C++ for Java Programmers

Hans Dulimarta

Contents

Listings

List of Tables

List of Figures

Introduction

The design of the Java programming language was significantly influenced by the C++ programming language. Designed to look much like C++, Java was expected to be a familiar language. The Java designer team thought that C++ is a complex programming and Java should be less complex. They decided to remove some of the C++ features that they believe "difficult" to understand.

While Java received so many revisions since 1995 until the present time (annually updated up to version 1.2 and biannually updated up to Java 6, with a 5-year hiatus prior to the release of Java 7), C++ received only four major revisions (1985, 1989, 1998, 2011). Among these revisions, the latest release of the C++ standard in 2011 (dubbed C++11) includes a great number of enhancement to the language. A program written in C++11 uses new styles for better performance. In fact, Stroustrup (the inventor of C++) claims:

> ... you can do better with modern C++; if you stick to older styles, you will be writing lower-quality and worse-performing code.

This manuscript is prepared with the main intention that Java programmers are able to uncover the missing features of C++ that are not available in Java and use them in real-world programming tasks. It is my hope that after reading this manuscript, readers with Java background can experience a smoother transition to use new C++11 features. Although the C++(11) features are described in comparison with their counterpart in Java, readers should write their C++ code using patterns and styles specific to C++11 so their code does not look like a C++-ized ("C-plus-plus-ized") variant of the original Java program.

C++ has been used for developing many systems, among the most notable ones are: Amazon server infrastructure, Google Search Engine, Google ChromeOS, and the Java Virtual Machine (JVM). It is interesting to know that the Java Virtual Machine (a software that interprets and runs Java programs) is actually written in C++.[1]

1 Execution Environment

The Java execution environment uses the "Compile-Load-Interpret" paradigm,

- Each Java file (`.java`) is individually compiled into a `.class` file.
- At runtime, the Java Virtual Machine (JVM) loads these classes (on-demand) to memory and interprets the Java bytecode.
- Classes loaded to memory can be either user-defined classes or classes from the Java run-time libraries.

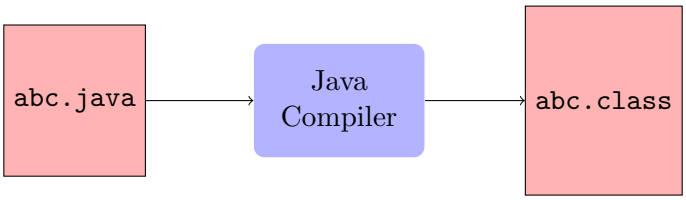

Figure 1: Java Compilation

While a Java program runs on top of the Java Virtual Machine, the JVM itself runs natively on top of the hardware. The C++ execution environment is different. Every C++ program runs natively on

[1]Source code of OpenJDK is available at `http://hg.openjdk.java.net/jdk8/jdk8/hotspot/`

hardware. In order to run a binary executable a C++ code goes through a 4-step process: "Preprocess-Compile-Link-Run".

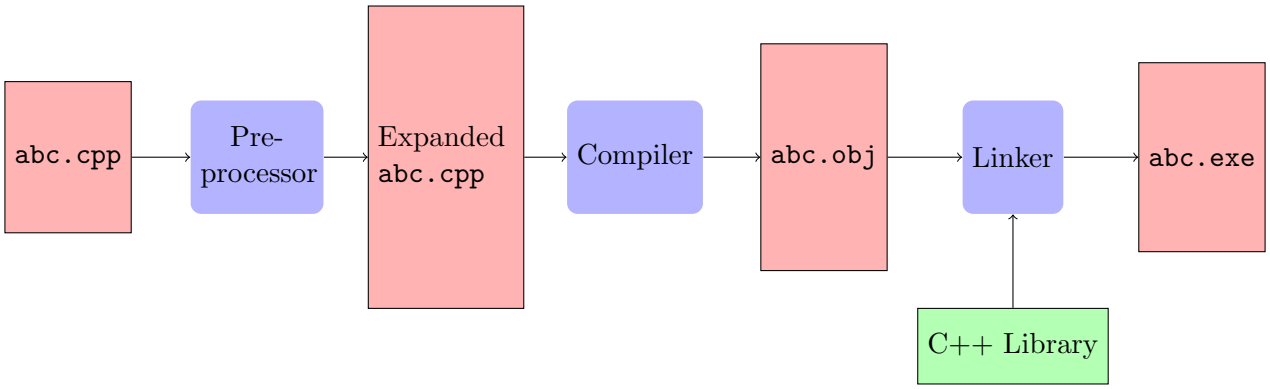

Figure 2: C++ Compilation Phases

- During the **preprocessing stage**, all the pound directives (`#include`, `#if`, `#endif`, ...) in a C++ source file (.cc or .cpp) are resolved to produce an "expanded" C++ source file, i.e., after all the files referred by all the `#include` directives are copied into the original source code
- During the **compilation stage**, each expanded source file is individually compiled into an object file (`.o` on Mac/Linux or `.obj` on Windows).
- During the **linking state** these object files are then combined together with the C++ libraries to make the binary executable of the program.
- Finally, the operating system then loads and runs the binary executable code. The code runs directly on an a real CPU (unlike Java bytecodes that are interpreted and run on a virtual machine)

1.1 Memory Layout

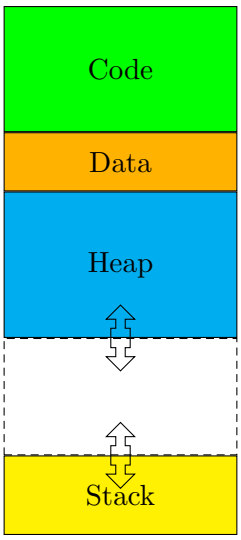

Figure 3: C++ Runtime Memory Layout

When a C++ compiler produces a binary executable, it generates **code** and **data segments** to place the instructions and data respectively. The size of these two segments is fixed and determined by the size of the code and **global data** used by the program. The code segment begins at the lowest memory address and the data segment begins immediately where the code segment ends. When the program runs, the operating system loads these two segments to memory. At runtime the program may create new data items. To accommodate these extra data requirements, the system reserves two additional sections: **heap** and **stack segments**. The heap segment is allocated by the operating system after the data segment, and the stack segment is allocated at starting at the highest address. Unlike **code** and **data segments** whose size is fixed, the heap and stack segments may increase/decrease, and they grow towards each other:

- The stack segment grows (towards low memory address) every time the program invokes a function. The longer the chain of calls, the higher the stack grows. It shrinks (towards high memory address) when the functions return. Function parameters and function local variables are allocated on the stack segment. The amount of stack growth during a function invocation is determined by the number of local variables and parameters of the function.

- The heap segment holds memory area for C++ object explicitly created by invoking `new`. This memory area will be deallocated from the heap segment whenever the program explicitly destroy the object by invoking `delete`. [2]

```
1  #include <string>
2  void beginAphelion() {
3      std::string moon = "Titan";
4      /* more code here */
5  }
6
7  int main() {
8      double distance = 0.0;
9      printf ("Begin here\n");
10     beginAphelion();
11     return 0;
12 }
```

The program to the left shows a short main and another global function. The string

`"Begin Here\n"`

and

`"Titan"`

are allocated by the compiler in the data section.

- When the program is about to begin execution, the heap and stack size is zero.
- When `main` becomes active, the **stack will grow** by 8 bytes (number of bytes for `double`) to allocate the storage for the local variable `distance` at line 8
- When `printf` is invoked, the stack will grow as much as the local variables and parameters used internally by `print`. After printf returns the stack will shrink to its prior height.
- When `main` invokes `beginAphelion()` and execution continues inside the function, **the stack grows** further by at least 5 bytes to reserve the space for the string object `moon` holding "Titan" (line 3).

1.2 Online Resources

Java programmers can use the official online documentation at Oracle.com. Although C++ was created by Bjarne Stroustrup when he was employed by AT&T Bell Labs, but they don't publish an official Programmer Reference manual of any kind. The best online references so far are

- `http://cppreference.com`
- `http://www.josuttis.com/cpp.html`
- `http://cplusplus.com`

[2] Unlike Java that provides Garbage Collector, C++ does not provide automatic garbage collectors, memory cleanup is the programmer's responsibility

2 C++ Class Structure

2.1 Access Modifiers

In Java, the access modifiers `public`, `private`, and `protected` are used before every declaration of an identifier (except for identifiers with package visibility). In C++ these keywords **begin a section** and identifiers declared within a given section inherit the section's access modifier. Compare the examples in Listing 1.

Java

```java
// Filename: Car.java
public class Car {
  private static final double
    MPG = 22.3;
  private static final double
    TANK_CAPACITY = 12.0;
  private String VIN;
  private double mileage;
  private double fuelAmount;

  public Car(String id) {
    VIN = id;
    mileage = 0; /* in miles */
    fuelAmount = TANK_CAPACITY;
  }

  public void drive(double distance) {
    mileage += distance;
    fuelAmount -= distance / MPG;
  }

  public void addFuel(double gasAmount) {
    fuelAmount += gasAmount;
  }

  public double getOdometer() {
    return mileage;
  }

  public double checkFuelAmount() {
    return fuelAmount;
  }
}
```

C++

```cpp
// Filename: Car.cpp (or Car.cc)
#include <string>

class Car {
private:
  const double MPG = 22.3;
  const double TANK_CAPACITY = 12;
  std::string VIN;
  double mileage;
  double fuelAmount;

public:
  Car (std::string id) {
    VIN = id;
    mileage = 0;
    fuelAmount = TANK_CAPACITY;
  }

  void drive (double distance) {
    mileage += distance;
    fuelAmount -= distance / MPG;
  }

  void addFuel (double gasAmount) {
    fuelAmount += gasAmount;
  }

  double checkFuelAmount() {
    return fuelAmount;
  }
};     // <== SEMICOLON REQUIRED!!!!
```

Listing 1: Access Modifiers: `private` and `public`

2.2 Functions

Unlike Java where every methods must be declared inside a class, C++ allows functions to be declared either inside or outside a class. When a function is declared inside a class declaration, it is called a **member function**, otherwise it is a **non-member** or **global function**. All the functions shown in Listing 1, are member functions. Any C++ program is required to define **at least one global function**, the `main()` function.

Java

```
class Java {
  private static final double
    MPG = 22.3;
  ...

  public Car(String id) {
    ...
  }

  ...
  /*- main() is a CLASS function -*/
  public static void main (String[] args) {
    /* more code here */
  }
}
```

C++

```
#include <string>
class Car {
private:
  const double MPG = 22.3;


public:
  Car (std::string id) {
    ...
  }

  /* more code here */

};

/*- main() is a GLOBAL function -*/
int main (int argc, char* argv[]) {
   /* more code here */
   return 0;
}
```

Listing 2: The `main()` function in C++ is global (not a member of any class). The first parameter `argc` represents the "argument count", it is equivalent to Java's `args.length`. The second parameter `argv` is an array of "argument values".

2.2.1 Using `const` for Non-Mutable Functions

In addition to using the `const` keyword as shown in Listing 1 for declaring `TANK_CAPACITY`, C++ also uses `const` for declaring non-mutable member functions, i.e., member functions that do not alter their instance variables. Refer to the `Die` class example in Listing 3. Out of the four member functions of `Die`, the two non-mutable functions are `getValue()` and `isEven()`, therefore these two function should be declared as `const` member function.

"Const-correctness" is a concept that must be understood by C++ programmers who plan to use the C++ standard (template) library (STL). An attempt to invoke a mutable function from a const object will be flagged as a compile error.

