

**De La Salle University • College of Computer Studies**

**Data Types**

Name (last name first) : Fernandez, Ryan Austin

Poblete, Clarisse Felicia M.

San Pedro, Marc Dominic

Tan, Johansson E.

Section : G01

Date of Submission : March 31, 2016

**1. What are the various collection types in your various programming languages?**

C++ - Lists, Stacks, Queues, Sets, Maps, Tuples

C# - Lists, Dictionaries, Stack, Queue, Sets, Iterators, Tuples

Python - list, dict, and tuple

Javascript – arrays and Objects

Scala – Lists, Sets, Maps, Tuples, Options, and Iterators

**2. Explain why there is a limitation in declaring very large arrays.**

Arrays require a contiguous segment of memory. Therefore, very large arrays require very large contiguous segments of memory, which may be difficult to allocate in stack due to the limitations of memory allocated by the operating system.

**3. Write a solution on the various languages on how such a large array can be created. Explain the rationale behind the solution.**

Since, based on our answer to the previous question, it is difficult to allocate large contiguous segments of memory, our solution is to divide the large array into chunks that can be stored in separate sections of the memory. By creating another array of pointers to these separate chunks, we can give the illusion of a single large array.

**C++**

int main()

{

vector<int> array(400000000);

array[399999999] = 24601;

cout << array[399999999];

return 0;

}

This is contiguous so accessing it does take time as a bigger number is given for the array. An alternative is this:

class BigArray

{

map<long, int \*> array;

int partitionSize;

public:

BigArray(long size)

{

partitionSize = 10000;

long nArrs = size / partitionSize;

long lastArr = size % partitionSize;

long i;

for (i = 0; i < nArrs; i++)

{

int\* newArray = new int[partitionSize];

array[i] = newArray;

}

if (lastArr != 0)

{

int\* newArray = new int[lastArr];

array[i] = newArray;

}

}

void set(long index, int value)

{

long mapIndex = index / partitionSize;

long offset = index % partitionSize;

int\* arr = array[mapIndex];

arr[offset] = value;

}

int get(long index)

{

long mapIndex = index / partitionSize;

long offset = index % partitionSize;

int\* arr = array[mapIndex];

return arr[offset];

}

};

int main()

{

long size = 400000000;

BigArray\* bigArray = new BigArray(size);

bigArray->set(size - 2, 24601);

cout << bigArray->get(size - 2);

return 0;

}

**C#**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BigArray

{

class BigArray

{

Dictionary<long, int[]> nums;

int partitionSize = 10000;

public BigArray(long size)

{

nums = new Dictionary<long,int[]>();

long nArrs = size / partitionSize;

long lastArr = size % partitionSize;

for (long i = 0; i < nArrs; i++)

{

nums.Add(i, new int[partitionSize]);

}

if (lastArr!=0)

{

nums.Add(nArrs, new int[lastArr]);

}

}

public void set(long index, int value)

{

long mapIndex = index / partitionSize;

long offset = index % partitionSize;

int[] arr = nums[mapIndex];

arr[offset] = value;

}

public int get(long index)

{

long mapIndex = index / partitionSize;

long offset = index % partitionSize;

int[] arr = nums[mapIndex];

return arr[offset];

}

}

}

// sample code

long size = 400000000;

BigArray ba = new BigArray(size);

Console.WriteLine(ba.get(size - 2)); // outputs 0

ba.set(size - 2, 24601);

Console.WriteLine(ba.get(size - 2)); // outputs 24601

**Python**

import math

class LargeArray:

class SubArray:

def \_\_init\_\_(self):

self.array = []

def \_\_getitem\_\_(self, index):

return self.array[index]

def \_\_setitem\_\_(self, index, value):

self.array[index] = value

def append(self, value):

self.array.append(value)

def \_\_init\_\_(self):

self.array = []

self.size = 0

def \_\_getitem\_\_(self, index):

return self.array[int(math.floor(index / 1000))][index % 1000]

def \_\_setitem\_\_(self, index, value):

self.array[int(math.floor(index / 1000))][index % 1000] = value

def append(self, value):

if self.size % 1000 == 0:

self.array.append(self.SubArray())

self.array[len(self.array) - 1].append(value)

self.size += 1

# Example use

# Instantiating array

lArray = LargeArray()

