

**Faculty of Engineering**

**Department of Electrical and Electronic Engineering**

**Name: Swaraj Dube**

**Student ID: 011114**

**Name of Module: Engineering software design and implementation**

**Code of Module: H63ESD**

**Module Convenor: Dr Lim Wee Gin**

**Effective Date: 22/4/16**

Table of contents

1. Approach to solving the problem…………………………………………………………………………………………………...Page 1
2. Design patterns review………………………………………………………………………………………………………………..Page 2-3
3. Singleton design pattern……………………………………………………………………………………………………..Page 2 - 3
4. Builder design pattern………………………………………………………………………………………………………..Page 4 – 5
5. Template…………………………………………………………………………………………………………………………………..Page 5 – 7
6. Data Structures review…………………………………………………………………………………………………………….Page 8 – 13
   1. List……………………………………………………………………………………………………………………………………….Page 8 -11
   2. Queue………………………………………………………………………………………………………………………………Page 11 - 13
7. Approach to solving the problem



The flowchart above shows how this program approaches this problem. First the cylindrical container parameters are added of which 75% of the volume is already filled with water. Two cubical sensors are to be inserted into the container and in order to make the program flexible, the program prompts the user to enter parameters for two sensor which means both the sensors need not be of the same dimensions. Upon insertion of the sensors inside the container, if there is no overflow then and the sensors successfully manage to fit inside the container, the program calculates the number of stones that could be fit in using the following formula:

The reason why the user is prompted only once to enter the stone volume is because if there are stones each with different dimensions to be put inside the container, the program would be able to calculate the maximum number of stones that can be put inside however the program will not be able to differentiate between the maximum number of stones for each cylindrical stone hence the program is designed in a manner such that the user can input parameters of the stone only once.

Page 1

1. Design patterns review
   1. Singleton design pattern

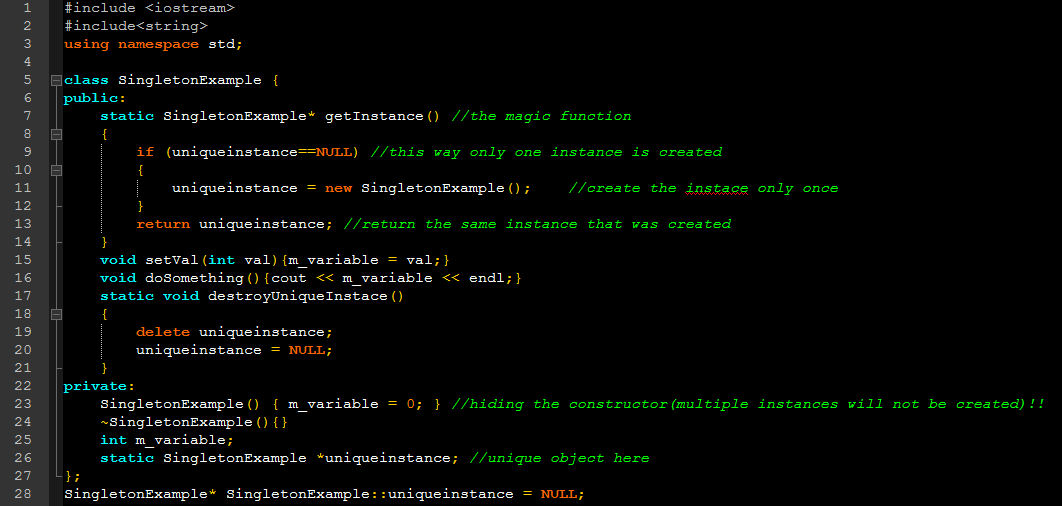
This design pattern is one of the most commonly used pattern in software engineering. The specialty of this pattern is that it creates only one instance of a class. This can come in handy when only one object is required across the entire program. However this pattern has caused some level of controversy as a number of programmers think this pattern is used in many cases where the pattern is not beneficial and the biggest problem is that it introduces global state into the problem which is opposing encapsulation.

Some notes on singleton:

* Another interesting note here is that other design patterns such as “abstract factory”, “builder” and “prototype” also use singletons.
* Static objects are often singletons.
* They are preferred to global variables.

