```
            delete ref;
            free(ref_count);
            ref = NULL;
            ref_count = NULL;
        }
    }

    T * ref;
    unsigned * ref_count;
};
```

The code for this problem is complicated, and you probably wouldn't be expected to complete it flawlessly.

12.10 Malloc: Write an aligned malloc and free function that supports allocating memory such that the memory address returned is divisible by a specific power of two.

EXAMPLE

`align_malloc(1000,128)` will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

`aligned_free()` will free memory allocated by `align_malloc`.

pg 164

SOLUTION

Typically, with `malloc`, we do not have control over where the memory is allocated within the heap. We just get a pointer to a block of memory which could start at any memory address within the heap.

We need to work with these constraints by requesting enough memory that we can return a memory address which is divisible by the desired value.

Suppose we are requesting a 100-byte chunk of memory, and we want it to start at a memory address that is a multiple of 16. How much extra memory would we need to allocate to ensure that we can do so? We would need to allocate an extra 15 bytes. With these 15 bytes, plus another 100 bytes right after that sequence, we know that we would have a memory address divisible by 16 with space for 100 bytes.

We could then do something like:

```
1  void* aligned_malloc(size_t required_bytes, size_t alignment) {
2    int offset = alignment - 1;
3    void* p = (void*) malloc(required_bytes + offset);
4    void* q = (void*) (((size_t)(p) + offset) & ~(alignment - 1));
5    return q;
6  }
```

Line 4 is a bit tricky, so let's discuss it. Suppose `alignment` is 16. We know that one of the first 16 memory address in the block at p must be divisible by 16. With `(p + 15) & 11...10000` we advance as need to this address. ANDing the last four bits of p `+ 15` with `0000` guarantees that this new value will be divisible by 16 (either at the original p or in one of the following 15 addresses).

This solution is *almost* perfect, except for one big issue: how do we free the memory?

We've allocated an extra 15 bytes, in the above example, and we need to free them when we free the "real" memory.

We can do this by storing, in this "extra" memory, the address of where the full memory block begins. We will store this immediately before the aligned memory block. Of course, this means that we now need to allocate even *more* extra memory to ensure that we have enough space to store this pointer.

Therefore, to guarantee both an aligned address and space for this pointer, we will need to allocate an additional `alignment - 1 + sizeof(void*)` bytes.

The code below implements this approach.

```
1   void* aligned_malloc(size_t required_bytes, size_t alignment) {
2     void* p1; // initial block
3     void* p2; // aligned block inside initial block
4     int offset = alignment - 1 + sizeof(void*);
5     if ((p1 = (void*)malloc(required_bytes + offset)) == NULL) {
6       return NULL;
7     }
8     p2 = (void*)(((size_t)(p1) + offset) & ~(alignment - 1));
9     ((void **)p2)[-1] = p1;
10    return p2;
11  }
12
13  void aligned_free(void *p2) {
14    /* for consistency, we use the same names as aligned_malloc*/
15    void* p1 = ((void**)p2)[-1];
16    free(p1);
17  }
```

Let's look at the pointer arithmetic in lines 9 and 15. If we treat p2 as a `void**` (or an array of `void*`'s), we can just look at the index - 1 to retrieve p1.

In `aligned_free`, we take p2 as the same p2 returned from `aligned_malloc`. As before, we know that the value of p1 (which points to the beginning of the full memory block) was stored just before p2. By freeing p1, we deallocate the whole memory block.

12.11 2D Alloc: Write a function in C called `my2DAlloc` which allocates a two-dimensional array. Minimize the number of calls to `malloc` and make sure that the memory is accessible by the notation `arr[i][j]`.

pg 164

SOLUTION

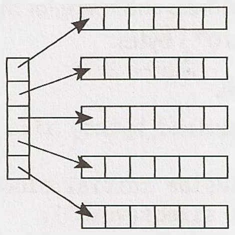
As you may know, a two-dimensional array is essentially an array of arrays. Since we use pointers with arrays, we can use double pointers to create a double array.

