#### 

#### **Assignment Operator**

* + Check if both cubes are the same one

|  |  |
| --- | --- |
| 1  2  3 | Cube & Cube::operator=(const Cube & other){  if (this != &other) { //If I’m not copying myself  \_destroy();  \_copy(other);  }  return \*this;  }; |

#### 

#### **Virtual**

* + Allow us to override the function in derived classes
  + Evaluate rules:
    - Check the type of the variable
    - Check the matching function in that type. If virtual, go to derived type (the actual instance type)
    - If function not found, check the base type, repeat

|  |  |  |  |
| --- | --- | --- | --- |
| **Cube.cpp** | | **RubikCube.cpp** | |
| 1  2  3  4  5  6  7  8  9  10  11  12  1314  15//  16 | 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;  }  // In .h file:  virtual print\_5() = 0;  **// pure virtual function** | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | // 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;  } |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **main.cpp** | **Cube c;** | **RubikCube c;** | **RubikCube rc;**  **Cube &c = rc;**  **(polymorphed into the base type Cube)** |
| c.print\_1(); | ~~Cube~~ **Error! Can’t create an instance of an abstract class, does not compile** | Cube | Cube |
| c.print\_2(); | ~~Cube~~ **Error! Can’t create an instance of an abstract class, does not compile** | Rubik | Cube |
| c.print\_3(); | ~~Cube~~ **Error! Can’t create an instance of an abstract class, does not compile** | Cube | Cube  (go to derived type, didn’t find 3, go back) |
| c.print\_4(); | ~~Cube~~ **Error! Can’t create an instance of an abstract class, does not compile** | Rubik | Rubik  (derived type function evaluated because of **virtual**) |
| c.print\_5(); | **Error! Can’t create an instance of an abstract class, does not compile** | Rubik | Rubik |

* + Pure virtual function
    - No implementation
    - Makes Cube an **abstract class**
    - Act as a placeholder function that every derived class must implement
  + In a chain of inheritance, every class that was used as a base class need to have its functions **virtual**
    - Shape -> Cube -> RubikCube -> MyCube, then functions in Shape/Cube/RubikCube are going to be virtual, so they can be overridden

#### **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 way

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