2.3 Interface vs. Implementation

In Java, interfaces and classes are written into a `.java` file. Without any specific file naming scheme, Java programmers will not be able to distinguish an interface from a class implementing the interface only from the name of the file. Java distinguishes these two types of module by two different keywords used in a declaration `interface` and `class`.

In C++ the design of a class can be split into two separate files. Its interface goes to a `.h` file and its implementation goes to a `.cpp` file.

- The interface contains the declaration of the class **instance variables** and **function prototypes** (function headers only)
- The implementation file defines the full definition of all the functions (function header and body) and static variables.

```
1  class Die {
2  private:
3      int value;
4  public:
5      Die() {
6      }
7
8      int getValue() const {   /* non mutable function */
9          return value;
10     }
11
12     void setValue (int v) {
13         value = v;
14     }
15
16     void roll() {
17         // generate a random
18         // number 1-6
19         value = (rand() % 6) + 1;
20     }
21
22     bool isEven() const {   /* non mutable function */
23         return (value % 2) == 0;
24     }
25 };
```

Listing 3: `Die` class declaration with non-mutable member functions.

The split design of the `Car` class is shown in Listing 4. When the function definitions are written separately into the implementation file, each function name (and instance variables) must be **prefixed with the class name using the scope operator** `::`. Some examples are highlighted in Table 1.

When working with multiple classes, there may be dependency among these classes. In a C++ program these dependencies are evident from the `#include` directives. It is possible that during parsing the directives, the preprocessor encounters several inclusion requests of the same file. For instance, consider the following code snippets:

```
1  // in main.cpp
2  #include "Car.h"
3  #include "Driver.h"
4
5  int main () {
6     ...
7     return 0;
8  }
```

```
1  // in Driver.h
2  #include <cstdlib>
3  #include "Car.h"
4  class Driver {
5     ...
6  };
```

```
// in Car.h
#ifndef GVSU_CAR_H
#define GVSU_CAR_H

#include <string>
class Car {
   ...
};

#endif
```

A careful inspection shows that `main.cpp` gets two copies of `Car.h`: the first copy is by direct `#include` at line 2 and the second copy is through an *indirect* `#include` via `Driver.h` (line 3). In order to avoid

`const double MPG = 22.3;`	`const double Car::MPG = 22.3;`
`Car (std::string id) { ...}`	`Car::Car (std::string id) { ...}`
`void drive (double distance) { ...}`	`void Car::drive (double distance) { ...}`

Table 1: Scope operators in function and variable declarations are required when a class implementation is separate from its interface

"multiple declarations" caused by multiple inclusions of header files, we protect the content of each header file using `#ifndef`, `#define`, and `#endif` directives.

During the first `#include "Car.h"` by `main.cpp`, the symbol `GVSU_CAR_H` has not been defined yet, so the pre-processor will copy the Car class declaration into `main.cpp` and `GVSU_CAR_H` is now defined. During the second attempt to include `Car.h` (via `Driver.h`, the symbol `GVSU_CAR_H` has been defined, and therefore the rest of `Car.h` will be skipped, thus avoiding multiple declarations of the same class.

```cpp
#include <string>

class Car {
private:
  const double MPG = 22.3;
  const double TANK_CAPACITY = 12;
  std::string VIN;
  double mileage;
  double fuelAmount;

public:
  Car (std::string id) {
    VIN = id;
    mileage = 0;
    fuelAmount = TANK_CAPACITY;
  }

  void drive (double distance) {
    mileage += distance;
    fuelAmount -= distance / MPG;
  }

  void addFuel (double gasAmount) {
    fuelAmount += gasAmount;
  }

  double checkFuelAmount() {
    return fuelAmount;
  }
};     // <== SEMICOLON REQUIRED!!!!
```

```cpp
// Filename: Car.h
#ifndef GVSU_CAR_H
#define GVSU_CAR_H
#include <string>

class Car {
private:
  // initial value of these two
  // constants is set in Car.cpp
  static const double MPG;
  static const double TANK_CAPACITY;
  std::string VIN;
  double mileage;
  double fuelAmount;

public:
  Car (std::string id);
  void drive (double distance);
  void addFuel (double gasAmount);
  double checkFuelAmount() const;
};
#endif
```

```cpp
// Filename: Car.cpp
#include "Car02.h"

const double Car::MPG = 22.3;
const double Car::TANK_CAPACITY = 12;

Car::Car (std::string id) {
    VIN = id;
    mileage = 0;
    fuelAmount = TANK_CAPACITY;
}

void Car::drive (double distance) {
    mileage += distance;
    fuelAmount -= distance / MPG;
}

void Car::addFuel (double gasAmount) {
    fuelAmount += gasAmount;
}

double Car::checkFuelAmount() const {
    return fuelAmount;
}
```

Listing 4: Separating the `Car` interface from its implementation

2.4 Java References and C++ Pointers

In Java, objects are created only by invoking the right constructor using the `new` operator. Thus, object instantiation in Java is explicit.

```
1  /* Java */
2  private void automotiveTask() {
3    Car his, hers;   /* two reference variables, ZERO object */
4
5    his = new Car ("1F.......");  /* now one object created */
6    hers = his; /* they both SHARE one car */
7
8    his.addFuel (5.0);
9    hers.drive (32.6);
10 }
```

C++ provides two different ways of creating objects (refer to Figure 4):

- Heap objects: created **explicitly** by the programmer by invoking `new`. These objects are allocated inside the heap segment. (Line 6 in the C++ example in Figure 4). These objects are destroyed explicitly by the programmer by invoking `delete` (at line 12).
- Automatic objects: created **automatically** by the runtime system when **the block** where the object is declared becomes active. (Line 4 in the C++ example in Figure 4). These objects are automatically destroyed when the block becomes out of scope, so the `hers` object is automatically destroyed at line 14.

During the entire lifetime of the function `automativeTask()` in Figure 4, we find **two local variables** and **two objects**:

- The local variable `his` is a 4-byte (or 8-byte) pointer allocated on the stack segment[3]
- The local variable `hers` is a multi-byte object whose size depends on the data requirement of the `Car` class. For the sake of disambiguation we will assume that a `Car` object takes 628 bytes of memory. This object is also allocated on the **stack segment**, and the stack goes up by 628 bytes. This object will be **automatically removed from the stack** when the function terminates (at line 12)
- The `Car` object instantiated at line 6 is allocated 628 bytes in the **heap segment**, but the pointer `his` itself is allocated 4 (or 8) bytes on the **stack segment**.

2.4.1 Memory Leaks

It is important to understand that, unlike Java, C++ does not have automatic garbage collector. **Without an explicit call to `delete` at line 12**, the function will suffer from **memory leak**. The chunk of memory allocated in the heap at line 6 will stay in the heap segment after the function terminates, but the pointer that held its address (`his`) has disappeared!

[3]The actual byte size depend on the CPU architecture. Pointers take 4 bytes on a 32-bit machine, and 8 bytes on a 64-bit machine

```
/* C++ */
void automotiveTask() {
  Car *his; /* he owns no car */
  Car hers("1F6DP5ED7B0858285"); /* she owns a Ford */

  his = new Car ("1HTSF30Y68E387874"); /* he now owns a Honda */

  his->addFuel (5.0);
  (*his).addFuel (2.5);
  /* total fuel added was 7.5 gallons */
  hers.drive (32.6);
  delete his;    // without this statement there is a memory leak

}
```

Figure 4: Heap objects and Automatic Objects in C++. `hers` is automatically destroyed when `automotiveTask()` terminates

De-reference/Indirection The example shown in Figure 4 also demonstrates the two different techniques of invoking a member function from an object pointer. The first technique uses the arrow operator[4]

```
his  ->  addFuel (5.0);
```

the alternate technique is to use the **indirection** operator to obtain an object from its pointer

```
(*his) . addFuel (2.5);
```

Using the second technique, the indirection operator (`*`) is first applied to a pointer to obtain its pointed data (object), then the member selection operator (`.`) is applied to that object to invoke the function.

2.5 C++ References

C pointers are often a stumbling block for many C programmers. In order to ease the burden of using pointers, C++ provides references. However, the concept is totally different from Java references. As shown in the previous section, a Java reference variable translates to a C++ pointer. **A C++ reference is not a pointer**, but it is an **alias** to an existing data; by defining an alias we are giving a new name to the existing data. It is important to understand that **no duplication** is being carried out when an alias is defined for the data. C++ references are declared using the `&` symbol.

A naïve example of using a reference is:

```
Car hers("1F6DP5ED7B0858285"); /* she owns a Ford */
Car blue_ford& = hers;

/* the following two operations apply to the same car */
blue_ford.addFuel (8.3);
hers.drive (50);
```

A more practical use of an alias is to selectively name an element of an array:

[4]the syntax is intentionally spaced out to emphasize the different components

```
int days_of_months[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int& feb = days_of_months[1];
```

A loop variable in a for-range loop to allow altering array elements. Assume the two for-range loops below apply to the following array:

```
float budgets[] {6123.50, 127.35, 874.12, 192.48, 61.03};
```

Copy of loop variable	For-range loop with reference

`/* failed attempt to increase budget 5% */`
`for (float b : budgets) {`
`  b *= 1.05;`
`}`
	`for (float& b : budgets) {`
	`  b *= 1.05;`
	`}`
	`/* budgets increased by 5% */`

Listing 5: For-range loop variables: without and with references

In each iteration of the left for-range loop, the loop variable `b` gets a **copy of one cell** in the array, and multiplication by 1.05 applies only the the copy (not the actual cell). In the right for-range **b** is an **alias**, therefore multiplying by 1.05 alters the array cells themselves.

2.5.1 Using References and Pointers as Function Parameters

Another technique for passing parameters to a C++ function is via pointers. There are two parts to using this technique:

- The invoked function declares its parameter(s) using a pointer syntax:

```
16 void drag_race (Car* c_ptr, Car& on wheels) {
17
18 }
```

- The caller is required to use the address-of operator (`&`) when supplying the actual argument:

```
31 Car white_honda("5K89172973123");
32 Car black_ford ("1Z7GF523KJD77");
33
34 /* addr-of is needed for the first arg, but not for the second */
35 drag_race (&white_honda, black_ford);  // pass address-of white_honda
```

Passing an object via its address (using pointer declaration on the function parameters) is as efficient as passing it by reference. Compare to pass-by-value, these two techniques do not suffer from **copying a large amount of bytes** to copy the actual object. Regardless of the actual size of the `Car` object (it may occupy megabytes of data), during the invocation of `drag_race` no more than 8 bytes of data are transferred from line 35 to line 16 in the above code snippet.