The basic idea is to create a one-dimensional array of pointers. Then, for each array index, we create a new one-dimensional array. This gives us a two-dimensional array that can be accessed via array indices.

The code below implements this.

```
int** my2DAlloc(int rows, int cols) {
  int** rowptr;
  int i;
  rowptr = (int**) malloc(rows * sizeof(int*));
  for (i = 0; i < rows; i++)  {
    rowptr[i] = (int*) malloc(cols * sizeof(int));
  }
  return rowptr;
}
```

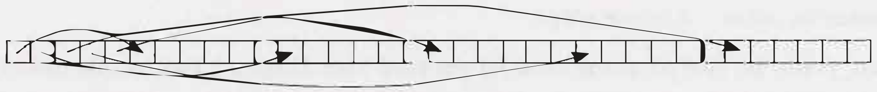
Observe how, in the above code, we've told `rowptr` where exactly each index should point. The following diagram represents how this memory is allocated.



To free this memory, we cannot simply call `free` on `rowptr`. We need to make sure to free not only the memory from the first `malloc` call, but also each subsequent call.

```
1   void my2DDealloc(int** rowptr, int rows) {
2     for (i = 0; i < rows; i++) {
3       free(rowptr[i]);
4     }
5     free(rowptr);
6   }
```

Rather than allocating the memory in many different blocks (one block for each row, plus one block to specify *where* each row is located), we can allocate this in a consecutive block of memory. Conceptually, for a two-dimensional array with five rows and six columns, this would look like the following.



If it seems strange to view the 2D array like this (and it probably does), remember that this is fundamentally no different than the first diagram. The only difference is that the memory is in a contiguous block, so our first five (in this example) elements point elsewhere in the same block of memory.

To implement this solution, we do the following.

```
1   int** my2DAlloc(int rows, int cols) {
2     int i;
3     int header = rows * sizeof(int*);
4     int data = rows * cols * sizeof(int);
5     int** rowptr = (int**)malloc(header + data);
6     if (rowptr == NULL) return NULL;
7
8     int* buf = (int*) (rowptr + rows);
9     for (i = 0; i < rows; i++) {
10      rowptr[i] = buf + i * cols;
11    }
12    return rowptr;
13  }
```

You should carefully observe what is happening on lines 11 through 13. If there are five rows of six columns each, `array[0]` will point to `array[5]`, `array[1]` will point to `array[11]`, and so on.

Then, when we actually call `array[1][3]`, the computer looks up `array[1]`, which is a pointer to another spot in memory—specifically, a pointer to `array[5]`. This element is treated as its own array, and we then get the third (zero-indexed) element from it.

Constructing the array in a single call to `malloc` has the added benefit of allowing disposal of the array with a single `free` call rather than using a special function to free the remaining data blocks.

13

Solutions to Java

13.1 Private Constructor: In terms of inheritance, what is the effect of keeping a constructor private?

pg 167

SOLUTION

Declaring a constructor `private` on class A means that you can only access the (private) constructor if you could also access A's private methods. Who, other than A, can access A's private methods and constructor? A's inner classes can. Additionally, if A is an inner class of Q, then Q's other inner classes can.

This has direct implications for inheritance, since a subclass calls its parent's constructor. The class A can be inherited, but only by its own or its parent's inner classes.

13.2 Return from Finally: In Java, does the finally block get executed if we insert a return statement inside the try block of a try-catch-finally?

pg 167

SOLUTION

Yes, it will get executed. The `finally` block gets executed when the `try` block exits. Even when we attempt to exit within the `try` block (via a `return` statement, a `continue` statement, a `break` statement or any exception), the `finally` block will still be executed.

Note that there are some cases in which the `finally` block will not get executed, such as the following:

- If the virtual machine exits during `try/catch` block execution.
- If the thread which is executing during the `try/catch` block gets killed.

13.3 Final, etc.: What is the difference between `final`, `finally`, and `finalize`?

pg 167

SOLUTIONS

Despite their similar sounding names, `final`, `finally` and `finalize` have very different purposes. To speak in very general terms, `final` is used to control whether a variable, method, or class is "changeable." The `finally` keyword is used in a `try/ catch` block to ensure that a segment of code is always executed. The `finalize()` method is called by the garbage collector once it determines that no more references exist.

