C++ Basics

Ruijun Ni

October 2, 2022

1 C++ Basics

1.1 Input/Output

To get an input value and put it into variable x: $cin >> x$; where *cin* is short for *characters in*. Moreover, we can also get all remaining text on the current input line and put it into x with $getline(cin, x)$. To output text: $cout <<$ "*desiredtext*";. *cout* is short for *characters out*. Additionally, $cout << endl$ starts a new line. Below is an example.

```
#include <iostream>
using namespace std;

int main() {
   int wage;
   cin >> wage;
   cout << "Salary is " << wage * 40 * 52 << endl;
   return 0;
}
```

In C++ input and output are performed in the form of a sequence of bytes called **streams**. To enable the program to get input and put output, the first two lines of code should be added at the top of the program to include the necessary library.

1.2 Switch statements

Switch statement is an alternative of if-else statements. The program will go from the first case to the last case until it reaches a "break" or reaches the end of the statement. Below is an example. If the condition satisfies, that case's statement would be executed, and the program will jump to the end. If no case matches, the default case statement will be executed.

```
switch (a) {
  case 0:
     // Print "zero"
     break;
  case 1:
     // Print "one"
     break;

  case 2:
     // Print "two"
     break;

   default:
     // Print "unknown"
     break;
}
```

1.3 Character operations

Character operations can be done with functions in the c++ library *cctype*. Functions include **isalpha(c)**, **isdigit(c)**, **isspace(c)**, **isupper(c)**, **islower(c)**, etc. Below is an example. We included the cctype libary and checked if the two characters are alpha with **isalpha(c)** function.

```
#include <iostream>
#include <cctype>
using namespace std;

int main() {
   char let0;
   char let1;

   cout << "Enter a two-letter state abbreviation: ";
   cin >> let0;
   cin >> let1;

   if ( ! (isalpha(let0) && isalpha(let1)) ) {
      cout << "Error: Both are not letters." << endl;
   }
   else {
      let0 = toupper(let0);
      let1 = toupper(let1);
      cout << "Capitalized: " << let0 << let1 << endl;
   }

   return 0;
}
```

1.4 Floating-point comparison

Floating-point should not be compared using == because some numbers might not be exact due to the bit length limitation. To solve this problem, we can use $fabs(x - y) < 0.001$ to compare two floating point numbers. $fabs(x)$ is to compute the absolute value of x. Below is a good example,

```
numMeters = 0.7;
numMeters = numMeters - 0.4;
numMeters = numMeters - 0.3;

// numMeters expected to be 0,
// but is actually -0.0000000000000000555112
if (numMeters == 0.0) {
   // Equals 0.
}
```

1.5 Enumerations

An **enumeration type (enum)** is a user-defined data type that declares a name for a new type and possible values for that type which makes the program easy to read and maintain. For example, in the GroceryItem type, there are there values.

```
int main() {
   // declares a new enumeration type named LightState
   enum GroceryItem {GR_APPLES, GR_BANANAS, GR_JUICE, GR_WATER};
   // declares a new variable lightVal of that type
   GroceryItem userItem;
   userItem = GR_APPLES;
   if (userItem == GR_APPLES || userItem == GR_BANANAS) {
      cout << "Fruit" << endl;
   }
   return 0;
}
```

2 Arrays, Vectors, and Classes

2.1 Vectors

We declare a vector via:

```
     vector<dataType> vectorName(numElements);
```

We can access the ith element in the vector using myVector.at(i), but please pay attention to the out of boundary error when choosing the wrong index. A vector's **size()** function returns the number of vector elements, and we can iterate through vectors using loops. Below is an example. Also keep in mind that we need to include vector at the top of the file.

```
#include <vector>
int main() {
     vector<int> myVector(3);  // declaration
     myVector.at(0) = 122;
     myVector.at(1) = 119;
```

```
    myVector.at(2) = 117;

    // Iterating through myVector
    for (i = 0; i < myVector.size(); ++i) {
       // Loop body accessing myVector.at(i)
    }

    return 0;
}
```

Vector can be initialized by several ways. Firstly, we can create a vector with three elements, each with value -1 via:

```
vector<int> myVector(3, -1);
```

We can also initialize each vector element with different values by specifying the initial values in braces {} separated by commas:

```
vector<int> myVector = {5, 7, 11};
```

push_back() function is used to append a new element to the end of a vector. **pop_back()** function is to remove the last element. **back()** function is to return the last element without changing the vector. Below is an example of push_back(), pop_back(), and back().