# Adding items to array

for i in range(0, 4200):

lArray.append(i + 1)

# Modifying items in array

for i in range(9, 4200, 10):

lArray[i] = lArray[i] \* 1000

# Getting items in array

for i in range(0, 100):

print str(lArray[i])

**Javascript**

function SubArray(size) {

this.array = [];

for(var i = 0; i < size; i++) {

this.array[i] = 0;

}

}

function LargeArray(size) {

this.array = [0,0,0];

this.size = size

this.chunks = Math.floor((size - 1) / 1000 + 1);

for(var i = 0; i < this.chunks; i++) {

if( i < this.chunks - 1 ) {

this.array[i] = new SubArray(1000);

} else {

this.array[i] = new SubArray(size % 1000);

}

}

this.set = function(position,value) {

this.array[Math.floor(position / 1000)]

.array[position % 1000] = value;

}

this.get = function(position) {

return this.array[Math.floor(position / 1000)]

.array[position % 1000];

}

}

**Scala**

class LargeArray(val size : Int) {

val ars = new Array[Array[Int]]((size - 1) / 1000 + 1)

var i = 0

while( i < ars.length ) {

if( i == ars.length - 1 ) {

ars(i) = new Array[Int](size % 1000);

} else {

ars(i) = new Array[Int](1000);

}

i += 1

}

def get(i : Int) : Int = ars(i / 1000)(i % 1000)

def set(i : Int, value : Int) {

ars(i / 1000)(i % 1000) = value

}

}

**4. What are the potential impact if you have a large temporary array declared in a recursive function? Discuss the impact if the variable resides statically, stack-based, or in the heap.**

For each call to the function, the environment creates a new frame. The parameters are copied to the new frame. If a temporary array is passed to the recursive calls, the same large array is repeatedly copied if it was declared in the stack. For static, it is no problem since the same value is being referenced. If the array is on the heap, only the reference is copied repeatedly.

**5. Explain the difference between Lists, Sets, Tuples, and Dictionaries.**

Lists are simple collections of data where the order does not matter. Sets are unordered collections where no duplicates are allowed. Tuples are ordered collections. Dictionaries map a set of keys to a set of values.

**6. Explain the Option/Optional Type.**

The Option is a collection in Scala that only contains zero or one elements. It could represent an actual value or a None value.

In C#, a function parameter p1 of a certain function f can be specified as optional to mean that a call to f may or may not necessarily supply a value to p1. Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used.

Optional parameters must be defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters.

Sample:

public void ExampleMethod(int required, string optionalstr = "default string",  
 int optionalint = 10)

Given this method signature, the following call

ExampleMethod(3, ,4);

will cause an error, since an argument was provided for the third parameter but not the second. Instead, if you want to override the default value of optionalInt, what should be done is:

ExampleMethod(3, optionalint: 4);

(Taken from https://msdn.microsoft.com/en-us/library/dd264739.aspx?f=255&MSPPError=-2147217396)

**7. What are Iterators? How do the different languages use iterators?**

An Iterator is a design pattern that abstracts moving through a list of objects.

**C++**

In C++, an iterator is used to iterate through a range of elements. The iterator is first pointed to some element on that range of elements. This following code uses an iterator through a vector of integers:

vector<int> intVector;

vector<int>::iterator vectorIterator;

intVector.push\_back(2);

intVector.push\_back(4);

intVector.push\_back(6);

intVector.push\_back(0);

intVector.push\_back(1);

for(vectorIterator = intVector.begin(); vectorIterator != intVector.end(); vectorIterator++)

cout<<\*vectorIterator<<" ";

In C++, iterators are classified into 5: Input, Output, Forward, Bidirectional and Random-Access.

Input iterators can only advance one at a time to the next element, is used for reading and can be compared to another iterator.

Output iterators can only advance one at a time to the next element and is used for storing.

Forward iterators are input and output iterators combined although they are can only move forward.