Further detail on these keywords and methods is provided below.

final

The final statement has a different meaning depending on its context.

- When applied to a variable (primitive): The value of the variable cannot change.
- When applied to a variable (reference): The reference variable cannot point to any other object on the heap.
- When applied to a method: The method cannot be overridden.
- When applied to a class: The class cannot be subclassed.

finally keyword

There is an optional `finally` block after the `try` block or after the `catch` block. Statements in the `finally` block will always be executed, even if an exception is thrown (except if Java Virtual Machine exits from the `try` block). The `finally` block is often used to write the clean-up code. It will be executed after the `try` and `catch` blocks, but before control transfers back to its origin.

Watch how this plays out in the example below.

```
public static String lem() {
  System.out.println(“lem”);
  return “return from lem”;
}

public static String foo() {
  int x = 0;
  int y = 5;
  try {
    System.out.println(“start try”);
    int b = y / x;
    System.out.println(“end try”);
    return “returned from try”;
  } catch (Exception ex) {
    System.out.println(“catch”);
    return lem() + “ | returned from catch”;
  } finally {
    System.out.println(“finally”);
  }
}

public static void bar() {
  System.out.println(“start bar”);
  String v = foo();
  System.out.println(v);
  System.out.println(“end bar”);
}

public static void main(String[] args) {
  bar();
}
```

The output for this code is the following:

```
start bar
```

```
2  start try
3  catch
4  lem
5  finally
6  return from lem | returned from catch
7  end bar
```

Look carefully at lines 3 to 5 in the output. The `catch` block is fully executed (including the function call in the `return` statement), then the `finally` block, and then the function actually returns.

finalize()

The automatic garbage collector calls the `finalize()` method just before actually destroying the object. A class can therefore override the `finalize()` method from the `Object` class in order to define custom behavior during garbage collection.

```
protected void finalize() throws Throwable {
  /* Close open files, release resources, etc */
}
```

13.4 Generics vs. Templates: Explain the difference between templates in C++ and generics in Java.

pg 167

SOLUTION

Many programmers consider templates and generics to be essentially equivalent because both allow you to do something like `List<String>`. But, *how* each language does this, and *why*, varies significantly.

The implementation of Java generics is rooted in an idea of "type erasure." This technique eliminates the parameterized types when source code is translated to the Java Virtual Machine (JVM) byte code.

For example, suppose you have the Java code below:

```
Vector<String>  vector = new Vector<String>();
vector.add(new String("hello"));
String  str = vector.get(0);
```

During compilation, this code is re-written into:

```
Vector vector = new Vector();
vector.add(new String("hello"));
String  str = (String) vector.get(0);
```

The use of Java generics didn't really change much about our capabilities; it just made things a bit prettier. For this reason, Java generics are sometimes called "syntactic sugar."

This is quite different from C++. In C++, templates are essentially a glorified macro set, with the compiler creating a new copy of the template code for each type. Proof of this is in the fact that an instance of `MyClass<Foo>` will not share a static variable with `MyClass<Bar>`. Two instances of `MyClass<Foo>`, however, will share a static variable.

To illustrate this, consider the code below:

```
/*** MyClass.h ***/
template<class T> class MyClass {
 public:
  static int val;
  MyClass(int v) { val = v; }
};

```

```
/*** MyClass.cpp ***/
template<typename T>
int MyClass<T>::bar;

template class MyClass<Foo>;
template class MyClass<Bar>;

/*** main.cpp ***/
MyClass<Foo> * foo1 = new MyClass<Foo>(10);
MyClass<Foo> * foo2 = new MyClass<Foo>(15);
MyClass<Bar> * bar1 = new MyClass<Bar>(20);
MyClass<Bar> * bar2 = new MyClass<Bar>(35);

int f1 = foo1->val; // will equal 15
int f2 = foo2->val; // will equal 15
int b1 = bar1->val; // will equal 35
int b2 = bar2->val; // will equal 35
```

In Java, static variables are shared across instances of `MyClass`, regardless of the different type parameters.