How the singleton works

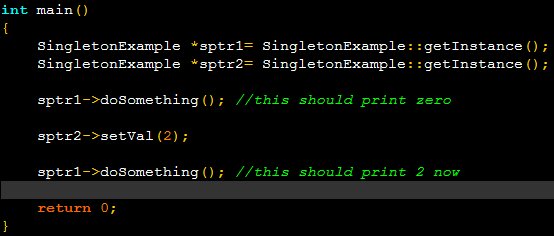
Firstly, this patterns needs a way to access the singleton class’ members without creating a class object and using the same class instance even though multiple pointer objects have been created of that class. The pattern is created using a static function that creates a new instance of the class if it does not exist and once it exists then a new instance can never be created if this function is called again because an instance has already been created and the function will not create another instance because the condition is that for a new instance to be create there must be no instances present. If the instance is already present the function simply return the reference to that object. Another integral feature that make this design pattern singleton is that the constructor of the class is hidden i.e. it is made private to make sure that the object cannot be instantiated in any other way. Another special feature about this pattern is that it can be lazily constructed which mean that no memory or resources are required until there is a requirement.

Example of singleton design pattern

Page 2

The figure shown above shows a singleton class named “SingletonExample”. It can be noted that the constructor of the class is privatized which prevents multiple instances. At line 26 a static pointer variable has been created that returns a pointer to “SingletonExample”, this variable is used to store the unique instance of the class.

The getInstance() function at line 7 is the main function of this class, which is what creates a unique instance of the class as it can be seen in the function body that if an only if “uniqueinstance” is null a new instance is created and once this instance is created, there is now way to make it null unless destroyUniqueInstance() method is called else not matter how many times getInstance() function is called , it returns the same instance that was created and that is the beauty of this design pattern. If suppose the main program looks like this



As it can be seen that two pointer variables have been used however there is only one instance hence any modifications made using one pointer variable can also be accessed by the other pointer variable.

Finally some advantages and disadvantages of using this design pattern

Advantages of Singleton design pattern

* Lazy Instantiation

This amazing thing about this feature is that singleton variable is not allocated memory until and unless the function to return this reference is called. This way the program capitalizes on saving and utilizing memory.

* Static initialization

In this type of initialization, memory is allocated to the singleton variable only when it is declared and the main advantage of this type of initialization is that there is no need to use any synchronization constructs.

Disadvantages of Singleton design pattern

Singleton generally takes over static classes which has the following disadvantages:

* One cannot specify any creation logic with static methods
* Grants global stage to program which opposes encapsulation
* Singleton classes cannot be sub classed
* A real singleton class is not easily extensible.
* Inheritance causes problems because any child class inheriting from the parent singleton class will create multiple instances of the single class which violates singleton principles as discussed above.

As for this program the singleton design patterns have been used for the following classes: “CheckFit.h”, “Shape.h”, “ShapeBuilder.h”.

Page 3

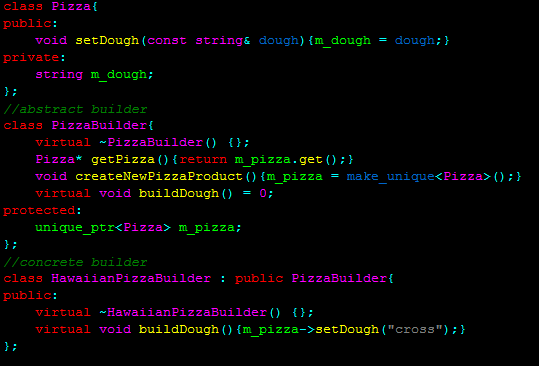
* 1. Builder design pattern

This is yet another extremely useful design pattern as this pattern is an object creation design pattern. This pattern has the ability to use the same construction process for creating different object representations. The way this pattern works is quite simple. Firstly there is an abstract base class and a concrete base class. The abstract base class is responsible for creating the standard construction process and the concrete derived class defines the implementation.

Some of the advantages of this pattern are:

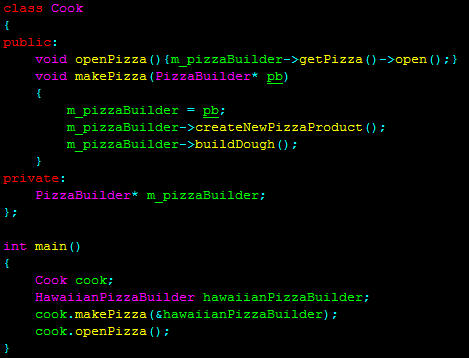
* Encapsulation of code.
* Can vary internal representations.
* Creation of multiple object representation using one construction process.

The main disadvantage of this design pattern is that one needs to make a new builder class for each different type of product.

Example:

As it can be seen from the figure above is that the class ‘PizzaBuilder’ is the abstract builder class which creates the construction process through the ‘unique\_ptr’ object, to understand the role of these classes we’ll look at the following figure on the next page. This program that is designed for Mr.Takahashi also uses the builder design pattern in the class “SetShape.h” where the concrete builder class is “ShapeBuilder.h”.

