CS 225

Data Structures

September 13 – Inheritance and Templates

G Carl Evans

Polymorphism

The idea that a single interface my take multiple types or that a single symbol may be different types.

In Object-Orientated Programming (OOP) a key example is that a single object may take on the type of any of its base types.

Virtual

Method Dispatch

1) Look at the type the method is called on
2) Look for the method in that type if found
   A. If type is virtual use runtime type and goto 2 ignoring virtual from now on
   B. Use method that method
3) No method found change to base type and goto 2

Cube.cpp

```cpp
Cube::print_1() {
  cout << "Cube" << endl;
}

Cube::print_2() {
  cout << "Cube" << endl;
}

virtual Cube::print_3() {
  cout << "Cube" << endl;
}

virtual Cube::print_4() {
  cout << "Cube" << endl;
}
```

Cube.h

```cpp
// In .h file:
virtual print_5() = 0;
```

RubikCube.cpp

```cpp
// No print_1() in RubikCube.cpp



RubikCube::print_2() {
  cout << "Rubik" << endl;
}

// No print_3() in RubikCube.cpp



RubikCube::print_4() {
  cout << "Rubik" << endl;
}

RubikCube::print_5() {
  cout << "Rubik" << endl;
}
```

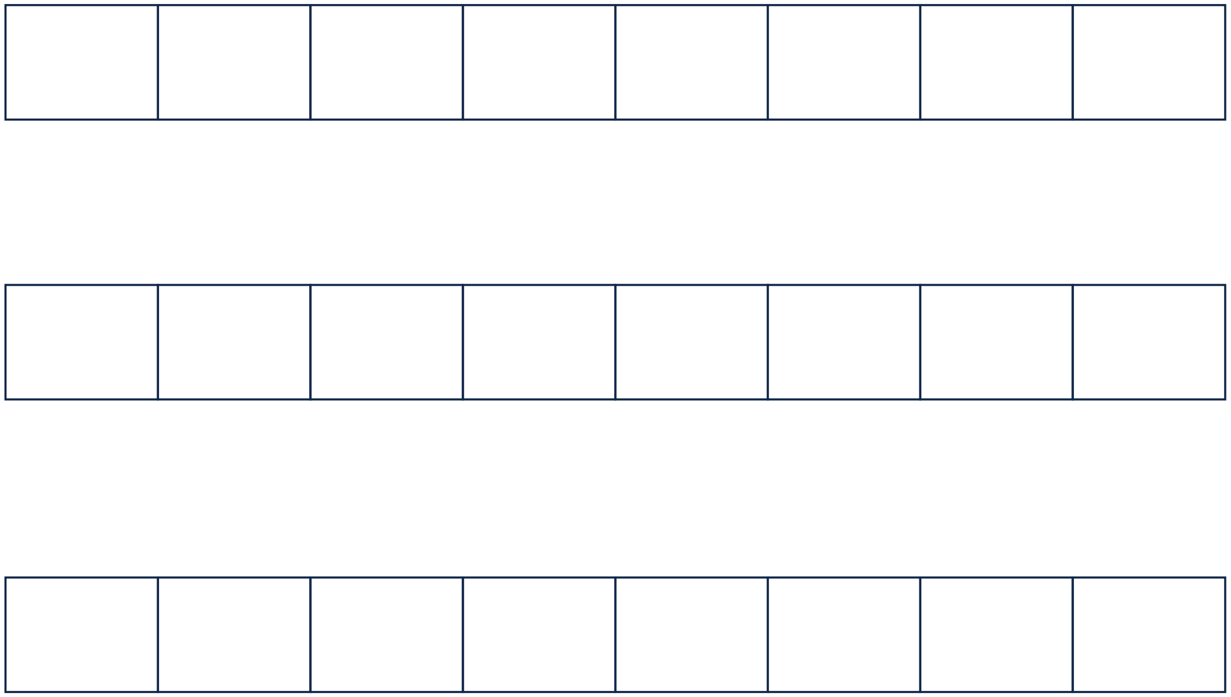
Runtime of Virtual Functions

virtual-main.cpp	`Cube c;`	`RubikCube c;`	`RubikCube rc;` `Cube &c = rc;`
`c.print_1();`			
`c.print_2();`			
`c.print_3();`			
`c.print_4();`			
`c.print_5();`			

Pure Virtual

List ADT

What types of “stuff” do we want in our list?