Java generics and C++ templates have a number of other differences. These include:

- C++ templates can use primitive types, like `int`. Java cannot and must instead use `Integer`.
- In Java, you can restrict the template's type parameters to be of a certain type. For instance, you might use generics to implement a `CardDeck` and specify that the type parameter must extend from `CardGame`.
- In C++, the type parameter can be instantiated, whereas Java does not support this.
- In Java, the type parameter (i.e., the `Foo` in `MyClass<Foo>`) cannot be used for static methods and variables, since these would be shared between `MyClass<Foo>` and `MyClass<Bar>`. In C++, these classes are different, so the type parameter can be used for static methods and variables.
- In Java, all instances of `MyClass`, regardless of their type parameters, are the same type. The type parameters are erased at runtime. In C++, instances with different type parameters are different types.

Remember: Although Java generics and C++ templates look the same in many ways, they are very different.

13.5 TreeMap, HashMap, LinkedHashMap: Explain the differences between `TreeMap`, `HashMap`, and `LinkedHashMap`. Provide an example of when each one would be best.

pg 167

SOLUTION

All offer a key->value map and a way to iterate through the keys. The most important distinction between these classes is the time guarantees and the ordering of the keys.

- `HashMap` offers `O(1)` lookup and insertion. If you iterate through the keys, though, the ordering of the keys is essentially arbitrary. It is implemented by an array of linked lists.
- `TreeMap` offers `O(log N)` lookup and insertion. Keys are ordered, so if you need to iterate through the keys in sorted order, you can. This means that keys must implement the `Comparable` interface. `TreeMap` is implemented by a Red-Black Tree.
- `LinkedHashMap` offers `O(1)` lookup and insertion. Keys are ordered by their insertion order. It is implemented by doubly-linked buckets.

Imagine you passed an empty `TreeMap`, `HashMap`, and `LinkedHashMap` into the following function:

```
void insertAndPrint(AbstractMap<Integer, String> map) {
  int[] array = {1, -1, 0};
  for (int x : array) {
    map.put(x, Integer.toString(x));
  }

  for (int k : map.keySet()) {
    System.out.print(k + ", ");
  }
}
```

The output for each will look like the results below.

HashMap	LinkedHashMap	TreeMap
(any ordering)	{1, -1, 0}	{-1, 0, 1}

Very important: The output of `LinkedHashMap` and `TreeMap` must look like the above. For `HashMap`, the output was, in my own tests, `{0, 1, -1}`, but it could be any ordering. There is no *guarantee* on the ordering.

When might you need ordering in real life?

- Suppose you were creating a mapping of names to `Person` objects. You might want to periodically output the people in alphabetical order by name. A `TreeMap` lets you do this.
- A `TreeMap` also offers a way to, given a name, output the next 10 people. This could be useful for a "More" function in many applications.
- A `LinkedHashMap` is useful whenever you need the ordering of keys to match the ordering of insertion. This might be useful in a caching situation, when you want to delete the oldest item.

Generally, unless there is a reason not to, you would use `HashMap`. That is, if you need to get the keys back in insertion order, then use `LinkedHashMap`. If you need to get the keys back in their true/natural order, then use `TreeMap`. Otherwise, `HashMap` is probably best. It is typically faster and requires less overhead.

13.6 Object Reflection: Explain what object reflection is in Java and why it is useful.

pg 168

SOLUTION

Object Reflection is a feature in Java that provides a way to get reflective information about Java classes and objects, and perform operations such as:

1. Getting information about the methods and fields present inside the class at runtime.
2. Creating a new instance of a class.
3. Getting and setting the object fields directly by getting field reference, regardless of what the access modifier is.