2.5.2 Pointers vs. References?

Looking at the syntax of C++ reference declaration, experience C programmers may begin to think that C++ references are a polished mechanism for avoiding pointers. This may be true to some degree, but there are conceptual differences between pointers and references (refer to the example in Code 6):

- Once an alias has been attached for a target, the alias will be "permanently" attached to that target and it cannot be used to alias a different target
- A pointer is more flexible. Anytime in our code, a pointer can be alter to hold the address of a different target

In C++, the `&` symbol can be used for three different purposes:

- As a bitwise AND
- As an alias declaration
- As an "address-of", to obtain the memory address of a known data. Used for this purpose, the `&` operator usually works in conjunction with the pointer ('`*`') operator

The last two may be confusing to new C++ programmers, but the guidelines to be used for distinguishing the last two cases are:

- When used in a declaration, `&` is an alias for an existing data (line 3 in Code 6)
- When used in a statement, `&` reads "address-off" the variable to its right (lines 9–10 in Code 6)

```
1  int diff[] {+2, +4, -3, +9, -2};
2
3  int& negative = diff[2];
4  int& negative = diff[4]; /* ERROR: duplicate */
5  &negative = diff[4];     /* ERROR: invalid expression */
6  negative = diff[4];      /* diff[2] = diff[4] */
7
8  int* data_ptr;
9  data_ptr = &diff[3];     /* keep the address of the fourth cell */
10 data_ptr = &diff[0];     /* keep the address of the first cell */
```

Listing 6: References vs. Pointers

In general, using C++ references is (syntactically) easier than C++ pointers. However, there are several limitations on using C++ references that do not apply to C++ pointers:

- When a reference variable is declared, it must be simultaneously initialized to the aliased data item

```
string city {"Philadelphia"};
string& philos;  // error: not init'd
string& philly = city;
char&   first = city[0];
```

A pointer can be declared without being initialized to any valid address

```
string city {"Philadelphia"};
string *place;  // OK: uninitialized
char   *front;
place = &city;  // init with addr-of city
front = &city[0];  // addr-of the first char
```

- Once a reference variable is declared **in a block** to be an alias of one data item, it cannot be "reassigned" as an alias for another data item **within the same block**

```
char& front = &city[0];
front = &city[4]; // replace 'P' with 'a'
char& front = &city[8];  // Error
```

(Within the same block) A pointer can be "reassigned" to point to a different data item

```
char *pch;
pch  = &city[0];
cout << *pch << endl;  // prints 'P'
pch = &city[5];
cout << *pch << endl;  // prints 'd'
```

- We can't declare an array of references
- A function that takes a reference parameter cannot be invoked with a "null" argument to indicate "No object"

We can declare an array of pointers
A function that takes a pointer parameter can be invoked with `nullptr` to indicate "No object"

2.6 Constructor, Destructor, Copy Constructor, and Move Constructor

In a Java class constructors are methods with a special role of initializing the class instance variables. In addition to the kind of constructors we know from Java, C++ introduces two new types of constructor:

- Copy constructor: a constructor that initializes an object by copying the current properties of a source object. For this reason, a copy constructor is required to take an input parameter of the source object. In order to avoid **indefinite copying** when the source object is passed into the copy constructor, the source object is pass by reference and as a read-only object. Therefore, a copy constructor has the following prototype:

```
MyClass (const MyClass& source) {
   /* put code here to COPY attributes from the "source" object
    * for initializing the attributes of "this" object
    */
}
```

- Move Constructor: a constructor that initializes an object by "stealing" the current properties of a source object. After the properties are "stolen", the source object should cease to exist. Move constructors are useful when the source object is a **temporary object** which a very short lifespan. Usually these are objects generated by the compiler to hold the intermediate result between operations. In order to indicate that the source object is a temporary object, the input parameter of the move constructor takes an different kind of reference (**rvalue** references) and it is passed as a non read-only object so its attributes can be stolen.

```
MyClass (MyClass&& source) {   /* double ampersand for rvalue ref */
   /* put code here to STEAL attributes from the "source" object
    * for initializing the attributes of "this" object.
    * Make the attributes of the "stolen source" disappear!
    */
}
```

Prior to the addition of the "move" feature in C++11, older C++ programs must rely on "copy-construct and destroy" steps to copy-construct a **temporary object**:

1. Copy the temporary object into the object being created
2. Destroy the temporary object (after copying is complete)

- A C++ destructor is a special function that executes when an object is being destroyed. What we write in a destructor are C++ statements to "undo"/"cleanup" any effects done by any of the constructors. When there is nothing to "undo", we are not required to provide a destructor for the class. A common example is when one of the constructors allocates memory, or open a file then the destructor is responsible for deallocating the memory and closing the file.

```
1  // Filename: Car.h
2  #ifndef GVSU_CAR_H
3  #define GVSU_CAR_H
4  #include <string>
5  using namespace std;
6  class Car {
7    /* more code here */
8
9  public:
10   Car (string& vin);                              /* constructor #1 */
11   Car (string& make, string& model, int year);  /* constructor #2 */
12
13   Car (const Car &  source_to_copy);          /* copy constructor */
14   Car (Car &&       temporary_source);        /* move constructor */
15
16   ~Car();                                          /* destructor */
17
18   /* more code here */
19 private:
20   string vin, model;
21   int year;
22   double mileage, fuelAmount;
23 };
24 #endif
```

Listing 7: Various forms of constructors

2.6.1 The Design of Constructors and Destructors

A class may declare several "ordinary" constructors (Code 7 shows two), but it can declare **at most** one copy constructor, one move constructor, and one destructor. These three special functions have a unique signature.

```
Car::Car (string& vin) {
    this->vin = vin;
    //this->model = "......"; /* decode from the VIN */
    //this->year = xxxx       /* decode from the VIN */
    /* more code here */
}

Car::Car(string& make, string& model, int year) {
    /* more code here */
}

Car::~Car() {
    /* empty destructor, nothing to undo */
}

Car::Car (const Car& source) {    /* copy constructor */
    this->vin = source.vin;
    this->model = source.model;
    this->year = source.year;
    this->mileage = source.mileage;
    this->fuelAmount = source.fuelAmount;
}

Car::Car (Car&& source) {        /* move constructor */
    /* attempt to use move semantic on string */
    this->vin = std::move(source.vin);
    this->model = std::move(source.model);

    /* can't use std::move on primitive types */
    this->year = source.year;
    this->mileage = source.mileage;
    this->fuelAmount = source.fuelAmount;
}
```

Listing 8: Implementation of Car special functions

- A copy constructor always takes an lvalue reference parameter (declared using one ampersand)
- A move constructor always takes an rvalue reference parameter (declared with two ampersands). Usually moving involves changing pointers from the source data items to the corresponding destinations
- A destructor has a tilde (~) prefix and takes no parameters

2.6.2 Invoking The Constructors and Destructors

Out of the four special functions above (constructor(s), copy constructor, move constructor, and destructor), only the constructor and destructor can be "invoked" explicitly in a C++ program, the copy and move constructors are not directly invocable but they execute automatically on several different occasions. In general, these four special functions serve as a *hook function* on these special occasions:

- The **Constructors** will be used on the following two occasions:
  - A variable of a class type is declared

  - An object is instantiated for a pointer of the class type

- The **destructor** will be used on the following two occasions:
  - A local variable of a class type is about to be out of scope
  - An object is explicitly removed from its pointer

```
1  void sampleFunction() {
2    Car black_jeep ("1GYER61772337");       /* constructor executes automatically */
3    Car *future_car;                                  /* no constructor invocation */
4
5    future_car = new Car ("1Z76DFYY73622");             /* constructor is invoked */
6
7    /* some more code here */
8
9    delete future_car;                      /* destructor is invoked (explicitly) */
10 }              /* black_jeep is out of scope destructor executes automatically */
```

- The copy (or move) constructor executes whenever an object is being constructed by using the properties of a source object. When the source object is temporary (or soon to expire), move constructor will execute[5].

  A few occassions where copy (or move) constructor will execute:
  - An object is created by "cloning" an existing object
  - An argument object is **passed-by-value** to a parameter of a function.
  - A function returns its result using `return-by-value`

```
/* copy constructor needed only for c1 */
Car test_function (Car c1, Car& c2) {
  c1.addFuel (10.0);                  /* it does not alter the passed object */
  c2.addFuel (4.0);                     /* it adds fuel to the passed object */

  return c2;
}                         /* destructor executes automatically to destroy c1 */

int main() {
   Car blue_saturn("1G8AK5593Z634523");              /* constructor executes */
   Car white_civic("1H8723478YF61234");              /* constructor executes */

   /* create a "clone" of my_car */
   Car  my_car {blue_saturn};  /* copy constructor executes to create my_car */

   Car mini;                             /* car (default) constructor executes */
   mini = test_function (blue_saturn, white_civic);
}
```

  During the invocation of `test_function`:

[5]it is the responsibility of the C++ compiler to determine whether the source object is temporary or expiring. Our responsibility as a programmer is just to provide the correct implementation of moving the object

- The argument `blue_saturn` is passed-by-value, parameter `c1` **gets a copy** of `blue_saturn` and the Car copy constructor will execute on our behalf to create the copy.
- The argument `white_civic` is passed-by-reference, therefore `c2` is an alias of `white_civic` and no copying takes place

Inside `test_function`, because `c1` is a copy, the 10 gallons of fuel is added only to the copy, and not to `blue_saturn`. On the contrary, inside `test_function c2` is an alias of `white_civic` and the 4 gallons of fuel was added to `white_civic`. itself

When `test_function` terminates:

- The copy constructor will execute to create the returned objnect from `c2`
- The Car destructor will execute automatically (on our behalf) to destroy `c1`.

Finally when `main` terminates the Car destructor will automatically executes four times to destroy `blue_saturn`, `white_civic`, `my_car`, and `mini`.

2.6.3 Memberwise Initialization List

A constructor (copy and move constructors also included), may improve the performance of its initialization logic by using C++ **member wise initialization list**. Consider the Car constructor shown in the left column

```
1  class Car {
2  private:
3    string VIN;
4    float gasAmount, odometer;
5
6  public:
7    Car (string& s_vin)
8    {
9      VIN = s_vin;
10     gasAmount = 0.0;
11     odometer = 0.0;
12   }
13 };
```

```
1  class Car {
2  private:
3    string VIN;
4    float gasAmount, odometer;
5
6  public:
7    Car (string& s_vin)
8       : VIN (s_vin), gasAmount(0)
9    {
10     odometer = 0.0;
11   }
12 };
```

The constructor in the left column initializes the string variable `VIN` by performing the following two actions:

1. Create an empty string using the `string` default constructor
2. Replace that empty string with the provided parameter (`s_vin`)

A more efficient constructor is shown in the right column. Line 8 employs the memberwise initialization syntax to setup `VIN` and `gasAmount`. The initialization of `gasAmount` here will not improve its performance because it is a variable of primitive type, it is just to show how to use the syntax for initializing multiple instance variables. Using this technique, the two-step initialization process for `VIN` has been eliminated and replaced by **only one** invocation to `string` **copy constructor**.

2.7 Java Packages vs. C++ namespaces

Java organizes its classes into packages (and sub packages). Similarly, C++ organizes its classes into namespaces. For instance, Code 9 shows how to declare a class inside a namespace and the following code snippet shows an example of variable declaration:

```
#include "Car.h"

using namespace gvsu;

int main() {
    Car        red_kia ("1G97192731232");
    gvsu::Car  blue_chevy ("1GDSTDY231233");   // alternate syntax
}
```

```
// Filename: Car.h
#ifndef GVSU_CAR_H
#define GVSU_CAR_H
#include <string>

namespace gvsu {
class Car {
private:
  // initial value of these two
  // constants is set in Car.cpp
  static const double MPG;
  static const double TANK_CAPACITY;
  std::string VIN;
  double mileage;
  double fuelAmount;

public:
  Car (std::string id);
  void drive (double distance);
  void addFuel (double gasAmount);
  double checkFuelAmount() const;
};
}
#endif
```

```
// Filename: Car.cpp
#include "Car.h"

namespace gvsu {
const double Car::MPG = 22.3;
const double Car::TANK_CAPACITY = 12;

Car::Car (std::string id) {
    VIN = id;
    mileage = 0;
    fuelAmount = TANK_CAPACITY;
}

void Car::drive (double distance) {
    mileage += distance;
    fuelAmount -= distance / MPG;
}

void Car::addFuel (double gasAmount) {
    fuelAmount += gasAmount;
}

double Car::checkFuelAmount() const {
    return fuelAmount;
}
}
```

Listing 9: Declaring a class inside a namespace

3 Operator Overloading

C++ operator overloading is a feature that allows us to (re)define the behavior of C++ operator when they are applied to objects of our class. Some practical example features that can be implemented using C++ operator overloading:

- Use the arithmetic operators (`+`, `-`, `*`) to perform matrix operations or operations to a `RationalNumber` class
- Use `+` and `-` to push and pop an item to or from a Stack.
- Use `[ ]` to select a character from a string
- Use `-` operator (unary minus) to reverse a string

Consider the following function invocations on `Coin` objects.

```
Coin one (20);
Coin two (60);

Coin sum = g_add (one, two);        // g_add is a global function
Coin total = one . m_add (two);     // m_add is a member function
```

Now imagine if last two function invocations could be invoked using the `operator` keyword:

```
Coin sum = operator+ (one, two);        // imagine "operator+" is "g_add"
Coin total = one . operator+  (two);    // imagine "operator+" is "m_add"
```

or even **without** the "`operator`" keyword:

```
total = one + two;       // invoke a member function
```

Essentially, this is **operator overloading** in C++; it is acceptable to declare functions whose name begins with the keyword `operator` followed by a valid C++ operator. From now on, expect to see function with the following names in a C++ code: `operator =`, `operator []`, `operator >=`, `operator ()`, etc. However, unlike other functions that may take any number of arguments, **operator functions can only a fixed number of arguments depending on the arity of the operator**. Most C++ operators take either one operand (unary operators) or two operands (binary operators).

