Part B – Foundations

Dynamic Memory

Workshop 2
v0.95 (Submission Policy standards updated)

In this workshop, you will use *references* to modify content of variables in other scopes, overload functions and allocate memory at run-time and deallocate that memory when it is no longer required. We will be doing so via the use of a **struct** that represents a **Gift**.

Learning Outcomes

Upon successful completion of this workshop, you will have demonstrated the abilities to:

- allocate and deallocate dynamic memory for an array;
- allocate and deallocate dynamic memory for a single variable;
- incorporate dynamic memory along with user input;
- overload functions;
- create and use references;

Submission Policy

The workshop is divided into 2 sections;
in-lab - 50% of the total mark

To be completed before the end of the lab period and submitted from the lab.

DIY - 50% of the total mark

To be completed within **5 days** after the day of your lab (meaning it will be **due 2 days prior to following lab period for the next Workshop**).

The *in-lab* section is to be completed after the workshop is published, and before the end of the lab session. The *in-lab* is to be submitted during the workshop period from the lab.

If you attend the lab period and cannot complete the *in-lab* portion of the workshop during that period, ask your instructor for permission to complete the *in-lab* portion after the period. You must be present at the lab in order to get credit for the *in-lab* portion.

If you do not attend the workshop, you can submit the *in-lab* section along with your *DIY* section (see Submission Penalties below). The *DIY* portion of the lab is due on the day that is **5 days** after your scheduled *in-lab* workshop (by 23:59 or 11:59PM) (even if that day is a holiday).

The *DIY* (Do It Yourself) section of the workshop is a task that utilizes the concepts you have done in the **in-lab** section. This section is open ended with no detailed instructions other than the required outcome.

All your work (all the files you create or modify) must contain your **name, Seneca email and student number**.

You are responsible to back up your work regularly.

Ask your professor if there are any additional requirements for your specific section.

Reflection, Citation and Sources

After the workshop fully is completed (meaning the **in-lab** was already submitted and the **diy** is ready to be submitted), create a text file named **reflect.txt** that contains your detailed description of the topics that you have learned in completing this particular workshop and mention any issues that caused you difficulty and how you solved them. Add any other comments you wish to make.

There is a section below prior to the **diy submission procedure** that provides some questions and expectations you should aim to answer at a minimum for the reflection. Aiming higher than that will be to your benefit as a poorly done reflection can incur a **mark reduction** (see Submission Penalties below).

This reflection is a mandatory part of the workshop submission.

Also, when submitting Workshops, Project and assignment deliverables, a file called **sources.txt** must be present. This file will be submitted with your work automatically.

You are to write either of the following statements in the file "**sources.txt**":

I have done all the coding by myself and only copied the code that my professor provided to complete my workshops and assignments.

Then add your name and your student number as signature

OR:

Write exactly which part of the code of the workshops or the assignment are given to you as help and who gave it to you or which source you received it from.

You need to mention the workshop name or assignment name and also mention the file name and the parts in which you received the code for help.

Finally add your name and student number as signature.

By doing this you will only lose the mark for the parts you got help for, and the person helping you will be clear of any wrong doing.

SUBMISSION PENALTIES:

The following are penalties associated with the workshop if the submission policies above regarding due dates and other requirements are not met. It is advised to pay close attention to these penalties in order to avoid them.

- If the **in-lab** is submitted past the due date but before the rejection date, it will be **worth half of its total weight** for the Workshop. If an in-lab is worth 50% then it will be worth 25%.
- If the **diy** is submitted past the due date but before the rejection date, it will be **worth half of its total weight** for the Workshop. If the diy is worth 50% then it will be worth 25%.

- If a submitted **reflection** (reflect.txt) is deemed to be insufficient in depth or does not demonstrate a strong understanding of the core concepts used in the Workshop, a **potential maximum reduction of 30%** may be applied the overall mark of the Workshop.
- If any of in-lab, diy or reflection is missing the total mark of the workshop will be **zero**.

