### **Abstract Class:**

* + Requirement: one or more pure virtual functions
  + Syntax: nothing - no abstract keyword in cpp
  + As a result: cannot create an instance of an abstract class

### **Virtual Destructor**

* + All destructors in base classes must be virtual
  + Destructors will call the base classes destructors

|  |  |
| --- | --- |
| **virtual-dtor.cpp** | |
| 1  2  3  4  5  6  …  28  29  30  31  32  33  34  35  36 | class Cube {  public:  ~Cube() { std::cout << "~Cube() invoked." << std::endl; }  };  class RubikCube : public Cube {  public:  ~RubikCube() { std::cout << "~RubikCube() invoked." << std::endl; }  };  class CubeV {  public:  virtual ~CubeV() { std::cout << "~CubeV() invoked." << std::endl; }  };  class RubikCubeV : public CubeV {  public:  ~RubikCubeV() { std::cout << "~RubikCubeV() invoked." << std::endl; }  };  int main() {  std::cout << "Non-virtual dtor:" << std::endl;  Cube \*ptr = new RubikCube();  delete ptr;  std::cout << "Virtual dtor:" << std::endl;  CubeV \*ptrV = new RubikCubeV();  delete ptrV;  return 0;  } |

* In this case we have rubikcube dtor invoked first then cube dtor is invoked

#### **Abstract Data Type (ADT)**

* + English definition the basic operations of a data structure
  + ADT describes functionality but not implementation details

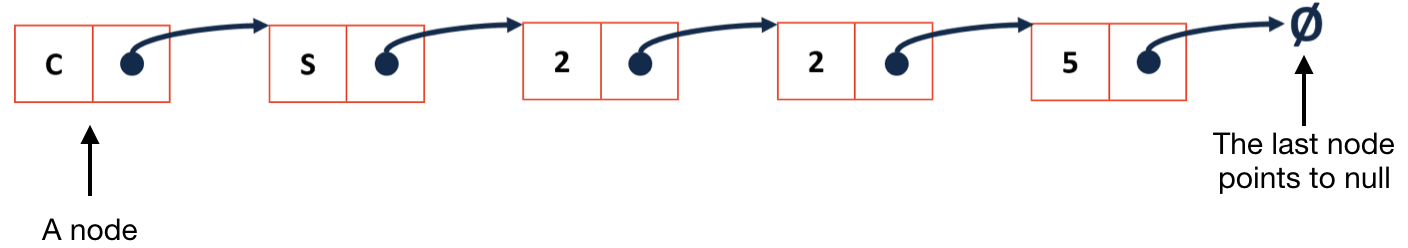
|  |  |
| --- | --- |
| **List ADT** | **Definition of Functionality** |
| Create the empty list | Creates an empty list. |
| Add data to the list | Store data. |
| Get data from the list | Access data. |
| Remove data from the list | Remove data. |
| Check if a list is empty/size | How much data is in the list. |

#### **Templates: a dynamic data type**

* + Using “Template <typename T>” so that we do not need to write the same function for various types
  + Template type are checked at compile time
    - maximum(3, 5): T = int
    - maximum(“world”, “hello”): T = string
    - maximum(cube(7), cube(42)) - but this may not complied since no > op defined
  + We can use other replace for T but using T is universally standard

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | template <typename T>  T maximum(T a, T b) {  T result;  result = (a > b) ? a : b;  return result;  } |

* There are two basic implementations of the list:
  + **Array** - sequential block of items.
  + **Linked Memory** - linked list.
* Linked Memory



* A node has two member variables:
  + A list node pointer that points to the next block (ListNode \* next).
  + The data stored in the block (T & data).
* A ListNode is available within List class, it is like a helper class. *Never* return a ListNode object/pointer outside of List, always return the data.

|  |  |
| --- | --- |
| **List.h** | |
| 1  2  3  4  5  6  …  28  29  30  31  32 | #ifndef LIST\_H  #define LIST\_H  template <typename T>  class List {  public:  /\* ... \*/  private:  class ListNode {  T & data;  ListNode \* next;  ListNode(T & data) : data(data), next(NULL) { }  };  };  #endif |