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main() {
   vector<string> groceryList; // Vector storing shopping list
   string groceryItem;         // Individual grocery items
   string userCmd;             // User input

   // Prompt user to populate shopping list
   cout << "Enter grocery items or type done." << endl;
   cin >> groceryItem;
   while (groceryItem != "done") {
      groceryList.push_back(groceryItem);
      cin >> groceryItem;
   }

   // Display shopping list
   cout << endl << "Enter any key for next item." << endl;
   while (groceryList.size() > 0) {
      groceryItem = groceryList.back();
      groceryList.pop_back();
      cout << groceryItem << "   ";
      cin >> userCmd;
   }
   cout << endl << "Done shopping." << endl;

   return 0;
}
```

C++ supports **Arrays** and **Vectors** for ordered lists. Arrays are declared as **int myList[10]**, accessed as **myList[i]**. Vectors are safer than arrays, so using vectors is a good practice.

2.2 User-defined function

Functions are a grouping of predefined statements created to reduce redundancy, improve readability, and enable modular and incremental program development. Functions include both function definitions and function calls.

2.2.1 Pass by reference

Variables can be **passed by value** or **passed by reference**. In Pass by value, parameters passed as an arguments create its own copy, so any changes made inside the function is made to the copied value not to the original value. In Pass by reference, parameters passed as an arguments does not create its own copy, it refers to the original value so changes made inside function affect the original value.

A **reference** is a variable type that refers to another variable. **int& maxValRef = usrInput3;** declares and initializes the reference variable. Below is another example of pass by reference:

```
void CoordTransform(int& x1, int& y1, int& xValNew, int& yValNew) {
     xValNew = (x1 + 1) * 2;
     yValNew = (y1 + 1) * 2;
}
```

2.2.2 Function name overloading

Function name overloading means a program has multiple functions with the same name but different number or types of parameters. For example, there are two PrintDate() functions, and the compiler will determine which function to call based on the argument types.

```
#include <iostream>
#include <string>
using namespace std;

void PrintDate(int currDay, int currMonth, int currYear) {
   cout << currMonth << "/" << currDay << "/" << currYear;
}

void PrintDate(int currDay, string currMonth, int currYear) {
   cout << currMonth << "␣" << currDay << ",␣" << currYear;
}

int main() {

   PrintDate(30, 7, 2012);
   cout << endl;
```

```
      PrintDate(30, "July", 2012);
      cout << endl;

      return 0;
   }
```

2.3 Objects and Classes

The class construct defines a new type that groups data and functions to form an object. The **member access operator**, ".", is to invoke a function on an object. **Private data members** are variables that member functions can access but class users cannot. Below is a complete class definition, and use of that class. Also in this example, we overload the default constructor.

```
#include <iostream>
#include <string>
using namespace std;

class Restaurant {                                // Info about a restaurant
   public:
      void SetName(string restaurantName);   // Sets the restaurant's name
      void SetRating(int userRating);        // Sets the rating (1-5, with 5
         best)
      void Print();                          // Prints name and rating on one
         line

   private:
      string name;
      int rating;
};

Restaurant::Restaurant() {   // Default constructor
   name = "NoName";          // Default name: NoName indicates name was not set
   rating = -1;              // Default rating: -1 indicates rating was not set
}

// Another constructor
Restaurant::Restaurant(string initName, int initRating) {
   name = initName;
   rating = initRating;
}

// Sets the restaurant's name
void Restaurant::SetName(string restaurantName) {
   name = restaurantName;
}

// Sets the rating (1-5, with 5 best)
void Restaurant::SetRating(int userRating) {
   rating = userRating;
}

// Prints name and rating on one line
void Restaurant::Print() {
   cout << name << " -- " << rating << endl;
}

int main() {
   Restaurant favLunchPlace;
   Restaurant favDinnerPlace;

   favLunchPlace.SetName("Central Deli");
   favLunchPlace.SetRating(4);

   favDinnerPlace.SetName("Friends Cafe");
   favDinnerPlace.SetRating(5);

   cout << "My favorite restaurants: " << endl;
   favLunchPlace.Print();
   favDinnerPlace.Print();

   return 0;
}
```

While an object's member function is called using the syntax object.Function(), we can also access the implicitly-passed object pointer via ***this***. For example,

