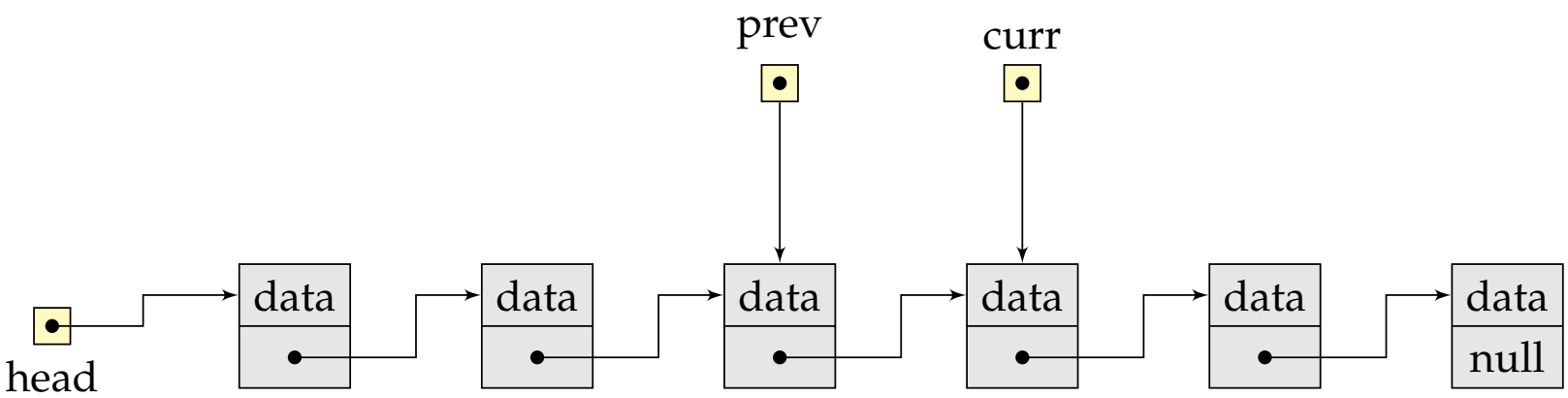
CSCI 2270: Data Structures

Lecture 08: Linked Lists

Ashutosh Trivedi



Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

Linked Lists: A mild introduction

Understanding Pointers

Sumner Downtown Promotion Association

Spring Treasure Hunt

Linked Lists: A mild introduction

Understanding Pointers

What is a Pointer?

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int x = 33;
```

Type	Name	Value	Address
int	x	33	0001
			0002
			0003
			0004
			0005

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int y[2] = {11, 14};
```

Type	Name	Value	Address
int	x	33	0001
int	y[0]	11	0002
int	y[1]	14	0003
			0004
			0005

What is a Pointer?

1. *A pointer is a data type that "points to" another value stored in memory.*
2. *A pointer is a variable that holds the "memory address" where a value lives.*

```
int* p = &x;
```

Type	Name	Value	Address
int	x	33	0001
int	y[0]	11	0002
int	y[1]	14	0003
			0004
int*	p	0001	0005

Declaring a pointer

```
int* p;
int *p;
string* q;
```

Pointer Operations

1. Address-of Operator &
2. Contents-of Operator *

Pointer Operations

1. Address-of Operator &
2. Contents-of Operator *

Examples:

```
int x = 25
```

Pointer Operations

1. Address-of Operator &
2. Contents-of Operator *

Examples:

```
int x = 25
int* p = &x
```

p points to the address of x

Pointer Operations

1. Address-of Operator &
2. Contents-of Operator *

Examples:

```
int x = 25
```

```
int* p = &x
```

p points to the address of x

```
std::cout<< *p
```

*p are the contents of p

Pointer Operations

1. Address-of Operator &
2. Contents-of Operator *

Examples:

```
int x = 25
```

```
int* p = &x
```
p points to the address of x

```
std::cout<< *p
```
*p are the contents of p

```
*p = *p + 1;
```
*p is an alias of x.

Pointers and Arrays

- In C++, arrays and pointers are intimately connected!

Pointers and Arrays

- In C++, arrays and pointers are intimately connected!
- `int x[20]` declares that x is an array of size 20.

Pointers and Arrays

- In C++, arrays and pointers are intimately connected!
- `int x[20]` declares that x is an array of size 20.
- Moreover, x is a pointer to the memory chunk that stores $x[0]$ value.

Pointers and Arrays

- In C++, arrays and pointers are intimately connected!
- `int x[20]` declares that x is an array of size 20.
- Moreover, x is a pointer to the memory chunk that stores $x[0]$ value.
- On the other hand, $\&x$ is a pointer to the memory chunk that stores whole array x value.

