CHAPTER 10: ARRAYS

* Syntax: type array\_name[# element]
* Access array element: array\_name[index]
* First element has index 0
* Last element has index # element - 1
* std::array and std::vector are better than raw arrays

CHAPTER 11: POINTERS

* Syntax: type \* pointer\_name
* To make pointer points to an object (ex: int object), use & (address of): int \*p=&a
* To make pointer does not point to any object, use nullptr: int \*p=nullptr
* Access data of object pointed by pointer, use \* (dereference): int b=\*p

|  |  |
| --- | --- |
| Command | Output |
|  |  |
|  |  |
|  |  |

CHAPTER 12: REFERENCES

* Reference is an alias for another variable
* Syntax: type &ref\_name
* Changing value of alias also change value of the original
* Const-reference is read-only alias to some object
  + Syntax: const int& a=b

|  |  |
| --- | --- |
| Command | Output |
|  |  |
|  |  |

CHAPTER 13: STRINGS

* Is in C++ standard library
* Including <string> header is needed
* Create string s: std::string s = text
* Add string or character: s +=added\_text
* Access string character: [] (like array) or .at member function
* Compare string: ==
* Input string: std::cin or std::getline
* Pointer to string: .c\_str() member function
* Create substring with a starting position in original string: .substr(position, length) member function
* Find substring: .find(string\_to\_find)
  + If string is not found, return std::string::npos

|  |  |
| --- | --- |
| Command | Output |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

CHAPTER 14: AUTOMATIC TYPE DEDUCTION

* Help to automatically deduce the type of an object
* Syntax: auto c=’a’ (char type)
* Can use as part of reference type: auto& y=x
* Can use as part of constant type: const auto x=123

CHAPTER 19: FUNCTIONS

* Has return type, a name, a list of parameters
* Syntax:

type function\_name(arguments) {

statement;

statement;

return something;

}

* “void” type: nothing, empty set of values
* Function declaration is needed. Argument name can be omitted.
* Return statement is needed for non-void function
* Argument can be passed to a function as value (value of original argument will not change), by reference (value of original value will change), by const reference (read only, for efficiency)
* Function overloading: many functions have the same name but different parameter types.