- The negation (`!`), decrement (`--`), increment (`++`) are a unary operator and they can be applied only to one object
- The division (`/`) and shift (`<<`) are a binary operator and they must be applied to two objects
- The minus (`-`) operator can be used as a unary (like in $-x$) or binary (like in $p - q$) operator

Operator overloading can be implemented in two different ways:

1. As a member function (a function of a class), when this function is invoked the left-hand side operand is an object of the class

[7] The invoking object `a` does not have to be an array
[7] The index expression does not have to be an integer

Operator Notation	Description	Expanded Expression
`a = b`	assignment	`a . operator = (b)`
`a[5]`	"array"-selection[6]	`a . operator[] (5)`
`a["happy"]`	"array"-selection[7]	`a . operator[] ("happy")`
`!a`	negation	`a . operator! ()`
`a(56)`	function call	`a . operator() (56)`
`a(56, 30.5, "feet")`	function call	`a . operator() (56, 30.5, "feet")`

2. As a global function, when this function is invoked the operands can be specified in any order as long as they may the parameter order of the function

Let's assume that we are about to utilize operator overloading to "insert a coin to a vending machine" using two different operators:

```
#include "Coin.h"
#include "VendingMachine.h"

int main() {

  VendingMachine vm;
  Coin c;

  vm << c;        // equivalent to  vm . operator<< (c)
  c >> vm;        // equivalent to  c  . operator>> (vm)

  return 0;
}
```

3.1 Implemented as Member Functions

When implemented as member functions, each function will be declared inside each respective class. Operator `<<` is a member function of the `VendingMachine` class and takes one argument (`Coin`). Likewise, operator `>>` is a member of the `Coin` class.

```
4  #include "Coin.h"
5
6  class VendingMachine {
7  public:
8    void operator<< (Coin& money) {
9      /* add your code here */
10   }
11 };
```

```
23 #include "VendingMachine.h"
24
25 class Coin {
26 public:
27   void operator>> (VendingMachine& machine) {
28     // invoke VendingMachine function (line 8)
29     // and pass "this coin" as argument
30     machine << *this;
31   }
32 };
```

Listing 10: Choices of operator overloading implementation for "insert coin to a vending machine" task

Notice that when an operator overloading is implemented as a member function, the **class of the invoking object** will be the host of the function. For instance, to enable the following operation

`vm << c;`

the `VendingMachine` class is required to implement the `operator <<` function and this function takes a `Coin` parameter. Likewise, to enable

`c >> vm;`

the `Coin` class is required to implement the `operator >>` function that takes a `VendingMachine` parameter.

3.2 Implemented as Global Functions

When implemented as global functions, each function takes two parameters and the order of these two parameters must be written correctly. For the above particular examples, the first argument is the left-hand-side object.

```
54 void operator<< (VendingMachine& machine, Coin& money) {
55    /* add your code here */
56 }
57
58 void operator>> (Coin& money, VendingMachine& machine) {
59    /* invoke the above function (line 54) */
60    operator<< (machine, money);
61 }
```

Some restrictions on operator overloading implementation:

- We can't invent a new operators, only valid C++ operators can be used
- We can't change the arity (number of operands) of the operators. For instance, the division operator (/) is binary, therefore it must be invoked with another argument:

```
obj1.operator/();       /* error */
obj1.operator/(obj2);   /* ok, it is equivalent to obj1 / obj2 */
```

- The left-hand side is an object of the class where the operator is defined. For instance, the following two lines are interpreted differently:

```
s1 + 23.4;    /* OK,  equivalent to   s1.operator+(23.4) */
23.4 + s1;    /* ERROR: can't invoke  "operator+" on 23.4 */
```

3.3 Implementation of `operator=`

The `operator =` functions[8] are frequently mixed up with the copy/move constructor described in Section 2.6. The following example should clarify the difference between them:

```
32 Coin c1(45);         // executes constructor
33 Coin *c2;            // executes none
34
35 Coin a{c1};          // executes copy constructor
36 Coin b = c1;         // executes copy constructor
37 a = c2;              // expanded expression: a . operator= (c2);
38 c2 = new Coin(20);   // execute constructor, but NOT operator=
```

[8] we have two variants: copy assignment and move assignment

- At lines 35 and 36, the Coin copy constructor is invoked to create `a` and `b` by cloning `c1`
- At line 37, the object `a` has been instantiated previously, so that line invokes the `operator=` function.
- Although line 38 uses the assignment (`=`) operator, we are copying a pointer, not a Coin object, so that statement will not execute Coin's `operator=`.

The prototypes of these assignment operators of the `Coin` class are shown in Listing 11. The main difference between the two designs is the function return value. The implementation with `void` return will not allow chaining of assignment operators as shown below:

```
c1 = c2 = mycoin;
// expanded expr: c1 . operator= (c2 . operator= (mycoin));
```

When `c2 . operator= (my coin)` return type is `void`, the invocation `c1 . operator= ()` will fail. The logic of the move assignment looks very similar to the copy assignment, expect for the attempt to move-assign the string object (Coin serial number).

```
1  class Coin {
2  private:
3    int value;
4    string serial;
5
6  public:
7    // copy assignment
8    void operator= (const Coin& src) {
9      value = src.value;
10     serial = src.serial;
11   }
12
13   // move assignment
14   void operator= (Coin&& src) {
15     value = src.value;
16     serial = std::move(src.serial);
17   }
18 }
```

```
1  class Coin {
2  private:
3    int value;
4    string serial;
5  public:
6    Coin& operator= (const Coin& src) {
7      value = src.value;
8      serial = src.serial;
9      return *this;
10   }
11
12   Coin& operator= (Coin&& src) {
13     value = src.value;
14     serial = std::move(src.serial);
15     return *this;
16   }
17 }
```

Listing 11: Possible implementation of Coin `operator=`

3.4 Stream Output (operator<<) as a Non-Member Function

Java programmers know how to print objects to standard output using the following syntax:

```
Book sawyer = new Book ("The Adventure of Tom Sawyer", "Mark Twain");

System.out.println ("I read " + sawyer + " last month");
```

where the last line is essentially equivalent to

```
System.out.println ("I read " + sawyer.toString() + " last month");
```

The equivalent C++ statement of

```
System.out.print (sawyer);     // in Java
```

is

```
// in C++
cout << sawyer;
cout . operator<< (sawyer);    // equivalent expanded expression
operator<< (cout, sawyer);     // alternate syntax of invocation
```

The expanded expression above gives a hint that the `operator<<` is invoked on the `cout` object (of the `ostream` class). Because `stream` is a class in the C++ library, we can't modify its declaration and it is not possible to implement the `operator <<` as a member function, we have to implement it as a global function.[9] The alternative solution is to implement a global function that takes two parameters: a `ostream` and a `Book`.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  class Book {
6  public:
7     Book (string, string);  /* constructor */
8     ~Book();                /* destructor */
9
10    /* other member functions of Book will go here */
11 private:
12    string author, title;
13 };
```

Listing 12: The `Book` class

```
void operator<< (ostream& os, const Book& b)
{
   os << b.getTitle() << " by " << b.getAuthor();
}
```

The above implementation is correct, but the `void` return type will prevent chaining of `<<` to print multiple data items when one of the items is a Book. To allow chaining, we must change the return type from `void` to `ostream&`

```
ostream& operator<< (ostream& os, const Book& b)
{
   os << b.getTitle() << " by " << b.getAuthor();
   return os;
}
```

[9] Had the operation were `sawyer >> cout`, then we could write the `operator>>` in the `Book` class.

The above code assumes that the Book class provides the two public functions getTitle() and getAuthor(). If it does not, we have to access the title and author private data members directly:

```
ostream& operator<< (ostream& os, const Book& b)
{
   os << b.title << " by " << b.author;
   return os;
}
```

And consequently, since the above operator<< function is **not a member function** of Book, we have to give it a special permission to access Book's private data by declaring it as a friend function as shown at line 8 of Listing 13.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  class Book {
6     /* a friend function is a standalone function that is given permission
7      * to access private identifiers (functions/data) of this class */
8     friend ostream& operator<< (ostream&, const Book& b);
9  public:
10    Book (string, string);  /* constructor */
11    ~Book();                /* destructor */
12
13    /* other member functions of Book will go here */
14 private:
15    string author, title;
16 };
```

Listing 13: Declaring a standalone function as friend

3.5 operator++ and operator--

The increment (or decrement) operator can be applied as either a prefix increment ++ obj— or postfix increment obj ++. Be sure to understand that operator++ is a function and takes **no argument**. The expanded expression of ++obj is

```
obj . operator++ ()
```

and the expanded expression of obj++ is

```
obj . operator++ (1);   /* takes an integer parameter */
```

3.6 operator[]

We normally use the square brackets [] to supply an non-negative zero-based integer index to select ane of the elements in a one-dimensional array. C++ operator overloading may turn this operator into a more **general-purpose selector**. For instance, we can provide a totally new behavior:

- Allow *negative numbers* to be used for indexing a one-dimensional array-like structure: `arr[-1]` selects the last element of `arr`.
- Use non-integer "index" to select a statistical property of a data set. For instance `dt_set["average"]` can be designed to return the average of numbers in a data set `dt_set`. Implementation of this feature can be outlined using the following class declaration:

```
#include <vector>
#include <string>
using namespace std;
class DataSet {
private:
  vector<float> dataitems;
public:
   float operator[] (const string& stat_name) {
     /* process your numbers here */
     return .....;
   }
};
```

Some programming languages allow arrays to be accessed using a negative index. For instance, in Python, `arr[-1]` refers to the *last* element of `arr`. C++ operator overloading allows us to mimic this Python syntax using only a few lines of code.

3.7 operator()

The "function call" operator is interesting as well as powerful. Let's assume that we are about to implement the "add gasoline and drive" task in the `Car` class. Using the "function call" syntax, we may write the code as follows:

```
Car blue_kia ("1GCEC19VX5E653978");

// add 5 gallons of gas before driving 106 miles
blue_kia (5, 106);
// expand expression is  blue_kia . operator()  (5, 106);
```

The most confusing part of implementing the function call operator is the parentheses. Be sure to understand that the first pair `()` is part of the function name `operator()`, and the next set of parentheses delimit the arguments passed to the function.