WORKSHOP DUE DATES

You can see the exact due dates of all assignments by adding -due after the submission command:

Run the following script from your account (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

```
~profname.proflastname/submit 244/NXX/WS02/in_lab -due<ENTER>
~profname.proflastname/submit 244/NXX/WS02/DIY -due<ENTER>
```

COMPILING AND TESTING YOUR PROGRAM

All your code should be compiled using this command on matrix:

```
g++ -Wall -std=c++11 -o ws (followed by your .cpp files)
```

After compiling and testing your code, run your program as follows to check for possible memory leaks: (assuming your executable name is "ws")

```
valgrind ws <ENTER>
```

IN-LAB - 50%

Create an empty module called **Gift** to represent gifts to be given on any special occasion.

> **Reminder:**
> To create an empty module called Gift:
> Create a header file called **Gift.h**, add the compilation safeguards (review Workshop 1 for details) and the sdds namespace. Then create a source file called **Gift.cpp**, include **Gift.h**. The code you create during this workshop should be enclosed / live in the sdds namespace. Additionally, include any

standard libraries you might need in Gift.cpp (eg <iostream>) as you develop the implementation.

Inside the Gift header file and `sdds` namespace create two **constant** variables:

`MAX_DESC`: This integer defines the maximum length of the Gift's description. The value should be 15.

`MAX_PRICE`: This double defines the maximum price of the Gift. The value should be 999.99.

Design and code a structure (struct) named `Gift` in the namespace `sdds`. The struct's definition should be placed in **the header file**.

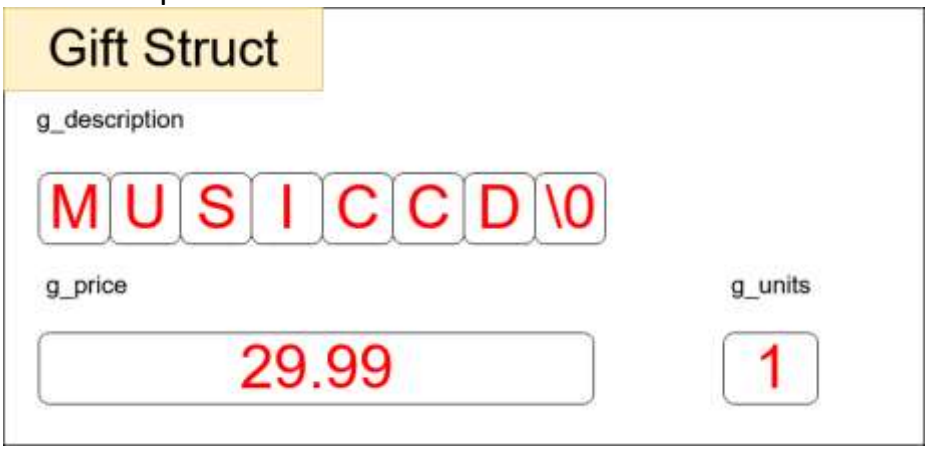
Your `Gift` structure should have the following data members:

`g_description`: A statically allocated array of **characters** that will store the description of the Gift. The size for this array will be based on MAX_DESC defined above. Remember to include additional space for the nullbyte.

`g_price`: A **double** that will hold the price of the Gift. At minimum the price can be 0 and the maximum will be based on MAX_PRICE.

`g_units`: A **integer** that will hold the number of units/copies of the Gift. For example, **2** concert-tickets, **3** gift-cards or **1** cake. The number of units has to be at least **1**.

A visual representation of an instance of the Gift struct could look like this:



Here we have a Gift that is a “MUSICCD” where the price is 29.99 and there is one copy/unit of it. Note that as the description is represented by a c-style character array we have a nullbyte “**\0**” at the last index.