Templates

template1.cpp

```cpp


T maximum(T a, T b) {
  T result;
  result = (a > b) ? a : b;
  return result;
}
```

List.h

```
#pragma once


class List {
  public:








  private:



};
```

List.hpp

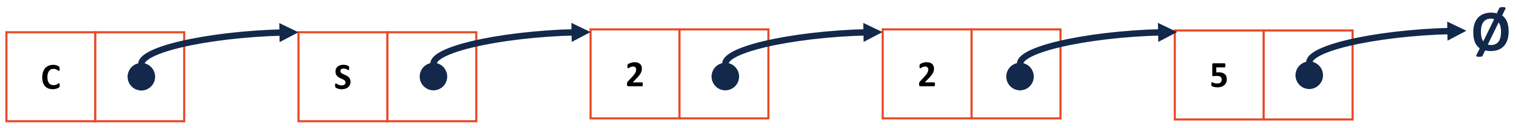
```
```

List Implementations

1.

2.

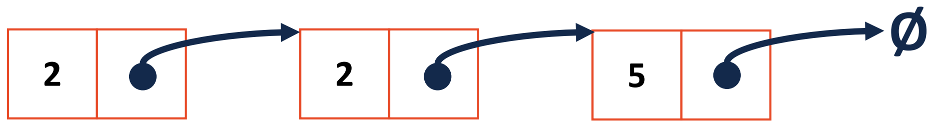
Linked Memory



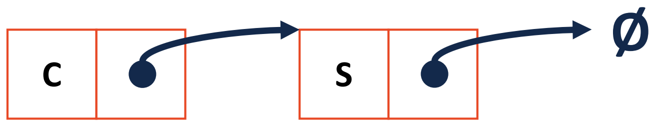
List.h

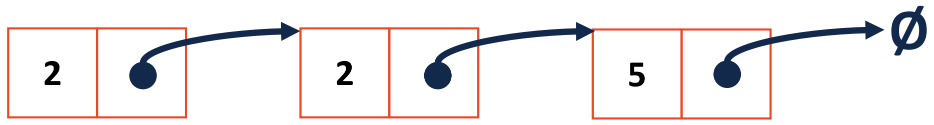
```
class ListNode {
  T & data;
  ListNode * next;
  ListNode(T & data) : data(data), next(NULL) { }
};
```

Linked Memory



Linked Memory





List.h

```
#pragma once

template <class T>
class List {
  public:
    /* ... */
  private:
    class ListNode {
      T & data;
      ListNode * next;
      ListNode(T & data) :
       data(data), next(NULL) { }
    };






};


```

List.hpp

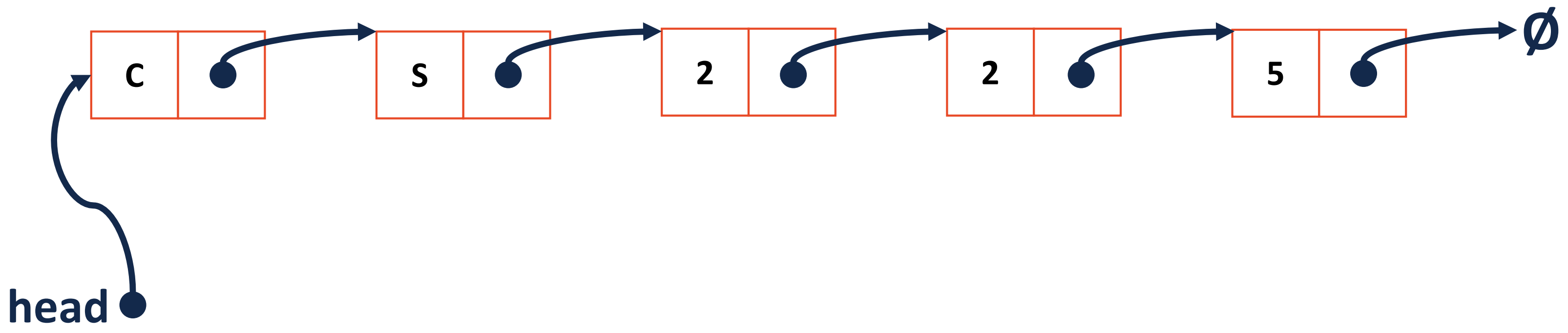
```
#include "List.h"

template <class T>
void List<T>::insertAtFront(const T& t)  {

















}
```

Running Time of Linked List `insertAtFront`

List.cpp

```
void List<T>::printReverse()
const {







}
```

Linked Memory



Running Time of Linked List `printReverse`

List.cpp

```
template <typename T>
T List<T>::operator[](unsigned index) {






}
```

List.cpp

```
ListNode *& List<T>::_index(int index) const {







}
```