Implemented as a member function of the `Car` class, the outline of this function is shown at lines 8-11 below:

```
1  class Car {
2  private:
3    /* data not shown */
4  public:
5    void addFuel (double f) { /* more code here */ }
6    void drive (double d)   { /* more code here */ }
7
8    void operator()  (double gasAmt, double drvDist) {
9       this->addFuel (gasAmt);
10      this->drive (drvDist);
11   }
12 };
```

3.7.1 Functors

A class that overloads the "function call" operator (like the Car example above) has a special name, we call it a **functor class**. The objects instantiated from this class are called **functors**. So, technically the `blue_kia` object above is a functor.

4 Templates

4.1 Java Generics and C++ Templates

After Java Generics was introduced in Java 5.0, Java programs are more type-safe. Prior to Generics, Java containers store their contents as references to `Object`:

```
java.util.ArrayList storage;
storage = new java.util.ArrayList();

storage.add (new Coin(30));
storage.add ("Atlanta");
storage.add (new Integer (404));
```

and it is the programmer's responsibility to know to right datatype stored in each container cell. After the introduction of generics, containers are declared by also specifying its cell type:

```
import java.util.Arraylist;

ArrayList<Coin> storage;     // "import" makes this declaration shorter to write
ArrayList<String> cities;
storage = new ArrayList<Coin>();
cities  = new ArrayList<String>();

storage.add (new Coin(30));
storage.add ("Atlanta");   // compile error
cities.add ("Atlanta");
System.out.println (cities.get(0));
```

C++ programers can benefit from a similar feature by using **templates**. The C++ counterpart of the above Java code would be:

```
#include <vector>
#include <string>
#include "Coin.h"

std::vector<Coin> storage;        // automatic object instantiation
std::vector<std::string> cities;  // automatic object instantiation

storage.push_back (Coin(30));

cities.push_back ("Atlanta");
cout << cities[0] << endl;
```

Let's design a `SafeBox`[10] class that represents a password-protected steel box for storing a single item. Ideally, a safe box should allow the user to store `any type` of item. A Java generic or C++ template class is the best choice for implementing such a class. Listing 14 shows the two different implementation of the class in Java and C++. In both class declarations, `E` represents the type of item stored in the safe box. In the Java class, this type is used at four different places (lines 5, 14, 39, and 43).

In the C++ version, the `saveWithCode` function is designed to use rvalue reference parameters to allow a move semantic applied to the item and user secret code.

[10]The idea of this design was inspired by the safe deposit boxes commonly found in hotels

```java
public class SafeBox<E> {
  private String secretCode;
  private int failedCount;
  private boolean isLocked;
  private E valuable;

  public SafeBox() {
    isLocked = false;
    valuable = null;
    secretCode = null;
    failedCount = 0;
  }

  public void saveWithCode (E uItem,
          String uCode) {
    if (isLocked) return;
    failedCount = 0;
    secretCode = uCode;
    isLocked = true;
    valuable = uItem;
  }

  public boolean unlock (String ucode) {
    if (failedCount < 3) {
      if (ucode.equals(secretCode))
        isLocked = false;
      else {
        failedCount++;
        System.err.println (
                "Incorrect code");
      }
      return !isLocked;
    }
    else
      throw new IllegalStateException (
              "Too many failed attempt");
  }

  public E removeContent() {
    if (isLocked)
      throw new IllegalStateException (
              "Box is locked");
    E item;
    item = valuable;
    valuable = null;
    System.out.println (
            "Item successfully removed");
    return item;
  }
}
```

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

template<typename E>
class SafeBox {
private:
  string secretCode;
  int failedCount;
  bool isLocked;
  E valuable;

public:
  SafeBox() {
    isLocked = false;
    failedCount = 0;
  }

  void saveWithCode (E&& uItem,
          string&& uCode) {
    if (isLocked) return;
    failedCount = 0;
    secretCode = uCode;
    isLocked = true;
    valuable = uItem;
  }

  bool unlock (const string& ucode) {
    if (failedCount < 3) {
      if (ucode == secretCode)
        isLocked = false;
      else {
        failedCount++;
        cerr << "Incorrect code" << endl;
      }
      return !isLocked;
    }
    else
      throw logic_error (
              "Too many failed attempt");
  }

  E removeContent() {
    if (isLocked)
      throw logic_error ("Box is locked");
    E item;
    item = valuable;
    cout << "Item successfully removed"
        << endl;
    return item;
  }
};
```

Listing 14: SafeBox class design in Java and C++

The corresponding main methods/functions that uses the `SafeBox` class are shown in Listing 15. The C++ counterpart show two instances of `SafeBox` objects: one for storing a `string` and one for storing an `int`.

```java
public class SafeBoxMain {
  public static void main (String[] args) {
    SafeBox<String> sbox;
    sbox = new SafeBox<String>();
    sbox.saveWithCode ("Hello",
            "h1Dd3nT3XT");
    sbox.unlock ("h1Dd3nT3XT");
    sbox.removeContent();
  }
}
```

```cpp
#include <iostream>
#include <string>
#include <utility>
#include "SafeBox.hpp"

using namespace std;

int main () {
  SafeBox<string> sbox;

  string msg = "Hello";
  // use std::move() to convert
  // to Rvalue ref
  sbox.saveWithCode (std::move(msg),
          "h1Dd3nT3XT");
  sbox.unlock ("h1Dd3nT3XT");
  string s = sbox.removeContent();

  SafeBox<int> ibox;
  ibox.saveWithCode (616, "areacode");
  ibox.unlock ("areacode");
  cout << ibox.removeContent() << endl;

  return 0;
}
```

Listing 15: The main method/function showing how the `SafeBox` class is used

Notice that line 6 of the C++ class declaration in Listing 14 tags the class with a `template` prefix. When a class declaration requires several template parameters, one of the parameters should be written in a separate `typename` declaration. For instance, if we were to customize the data type for the safe box secret code using a template parameter, we can change the class declaration to:

```cpp
template<typename E, typename CODE>
class SafeBox {
private:
  CODE secretCode;
  int failedCount;
  bool isLocked;
  E valuable;

public:
  /* more code here */
};
```

```cpp
template<class E, class CODE>
class SafeBox {
private:
  CODE secretCode;
  int failedCount;
  bool isLocked;
  E valuable;

public:
  /* more code here */
};
```

If desired, the keyword `class` can be used in place of `typename`.

Another technique for declaring a template class is to separate the "interface" from its function "implementation". However, unlike the non-template class declaration where the split design is written

into a `.h` and `.cpp` files, a template class declaration must be written into **a header only declaration**, and the common practice is to write the code in a file with `.hpp` extension. Listing 16 shows an example of this design approach. Pay special attention to lines 21, 26, 34, and 47 where each function must be declared using the `template` keyword an the scope prefix, `SafeBox<E>::`, must be written to include the template parameter.

```cpp
// File: SafeBox01.hpp
#ifndef SAFEBOX_H
#define SAFEBOX_H
#include <iostream>
#include <string>
#include <stdexcept>

template<typename E> class SafeBox {                    /*--- CLASS DECLARATION ---*/
private:
  std::string secretCode;
  int failedCount;
  bool isLocked;
  E valuable;
public:
  SafeBox();
  void saveWithCode (E&& uItem, std::string&& uCode);
  bool unlock (const std::string& ucode);
  E removeContent();
};

template<typename E> SafeBox<E>::SafeBox() {         /*--- FUNCTION DEFINITIONS ---*/
  isLocked = false;
  failedCount = 0;
}

template<typename E> void SafeBox<E>::saveWithCode (E&& uItem, std::string&& uCode) {
  if (isLocked) return;
  failedCount = 0;
  secretCode = uCode;
  isLocked = true;
  valuable = uItem;
}

template<typename E> bool SafeBox<E>::unlock (const std::string& ucode) {
  if (failedCount < 3) {
    if (ucode == secretCode) isLocked = false;
    else {
      failedCount++;
      std::cerr << "Incorrect code" << std::endl;
    }
    return !isLocked;
  }
  else
    throw std::logic_error ("Too many failed attempt");
}

template<typename E> E SafeBox<E>::removeContent() {
  if (isLocked) throw std::logic_error ("Box is locked");
  E item;
  item = valuable;
  std::cout << "Item successfully removed" << std::endl;
  return item;
}
#endif
```

Listing 16: A split design of the `SafeBox` class.

```cpp
// File: SafeBox02.hpp
#ifndef SAFEBOX_H
#define SAFEBOX_H
#include <iostream>
#include <string>
#include <stdexcept>
// use string as the default type when the second parmeter is unspecified
template<class E,class C = std::string> class SafeBox {
private:
  C secretCode;
  int failedCount;
  bool isLocked;
  E valuable;
public:
  SafeBox();
  void saveWithCode (E&& uItem, std::string&& uCode);
  bool unlock (const std::string& ucode);
  E removeContent();
};

template<class E, class C> SafeBox<E,C>::SafeBox() {        /*--- FUNCTION DEFINITIONS ---*/
  isLocked = false;
  failedCount = 0;
}

template<class E, class C> void SafeBox<E,C>::saveWithCode (E&& uItem, std::string&& uCode) {
  if (isLocked) return;
  failedCount = 0;
  secretCode = uCode;
  isLocked = true;
  valuable = uItem;
}

template<class E, class C> bool SafeBox<E,C>::unlock (const std::string& ucode) {
  if (failedCount < 3) {
    if (ucode == secretCode) isLocked = false;
    else {
      failedCount++;
      std::cerr << "Incorrect code" << std::endl;
    }
    return !isLocked;
  }
  else
    throw std::logic_error ("Too many failed attempt");
}

template<class E, class C> E SafeBox<E,C>::removeContent() {
  if (isLocked) throw std::logic_error ("Box is locked");
  E item;
  item = valuable;
  std::cout << "Item successfully removed" << std::endl;
  return item;
}
#endif
```

Listing 17: A split design of the `SafeBox` class with multiple template parameters.

5 Designing a Class

C++ class designers should consider including the following six special member functions in every class they design:

1. Constructor(s)
2. Destructor
3. Copy (clone) constructor
4. Move (clone) constructor
5. Copy assignment
6. Move assignment

As a general rule of thumb, **whenever you write any of the five functions, most likely you have to write them all.** Out of the five functions, the destructor is perhaps the easiest to comprehend, its main job is to reverse the operations carried out by the constructor. Moreover, constructor and destructor are easier to comprehend than the other four because these two functions are invoked explicitly by `new` and `delete`. Besides these two explicit invocations, constructor and destructor are also *implicitly* invoked by the C++ runtime system whenever local (automatic) objects are about to be created or destroyed on the stack.

In order to provide a more concrete example, we will look at the design of an `arr_list` as the counterpart of `java.util.ArrayList`. The Java implementation of `ArrayList` relies on a one-dimensional array whose size automatically grows when more data items are added to the array list. We will design our `arr_list` as a template class. The interface of `arr_list` is shown in Listing 18.

Observe a few details in the design of the `arr_list` class:

- `data` a pointer that holds an array of template type `E`. In Java, we would declare the array using the following syntax to declare an array (of float):

  ```
  float[] data;
  ```

  and then instantiate the array by specifying its capacity:

  ```
  data = new float[300];
  ```

  The equivalent declaration and instantiation in C++ is:

  ```
  float *data;
  data = new float[300];
  ```

- `num_items` (initially 0) and `capacity` keep track of the current number of items and the capacity of the array. When `num_items == capacity` the array must be reallocated to hold more items.
- The six special functions (constructor, destructor, copy constructor, move constructor, copy/move assignment) are declared next
- The next six functions are specific to array list insert/delete operations. Out of the six functions, `get()` and `size()` are non-mutable
- Anywhere data type `E` is used in the functions, we use C++ alias as much as possible. The only exception is the return of the `remove()` function, which tis a copy of the item. This design is intentional because after an item is removed, we must return its copy.

