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 |
|  |  |

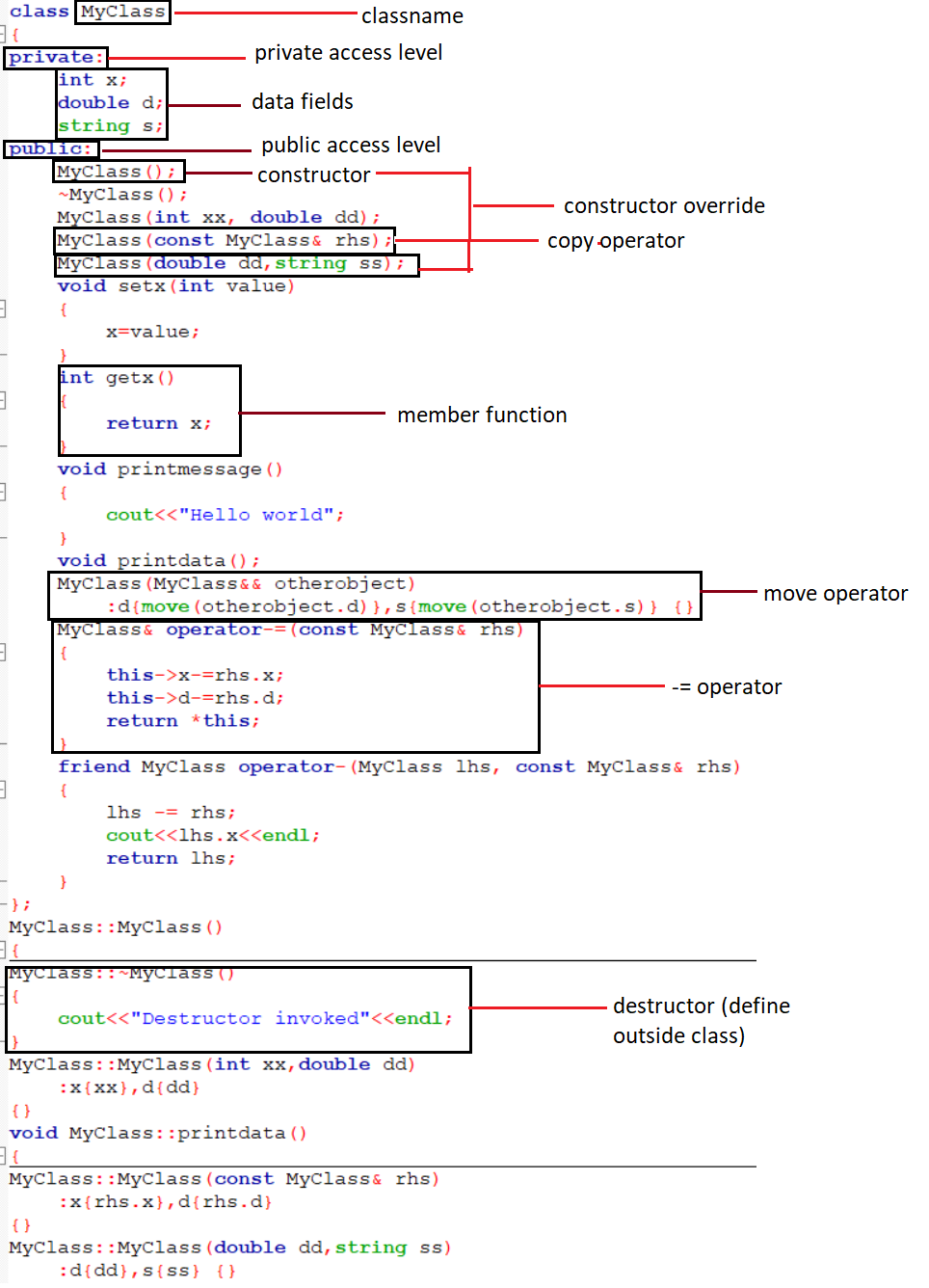
CHAPTER 21: SCOPE AND LIFETIME

* 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();  } |  |

CHAPTER 27: STATIC

* Static storage duration: The memory space for static objects is allocated when the program starts and deallocated when the program ends
* Syntax: **static type var;**
* Static in class: variable not part of the object.
* Declare static function inside a class:

**public:**

**static void myfunction();**

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  void myfunction()  {  static int x = 0; // defined only the first time, skipped every other  // time  x++;  std::cout << x << '\n';  }  int main()  {  myfunction(); // x == 1  myfunction(); // x == 2  myfunction(); // x == 3  } |  |
| #include <iostream>  class MyClass  {  public:  static int x; // declare a static data member  };  int MyClass::x = 123; // define a static data member  int main()  {  MyClass::x = 456; // access a static data member  std::cout << "Static data member value is: " << MyClass::x;  } |  |
| #include <iostream>  class MyClass  {  public:  static void myfunction(); // declare a static member function  };  void MyClass::myfunction()  {  std::cout << "Hello World from a static member function.";  }  int main()  {  MyClass::myfunction(); // call a static member function  } |  |

CHAPTER 28: TEMPLATES

* Templates: support generic programming (means we can define a function or a class without worrying about what types it accepts)
* Define a template: **template <typename T>**
* Instantiate a function template: call a function by supplying a specific type name, surrounded by angle brackets
* Template can have more than one parameter: **template <typename T, typename U>**
* Class can have template.
* Template specialization: make template behave differently for a specific type. Syntax: prepend function/class with **template <>**