|  |  |
| --- | --- |
| Command | Output |
|  |  |

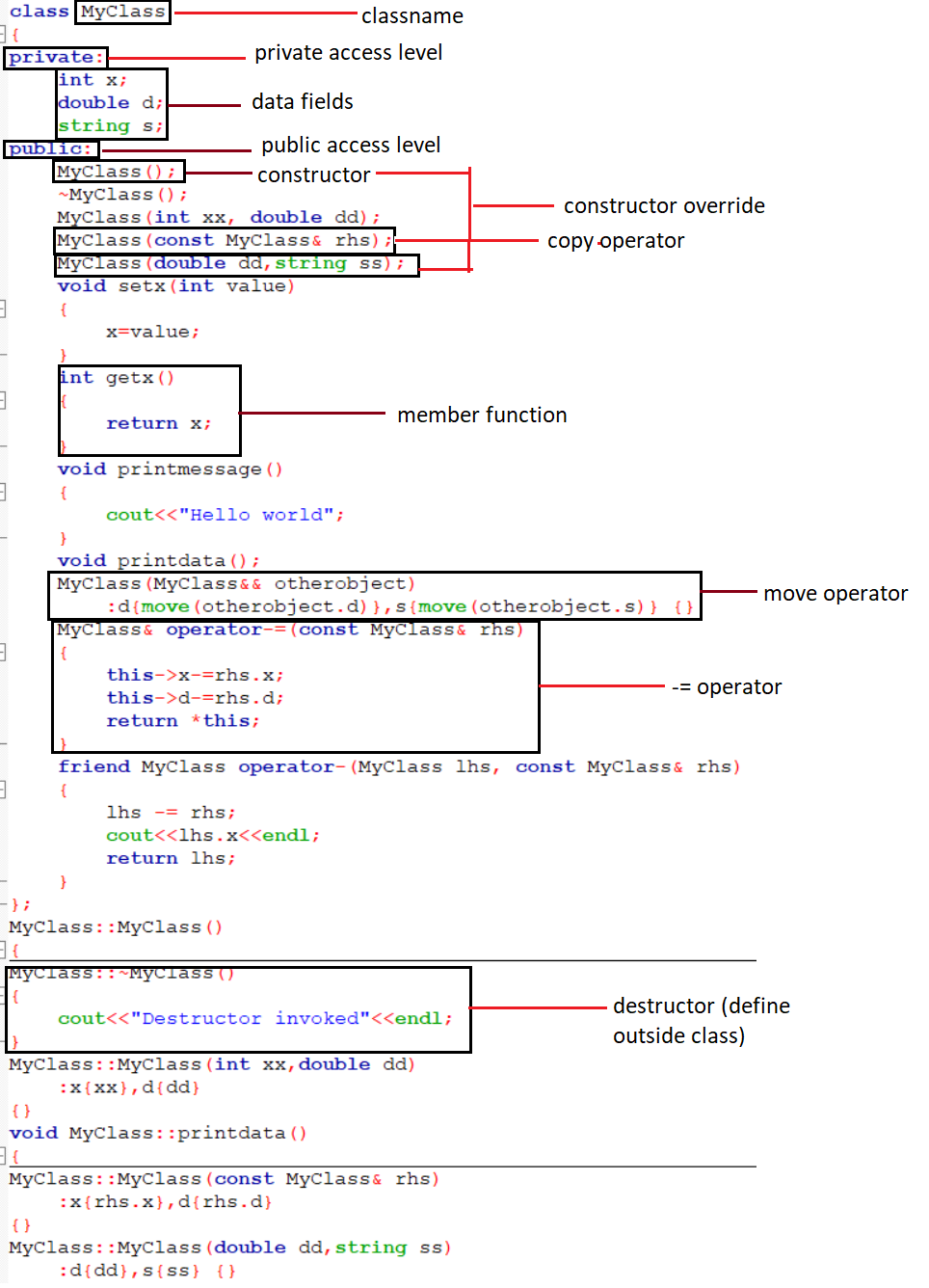
CHAPTER 21: SCOPE AND LIFETIME

* Variables only valid in some sections of the source code called **SCOPE**
* Local scope: inside a function
* Block scope: is marked by block of code start with { and end with }
* Lifetime: the time an object spends in memory, determined by storage duration
* Stack memory: memory automatically allocated at the beginning of a block and deallocated when the code block ends
* Heap memory: memory for an object is manually allocated and manually deallocated, is not determined by scope, but with operator new or smart pointers.
* Static storage duration: memory is allocated when the program starts and deallocated when program ends
* Allocate for an int pointer: int \*p=new int
* Allocate for an int array: int \*p=new int[3]
* Deallocate for an int pointer: delete p
* Deallocate for an int array: delete[] p