```
1  //  arr_list00.h
2  #ifndef CS263_arr_list_h
3  #define CS263_arr_list_h
4  template<typename E>
5  class arr_list {
6  private:
7      E* items;
8      int capacity;
9      int num_items;
10 public:
11     arr_list();                                        /* constructor */
12     ~arr_list();                                       /* destructor */
13     arr_list (const arr_list& src);                 /* copy constructor */
14     arr_list (arr_list&& src);                      /* move constructor */
15     arr_list& operator= (const arr_list& rhs);       /* copy assignment */
16     arr_list& operator= (arr_list&& rhs);            /* move assignment */
17
18     E& get(int pos) const;   /* retrive an item at a specific position */
19     int size() const;                      /* the number of items stored */
20     void add (const E&);                             /* append to the end */
21     void add (int pos, const E&);   /* insert into a specific position */
22     void clear();                                /* remove all data items */
23     E remove (int pos);       /* remove an item at a specific position */
24 };
25 #endif
```

Listing 18: Interface of `arr_list`

The complete code of the special functions is shows in Listing 19. The implementation of the other functions of `arr_list` is shown in Listing 20.

- The code structure of the copy constructor and copy assignment looks very similar. The first difference is the `delete [] items` statement at the beginning of the copy assignment required to delete the array of the current object. The other difference is the `return *this` statement to return the current object.

  Likewise, the code structure of the move constructor looks very similar to the move assignment. There is so much code duplication among these four functions, there should be a better way of writing them that avoids duplication.

```
36 /* constructor */
37 template<class E> arr_list<E>::arr_list() {
38     capacity = 5;
39     items = new E[capacity];
40     num_items  = 0;
41 }
42
43 template<class E> arr_list<E>::~arr_list() {      /* destructor */
44     delete [] items;
45 }
46
47 template<class E> /* copy constructor */
48 arr_list<E>::arr_list(const arr_list& src) {
49     capacity = src.capacity;
50     num_items = src.num_items;
51     items = new E[capacity];
52     for (int k = 0; k < num_items; k++)
53         items[k] = src.items[k];
54 }
55
56 template<class E> /* move constructor */
57 arr_list<E>::arr_list(arr_list&& src) {
58     capacity = src.capacity;
59     src.capacity = 0;
60     num_items = src.num_items;
61     src.num_items = 0;
62     items = src.items;
63     src.items = nullptr;
64 }
65
66 template<class E> /* copy assignment */
67 arr_list<E>&
68 arr_list<E>::operator= (const arr_list& src) {
69     delete [] items;  // remove current items
70     capacity = src.capacity;
71     num_items = src.num_items;
72     items = new E[capacity];
73     for (int k = 0; k < num_items; k++)
74         items[k] = src.items[k];
75     return *this;
76 }
77
78 template<class E> /* move assignment */
79 arr_list<E>&
80 arr_list<E>::operator= (arr_list&& src) {
81     delete [] items;
82     num_items = src.num_items;
83     capacity = src.capacity;
84     items = src.items;
85     src.num_items = 0;
86     src.capacity = 0;
87     src.items = nullptr;
88     return *this;
89 }
```

Listing 19: Implementation of `arr_list` special functions

```
91  template <class E> E& arr_list<E>::get(int pos) const {
92      return items[pos];
93  }
94
95  template <class E> void arr_list<E>::resize() {
96      int bigger_capacity = capacity + 10;
97      E* bigger = new E[bigger_capacity];        /* allocate a bigger array */
98      for (int k = 0; k < capacity; k++)  /* copy the data to the new array */
99          bigger[k] = std::move(items[k]);
100     delete [] items;                              /* delete the old array */
101     items = bigger;
102     capacity = bigger_capacity;
103 }
104
105 template<class E> void arr_list<E>::add(const E& z) {
106     if (num_items == capacity)
107         resize();
108     items[num_items] = z;
109     num_items++;
110 }
111
112 template<class E> void arr_list<E>::add(int pos, const E& z) {
113     if (num_items == capacity)
114         resize();
115     for (int k = num_items; k > pos; k--) /* make room for the new item */
116         items[k] = std::move(items[k-1]);
117     items[pos] = z;
118     num_items++;
119 }
120
121 template<class E> E arr_list<E>::remove(int pos) {
122     E del_item = std::move(items[pos]);
123     for (int k = pos; k < num_items; k++) /* fill in the "void" */
124         items[k] = std::move(items[k+1]);
125     num_items--;
126     return del_item;
127 }
128
129 template<class E>
130 std::ostream& operator<< (std::ostream& ss, const arr_list<E>& arr)
131 {
132     for (int k = 0; k < arr.size(); k++)
133         ss << arr.get(k) << ' ';
134     return ss;
135 }
136 #endif
```

Listing 20: Implementation of the other `arr_list` member functions

As mentioned previously, we find a lot of code duplication among the copy constructor, move constructor, and copy assignment. The improved implementation of the copy constructor and copy assignment is given in the right column of Listing 21.

- The STL `copy` function at line 60 of the improved implementation replaces the for-loop at line 52
- The improved copy-assignment operator is implemented using a combination of copy-construct (line 74) and swap (line 75). Upon entry to the function, `tmp` is a local copy of `src` (the object to copy from). After the swap completes, `tmp` is swapped with "this object". Upon exit of the function, `tmp` (which now contains the "old this") is destroyed (destructor is called), hence removing the "old current items" (mimicking line 69 of the original code)

Original

```
47 template<class E> /* copy constructor */
48 arr_list<E>::arr_list(const arr_list& src) {
49     capacity = src.capacity;
50     num_items = src.num_items;
51     items = new E[capacity];
52     for (int k = 0; k < num_items; k++)
53         items[k] = src.items[k];
54 }



66 template<class E> /* copy assignment */
67 arr_list<E>&
68 arr_list<E>::operator= (const arr_list& src) {
69     delete [] items;  // remove current items
70     capacity = src.capacity;
71     num_items = src.num_items;
72     items = new E[capacity];
73     for (int k = 0; k < num_items; k++)
74         items[k] = src.items[k];
75     return *this;
76 }
```

Improved

```
55 template<class E>  /* copy constructor */
56 arr_list<E>::arr_list(const arr_list& src) {
57     capacity = src.capacity;
58     num_items = src.num_items;
59     items = new E[capacity];
60     copy (src.items, src.items + num_items,
61             items);
62 }



71 template<class E>    /* copy assignment */
72 arr_list<E>&
73 arr_list<E>::operator= (const arr_list& src) {
74     arr_list<E> tmp {src};
75     swap(tmp);
76     return *this;
77 }
```

Listing 21: Improved implementation of `arr_list` copy constructor and copy assignment

The improved implementation relies on a private function `swap` whose main job is to swap all the attributes between **this** `arr_list` and a second object passed as `other`. The three statements inside this function invokes the swap function from the C++ standard library.

```
void swap (arr_list& other) {
    std::swap(capacity, other.capacity);
    std::swap(num_items, other.num_items);
    std::swap(items, other.items);
}
```

The improved move constructor is implemented using a combination of default constructor and swap (again). Upon entry to the move constructor function, the default constructor initializes an "empty" `arr_list` which is later swapped with the incoming source object. The syntax used at line 66 is known as the `constructor initialization list`. For this particular code, the initialization list invokes the

default constructor. The overall effect, is that "this" object was initialized from the source and the source object becomes empty. Similarly, the improved move assignment is implemented the same way. However, instead of creating an empty object, we first remove the current items (line 82) and then perform a swap.

Original

```
56 template<class E> /* move constructor */
57 arr_list<E>::arr_list(arr_list&& src) {
58     capacity = src.capacity;
59     src.capacity = 0;
60     num_items = src.num_items;
61     src.num_items = 0;
62     items = src.items;
63     src.items = nullptr;
64 }


78 template<class E> /* move assignment */
79 arr_list<E>&
80 arr_list<E>::operator= (arr_list&& src) {
81     delete [] items;
82     num_items = src.num_items;
83     capacity = src.capacity;
84     items = src.items;
85     src.num_items = 0;
86     src.capacity = 0;
87     src.items = nullptr;
88     return *this;
89 }
90
```

Improved

```
64 template<class E> /* move constructor */
65 arr_list<E>::arr_list(arr_list&& src)
66     : arr_list()
67 {
68     swap(src);
69 }


79 template<class E> /* move assignment */
80 arr_list<E>&
81 arr_list<E>::operator= (arr_list&& src) {
82     delete [] items;
83     swap(src);
84     return *this;
85 }
```

Listing 22: Improved implementation of `arr_list` move constructor and move assignment

6 Containers and Iterators

The Java Collection Framework (JCF) provides a set of classes that implement various data structures: linked list, trees, map, stack, queue, priority queue, etc. The `Collection` interface is one of the top interfaces that defines the behavior of the rest of the classes in the framework. In conjunction with these collection classes, Java provides iterator classes that can be use for accessing and manipulating data items in the collection.

6.1 Java Iterators vs. C++ Iterators

Although both languages provide iterators associated with a container, the detailed implementation of iterators by each language is different.

Java	C++
• An iterator is an object that lives **between** two consecutive data items. Upon creation, it stays at a position **before** the first data item and `hasNext()` returns true (when the container is not empty). • After it advances beyond the last item, invoking `hasNext()` returns false • Invoking `next()` on a Java iterator will perform two actions: 1. Return the data item to its right 2. Advance the iterator to the next position	• An iterator is an object that **points** to the current data item. • To make the iterator point to the first data item, an explicit invocation of `begin()` is required • Invoking the indirection/dereference operator only returns the current data item (it will not advance to the next data item) • The increment operator (`++`) advances the iterator to point to the next data item. When it is advanced beyond the last data item, it will "point to a dummy" item that indicates the "end of all data items".

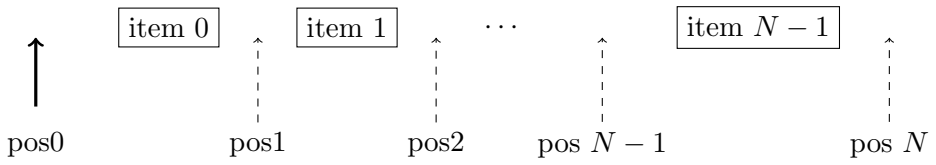

Figure 5: Conceptual depiction of Java iterators: they are like a text cursor, they stay **between** two adjacent data items. The thick arrow shows the current position of the iterator

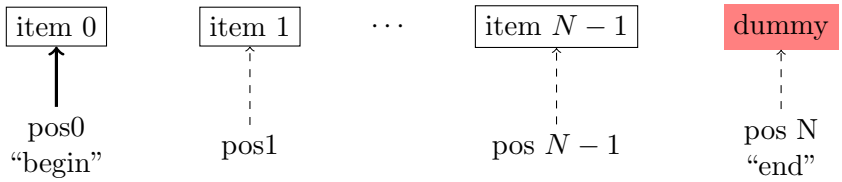

Figure 6: Conceptual depiction of C++ iterators: they are like a pointer, they point at the current object

In C++, a range of contiguous data items is identified by

- an **inclusive** marker "begin" that points to the first item in the range AND
- a **non-inclusive** marker "end" that "points" to a dummy last element

The common notation used for these two markers is borrowed from mathematics: `[begin,end)`.

6.2 Designing a C++ Iterator

Technically, an iterator is an object that keeps track of the current object of interest within a container. In general an iterator should support the following three main tasks: [11]

- **Retrieval**: access the current item stored in the container
- **Advance**: change the iterator position to advance to the next (or previous) data item
- **Compare**: whether two iterators point to the same data item

When an iterator is created, it has to be informed about the "current item", and it is the responsibility of the associated container to create and initialize a container to such a state.

For instance, if the container is a collection of words in a sequential text file, we can implement the iterator using an unsigned long integer variable to keep track of the offset of the current word relative to the beginning of the file. The **retrieve** action returns the word at the current offset and the **advance** action updates the unsigned long integer variable to the offset of the next word. If the collection is a linked list of nodes that store data items, we can implement the iterator using a pointer that keep tracks of the address of the current node. The **retrieve** action returns the data item stored at the current node, and the **advance** action changes the pointer to follow the next adjacent node.

Because the implementation details of an iterator depends on the characteristics of the associated container, it is common to implement the iterator class as an **inner class** of the associated container class. The container class itself must provide two functions (and the corresponding `const` variants of them) `begin()` and `end()` to instantiate two iterators that marks the beginning and the end of a range. Because an iterator mimics the behavior of C/C++ traditional pointers, it is a common practice to overload pointer related operators on an iterator class.

- Constructor(s) to place the iterator at the right position of the associated container.
- The **retrieval** task is implemented as C++ indirection/dereference (`operator*`. In order to support both read-only and read/write retrieval we overload both the mutable and non-mutable (`const`) versions
- The **advance** task is implemented as `operator ++` and `operator --` (both prefix and postfix variants)
- The **compare** task is obviously implemented as `operator ==` and `operator !=`

Table 2 shows the two inner classes of `arr_list` that implement the read-only (`ro_iter`) and read-write (`rw_iter`) iterators. All operations across the two classes, except for the **retrieval** (`operator*`) task, look the same.

By design, a "read/write" iterator should allow mutable operations to the data item associated with the iterator therefore `operator*` returns an alias of the data item pointed by the iterator. Also notice that in both classes, the prefix version of `operator++` returns an alias of the current iterator object while its postfix counterpart returns a **copy** of the current iterator object. A careful reader may also notice

[11] Out of the three tasks above, the compare operation is the easiest to implement.

so much code repetitions across both classes. This design can be further improved by deriving `rw_iter` as a child class of `ro_iter`. Listing 23 shows an improved version of the design, the `rw_iter` is derived from the `ro_iter` class. Using inheritance duplicate codes have been removed.

Read-only