```
void ElapsedTime::SetTime(int timeHr, int timeMin) {
   this->hours = timeHr;
   this->minutes = timeMin;
}
```

3 Pointers, Memory Allocation and Inheritance

3.1 Pointers

3.1.1 Definition

Pointer is a variable that holds a memory address. It has a **data type**. For example, a double pointer holds an address of a double. A pointer is **declared with a *** before the pointer's name; A pointer is **initialized with another variable's address that is obtained by *reference operator* (&)**. For example,

```
int* valPointer = &someInt;
```

3.1.2 Dereferencing a pointer

The **dereference operator (*)** is put behind a pointer variable's name **to get the <u>data value</u>** where the pointer variable points. For exaple,

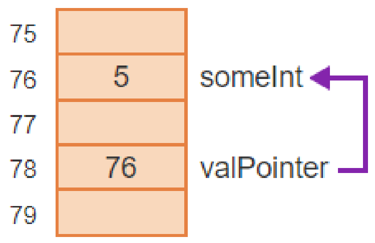


Figure 1: int* valPointer = someInt; (from zybook 8.2 Pointer basics)

```
cout << *valPointer;
```

Also, a null pointer is ***nullptr***.

3.1.3 Operators: new, delete, and -¿

The **new operator** allocates memory and returns a pointer to the allocated memory.

```
Point* sample = new Point;
```

The **delete operator** deallocates or frees memory.

```
delete pointerVariable;
```

We can also use *delete[] operator* to free an array. For example:

```
int main() {
   // Allocate points
   int pointCount = 4;
   Point* manyPoints = new Point[pointCount];

   // Display each point
   for (int i = 0; i < pointCount; ++i)
      manyPoints[i].Print();

   // Free all points with one delete
   delete[] manyPoints;

   return 0;
}
```

3.1.4 String functions with pointers

The C string library should be included via: $\#include < cstring >$. Strings pass as $char*$. There are three main string search functions:

1. $strchr(sourceStr, searchChar)$ returns pointer to first occurrence. Else returns null.
2. $strrchr(sourceStr, searchChar)$ returns pointer to last occurrence. Else returns null.
3. $strstr(str1, str2)$ returns a char pointer pointing to first occurrence of str2 within str1. Else, it returns null if str2 doesn't exist in str1.

Below is a good example about string and string searching.

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   const int MAX_USER_INPUT = 100;         // Max input size
   char userInput[MAX_USER_INPUT];         // User defined string
   char* stringPos = nullptr;              // Index into string

   // Prompt user for input
   cout << "Enter a line of text: ";
   cin.getline(userInput, MAX_USER_INPUT);

   // Locate exclamation point, replace with period
   stringPos = strchr(userInput, '!');

   if (stringPos != nullptr) {
      *stringPos = '.';
   }

   // Locate "Boo" replace with "---"
   stringPos = strstr(userInput, "Boo");

   if (stringPos != nullptr) {
      strncpy(stringPos, "---", 3);
   }

   // Output modified string
   cout << "Censored: " << userInput << endl;

   return 0;
}
```

3.2 Dynamic memory allocation

Figure 2 is a summary of constructors and destructors in c++. Destructor, copy constructor, and copy assignment operator are the so-called rule of three.

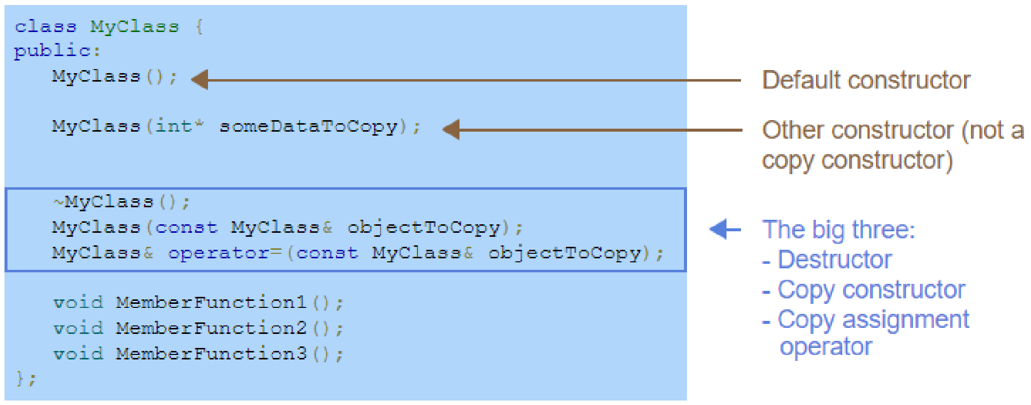
3.2.1 Destructors

Destructors is called automatically when a variable of that class is destroyed.

