







Branch: master ▾ OOP-Workshops / WS02 /

Create new file Upload files Find file History

 corbar	Workshop #2 has been released!	Latest commit 918507a 4 hours ago
..		
 images	Workshop #2 has been released!	4 hours ago
 part 1	Workshop #2 has been released!	4 hours ago
 part 2	Workshop #2 has been released!	4 hours ago
 readme.md	Workshop #2 has been released!	4 hours ago

 readme.md

Workshop #2: Dynamic Memory

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 custom type 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 two coding parts and one non-coding part:

- *Part 1*: worth 50% of the workshop's total mark, is due on **Wednesday at 23:59:59** of the week of your scheduled lab.
- *Part 2*: worth 50% of the workshop's total mark, is due on **Sunday at 23:59:59** of the week of your scheduled lab. Submissions of *part 2* that do not contain the *reflection* are not considered valid submissions and are ignored.
- *reflection*: non-coding part, to be submitted together with *Part 2*. Te reflection doesn't have marks associated to it, but can incur a **penalty of max 40% of the whole workshop's mark** if your professor deems it insufficient (you make your marks from the code, but you can lose some on the reflection).

If at the deadline the workshop is not complete, there is an extension of **one day** when you can submit the missing parts. **The code parts that are submitted late receive 0%.** After this extra day, the submission closes; if the workshop is incomplete when the submission closes (missing at least one of the coding or non-coding parts), **the mark for the entire workshop is 0%.**

Every file that you submit must contain (as a comment) at the top **your name, your Seneca email, Seneca Student ID** and the **date** when you completed the work.

If the file contains only your work, or work provided to you by your professor, add the following message as a comment at the top of the file:

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

If the file contains work that is not yours (you found it online or somebody provided it to you), **write exactly which part of the assignment are given to you as help, who gave it to you, or which source you received it from.** 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.

Compiling and Testing Your Program

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

```
g++ -Wall -std=c++11 -g -o ws file1.cpp file2.cpp ...
```

- `-Wall` : compiler will report all warnings
- `-std=c++11` : the code will be compiled using the C++11 standard
- `-g` : the executable file will contain debugging symbols, allowing *valgrind* to create better reports
- `-o ws` : the compiled application will be named `ws`

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

```
valgrind ws
```

To check the output, use a program that can compare text files. Search online for such a program for your platform, or use *diff* available on `matrix`.

Part 1 (50%)

Gift Module

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

Reminder: How to create a module

Add to the project a header file called `Gift.h`; inside the header file create a namespace called `sdds` that will contain all the functions and types that you are going to add to this module. Add the compilation safeguards (review **Workshop #1** for details).

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` (e.g., `<iostream>`) as you develop the implementation. Remember that standard headers should be included before custom headers.

Global Constants in this Module

Create two global constants in this module:

- `MAX_DESC` : the maximum length of gift description (not including the null terminator). Set it to 15.
- `MAX_PRICE` : the maximum price a gift can have. Set it to 999.999.

Gift Custom Type

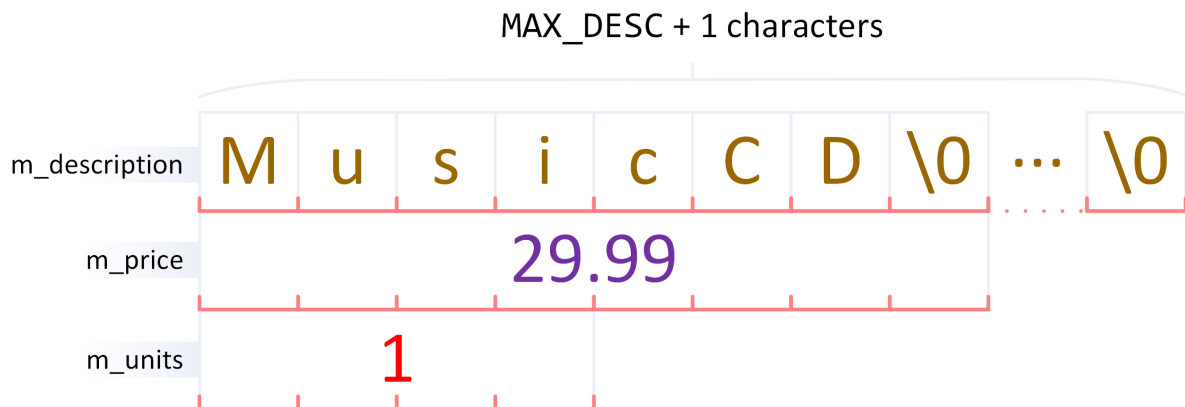
Design and code a structure (`struct`) named `Gift` in the namespace `sdds`. The structure's definition should be placed in the header file.