```
44 /* R/O iterator inner class */
45 class ro_iter {
46 private:
47     E* ptr;
48 public:
49   ro_iter (E* _ptr) {
50     ptr = _ptr;
51   }
52   bool operator== (const ro_iter& rhs)
53       const {
54     return this->ptr == rhs.ptr;
55   }
56   bool operator!= (const ro_iter& rhs)
57       const {
58     return this->ptr != rhs.ptr;
59   }
60   ro_iter& operator++ () {// prefix
61     ++ptr;
62     return *this;
63   }
64   ro_iter operator++ (int) { // postfix
65     ro_iter old{*this};
66     ++ptr;
67     return old;
68   }
69
70   E operator * () const {
71     return *ptr;
72   }
73 };
```

Read-write

```
75 /* R/W iterator inner class */
76 class rw_iter {
77 private:
78   E* ptr;
79 public:
80   rw_iter (E* _ptr) {
81     ptr = _ptr;
82   }
83   bool operator == (const ro_iter& rhs)
84     const {
85     return this->ptr == rhs.ptr;
86   }
87   bool operator == (const rw_iter& rhs)
88     const {
89     return this->ptr == rhs.ptr;
90   }
91   bool operator != (const ro_iter& rhs)
92     const {
93     return this->ptr != rhs.ptr;
94   }
95   bool operator != (const rw_iter& rhs)
96     const {
97     return this->ptr != rhs.ptr;
98   }
99   rw_iter& operator ++ () { // prefix
100    ++ptr;
101    return *this;
102  }
103  rw_iter operator ++ (int) { // postfix
104    rw_iter old{*this};
105    ++ptr;
106    return old;
107  }
108  E& operator * () {
109    return *ptr;
110  }
111 };
```

Table 2: Two variants of `array_list` iterators.

```
/* iterator stuff */
class ro_iter {
protected:
    E* ptr;
public:
    ro_iter (E* _ptr) {
        ptr = _ptr;
    }
    bool operator == (const ro_iter& other) const {
        return this->ptr == other.ptr;
    }
    bool operator != (const ro_iter& other) const {
        return this->ptr != other.ptr;
    }
    ro_iter& operator ++ () { /* prefix */
        ++ptr;
        return *this;
    }
    ro_iter operator ++ (int unused);

    E operator * () const;
};

class rw_iter: public ro_iter {
public:
    rw_iter (E* _ptr) :
        ro_iter(_ptr)  /* invoke parent constructor */
    {
    }
    E& operator * () {
        return *this->ptr;
    }
};
```

Listing 23: Improved designed of `array_list` iterators. The top box shows the "read-only" variant and the bottom box the "read-write" variant.

7 STL Classes

The C++ support for defining template classes has enabled Alexander Stepanov and David Musser to implement generic programming in C++. Their research project began in late 1970s but the template library they developed was adopted into the C++ standard after over a decade later. Today, we refer to the library as the C++ Standard Template Library (STL). Readers interested in their research papers can visit http://www.stepanovpapers.com/.

The STL classes can be categorized into the four (interacting) components:

- **Containers/Collections** define a number of generic data structures for storing data items
- **Iterators** provide a mechanism to access elements in the data structures using generic pointer-like objects
- **Algorithms** define a number of generic functions for manipulating data items stored in a generic data structure. In general, an STL algorithm takes one or more containers as input and produces another container as output
- **Functors** provide simple procedure that wrapped as lightweight objects. These function objects (functors) work in conjunction with generic algorithms[12]

Java programmers may recall how the Java Collection Framework provides only the first two (Collections and Iterators) out of the four components above. The last two components give C++ programmers more tools that are not available to Java programmers.

7.1 STL Containers

Let's start with concepts that are already familiar to Java programmers: containers and iterators. The following table compares a sample of Java collections and their C++ counterparts

Java	C++
`java.util.ArrayList`	`std::vector`
`java.util.LinkedList`	`std::list`
`java.util.Stack`	`std::stack`
`java.util.Queue`	`std::queue`
`java.util.Deque`	`std::deque`
`java.util.TreeSet`	`std::set`
`java.util.HashSet`	`std::unordered_set`
`java.util.TreeMap`	`std::map`
`java.util.HashMap`	`std::unordered_map`

Table 3 compares Java ArrayList method and C++ vector functions.

[12]Readers with C programming background may have learned how to use pointers to a C function, a technique that enable passing a function as an argument to another function

Java ArrayList method	C++ vector function
add(E element)	push_back()
add(int pos, E element)	insert()
addAll()	Use std::copy() in <algorithm>
clear()	clear()
contains()	Use std::find() in <algorithm>
get()	at(), operator[]
indexOf()	Use std::find() in <algorithm>
isEmpty()	empty()
iterator()	begin(), cbegin()
lastIndexOf()	Use std::find() in <algorithm>
remove()	erase(), pop_back()
removeAll(Collection c)	Use std::set_difference() in <algorithm>
set (int index, E element)	operator[]
size()	size()
trimToSize()	shrink_to_fit()
	front(), back()
	data()

Table 3: Java ArrayList methods vs. C++ vector functions

7.1.1 Using std::find

The lack of contains(), indexOf(), and lastIndexOf() counterpart in C++ is covered by <algorithm>'s find function. In Listing 24, pos is 4 at line 11, but it is 0 at line 21.

```
1  #include <algorithm>
2  using namespace std;
3
4  int main() {
5      vector<string> lakes {"Huron", "Ontario", "Michigan",
6          "Erie", "Superior"};
7
8      // vector<string>::iterator iter;
9      auto iter = std::find (lakes.begin(), lakes.end(), "Superior");
10     if (iter != lakes.end()) {
11         int pos = iter - lakes.begin();   // Java indexOf
12         cout << "Found at index " << pos << endl;
13     }
14     else
15         cout << "Not found" << endl;
16
17     /* use reverse iterator */
18     vector<string>::reverse_iterator kter;
19     kter = std::find (lakes.rbegin(), lakes.rend(), "Superior");
20     if (kter != lakes.rend()) {
21         int pos = kter - lakes.rbegin();  // Java lastIndexOf
22         cout << "Found at index " << pos << endl;
23     }
24
25     return 0;
26 }
```

Listing 24: Using std::find

7.2 STL Iterators

A C++ container may provide up to four sets of iterators. Each set is comprised of the "begin" and "end" pair. When an iterator is initialized to its "begin" position, it points to the first element of the container. An iterator initialized to its "end" position points to a "dummy" item **beyond** the actual last item in the container. With this convention, the "begin" is an inclusive iterator, while "end" is an exclusive iterator.[13]

The four different sets allow us to manipulate the container in read/write or read-only mode and in forward or reverse direction. These four pairs are:

- `begin()` and `end()`: forward read/write
- `cbegin()` and `cend()`: forward read only
- `rbegin()` and `rend()`: reverse read/write
- `crbegin()` and `crend()`: reverse read only

Figure 7 shows the conceptual model of iterator implementation for a linked list. To allow **valid positioning** of the "end", a tail dummy is created into the list. Likewise, to allow valid position of the "reverse end" a head dummy is created into the list.

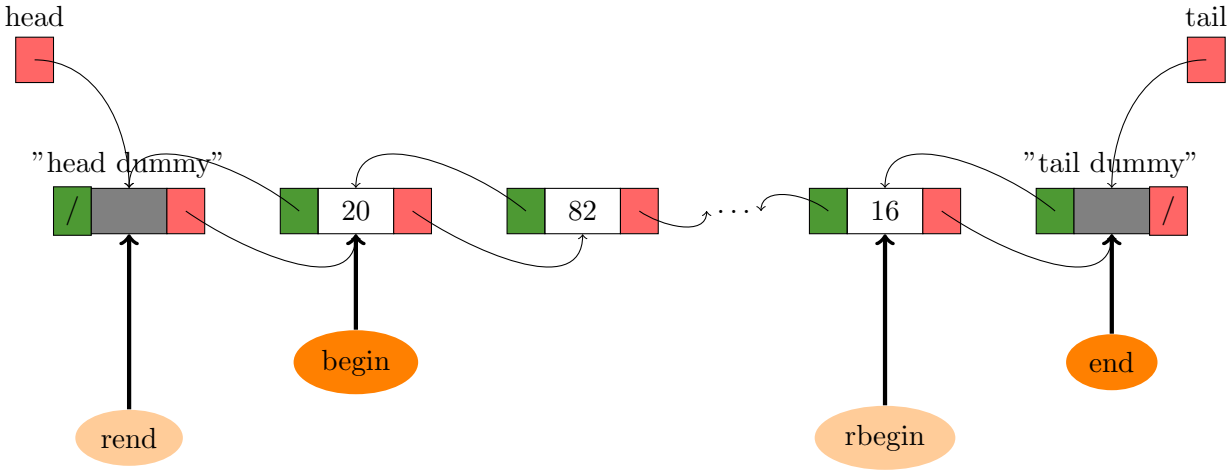

Figure 7: Linked List of integers: first data is 20, last data is 16

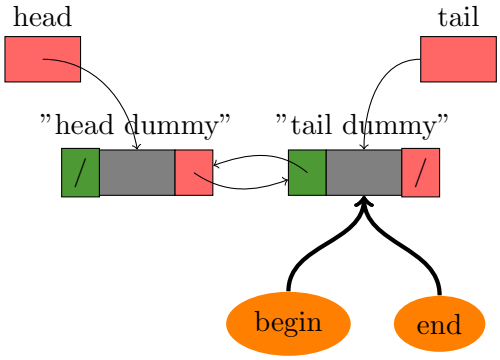

Figure 8: Position of iterators on an empty linked-list

[13] Mathematically this is equivalent to an interval whose one endpoint is close while the other end points is open.

Listing 25 shows short examples of using containers and iterators in both languages. The `auto` variable declaration is a C++11 feature that informs the compiler to infer the correct data type of a variable based on its surrounding context.