Page 4



The child class here is the ‘HawaiianPizzaBuilder’ class and it’s object’s reference is passed to the makePizza() function and the argument in the function is of type ‘PizzaBuilder’ which is the parent class which is valid and now the m\_pizzaBilder pointer variable point to the HawaiianPizzaBuilder class where the ‘unique\_ptr’ object creates a unique instance of this very object and that is how this pattern just uses one construction process to build unique instances of each object. Suppose if the class ‘PizzaBuilder’ has another child class then its unique instance would also be created in the same manner. It can be seen how important of a role the abstract class is playing here alongside unique\_ptr.

1. Templates

Templates are one of the greatest features in C++ that promote software reusability. They are a good way to make the code more reusable, it becomes a great tool when inheritance and operator overloading is involved. Generally there are two types of templates which are function templates and class templates. Function templates and class templates enable you to specify only with a single code segment, an entire range of specific functions or even classes. Even the STL (Standard Template Library) which is an integral part of algorithms uses special functions that are within a framework of connected templates.

In general there is an absolute need for writing templates especially when the program is bound to be very big because in case of software reuse it can come in very handy, if the program is huge and there needs to be a modification to the program then in some cases this can become very tiresome in absence of templates.

However to use or not use templates requires a lot of decision making. Listed below are some important advantages of using templates:

* Reduces repetition of code
* Provides static polymorphism
* Promotes flexibility, reusability , easier to implement changes\*
* Easier to write templates
* There is need to only create one generic version of the class or function instead of manually creating specializations.
* Templates are type safe.

Page 5

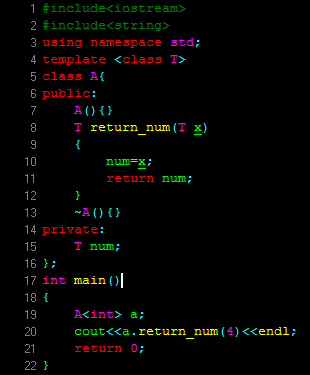
The following is the syntax for templates:

“template <class datatypeparameter>”

Or

“template <typename datatypeparameter >”

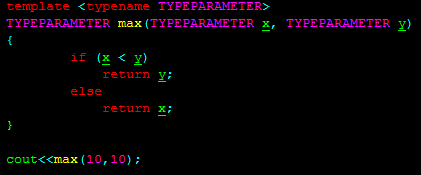
The best way to understand the power of templates if by understanding how it impacts on the use of data types and by observing the following example:

Consider this example:

Though this program is very small it can be seen how using templates here promote flexibility and reusability, this is a class template to be much more specific. It can be seen that the number 4 is passed in integer format to the function return\_num() which returns this number in integer format. Now suppose someone else wants this function to pass the number into this function in ‘double’ format and also return of type ‘double’, without templates if this had to be done is replace all the ‘T’ with double which consumes a lot of time and is not a good practice in this case, with templates all that needs to be done is specify the template parameter during object creation and all the ‘T’ are now replaced with the type specified during object creation. In this case the ‘T’ becomes integer and when type double is required as parameters and return type then the template is specified as double during object creation and all the rest is done.

There’s also function templates on the other hand which are functions that can input as parameters data of any type, the only thing that needs to be done is to specify the type of data while calling the function for example:

Page 6



In this case, two integers are being passed into the max() function and hence ‘TYPEPARAMETER’ becomes and integer and if parameters that were being passed into this function were of type double then ‘TYPEPARAMETER’ would become double and this simply shows us the amount of flexibility templates have to offer.

This is one of the main reason why templates have been used in this program designed for Mr. Takahashi as it can be seen in the program, there is a heavy use of templates. The template arguments are defined in the main file from line 19-22 and each file has class templates. The program has 6 main classes and each class has a lot of members due to the fact that the program is dealing with shapes (volume, area, length, width, height etc.), if observed carefully, the data type of the members depend on the template arguments in the main file as the program requires the output

to be calculated should be in integer format hence integer template arguments are used and if the output were to be of type double then the template arguments needs to be double, there is absolutely no need to change anything in the class templates and now suppose if templates were not to be used then the data type would be needed to be specified in every class and again if the requirements were changed then again new data types would be required and that too in 6 classes and even worse if the project contained more classes which is why templates especially in this project greatly promotes flexibility.

Templates however are not all positives, there are a lot of drawbacks with templates which are listed as follows:

* Historically compilers have a poor support for macros and templates were not widely used in the back.

This means that the code could be made less portable across compilers.