Bidirectional iterators are like forward iterators but can move backward.

Random-access iterators are like bidirectional iterators but they can do more. They can access ranges non-sequentially given an offset value. They also support inequality comparisons to other iterators. They also support += and -= as well as dereference using [].

**C#**

In C#, iterators are used to run through each element of a collection. An iterator method has a return type of IEnumerable, and makes use of the yield return statement to return individual elements, like in the following code:

static void Main()  
{  
 foreach (int number in EvenSequence(5, 18))  
 {  
 Console.Write(number.ToString() + " ");  
 }  
 // Output: 6 8 10 12 14 16 18  
 Console.ReadKey();  
}  
  
public static System.Collections.Generic.IEnumerable<int>  
 EvenSequence(int firstNumber, int lastNumber)  
{  
 // Yield even numbers in the range.  
 for (int number = firstNumber; number <= lastNumber; number++)  
 {  
 if (number % 2 == 0)  
 {  
 yield return number;  
 }  
 }  
}

You can also create an iterator for a custom class, by making it implement the IEnumerable interface, and writing a GetEnumerator() method. GetNumerator() serves as the iterator method for the class, and is called when the class is used in a foreach loop, as in the code below:

static void Main()  
{  
 DaysOfTheWeek days = new DaysOfTheWeek();  
  
 foreach (string day in days)  
 {  
 Console.Write(day + " ");  
 }  
 // Output: Sun Mon Tue Wed Thu Fri Sat  
 Console.ReadKey();  
}  
  
public class DaysOfTheWeek : IEnumerable  
{  
 private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };  
  
 public IEnumerator GetEnumerator()  
 {  
 for (int index = 0; index < days.Length; index++)  
 {  
 // Yield each day of the week.  
 yield return days[index];  
 }  
 }  
}

(taken from https://msdn.microsoft.com/en-us/library/dscyy5s0.aspx?f=255&MSPPError=-2147217396)

**Python**

Python iterators can be derived from an iterable object using the iter function. You can move through the elements of a python iterator using the next function.

Python also makes use of iteration and the iterators’ next function in its for loop.

**Javascript**

Javascript Iterators are not native to the language. You can create your own iterator with the same characteristics as the usual iterator.

**Scala**

Scala supports the usual hasNext() : boolean and next() : T methods, but it also supports min() : T and max() : T methods as well as the size() : int or length() : int methods. Scala can also concatenate iterators.

**8. What are various options in creating custom data types and complex types in the various programming languages?**

**C++**

C++ supports classes which could contain its own variables and functions. C++ also supports inheritance, even to the extent of multiple inheritance. C++ also supports interfaces which is implemented using abstract classes. At least one function in an abstract class should be declared as pure virtual function.

**C#**

C# allows users to create custom data types in the form of classes. C# also supports inheritance, abstract classes, and interfaces.

**Python**

Python supports classes which may contain their own local variables and function definitions.

**Javascript**

Javascript can express objects as “Functions” or as JSON formatted objects which maps keys to values, which may be objects as well.

**Scala**

Scala supports classes or objects. The main difference is that objects are singletons while classes can be instantiated.

**9. What are the difference between class and struct across the languages?**

**C++**

Struct in C++ has always has its data as public while class in C++ can be public, private or protected, depending on the programmer’s choice.

**C#**

The main difference between classes and structs in C# is that classes are reference types, while structs are value types. Also, structs are more limited in functionality as compared to classes. A struct cannot have a 0 parameter constructor, and it cannot extend other structs and/or classes. They can, however, implement interfaces.

**Python**

Python does not make use of structs. It only has support for classes.

**Javascript**

Javascript only supports the generic object, which acts as an associative array, and function definitions. It has no clear struct.

**Scala**

Scala has no struct.

**10. Differentiate between reference types and value types. When is the usage of each type ideal? Illustrate the difference by citing code and output differences.**

Reference types point to an allocated block of memory on the heap. Value types are either in a static or on the stack. Reference types are better for values that are transferred through multiple referencing environments i.e. function calls, recursive calls. Value types are better for global variables or for variables which are only used in a limited referencing environment.