```
auto one = 250;         /* one is int */
auto two = 250.0;       /* two is double */
auto three = 250.0F;    /* three is float */
```

Without using `auto` declaration, the C++ variable `iter` in Listing 25 has to be declared

```
vector<Book>::iterator iter = libs.begin();
```

```java
// Java
import java.util.ArrayList;
public class Sample {
  public static void main (String[] _) {
    ArrayList<Book> libs;
    libs = new ArrayList<Book> ();

    libs.add (new Book("634927493-0"));
    libs.add (new Book("776123519-4"));
    /* more books */
    Iterator<String> iter;
    iter = libs.listIterator();
    while (iter.hasNext()) {
       String b = iter.next();
       System.out.println (b);
    }
  }
}
```

```cpp
// C++
#include <vector>
#include <iostream>
#include "Book.h"
using namespace std;

int main () {
  vector<Book> libs;
  vector<Book*> plibs;

  libs.push_back (Book("634927493-0"));
  libs.push_back (Book("776123519-4"));
  plibs.push_back (new Book("634927493-0"));
  plibs.push_back (new Book("776123519-4"));
  /* more books */

  auto iter = libs.begin();
  while (tier != libs.end()) {
    cout << (*tier) << endl;
    ++iter;
  }

  auto b_iter = plibs.begin();
  while (b_iter != plibs.end()) {
    /* notice the two indirections */
    cout << *(*b_iter) << endl;
    ++b_iter;
  }
  /* much easier using the for-range loop
   * and avoid double indirections */
  for (Book *bp : plibs)
    cout << *bp << endl;

  /* delete each book via its pointer */
  for (Book *bp : plibs)
    delete bp;
  return 0;
}
```

Listing 25: Using containers and iterators in Java and C++

Listing 25 also shows how to work with both vector of objects and vector of pointers to object. At line 19, the expression `*iter` dereferences the iterator to obtain a `Book`. However, at line 26, `*b_iter` evaluates to a **Book pointer**, therefore to obtain the actual book from its pointer, another indirection is required. The for-range loop at lines 35–36 is required to avoid memory leak. Without these, when `plibs` is destroyed, the book objects created at lines 13-14 **will not be destroyed** and remain on the heap. The two book objects created at line 11-12 are created on the stack, and they are automatically destroyed when the main function terminates, therefore it is **not necessary** to invoke `delete` on them.

The "reverse iterators" seem to be counter intuitive. Based on the conceptual model shown in Figure 7 Java programmers may expect to use `operator--` (similar to the `prev()` method in Java) to advance a reverse iterator towards its "reverse end". This is not the case for C++ iterators, reverse iterators are advanced using `operator++` as shown in Listing 26.

```
1 vector<float> numbers {30, 17, -4, 56, 81, 37};
2
3 /* the ++ advances the iterator closer to "rend" */
4 for (auto iter = numbers.rbegin(); iter != numbers.rend(); ++iter)
5 {
6     cout << *iter << " ";
7 }
```

Listing 26: Using Reverse Iterators

7.3 vector<bool>

The STL vector of boolean is implemented differently from other vectors. To save memory space, one boolean element is assigned a single bit of data. In addition to most functions that apply to vectors, `vector<bool>` provides the `flip()` function that can be applied either to **the entire vector** or **selected elements**.

```
vector<bool> checked {true, true, false, true};

checked.flip();    // flip all elements
checked[2].flip(); // flip selected element
```

7.4 push_back vs. emplace_back

The "emplace" variants of insertion operation is a convenient way of inserting objects into a container **without explicitly** creating an object. These functions are smart enough to select the right constructor to use when inserting new objects into the associated container. Listing 27 shows two `std::vector` containers. The first holds pairs of string and int, the second holds `Measurement` objects. Compare the invocation of `emplace_back` and `push_back` at lines 32, 33, 39, 40. At line 32 `emplace_back` is invoked with a string and an integer arguments. Using the two arguments, `emplace_back` automatically creates a `pair`. Compare that to `push_back` where the programmer is responsible for creating the pair herself. Likewise, at lines 39 and 40, `emplace_back` correctly selects the right constructor from the `Measurement` class.

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;
class Measurement {
private:
    float amount;
    string unit_name;
public:
    Measurement (float a, string name) :
        unit_name(name) // use init-list for non-primitive
    {
        amount = a;
    }
    Measurement (string uname) : unit_name(uname) {
        amount = 1.0;
    }
    string to_str() const {
        ostringstream text;
        text << amount << " " << unit_name;
        return text.str();
    }
};

int main() {
    vector<pair<string,int>> what;
    vector<Measurement> mms;

    /* emplace automatically creates an object */
    what.emplace_back("abc", 5);
    what.emplace_back("def", 7);
    what.push_back(make_pair("xyz", -4));

    for (auto x : what)
        cout << x.first << endl;

    /* emplace automatically selects the proper constructor */
    mms.emplace_back(20, "kg");
    mms.emplace_back("mile"); // invoke the second constructor
    mms.push_back(Measurement(30.5, "cu ft"));
    for (auto m : mms)
        cout << m.to_str() << endl;

    vector<Measurement> temp {{"asdf"},{"day"}, {20, "week"}};
    for (auto m : temp)
        cout << m.to_str() << endl;

    return 0;
}
```

Listing 27: `emplace_back` automatically infers the proper constructor from its arguments

7.5 Functors

In order to work with functors, we need a class that overload the function call operator. The following function class overloads the `operator()` function, hence any object of type `plus2` behaves like a univariate function (function of one variable).

```
class plus2 {
  public:
    int operator()    (int v) {
      return v + 2;
    }
};
```

When an object of type `plus2` is declared, and the function call is applied, the object will return an integer value:

```
plus2  adder;

cout << adder(5) << endl;   // output 7
```

The `operator()` function may take more than one parameters.

```
class Expander {
public:
  string operator()   (int count, const string& s) {
    ostringstream dest;
    for (int k = 0; k < count; k++)
      dest << s;
    return dest.str();
  }
};


Expander  expo;
cout << expo(4, "Go!");   // output: Go!Go!Go!Go!
```

A function class may also provide constructors, so its object can be initialized like any other object

```
class Repeater {
private:
  string text;
public:
  Repeater (const string& s) : text{s} {
    /* no additional code required */
  }

  string operator()   (int count) {
    ostringstream dest;
    for (int k = 0; k < count; k++)
      dest << s;
    return dest.str();
  }
};


Repeater  rep{"Go!};
cout << rep(3);   // output: Go!Go!Go
```

7.6 STL Functors and Algorithms

The STL algorithm classes offer a library of routines/functions for common manipulation tasks on containers. Some of these functions work together with builtin functors. In order to gain the benefit of both, we usually add the following to our C++ code:

```
#include <algorithm>
#include <functional>
```

For instance, the sort function can be invoked as follows

```
vector<float> values {-10.3, 43.2, 2.1, 22.5};

sort (values.begin(), values.end());   /* sort in ascending order */
sort (values.begin(), values.end(), greater<float>()  );   /* descending */
```

Listing 28: Controlling sort order using a builtin functor

The third argument passed in the second invocation of `sort` is a builtin function object that compares two values and return a boolean true when the first value is greater than the second. Because `greater` is a template function, it must be invoked with the data type of the vector element (`float` in the above the example). Had we used `vector<string>` the third argument to sort should have been written as `greater<string>()`.

The rest of the builtin functors are organized into the following categories:

- Arithmetic operations: `plus`, `minus`, `multiplies`, `divides`,`modulus`,`negate`
- Comparisons: `equal_to`, `not_equal_to`, `greater`, `less`,`greater_equal`,`less_equal`
- Logical operations: `logical_and`, `logical_or`, `logical_not`

- Bit operations: `bit_and`, `bit_or`, `bit_xor`

Within the sort algorithm, the function object `greater<T>` compares two arguments from the container. What if a certain task requires a comparison between the elements in the container and an externally supplied data? Listing 29 shows an example of using `find_if`. The builtin functor `equal_to` checks if `param1 == param2`. By using `std::bind` at line 12 we change this builtin functor to check if `"Donut" == placeholder::_1`, where the placeholder refers to each element of the container. The second (unused) functor at line 14 checks if `placeholder::_1 == "Donut"`.

```
1  #include <iostream>
2  #include <set>
3  #include <algorithm>
4  #include <functional>
5
6  using namespace std;
7
8  int main() {
9      set<string> sweets {"Cupcake", "Bubble Gum", "Caramel", "Donut", "Licorice"};
10
11     // std::bind the first arg to "donut"
12     auto donut_matcher = bind(equal_to<string>(), "Donut", placeholders::_1);
13     // std::bind the second arg to "donut"
14     auto donut_sniffer = bind(equal_to<string>(), placeholders::_1, "Donut");
15
16     auto pos = find(sweets.begin(), sweets.end(), "Donut");
17     pos = find_if(sweets.begin(), sweets.end(), donut_matcher);
18
19     return 0;
20 }
```

Listing 29: Binding a functor's argument

Function binding can be used for building functor compositions. Listing **??** shows an example of searching for a value x, $40 < x < 50$. The techniques used are:

- Create a functor `below50` by binding `less<T>` to 50
- Create another functor `above40` by binding `greater<T>` to 40
- Create a boolean functor that binds these two using `logical_and`

The overall expression is `logical_and (greater(x,40), less(x,50))`

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
    vector<int> vals {2, 8, 90, 44, 28, 17, 72};

    auto below50 = bind(less<int>(), placeholders::_1, 50);
    auto above40 = bind(greater<int>(), placeholders::_1, 40);
    auto between_40_50 = bind(logical_and<bool>(), below50, above40);

    auto pos = find_if(vals.begin(), vals.end(), between_40_50);
    cout << "Item found: " << *pos << endl;
    return 0;
}
```

Listing 30: Functor Compositions

8 Lambdas (in progress)

The introduction of lambda expressions in C++11 allows programmers to **declare** inline (and usually) anonymous functions. It is important to emphasize a lambda expression only **declares** a function, it does not execute it.

For instance, a lambda expression for an inline function that takes no parameters is shown at lines 6–8 below. This lambda declaration **will not** execute, because it is just a function **declaration** that never gets invoked.

```
1  #include <iostream>
2
3  int main() {
4    int x = 10;
5
6    [] {
7      std::cout << "Printed from an inline function" << endl;
8    }
9  }
```

```
1  #include <iostream>
2
3  int main() {
4    int x = 10;
5
6    auto lamb = [] {
7      std::cout << "Printed from an inline function" << endl;
8    }
9
10   std::cout << "Hello" << std::endl;
11   lamb();  /* invoke the lambda declaration */
12 }
```

TODO: Show a table of traditional function declaration vs its lambda counterpart.