GIFTING(……) OVERLOADS AND DISPLAY FUNCTION

Add three **overloads** (functions with the same name but different signature) of a **gifting** function to the **Gift** module in `sdds` namespace. The prototypes should go in the **Gift.h** header file and the implementation in the **Gift.cpp** file.

```
void gifting(char*)
```

This function returns nothing (void) and takes a single **character pointer** as a parameter. It is expected that a `Gift` struct instance’s **description** will be passed in so it may be given a value from user input.

First the function will print to the standard output via the **cout** object the message:

```
"Enter gift description: "
```

Prior to accepting input however we should consider what would occur if the user gave an input greater than the size (MAX_DESC) of our description character array. **Exceeding the length of our array can cause undesired behavior**. We can adjust the number characters to accept from the user via the following function applied on the standard input object **cin**:

cin.width(x)

where x is the number of characters desired at maximum for our description + 1 (for the **nullbyte**). Place this before accepting input via **cin**. These kinds of functions that affect I/O will be explored more in future weeks of study.

Then we will accept input from the user via **cin** to our passed in **description**.

```
void gifting(double&)
```

This variant of the gifting function also returns nothing (void) and takes a **double reference** (consider what would occur if this was a plain double rather than a

reference to a double) as a parameter. Similar to the previous the expected behavior is that we pass in a `Gift` struct instance's price to this function so it may be given value from user input.

The function first prints out:

`"Enter gift price: "`

Then it accepts user input from the user and stores that into the passed in price.

If the price given however is either less than zero or greater than our defined MAX_PRICE, print out an error message:

`"Gift price must be between 0 and [MAX_PRICE]" <newline>`

And then loop the user input process until a valid price is given.

```
void gifting(int&)
```

This variant of the gifting function also returns nothing (void) and takes an **integer reference.** It is similar to the previous function except that we are taking input for the units of a `Gift`. First print out the message:

`"Enter gift units: "`

Then accept input from the user and store that into the passed in parameter for the units.

If the value given by the user is less than 1 however, print out the error message:

`"Gift units must be at least 1" <newline>`

And then loop the user input process until a valid units is given.

```
void display(const Gift&)
```

This function returns nothing and takes in an **unmodifiable reference** to a `Gift`. It will display/print the details of the passed in `Gift` to the standard output in the following format.

`"Gift Details:" <newline>`

```
"Description: [g_description]" <newline>
"Price: [g_price]" <newline>
"Units: [g_units]" <newline>
```

MAIN PROGRAM:

GiftMain.cpp

In order to complete this **in_lab** we also be working with a main program shown below. It is littered with some **TODO** comments to assist you. Complete each of the **TODO** tasks by utilizing your `Gift` struct and the **gifting** functions. As you do this you will be experiencing dynamic memory allocation of your `Gift`. You will notice that the number of gifts desired isn't known at compile time and we require user input to know how much memory we need to store them (thus needing dynamic memory).

Following the main program is a **Sample Output** that will be tested against your code. Your main should output the exact same as the Sample. The sample will feature in **RED**, the exact user inputs expected and in **GREEN** is what is generated by the program when you run it.

```
/*************************************************************************
// OOP244 Workshop 2: Dynamic Memory & Function Overloading
// File GiftMain.cpp
// Version 1.0
// Date       2020/01/05
// Author Michael Huang
// Description
// Tests Gift module and provides a set of TODOs to complete
// which the main focuses are dynamic memory allocation
//
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
//
/////////////////////////////////////////////////////////////////
*************************************************************************/

#include <iostream>
#include "Gift.h"
#include "Gift.h" // Intentional
using namespace std;
using namespace sdds;

int main() {
  int numberOfGifts = 0; // Initial number of Gifts

  // TODO 1: declare a pointer for a dynamic array of Gifts (remember to initialize it)

  // END TODO 1
```

```
  cout << "Enter number of Gifts to allocate: ";
  cin >> numberOfGifts;

  if (numberOfGifts < 1) return 1;

  // TODO 2: allocate dynamic memory for the gifts pointer using the numberOfGifts

  // END TODO 2

  for (int i = 0; i < numberOfGifts; ++i) {
    cout << "Gift #" << i + 1 << ": " << endl;
    // TODO 3: utilizing the gifting functions, allow input for the gift's description

    // END TODO 3
    cin.ignore(2000,'\n'); // clear input buffer
    // TODO 4: utilizing the gifting functions, allow the user to input the gift's price

    // END TODO 4
    cin.ignore(2000,'\n'); // clear input buffer
    // TODO 5: utilizing the gifting functions, allow the user to input the units of gift

    // END TODO 5
    cin.ignore(2000, '\n');
    cout << endl;
  }

  // TODO 6: display the details of each gift using the Gifts module display function

  // END TODO 6

  // TODO 7: deallocate the gifts pointer here to avoid memory leaks as we are done with
it

  // END TODO 7

  return 0;

}
```

