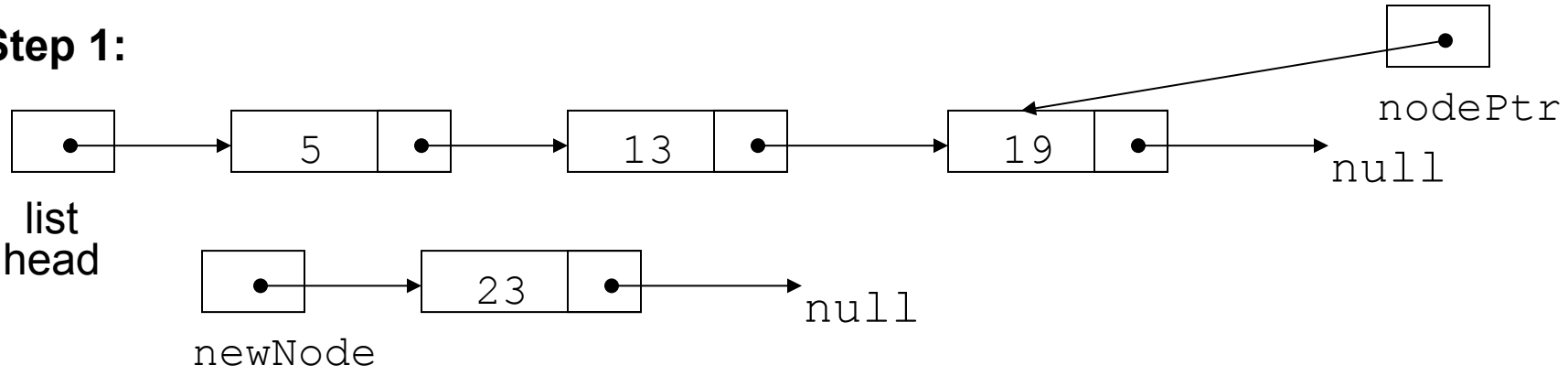
- Appending the node involves adding a node to the end of the list

- Basic process:
  - Create the new node (as already described)

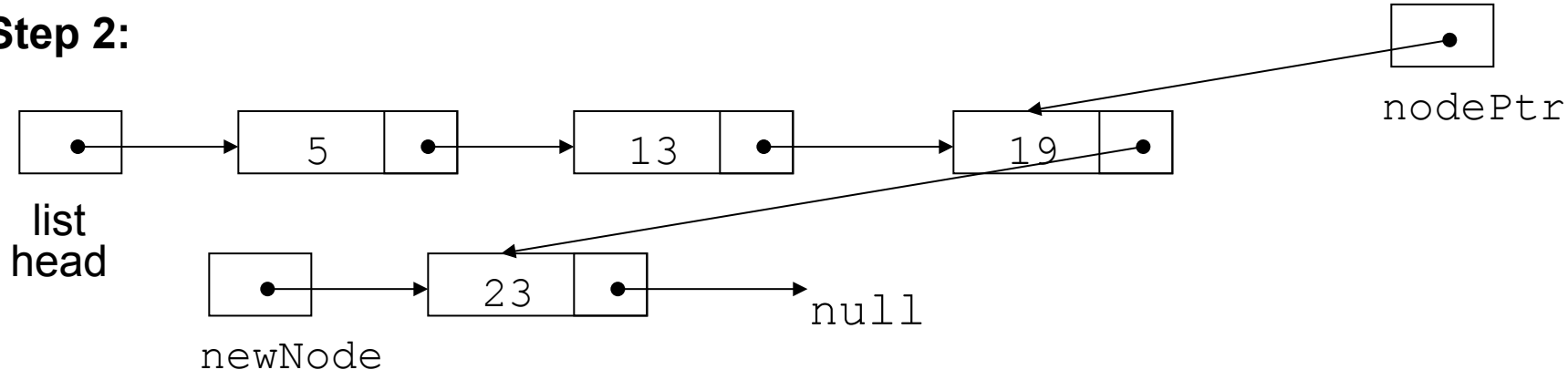
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - else,
      - traverse the list to the end
      - set pointer of last node to point to new node

APPENDING THE NODE

Step 1:



Step 2:

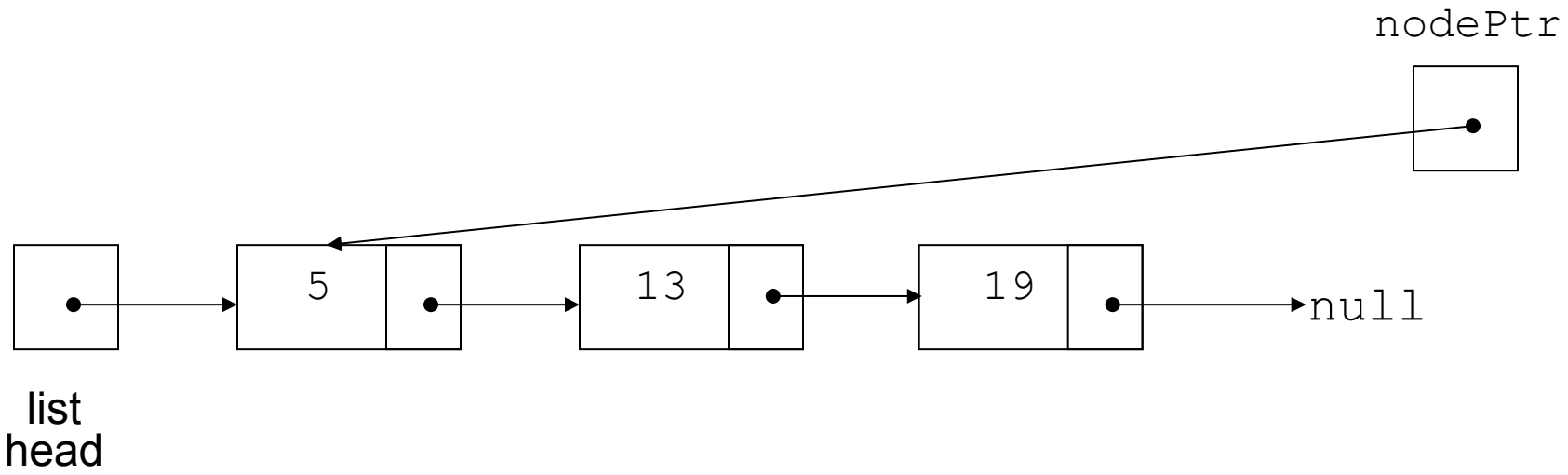


TRAVERSING THE LIST

- Visit each node in a linked list: display contents, validate data, etc.

- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not a null pointer
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

TRAVERSING A LINKED LIST



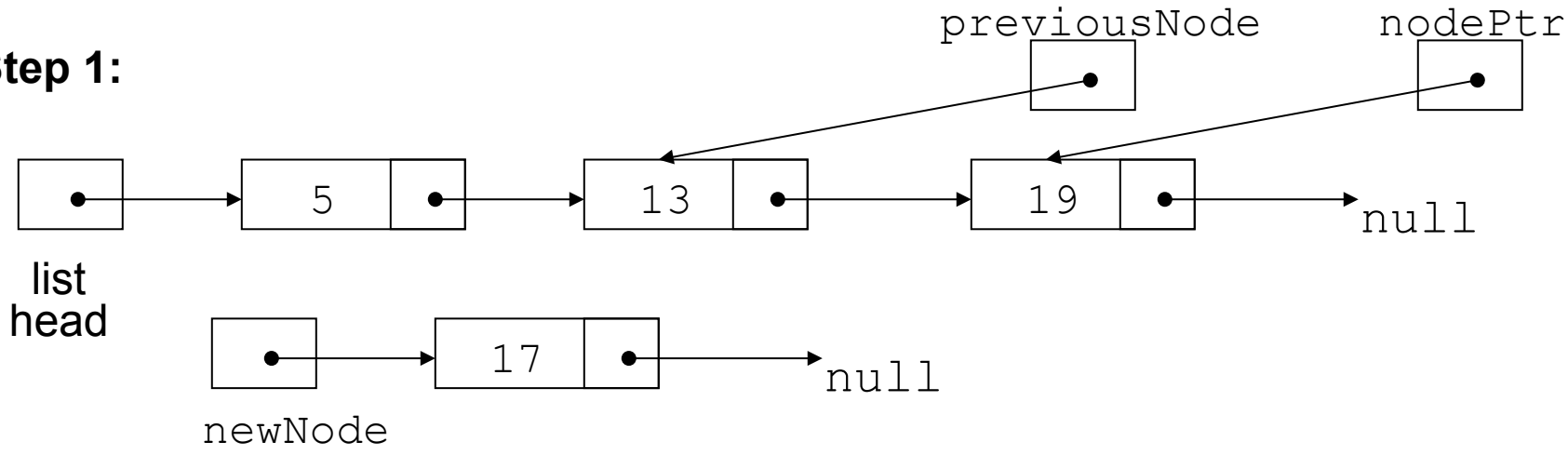
`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to the null pointer, and the list traversal stops