For example, without a destructor, deleting list1 by the code $deletelist1$ only deallocates the list1 object, but not the 2 nodes.

Implementation of a destructor: We can declare the LinkedList class destructors with $\sim LinkedList()$;.

Here is an example of the LinkedList class destructor, called when the list is deleted, frees all nodes. The object's destructor is called automatically when the object goes out of



```
class MyClass {
public:
   MyClass();

   MyClass(int* someDataToCopy);


   ~MyClass();
   MyClass(const MyClass& objectToCopy);
   MyClass& operator=(const MyClass& objectToCopy);

   void MemberFunction1();
   void MemberFunction2();
   void MemberFunction3();
};
```

Default constructor
Other constructor (not a copy constructor)
The big three:
- Destructor
- Copy constructor
- Copy assignment operator

Figure 2: Constructors (from zybook 8.11 Rule of three)

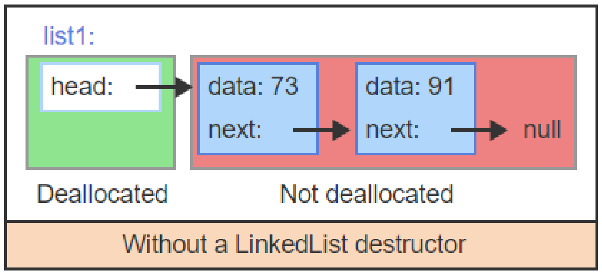


Figure 3: Without a destructor (from zybook 8.7 Destructors)

scope.

```
#include <iostream>
using namespace std;

// ... LinkedListNode class omitted ...
// ... LinkedList class omitted, except for destructor, below ...

LinkedList::~LinkedList() {
   cout << "In LinkedList destructor" << endl;

   // The destructor deletes each node in the linked list
   while (head) {
      LinkedListNode* next = head->next;
      delete head;
      head = next;
  }
}

int main() {
   // Create a linked list
   LinkedList* list = new LinkedList;
   for (int i = 1; i <= 5; ++i)
      list->Prepend(i * 10);

   // Free the linked list.
   // The LinkedList class destructor frees each node.
   delete list;

   return 0;
}
```

If we fail to free allocated memory, a **memory leak** may happen.

3.2.2 Copy constructor

Copy constructor is a constructor that can be called with a single pass by reference argument.

```
class MyClass {
  public:
      ...
      MyClass(const MyClass& origObject);
      ...
};
```

3.2.3 Copy assignment operator

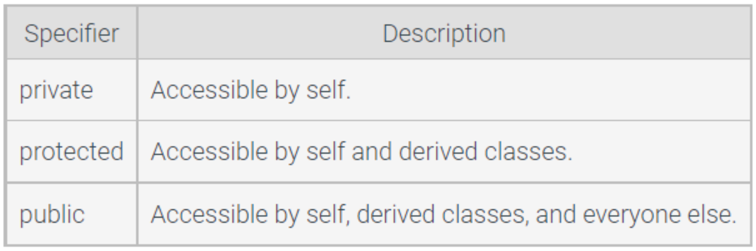
The **copy assignment operator** overloads the built-in function "operator=".For example,

```
MyClass& operator=(const MyClass& objToCopy);
```

3.3 Inheritance and Polymorphism

Inheritance means that the **derived class** (subclass) **inherits the properties** of the **base class** (super class).

There are three kinds of **access specifiers**, **private**, **protected**, and **public**. *Protected* allows access by self and derived classes.



Specifier	Description
private	Accessible by self.
protected	Accessible by self and derived classes.
public	Accessible by self, derived classes, and everyone else.

Figure 4: Access specifiers (from zybook 10.2 Access by members of derived classes)

Polymorphism allows subclasses to **override** functionalities of the body class. We need to make the member function in the base class **virtual** to enable runtime polymorphism.

A **pure virtual function** is a virtual function that has no definition in the base class, and all derived classes must override it. The function should always be assigned to 0. An **abstract class** is a class that **cannot be instantiated as an object**. Class with one or more pure virtual functions is abstract. For example, the code snippet below is wrong because abstract class can not be instantiated.

```
class Shape {
  protected:
    Point position;

  public:
    virtual ~Shape() { }
    virtual double ComputeArea() const = 0;  // pure virtual function
    // some functions...
};

int main(int argc, const char* argv[]) {
  Shape* shape = new Shape();
  ...
}
```