OUTPUT SAMPLE:

```
Enter number of Gifts to allocate: 4
Gift #1:
Enter gift description: Roses
Enter gift price: 5.99
Enter gift units: 5

Gift #2:
Enter gift description: Gift-card
Enter gift price: -99.99
Gift price must be between 0 and 999.99
Enter gift price: 1000
Gift price must be between 0 and 999.99
Enter gift price: 50
```

```
Enter gift units: 1

Gift #3:
Enter gift description: Hugs
Enter gift price: 0
Enter gift units: -5
Gift units must be at least 1
Enter gift units: 0
Gift units must be at least 1
Enter gift units: 25

Gift #4:
Enter gift description: cardcardcardcard
Enter gift price: 2
Enter gift units: 3

Gift #1:
Gift Details:
Description: Roses
Price: 5.99
Units: 5

Gift #2:
Gift Details:
Description: Gift-card
Price: 50
Units: 1

Gift #3:
Gift Details:
Description: Hugs
Price: 0
Units: 25


Gift #4:
Gift Details:
Description: cardcardcardcar
Price: 2
Units: 3
```

To complete your coding

a) remove all the **TODO** comments that indicated the tasks to complete and debugging statements from your code
b) Include in each file an appropriate header comments uniquely identify the file (as shown above)

c) Preface each function definition in the implementation file with a function header comment explaining what the function does in a single phrase (as shown above).
d) Make sure your function prototypes have meaningful argument names to help understand what the function does.

IN-LAB SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `Gift module` and the `GiftMain.cpp` program to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace `NXX`, i.e., NAA, NBB, etc.):

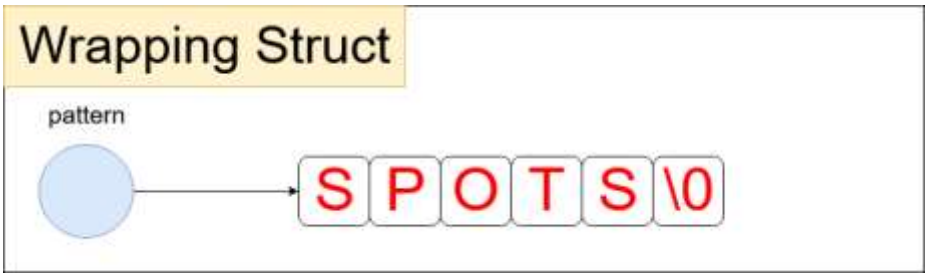
```
~profname.proflastname/submit 244/NXX/WS02/in_lab<ENTER>
```