* Hard to debug and understand

What this means is that if the code were to be run using different compilers and if errors were to be produced then each compiler could output a different error message and in general templates produce a lot of unnecessary messages even though the error is present at only one line.

* Code bloating

This is major concern amongst many programmers and this code bloating occurs especially in huge programs that heavily rely on templates because each time a template is used could cause the compiler to generate an extra instantiation of the template and given that there are many templates uses in the program could very well cause a code bloat.

Page 7

1. Data structures review

4.1 List

A list just like an array is collection of elements, several operations can be performed on a list mainly which are add, search, delete however they have a much greater advantage over arrays as lists are dynamic data structures which means that the size of the list need not be determined during compile time, the size of the list can grow or shrink during runtime. Let’s dig into the theory behind lists.

Firstly list is a linear data structure which means that the elements in the list have some sort of linkage with other elements and not are randomly stored in memory. Each element in a list is known as a node and each node points to the next node by means of pointers. The following figure shows how a node looks like:

Data address

A node has two parts: one part contains the data and the other part contains the address to the next node which is how each node is linked to the next via pointer representations. If we this think about this carefully then we can deduce that the last node in the list has an address of ‘NULL’ as it is not pointing to anything because it is literally the last element in the list.

What makes this data structure interesting is that a pointer to the head of the list is used to access the link which is what helps make programmers the following properties about the list:

* Slow search of order O(n)
* Fast traversing
* Fast insertion and removal at any place in the list
* No random access

The reason for a slow search is because if a list has 10,000 elements and it is required to search the 9,999th element, the search begins at the head of the list which is the 1st element and the search process repeats for 10,000 times till it finally finds the 9,999th element. It can be seen how much time is wasted in search here which is why the search time is of order O(n) which mean linear search time which is slow. The other reason is because each node in the list not only contains the address to the next node but also to the previous node which instantly means more memory requirements which also implies more cache misses.

The reason why traversing is fast is because each node contains the address to the next node which mean each node is connected to the next node by containing the address to the next link and due to this very linkage due to the pointer representations, the traversal time becomes faster because the compiler does not need to search for various elements in different parts of the memory like is does of unordered data structures.

Lists can also be of different types:

* Single lists where each node only points to the next node.
* Double lists where each node points both to the next and the previous node.
* Circular lists where the end of the node points to the head of the list.

Some of the applications of linked lists are:

* Representation of a sparse matrix
* Dynamic memory management in allocating and releasing memory at run time
* Microsoft image viewer
* Implementation of other data structures such as stacks, queues, graphs and trees

A sample code is shown on the next page that shows how a list is implemented:

Page 8

#include<iostream>

#include<string>

using namespace std;

class list {

public:

list();

void insert\_at\_beg(string n, double m);

void insert\_at\_end(string n, double m);

void insert\_at\_specific\_loc(int loc, string n, double m);

void del(string n);

void show();

~list();

private:

struct node {

string name;

double mark;

node \*link;

}\*p;

};

list::list()

{

p = NULL;

}

void list::insert\_at\_beg(string n, double m)

{

node \*temp = NULL;

temp = new node;

temp->name = n;

temp->mark = m;

temp->link = p;

p = temp;

}

void list::insert\_at\_end(string n, double m)

{

node \*temp = NULL, \*r = NULL;

if (p == NULL)

{

temp = new node;

temp->name = n;

temp->mark = m;

temp->link = NULL;

p = temp;

}

else

{

temp = p;

while (temp->link != NULL)

{

temp = temp->link;

}

r = new node;

r->name = n;

r->mark = m;

r->link = NULL;

temp->link = r;

}

}

void list::insert\_at\_specific\_loc(int loc, string n, double m)

{

node \*temp = p, \*r = NULL;

for (nti = 0; i<loc; i++)

{

if (temp == NULL)

{

cout << “location not found” << endl;

return;

}

temp = temp->link;

}

r = new node;

r->name = n;

r->mark = m;

r->link = temp->link;

temp->link = r;

}

void list::del(string n)

{

node \*temp = p, \*old = NULL;

while (temp != NULL)

{

if (temp->name == n)

{

if (temp == p)

{

old = p->link;

delete p;

p = old;

return;

}

else

{

old->link = temp->link;

delete temp;

return;

}

}

else

{

old = temp;

temp = temp->link;

}

}

}

void list::show()

{

node \*temp = p;

while (temp != NULL)

{

cout << “name is: “ << temp->name << “ and marks are: “ << temp->mark << endl;

temp = temp->link;

}

}

list::~list()

{

node \*q = NULL;

while (p != NULL)