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

- `m_description` : a statically allocated array of characters that will store the description of the gift. The array should have `MAX_DESC` characters, not counting the null byte.
- `m_price` : a floating point number in double precision that will hold the price of the gift. At minimum, the price can be 0 and the maximum will be `MAX_PRICE`.
- `m_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` type could look like this:

Gift Structure



In the image above, we have an instance of type `Gift`, containing a single music CD at a price 29.99. The description is set to the string `MusicCD`. Note that the description has `MAX_DESC + 1` characters, but only 8 are in use (7 to store `MusicCD` and one to store the null terminator); the rest of the characters are not necessary for this instance, and are initialized with `\0`, in order not to contain *garbage* (random values that do not make semantically sense for the program).

Global Functions

Overload three times a function called `gifting` (remember that overloaded functions are functions with the same name, but different signature):

- `void gifting(char* desc)` : reads a gift description from the keyboard and stores the read characters into the array received as a parameter. It is expected that a `Gift` structure 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, we should consider what would occur if the user gave a string greater than the size (`MAX_DESC`) of our description character array. **Exceeding the length of our array can cause undesired behavior (this is a bug called *buffer overflow*).** We can adjust the number characters to accept from the user via the `width()` 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 null byte). These kinds of functions that affect I/O will be explored more in future weeks of study.

- `void gifting(double& price)` : reads a number from the keyboard and stores it into the parameter (consider what would happen if this was a plain `double` rather than a reference to a `double`). Similar to the previous function, the expected behavior is that we pass in a `Gift` structure instance's price to this function so it may be given value from user input.

First, the function will print to the standard output via the `cout` object the message `Enter gift price: .` Then read a number from the keyboard. If the number read is not within the interval `[0, MAX_PRICE]`, then print to the screen the message `Gift price must be between 0 and [MAX_PRICE]<ENDL>`, and read another number. This function should continue reading numbers until a valid one is given.

If the user provides a valid number, store it in the parameter.

- `void gifting(int& units)` : reads a number from the keyboard and stores it into the parameter.

First, the function will print to the standard output via the `cout` object the message `Enter gift units: .`

Similar to the previous function, this overload validates the number read to be at least 1. If the user input is invalid, print to the screen the message `Gift units must be at least 1<ENDL>` and read again. Continue looping until the user types a valid number.

If the user provides a valid number, store it in the parameter.

- `void display(const Gift& theGift)` : display to the screen the content of the parameter in the following format:

```
Gift Details:<ENDL>
Description: [DESCRIPTION]<ENDL>
Price: [PRICE]<ENDL>
Units: [UNITS]<ENDL>
```

main Module (partially supplied)

In order to complete this part, you have to update the main function. It is littered with some `TODO` comments to assist you. Complete each of the `TODO` tasks by utilizing your `Gift` custom type 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).

Sample Output

The output should look like the one from the `sample_output.txt` file.

Submission


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

Upload your source code to your `matrix` account. Compile and run your code using the `g++` compiler as shown above and make sure that everything works properly.

Then, run the following command from your account (replace `profname.proflastname` with your professor's Seneca userid):

```
~profname.proflastname/submit 244/w2/p1
```

and follow the instructions.

 **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.

Part 2 (50%)

The `Gift` module will be updated to include a new custom type called `Wrapping` in order to allow a gift to be decorated with colourful paper.

Global Constants

Add another constant to the module:

- `MAX_WRAP` : represents the maximum number of wrapping layers a gift can have. Initialize it to 20.

Wrapping Custom Type

The `Wrapping` custom type should contain a single attribute:

- `m_pattern` : a dynamically allocated array of characters. 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.

A visual representation of an instance of the `Wrapping` type could look like this:

Wrapping Structure



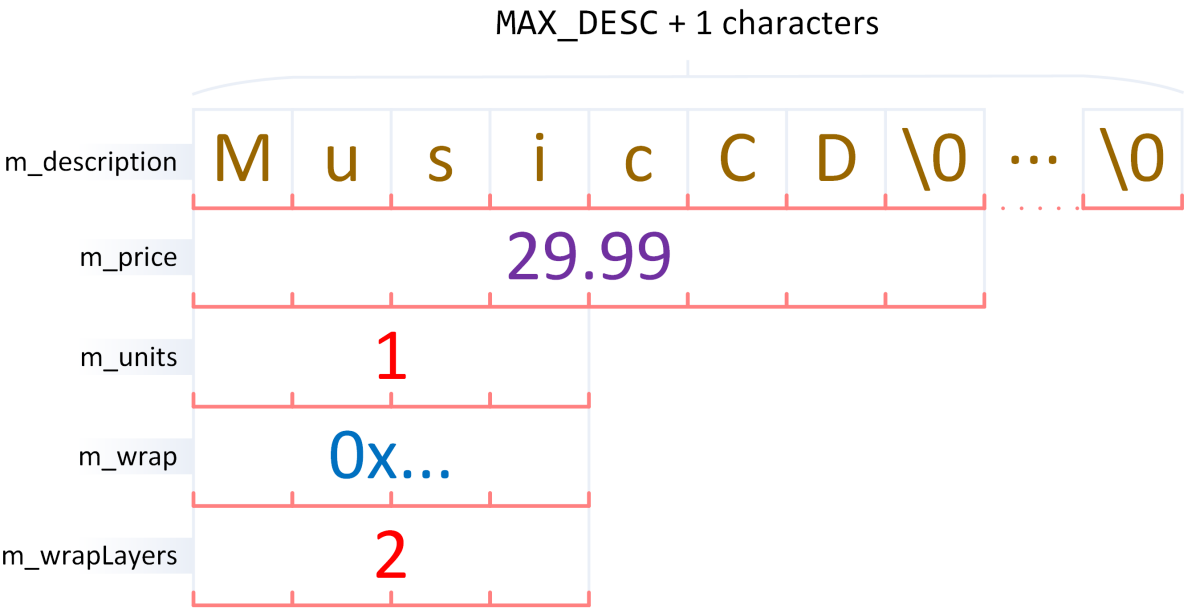
Gift Custom Type (update)

Update the `Gift` custom type to have two more attributes:

- `m_wrap` : a dynamically allocated array of objects of type `Wrapping` . The size of this array will be found at runtime (hence the dynamic nature of the array).
- `m_wrapLayers` : the number of layers used to wrap the gift (the size of the array `m_wrap`).

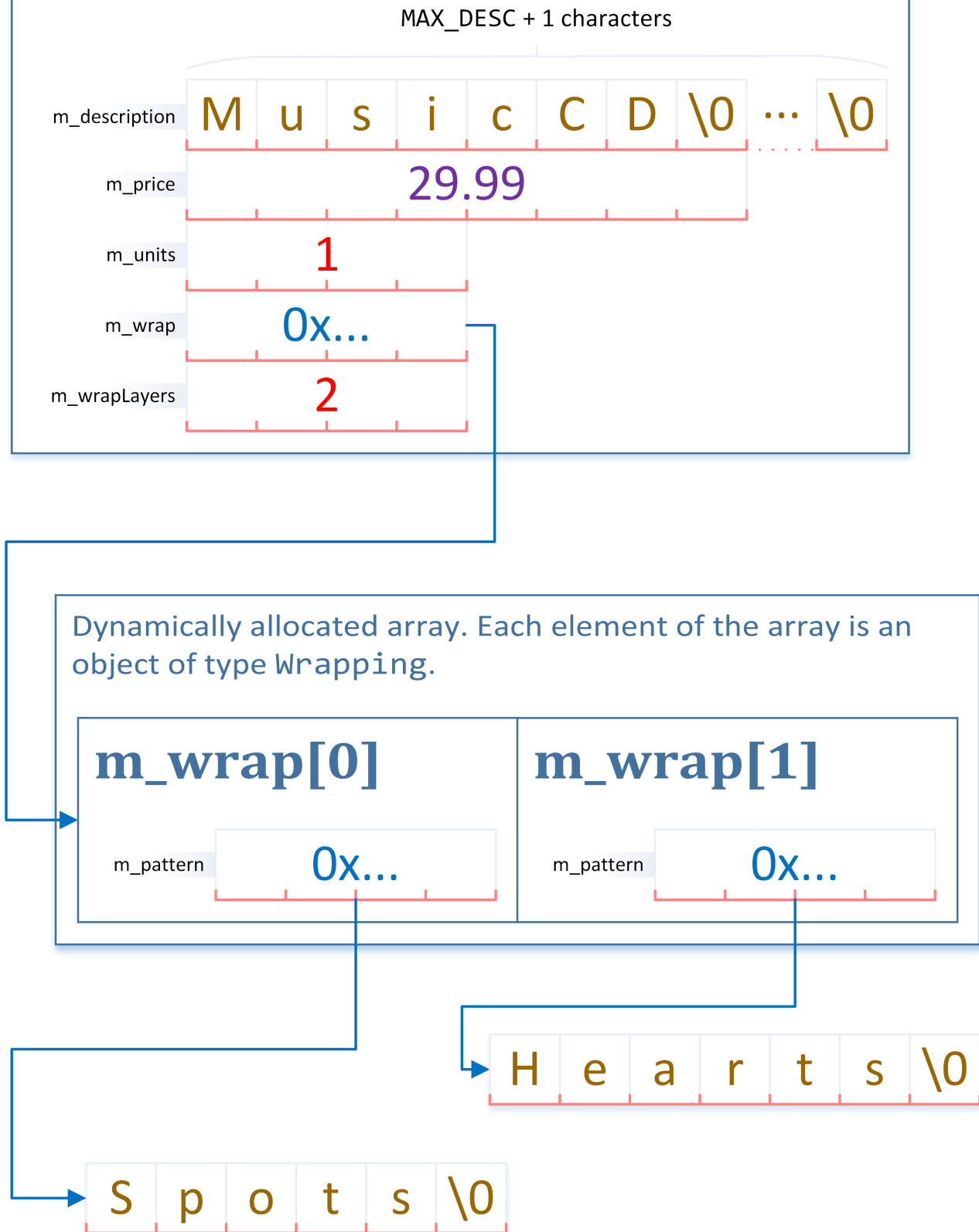
A visual representation of an instance of the `Gift` type could look like this:

Gift Structure



The following picture shows an instance of type `Gift` where the wrappings are defined:

Gift Structure



In the image, the memory is represented by boxes with a red bottom border; each box is one byte. All other information is metadata to help you better understand the memory layout, but will not remain in the executable file (obtained after compilation). The number of boxes associated to a variable represent the number of bytes used by that variable (i.e., `m_price` is a variable of type `double` and uses 8 bytes, while `m_units` uses only 4 bytes, etc.). The arrows represent the memory addresses to where pointers point.

Note in the image the difference between static memory (used by `m_description`) and dynamic memory (used by `m_pattern`); `m_description` has preallocated `MAX_DESC + 1` bytes which can be too much (resulting in wasted memory), or too little for the needs of the program. In case of the dynamic memory used by `m_pattern` we always have exactly the amount required.

Global Functions

As you work on these functions, do examine the main program supplied to see if you can get some hints on how they should work.

- `bool wrap(Gift& theGift)` : wraps the gift if it's not already wrapped.

If the gift is already wrapped, print to the screen `Gift is already wrapped!<ENDL>` and do nothing else.

Otherwise, print the message `Wrapping gifts...<ENDL>` and then ask the user for the number of layers: `Enter the number of wrapping layers for the Gift: .` If the user types a number smaller than 1, this function prints an error message `Layers at minimum must be 1, try again.<ENDL>` and ask for another number. It repeats asking until the user types a valid number.

When the number of layers has been retrieved, allocate dynamic memory to store that number of layers in the gift. Then iterate over the wrapping array and ask the user to provide the pattern for each one by prompting `Enter wrapping pattern #X: .` Do recall that there is dynamic memory in play here for the string that is stored in the wrapping of the `Gift`. **Do not allocate more memory than is needed for a pattern.**

This function should return `true` if wrapping has been performed; `false` otherwise. After this function finishes, the gift should be wrapped.

- `bool unwrap(Gift& theGift)` : unwraps the gift if it's wrapped.

If wrapping cannot happen, print the message `Gift isn't wrapped! Cannot unwrap.<ENDL>` and do nothing else.

Otherwise, print the message `Gift being unwrapped.<ENDL>` and deallocate all the dynamic memory used by the parameter.

Return `true` if the gift has been unwrapped, `false` otherwise. After this function finishes the gift should be unwrapped.

- `void gifting(Gift& theGift)` : prints to the screen the message `Preparing a gift...<ENDL>` and then calls the other `gifting()` function. At the end also call the `wrap()` function
- `void display(const Gift& theGift)`: update this function from previous part, to account for the wrapping.

If the gift doesn't have wrapping, this function should display:

```
Gift Details:<ENDL>
Description: [DESCRIPTION]<ENDL>
Price: [PRICE]<ENDL>
Units: [UNITS]<ENDL>
Unwrapped!<ENDL>
```

If the gift has wrapping with `N` layers, this function should display:

```
Gift Details:<ENDL>
Description: [DESCRIPTION]<ENDL>
Price: [PRICE]<ENDL>
Units: [UNITS]<ENDL>
Wrap Layers: [NUMBER_OF_LAYERS]<ENDL>
Wrap #1 -> [PATTERN]<ENDL>
Wrap #2 -> [PATTERN]<ENDL>
...
Wrap #N -> [PATTERN]<ENDL>
```

main Module (supplied)

Do not modify this module! Look at the code and make sure you understand it.

Sample Output

The output should look like the one from the `sample_output.txt` file.

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.

Submission


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

Upload your source code to your `matrix` account. Compile and run your code using the `g++` compiler as shown above and make sure that everything works properly.

Then, run the following command from your account (replace `profname.proflastname` with your professor's Seneca userid):

```
~profname.proflastname/submit 244/w2/p2
```

and follow the instructions.

 **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.