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

DIY (DO IT YOURSELF) – 50%

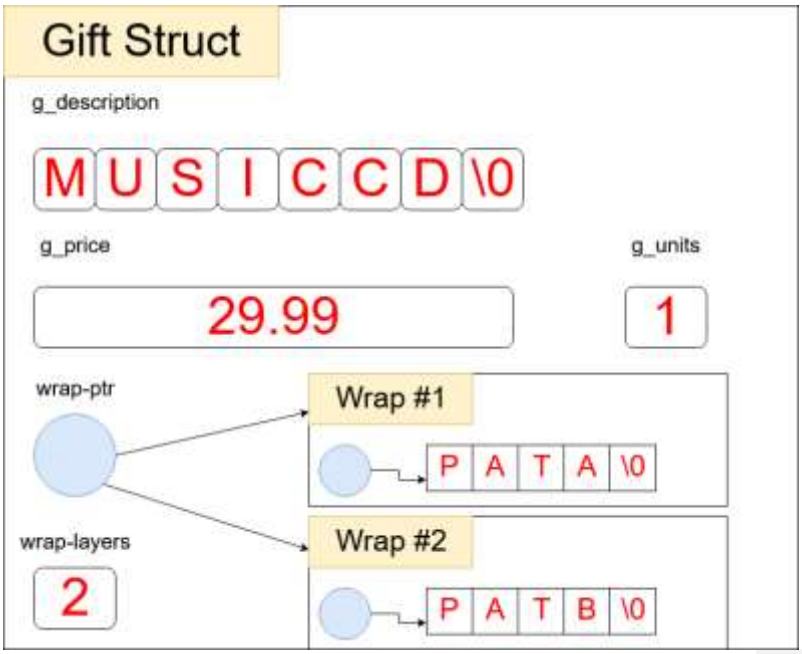
The **Gift** module will be updated to include a new **struct** called **Wrapping** in order to allow our `Gift` to be decorated with colourful paper. The Wrapping struct will be very simple and could look like this:



It will contain only a single data member that is a **character pointer** called **pattern**. This pointer will be used to allocate memory in order to store strings of **dynamic** length (we won't know this length until **runtime**) to describe the pattern of our wrapping paper.

Additionally, introduce another **constant integer** into the **Gift** module that represents the maximum length for the **pattern**. This value should be **20**.

After having defined the **Wrapping** struct, update the `Gift` struct to incorporate it:



Note that we're incorporating **Wrapping** into our `Gift` in two ways:

1. We have a **Wrapping pointer**, wrap-ptr that will be **dynamically allocated** to store 1 or more Wrappings. The above image shows two wrappings specifically but that's only for example purposes.
2. An **integer,** wrap-layers that tracks the number of wrappings on this `Gift`.

WRAP/UNWRAP FUNCTIONS.

We will be creating two functions related to **Wrapping**. As you work on these functions, do examine the **WrapGift.cpp** main program to see if you can get some hints on how these functions should work.

```
bool wrap(Gift&)
```

The **wrap** function will take in a `Gift` **reference** as its parameter and returns a ***Boolean*** value. The returned value should indicate if the wrapping was successful or otherwise. A Gift that is already wrapped cannot be wrapped until it is unwrapped and any other unwrapped Gift can be wrapped.

The main purpose of this function is to allow for a **user** to **input** values to determine how many layers of wrapping will be applied to this `Gift` and what the **patterns** of each layer are.

The basic process for this function could be as follows:

1. If a `Gift` **cannot** be wrapped (perhaps because it is already wrapped) simply print out: `"Gift is already wrapped!" <newline>`
2. If a `Gift` **can** be wrapped then proceed to prompt the **user** to do so with the line:

   ```
   "Enter the number of wrapping layers for the Gift: "
   ```

   and then accept their input. Do note that minimum number of layers is 1 and if a value provided is less than that print out:

   ```
   "Layers at minimum must be 1, try again."
   ```

   And have the user attempt again until a **valid** value is given.
3. Based on the number of layers, allocate memory to store that many **Wrappings** in the `Gift`.
4. Following that, provide values to the **pattern** of the **Wrappings** allocated via user input. The user should be prompted in the manner of:

   ```
   "Enter wrapping pattern #X"
   ```

Where X is the current wrapping being inputted into. Do recall that there is **dynamic memory** in play here for the string that is stored in the **Wrapping** of the `Gift`.

Avoid using allocating more memory than is needed for a pattern. For example, if a user entered "Spots" for the pattern of the Wrapping, we should only allocate enough memory to hold that text and no more.