{

q = p->link;

delete p;

p = q;

}

}

int main()

{

list l;

l.insert\_at\_beg(“swaraj”, 81);

l.insert\_at\_beg(“tiancheng”, 94);

l.insert\_at\_end(“david”, 98);

l.show();

l.del(“tiancheng”);

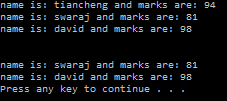
cout << “\n\n”;

l.show();

return 0;

}

Page 10

The output of the following program would be:

The output of this program itself helps us understand about lists. The function insert\_at\_beg() inserts elements from the head of the list as it is inserted from the front which means the first element to be entered will be the first element and the next time an element is pushed back this elements becomes the head the previous first element now becomes the 2nd node which is what can be seen from the output of the program. This however is not the case when inserting from the back, every time a new element is inserted from the back, the new element keeps on appending.

4.2 Queue

Part of the STL library, a queue is a FIFO data structure (First In First Out) which means the first element to be inserted is the first element to be removed. The best way to think about a queue is a real life queue which is as follows:

The first person waiting in the queue is the first person to leave as well and that is exactly how this data structure works.

This means that elements can only be inserted at the back and removed from the front. This data structure is very much similar to stack which has restriction on where to add and remove elements. However this data structure is not as simple to implement as the stack because in the stack when an element is popped it just goes i.e. the rest of the elements stay in their position like a pile of coins whereas in a queue, if a person moves out of the queue then the 2nd person in line must fill that position and the 3rd person in line must fill up the 2nd position which means the entire que must move forward. Thinking about this in terms of programming, when an element in popped, all the other elements must move from their position s step forward and this way the queue becomes very slow and thus the real challenge becomes to make the queue fast which has various ways of implementation that depends on the programmer. A queue grows at the rear and shrinks from the front.

The queue has the following properties:

* Fast insertion at the end
* Fast removal at the front
* Slow search of order O(n)
* Fast traversal

The following page shows a sample code of how a queue is implemented and manipulated.

Page 11

#include<iostream>

#include<string>

using namespace std;

class queue {

public:

queue();

void insert(char item);

void del(); //in queues we can only delete the first node in the queue

void show();

~queue();

private:

struct node {

char data;

node \*link;

}\*front, \*rear;

};

queue::queue() { front = rear = NULL; }

void queue::insert(char item)

{

node \*temp = NULL;

temp = new node;

if (temp == NULL) //queue is full

{

cout << "the queue is full\n";

return;

}

temp->data = item; //store data

temp->link = NULL;

if (front == NULL) //if no elements are found in the list, first element is both rear and front

{

rear = front = temp;

return;

}

rear->link = temp;

rear = temp;

}

void queue::del()

{

node \*temp = front;

if (front == NULL)

{

cout << "queue is empty\n";

return;

}

front = temp->link; //assign the next element as the head

delete temp; //delete the previous head

}

void queue::show()

{

node \*temp = front;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->link; //point to the next node to print that element

}

}

queue::~queue()

{

node \*q = NULL;

while (front->link != NULL)

{

q = front->link;

delete front; //delete the nodes in the queue

front = q;

}

delete front; //delete the final node with null address

}

int main()

{

queue q;

q.insert('A');

q.insert('B');

q.insert('C');

q.show();

q.del();

cout << "\n\nafter deletion \n\n";

q.show();

return 0;

}

This will be the program output:

A B C

B C

At first A, B and C are inserted but once the del() function is called it pops the head element which in this case is A and make B the head. The pointer variable ‘front’ points to the first node in the queue and the pointer variable ‘rear’ points to the last node in the queue.

Most of these properties are somewhat similar to those of the list which include slow search or fast traversals which have been explained why discussing the list. Queues can be very useful in computer science applications such as:

* Scheduling of CPU , disk
* Resource sharing like printers which use print queue
* Asynchronous data transfer between process like IO buffers, pipes

If we observe the applications carefully, it can be noted that all these tasks are FIFO oriented for example a print queue, the first printing job will be the first one to get printed or scheduling the CPU where in the first task to be scheduled will be performed.

The program designed for Mr. takahashi uses vectors as the data structure in the main file. The benefit of using vector in this program is that is separately stores object properties that includes volume, area, length, width, height. The shapes also need to be fit in a chronological order inside the vector for example first is the cylinder container, the next are the sensor modules and the third are the stones which are inserted very quickly in the vector due to its properties. The random operator also makes it very easy to get the objects’ properties and compare them with each other to check for overflow or an oversized object that is attempted to be fit inside.

Page 13