The code below offers an example of object reflection.

```
/* Parameters */
Object[] doubleArgs = new Object[] { 4.2, 3.9 };

/* Get class */
Class rectangleDefinition = Class.forName("MyProj.Rectangle");

```

```
/* Equivalent: Rectangle rectangle = new Rectangle(4.2, 3.9); */
Class[] doubleArgsClass = new Class[] {double.class, double.class};
Constructor doubleArgsConstructor =
  rectangleDefinition.getConstructor(doubleArgsClass);
Rectangle rectangle = (Rectangle) doubleArgsConstructor.newInstance(doubleArgs);

/* Equivalent: Double area = rectangle.area(); */
Method m = rectangleDefinition.getDeclaredMethod("area");
Double area = (Double) m.invoke(rectangle);
```

This code does the equivalent of:

```
Rectangle rectangle = new Rectangle(4.2, 3.9);
Double area = rectangle.area();
```

Why Is Object Reflection Useful?

Of course, it doesn't seem very useful in the above example, but reflection can be very useful in some cases. Three main reasons are:

1. It can help you observe or manipulate the runtime behavior of applications.
2. It can help you debug or test programs, as you have direct access to methods, constructors, and fields.
3. You can call methods by name when you don't know the method in advance. For example, we may let the user pass in a class name, parameters for the constructor, and a method name. We can then use this information to create an object and call a method. Doing these operations without reflection would require a complex series of if-statements, if it's possible at all.

13.7 Lambda Expressions: There is a class `Country` that has methods `getContinent()` and `getPopulation()`. Write a function `int getPopulation(List<Country> countries, String continent)` that computes the total population of a given continent, given a list of all countries and the name of a continent.

pg 168

SOLUTION

This question really comes in two parts. First, we need to generate a list of the countries in North America. Then, we need to compute their total population.

Without lambda expressions, this is fairly straightforward to do.

```
int getPopulation(List<Country> countries, String continent) {
  int sum = 0;
  for (Country c : countries) {
    if (c.getContinent().equals(continent)) {
      sum += c.getPopulation();
    }
  }
  return sum;
}
```

To implement this with lambda expressions, let's break this up into multiple parts.

First, we use `filter` to get a list of the countries in the specified continent.

```
Stream<Country> northAmerica = countries.stream().filter(
  country -> { return country.getContinent().equals(continent);}
);
```

Second, we convert this into a list of populations using map.

```
Stream<Integer> populations = northAmerica.map(
   c -> c.getPopulation()
);
```

Third and finally, we compute the sum using reduce.

```
int population = populations.reduce(0, (a, b) -> a + b);
```

This function puts it all together.

```
int getPopulation(List<Country> countries, String continent) {
   /* Filter countries. */
   Stream<Country> sublist = countries.stream().filter(
      country -> { return country.getContinent().equals(continent);}
   );

   /* Convert to list of populations. */
   Stream<Integer> populations = sublist.map(
      c -> c.getPopulation()
   );

   /* Sum list. */
   int population = populations.reduce(0, (a, b) -> a + b);
   return population;
}
```

Alternatively, because of the nature of this specific problem, we can actually remove the `filter` entirely. The `reduce` operation can have logic that maps the population of countries not in the right continent to zero. The sum will effectively disregard countries not within `continent`.

```
int getPopulation(List<Country> countries, String continent) {
   Stream<Integer> populations = countries.stream().map(
      c -> c.getContinent().equals(continent) ? c.getPopulation() : 0);
   return populations.reduce(0, (a, b) -> a + b);
}
```

Lambda functions were new to Java 8, so if you don't recognize them, that's probably why. Now is a great time to learn about them, though!

13.8 Lambda Random: Using Lambda expressions, write a function `List<Integer> getRandomSubset(List<Integer> list)` that returns a random subset of arbitrary size. All subsets (including the empty set) should be equally likely to be chosen.

pg 439

SOLUTION

It's tempting to approach this problem by picking a subset size from 0 to N and then generating a random subset of that size.

That creates two issues:

1. We'd have to weight those probabilities. If N > 1, there are more subsets of size N/2 than there are of subsets of size N (of which there is always only one).
2. It's actually more difficult to generate a subset of a restricted size (e.g., specifically 10) than it is to generate a subset of any size.