5. In the end the function should indicate success or lack of in the **wrapping** process.

```
bool unwrap(Gift&)
```

The **unwrap** function will take in a `Gift` **reference** as its parameter and returns a *Boolean* value. The returned value should indicate if the unwrapping was successful or otherwise A Gift that isn't currently wrapped cannot be unwrapped until it is wrapped.

In the case where an unwrapping process **can proceed** the following should occur:

1. Print out the line: `"Gift being unwrapped." <newline>`
2. Clear the memory that was initially allocated to store the Wrapping(s) of the Gift. Recall that the **Wrappings** are dynamically allocated and the **pattern** data member in the Wrappings were also dynamically allocated.
3. Set the values of the number of layers and the wrap-ptr to sensible values that match the current state (what values would make sense for a **Gift with no Wrappings**?)

In the opposite case where **we cannot unwrap** this Gift, simply print out the line: `"Gift isn't wrapped! Can't unwrap."`

In **both** cases, the function, at the end should indicate success or lack of in the **unwrapping** process.

NEW GIFTING FUNCTION.

Create a new gifting function like the previous that takes in a **Gift reference** and returns nothing. This function will first print out:

`"Preparing a gift..." <newline>`

And then should combine the use of the previous gifting functions to set the gift's description, price and units then attempt to **wrap** the gift.

UPDATE DISPLAY FUNCTION.

Having adding **Wrapping** to our `Gift` struct, we should update the **display** function we defined from the in-lab. If a `Gift` isn't **wrapped** (we can define the meaning of not wrapped as the **wrap-ptr** data member being **the null address**) then we should display the following:

```
"Gift Details:" <newline>
"Description: [g_description]" <newline>
"Price: [g_price]" <newline>
"Units: [g_units]" <newline>
"Unwrapped" <newline>
```

In the case that it is **wrapped**, we should display

```
"Gift Details:" <newline>
"Description: [g_description]" <newline>
"Price: [g_price]" <newline>
"Units: [g_units]" <newline>
"Wrap Layers: [wrap-layers]" <newline>
"Wrap #1: [first wrapping]" <newline>
"Wrap #2: [second wrapping]" <newline>
. . .
```

Note that the number of "**Wrap #**" outputs will depend on how many Wrappings the `Gift` actually has.

MAIN PROGRAM:

WrapGift.cpp

```
/*****************************************************************
// OOP244 Workshop 2: Dynamic Memory & Function Overloading
// File WrapGift.cpp
// Version 1.0
// Date       2020/01/05
// Author Michael Huang
// Description
// Tests Gift module and provides a set of TODOs to complete
// which the main focuses are dynamic memory allocation
//
```

```
// Revision History
// -----------------------------------------------------------
// Name            Date            Reason
//
/////////////////////////////////////////////////////////////////
*****************************************************************************/

#include <iostream>
#include "Gift.h"
#include "Gift.h" // intentional
using namespace std;
using namespace sdds;

void initWrap(Gift& g) {
  g.wrap = nullptr;
  g.wrap_layers = 0;
}

ostream& line(int len, char ch) {
       for (int i = 0; i < len; i++, cout << ch);
       return cout;
}
ostream& number(int num) {
       cout << num;
       for (int i = 0; i < 9; i++) {
              cout << " - " << num;
       }
       return cout;
}

int main() {

  Gift g1, g2; // Unwrapped Gifts

  cout << "Preparing an empty Gift to be wrapped and display it" << endl;
  line(64, '-') << endl; number(1) << endl;
  gifting(g1.g_description);
  gifting(g1.g_price);
  gifting(g1.g_units);
  initWrap(g1); cout << endl;
  display(g1);

  cout << "\nAttempting to wrap the previous Gift: " << g1.g_description << endl;
  line(64, '-') << endl; number(2) << endl;
  if (wrap(g1)) cout << "Success!" << endl; else cout << "Wrapping failure!" << endl;

  cout << "\nAttempting to rewrap the previous Gift: " << g1.g_description << endl;
  line(64, '-') << endl; number(3) << endl;
  if (!wrap(g1)) cout << "Success2!" << endl; else cout << "Already wrapped failure!" <<
endl;

  cout << "\nAttempting to unwrap the previous Gift: " << g1.g_description << endl;
  line(64, '-') << endl; number(4) << endl;
  if (unwrap(g1)) cout << "Success3!" << endl; else cout << "Unwrapping failure!" <<
endl;

  cout << "\nAttempting to un-unwrap the previous Gift: " << g1.g_description << endl;
  line(64, '-') << endl; number(5) << endl;
```

```
  if (!unwrap(g1)) cout << "Success4!" << endl; else cout << "Already unwrapped failure!"
<< endl;

  cout << "\nPrepare another empty Gift via the combined function and display it" <<
endl;
  line(64, '-') << endl; number(6) << endl;
  initWrap(g2);
  gifting(g2); cout << endl;
  display(g2);

  cout << "\nEnd of main" << endl;
  unwrap(g2); // Unwrap in the end
  return 0;
}
```