INSERTING A NODE

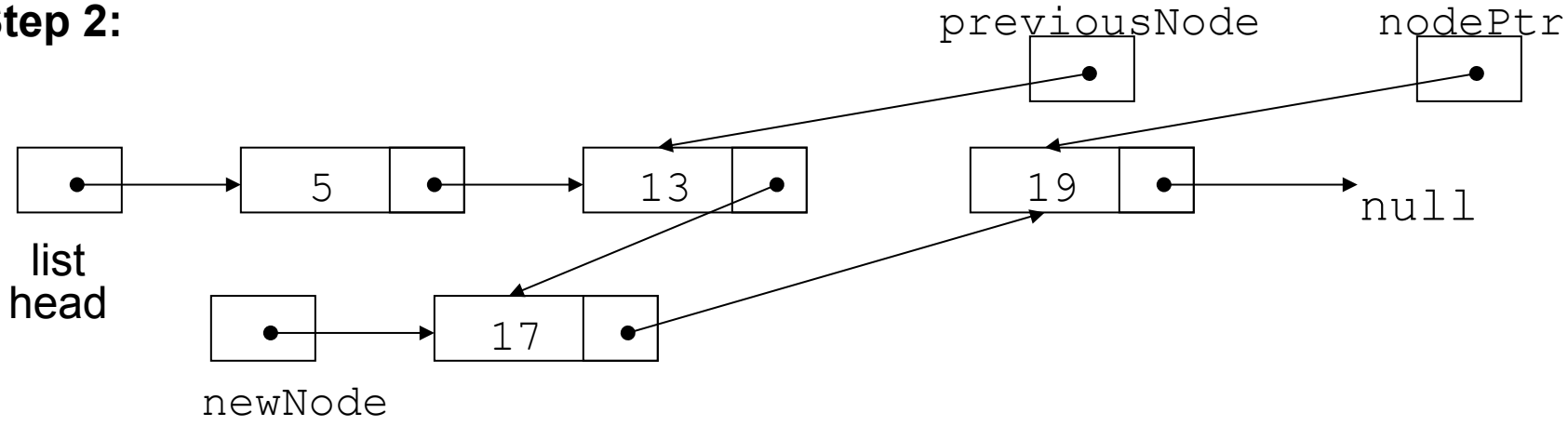
- This can be used to maintain a linked list *in order*
- Requires two pointers to traverse the list:
  - One pointer to locate the node with data value greater than that of node to be inserted
  - One pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

INSERTING A NODE

Step 1:



Step 2:

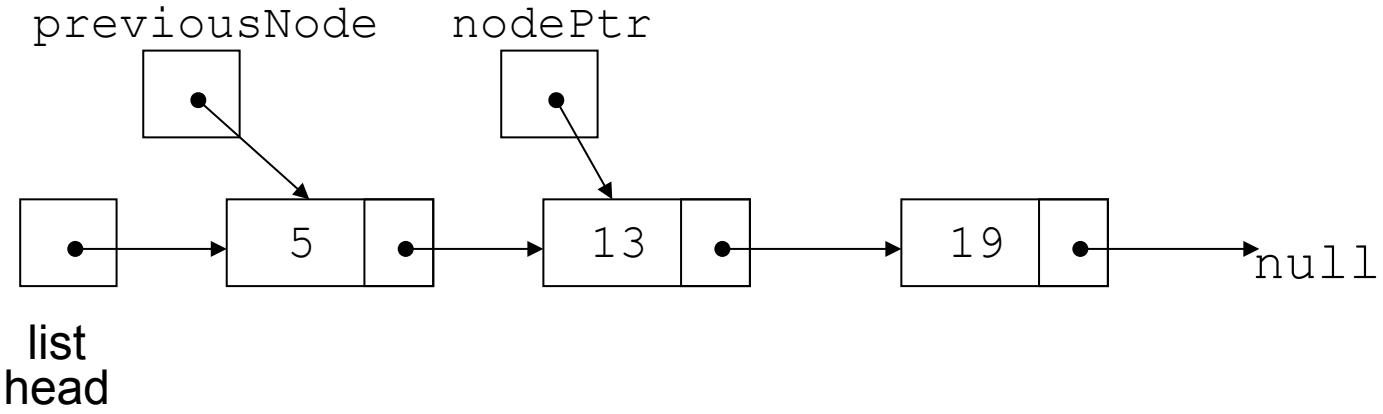


DELETING A NODE

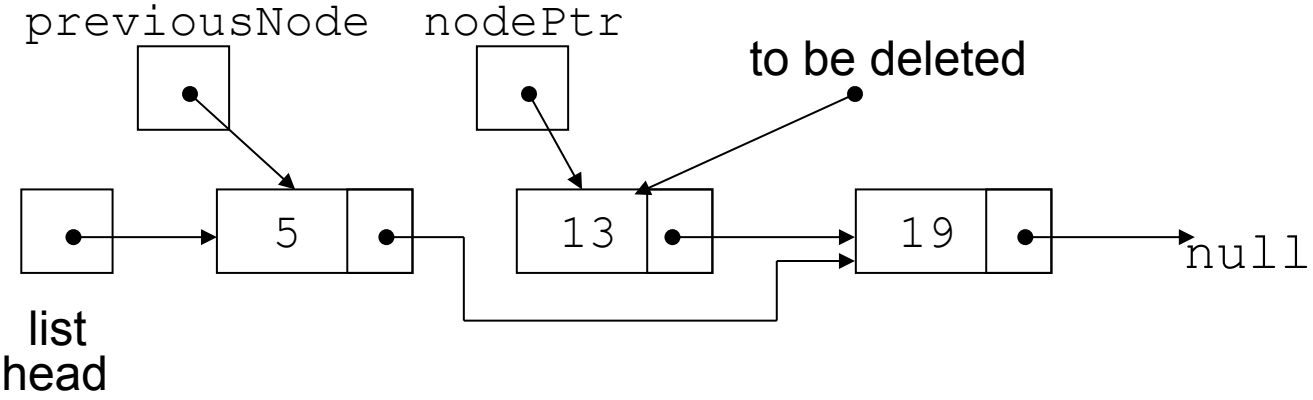
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers:
  - one to locate the node to be deleted,
  - one to point to the node before the node to be deleted

DELETING A NODE

Step 1:



Step 2:



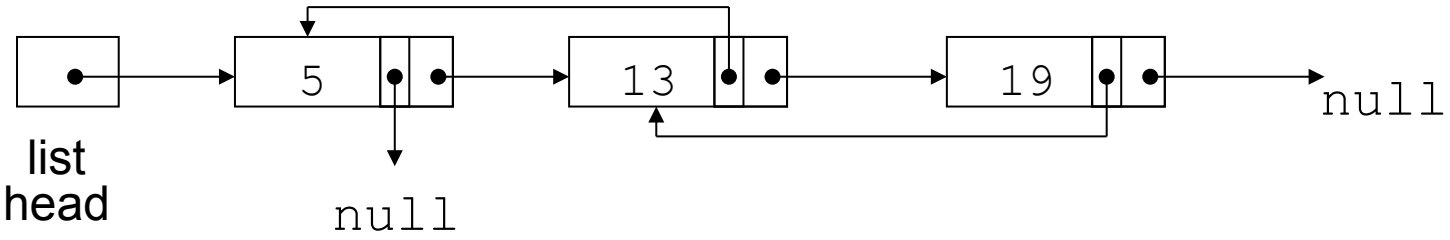
DESTROYING THE LIST

- All nodes must used in the list for it to be completely destroyed
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `nullptr`

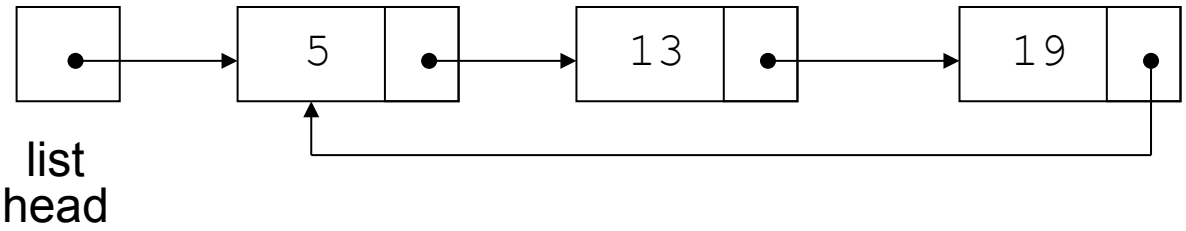
TIPS ON LINKED LISTS

- You can create a template to store linked lists of any type

- Variations of the linked list include:
  - Doubly Linked List



  - Circularly Linked List



THE STL LIST CONTAINER

- list is implemented as a doubly linked list
- foward_list is implemented as a singly linked list
- The STL includes member functions for
  - locating beginning, end of list: `front, back, end`
  - adding elements to the list: `insert, merge, push_back, push_front`
  - removing elements from the list: `erase, pop_back, pop_front, unique`