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  template <typename T>  void myfunction(T param)  {  std::cout << "The value of a parameter is: " << param<<std::endl;  }  int main()  {  myfunction<int>(123);  myfunction<double>(123.456);  myfunction<char>('A');  } |  |
| #include <iostream>  template <typename T, typename U>  void myfunction(T t, U u)  {  std::cout << "The first parameter is: " << t << '\n';  std::cout << "The second parameter is: " << u << '\n';  }  int main()  {  int x = 123;  double d = 456.789;  myfunction<int, double>(x, d);  } |  |
| #include <iostream>  template <typename T>  class MyClass  {  private:  T x;  public:  MyClass(T xx)  :x{ xx }  {  }  T getvalue()  {  return x;  }  };  int main()  {  MyClass<int> o{ 123 };  std::cout << "The value of x is: " << o.getvalue() << '\n';  MyClass<double> o2{ 456.789 };  std::cout << "The value of x is: " << o2.getvalue() << '\n';  } |  |
| #include <iostream>  template <typename T>  class MyClass  {  private:  T x;  public:  MyClass(T xx);  };  template <typename T>  MyClass<T>::MyClass(T xx)  : x{xx}  {  std::cout << "Constructor invoked. The value of x is: " << x << '\n';  }  int main()  {  MyClass<int> o{ 123 };  MyClass<double> o2{ 456.789 };  } |  |
| #include <iostream>  template <typename T>  void myfunction(T arg)  {  std::cout << "The value of an argument is: " << arg << '\n';  }  template <>  // the rest of our code  void myfunction(int arg)  {  std::cout << "This is a specialization int. The value is: " << arg <<  '\n';  }  int main()  {  myfunction<char>('A');  myfunction<double>(345.678);  myfunction<int>(123); // invokes specialization  } |  |

CHAPTER 29:

* Enumeration (enum): a type whose values are user-defined named constants called enumerators
* 2 types:
  + Unscoped: enumerators leak into outside scope
    - Syntax: **enum name{};**
  + Scoped: enumerators don’t leak into outer scope, are not implicittly convertible to other types.
    - Syntax: **enum class name{};**
    - Access enumerator value: prepend enumerator with enum name and scope resolution operator ::
    - Define underlying type for scoped enum: **enum class name:char{};**

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  using namespace std;  enum class MyEnum  {  myfirstvalue,  mysecondvalue,  mythirdvalue  };  int main()  {  MyEnum myenum=MyEnum::myfirstvalue;  int a=static\_cast<int>(myenum);  cout<<a;  return 0;  } |  |

CHAPTER 31: ORGANIZING CODE

* Headers (.h or .hpp): source codes files, store various declarations
* Source files: store definitions and main program
  + Use standard library header: **#include <name>**
  + Use user-defined header: **#include “name”**
* Header guards: prevent header being included more than once.
  + Syntax: surround the header code with

**#ifndef HEADERNAME**

**#define HEADERNAME**

**…**

**#endif**

* Namespace: another way to group parts of C++ code.
  + Declare namespace: **namespace Name{}**
  + To refer object inside namespace, use **name::object**

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  using namespace std;  namespace abc  {  int aaa;  int bbb;  int comp(int x,int y)  {  return x<y;  }  }  using namespace abc;  int main()  {  int aaa=0;  abc::aaa=10;  abc::bbb=11;  int x=10,y=11;  cout<<abc::comp(x,y);  return 0;  } |  |

CHAPTER 33: CONVERSIONS

* Conversion: convert type to other types.
* Implicit conversions: true for all built in types.
  + Narrowing conversions: information loss during conversions.
  + Integral promotion: smaller integer (char or short) gets promoted/converted to integers in arithmetic operations.
  + Any built-in type can be converted to boolean (value=0 turn is false, else is true)
  + Boolean type can be converted to int (false is 0, true is 1)
  + Void pointer: cannot be dereferenced. To access object pointed to by void pointer, cast the void pointer to other type by function **static\_cast<type\*>**
  + Arrays are implicitly convertible to pointers
* Explicit conversions:
  + Function **static\_cast<cast\_type>(value)** (compile-time conversions)
  + **Dynamic\_cast**: converts pointer of base class to pointers to derived class and vice versa.
  + **Reintrepre\_cast**: dangerous, should not use.

|  |  |
| --- | --- |
| Command | Output |
| #include <iostream>  class MyBaseClass  {  public:  virtual ~MyBaseClass() {}  };  class MyDerivedClass : public MyBaseClass {};  int main()  {  MyBaseClass\* base = new MyDerivedClass;  MyDerivedClass\* derived = new MyDerivedClass;  // base to derived  if (dynamic\_cast<MyDerivedClass\*>(base))  {  std::cout << "OK.\n";  }  else  {  std::cout << "Not convertible.\n";  }  if (dynamic\_cast<MyBaseClass\*>(derived))  {  std::cout << "OK.\n";  }  else  {  std::cout << "Not convertible.\n";  }  delete base;  delete derived;  } |  |
| #include <iostream>  class MyBaseClass  {  public:  virtual ~MyBaseClass() {}  };  class MyDerivedClass : public MyBaseClass {};  class MyUnrelatedClass {};  int main()  {  MyBaseClass\* base = new MyDerivedClass;  MyDerivedClass\* derived = new MyDerivedClass;  MyUnrelatedClass\* unrelated = new MyUnrelatedClass;  // base to derived  if (dynamic\_cast<MyUnrelatedClass\*>(base))  {  std::cout << "OK.\n";  }  else  {  std::cout << "Not convertible.\n";  }  // derived to base  if (dynamic\_cast<MyUnrelatedClass\*>(derived))  {  std::cout << "OK.\n";  }  else  {  std::cout << "Not convertible.\n";  }  delete base;  delete derived;  delete unrelated;  } |  |