Output Sample:

User input is in **red**. Program output is in **green**.

```
Preparing an empty Gift to be wrapped and display it
----------------------------------------------------------------
1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1 - 1
Enter gift description: Roses
Enter gift price: 5.99
Enter gift units: 5

Gift Details:
Description: Roses
Price: 5.99
Units: 5
Unwrapped

Attempting to wrap the previous Gift: Roses
----------------------------------------------------------------
2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2 - 2
Wrapping gifts...
Enter the number of wrapping layers for the Gift: -1
Layers at minimum must be 1, try again.
Enter the number of wrapping layers for the Gift: 0
Layers at minimum must be 1, try again.
Enter the number of wrapping layers for the Gift: 3
Enter wrapping pattern #1: Spots
Enter wrapping pattern #2: Stripes
Enter wrapping pattern #3: Zigzags
Success!

Attempting to rewrap the previous Gift: Roses
----------------------------------------------------------------
3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3 - 3
Gift is already wrapped!
Success2!

Attempting to unwrap the previous Gift: Roses
```

```
------------------------------------------------------------------
4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4 - 4
Gift being unwrapped.
Success3!

Attempting to un-unwrap the previous Gift: Roses
------------------------------------------------------------------
5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5 - 5
Gift isn't wrapped! Can't unwrap.
Success4!

Prepare another empty Gift via the combined function and display it
------------------------------------------------------------------
6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6 - 6
Preparing a gift...
Enter gift description: Hand-knit-scarf
Enter gift price: 0
Enter gift units: 1
Wrapping gifts...
Enter the number of wrapping layers for the Gift: 1
Enter wrapping pattern #1: Hearts

Gift Details:
Description: Hand-knit-scarf
Price: 0
Units: 1
Wrap Layers: 1
Wrap #1: Hearts

End of main
Gift being unwrapped.
```

REFLECTION

Study your final solutions for each deliverable of the Workshop, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time and the result is suggested to be at least 150 words in length.**

Create a file named `reflect.txt` that contains your **detailed description of the topics that you have learned** in completing this workshop and mention any issues that caused you difficulty. Some possible points of discussion for the reflection — **but do not limit it to just these** —:

1. How does dynamic memory differ from static memory?
2. What errors if any did you encounter when incorporating dynamic memory?
3. For the overloaded functions used in this workshop, how did they differ?

4. Describe what you have learned in this workshop

DIY SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `WrapGift.cpp` program and the updated `Gift` module to your `matrix` account. Compile and run your code and make sure that everything works properly.

Then, run the following script from your account during the lab (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace NXX, i.e., NAA, NBB, etc.):

`~profname.proflastname/submit 244/NXX/WS02/diy`<ENTER>

and follow the instructions generated by the command and your program.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.