|  |  |
| --- | --- |
| Command | Output |
|  |  |
|  |  |

CHAPTER 23: CLASSES

* Class: user-defined type, has members
* Member: data or functions
* Object: instance of a class
* Member field: data of some type
* 2 ways to define member function:
  + Define inside the class
  + Define outside the class. We write the function type first, followed by a class name, followed by a scope resolution :: operator followed by a function name, list of parameters and a function body
* Access specifiers: define access level for members. 3 access specifiers: public, protected, private. By default, members have private access level. For struct, public access level is the default access level.
* Public: member is accessible anywhere
* Private: member is only accessible inside the class
* Constructors: member function with the same name as class, used to initialize an object of a class, cannot be invoked directly
  + Default constructor: constructor without params or with default params. Can be called without arg
  + Can have arbitrary params (user-provided constructors)
  + Member initializer list: a better, more efficient way to initialize an object. It starts with a colon, followed by member names and their initializers, where each initialization expression is separated by a comma
  + Copy constructor: invoked when object is initialized with another object of the same class. If no copy constructor is provided, shallow copy provided by compiler will be used
    - User-define copy constructor has param signature: **MyClass(const MyClass& rhs)**
    - Copy assignment: copy another object after initialization, has signature: **MyClass& operator=(const MyClass& rhs)**
  + Move constructor: move data from one object to the other (move semantic).
    - Signature: **MyClass (MyClass&& rhs)**
    - Use std::move function
    - If a user does not provide a move constructor, the compiler provides an implicitly generated default move constructor.
    - Move assignment: assign value to an object after initialization
    - Signature: **MyClass& operator=(MyClass&& otherobject)**
* Operator overloading: create meaning for expression “**+ - \* / % ^ & | ~ ! = < > == != <= >= += -= \*= /= %= ^= &= |= << >> >>= <<= && || ++ -- , ->\* -> () []”** 
  + Signature: **MyClass& operator++(MyClass&& otherobject)**
* Destructor: function that gets invoked when an object is destroyed (goes out of scope or pointer to that object is deleted
  + The name of the destructor is tilde ~ followed by a class name: **~MyClass() {}**



CHAPTER 24: EXERCISES

|  |  |
| --- | --- |
| Command | Output |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

CHAPTER 25: INHERITANCE AND POLYMORPHISM

* Inheritance: class can be built from an existing class (derived).
  + Syntax: **class MyDerivedClass : public MyBaseClass {};**
  + Derived class and objects of a derived class can access public members of a base class
  + The protected access specifier allows access to the base class and derived class, but not to objects (1)
  + A pointer to a derived class is compatible with a pointer to a base class (2)
* Polymorphism: function that morph object into different types
  + Polymorphism in C++ is achieved through an interface known as virtual functions.
  + A virtual function is a function whose behavior can be overridden in subsequent derived classes
  + Functions can be pure virtual by specifying the = 0: do not have definitions (interfaces)
    - Must be re-defined in the derived class
  + Classes with >=1 pure virtual function are abstract classes and cannot be instantiated (only be used as base classes)
  + base class must have a virtual destructor if it is to be used in a polymorphic scenari

CHAPTER 26: EXERCISE

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  #include <string>  class Person  {  private:  std::string name;  public:  explicit Person(const std::string& aname)  : name{ aname }  {}  std::string getname() const  {  return name;  }  };  class Student : public Person  {  private:  int semester;  public:  Student(const std::string& aname, int asemester)  : Person::Person{ aname }, semester{ asemester }  {}  int getsemester() const  {  return semester;  }  };  int main()  {  Person person{ "John Doe." };  std::cout << person.getname() << '\n';  Student student{ "Jane Doe", 2 };  std::cout << student.getname() << '\n';  std::cout << "Semester is: " << student.getsemester() << '\n';  } |  |

CHAPTER 38.1.1: STD::VECTOR

* Vector: sequence of contiguous elements of any types.
* Use with **<vector>** header
* Syntax: **std::vector<int> v = { 1, 2, 3, 4, 5 };**
* Insert element ad the end: **vector.push\_back(element)**
* Get vector size**: vector.size()**
* Access vector element: **use [index]** or **at(index)**
* Other sequential containers: std::list (double linked list), std::forward\_list (singly linked list), std::deque (double-ended queue)

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  #include <vector>  int main()  {  std::vector<int> v = { 1, 2, 3, 4, 5 };  std::cout << "The third element is:" << v[2] << '\n';  std::cout << "The fourth element is:" << v.at(3) << '\n';  } |  |
| #include <iostream>  #include <vector>  int main()  {  std::vector<int> v = { 1, 2, 3, 4, 5 };  std::cout << "The vector's size is: " << v.size();  } |  |