Pointers and Arrays

- In C++, arrays and pointers are intimately connected!
- `int x[20]` declares that x is an array of size 20.
- Moreover, x is a pointer to the memory chunk that stores $x[0]$ value.
- On the other hand, $\&x$ is a pointer to the memory chunk that stores whole array x value.

```
1  int main() {
2    int x [20];
3
4    x[0] = 1;
5    for (int i=1; i<20; i++) x[i] = x[i-1]+i;
6    for (int i=0; i<20; i++) std::cout<< "x["<<i<<"] = "<<x[i]<<" at address: " << x+i << "\n";
7    std::cout<< std::endl;
8
9    int* p = x;
10   std::cout<< "*p = " << *p << std::endl;
11   std::cout<< "*(p)+4=" << *(p)+4 << std::endl;
12   std::cout<< "*(p+4) = " << *(p+4) << std::endl;
13   //  std::cout<< "(*p)[4] = " << (*p)[4]<< std::endl;
14
15   int (*ptr)[20]; // pointer to an array of 20 integers
16   ptr = &x;
17
18   std::cout << "*ptr+4 = " << *ptr+4 << std::endl;;
19   std::cout << "(*ptr)+4 = " << (*ptr)+4 << std::endl;
20   std::cout << "*(ptr+4) = " << *(ptr+4) << std::endl;
21   std::cout << "(*ptr)[4] = " << (*ptr)[4] << std::endl;
22
23   return 0;
24 }
```

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?
- If you wish to modify the structure of the array, you need to pass it by pointers.

Arrays as Parameters

- When you pass an array (by value) to a function, the address to the memory chunk storing the array gets passed.
- It means that the invoked function can change the values stored at individual indices.
- However, if you need to change structure of the array itself, it can not be done if you are passing by value! Why?
- If you wish to modify the structure of the array, you need to pass it by pointers.

```
1  void plusFour(int *array, int capacity) {
2    for (int i=0; i< *capacity; i++) {
3      array[i] = array[i] + 4;
4    }
5  }
6
7  int main(){
8    int capacity = 10;
9    int array[10];
10
11   for (int i=0; i < capacity; i++) array[i] =1;
12
13   plusFour(array, capacity);
14
15   for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
16 }
```

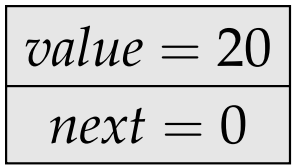
Arrays as Parameters (Contd.)

```
1  void plusFour(int *array, int capacity) {
2    int *newarray = new int[capacity];
3    for (int i=0; i< capacity; i++) {
4      newarray[i] = array[i] + 4;
5    }
6
7    array = newarray;
8    std::cout << "Inside called function:" << std::endl;
9    for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
10
11   std::cout<<std::endl;
12 }
13
14 int main(){
15   int capacity = 10;
16   int array[10];
17
18   for (int i=0; i < capacity; i++) array[i] =1;
19
20   plusFour(array, capacity);
21   std::cout << "After returning:" << std::endl;
22   for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
23 }
```

Arrays as Parameters (Contd.)

```
1  void plusFour(int** pArray, int capacity) {
2    int *newarray = new int[capacity];
3    for (int i=0; i< capacity; i++) {
4      newarray[i] = (*pArray)[i] + 4;
5    }
6
7    delete[] (*pArray);
8    *pArray = newarray;
9
10   std::cout << "Inside called function:" << std::endl;
11   for (int i=0; i< capacity; i++) std::cout << (*pArray)[i] << " ";
12
13   std::cout<<std::endl;
14 }
15
16 int main(){
17   int capacity = 10;
18   int* array = new int[capacity];
19
20   for (int i=0; i < capacity; i++) array[i] =1;
21
22   plusFour(&array, capacity);
23   std::cout << "After returning:" << std::endl;
24   for (int i=0; i< capacity; i++) std::cout << array[i] << " ";
25 }
```

Pointer to struct and classes




```
1  #include<iostream>
2
3  //self-referential structure
4
5  struct Node {
6    int value;
7    struct Node* next; //Self-referential structures
8  };
```

Pointer to struct and classes (Contd.)

```
1  int main() {
2
3    // Lets define a node
4    Node n1;
5    n1.value = 5;
6    n1.next = 0;
7
8    std::cout << "n1 data: " << n1.value << std::endl;
9
10   Node* n2 = 0;
11   n2 = &n1; // n2 now stores the address of n1
12
13   (*n2).value = 11;
14   std::cout << "n1 data: " << n1.value << std::endl;
15
16
17   Node* n3 = new Node;
18   (*n3).value = 55;
19   (*n3).next = &n1;
20
21   std::cout << "n3 data: " << (*n3).value << std::endl;
22
23
24   n3->value = 66;
25   n3->next = &n1;
26
27   std::cout << "n3 data: " << n3->value << std::endl;
28   std::cout << "n3 next pointer's data: " << n3->next->value << std::endl;
29
30
31   return 0;
32 }
```

Pointer to struct and classes (Exercise)

```
int main() {

  // Lets define a node
  Node n1;
  n1.value = 5;
  n1.next = 0;

  std::cout << "n1 data: " << n1.value << std::endl;

  Node* n2 = 0;
  n2 = &n1; // n2 now stores the address of n1

  (*n2).value = 11;
  std::cout << "n1 data: " << n1.value << std::endl;


  Node* n3 = new Node;
  (*n3).value = 55;
  (*n3).next = &n1;

  std::cout << "n3 data: " << (*n3).value << std::endl;


  n3->value = 66;
  n3->next = &n1;

  std::cout << "n3 data: " << n3->value << std::endl;
  std::cout << "n3 next pointer's data: " << n3->next->value << std::endl;


  return 0;
}
```

Why we need pointers?

- To refer to memory allocated on the heap using “new”.
- Refer to and share large data-structures among functions (recall that passing-by-reference makes called function to copy the whole structure).
- A beautiful application in data-structure: Linked-Lists!