Final Project Report

Program Overview/ Explanation

The overall idea for my program is to allow the user to create and interact with robots. I came up with this topic/theme partially because it was entertaining (in a certain case, I get to have HAL say "I'm sorry Dave, I'm afraid I can't do that"!) and it seemed like a good way to cover all of the requirements. I tried to add a bit of fun.

There is a generic base type of robots, and two specialty sub-types of robots that are derived from the generic robot class. Those specialty robots are entertainers and personal assistants. The different types of robots have some different characteristics and tasks that they can do. This inheritance hierarchy covers the requirements for inheritance, polymorphism, and classes, as well as a number of other requirements that are spread throughout.

The main driver program, located in RMS.cpp, lets the user create, interact with, and remove the different types of robots stored in a list I call the Robot Management System.

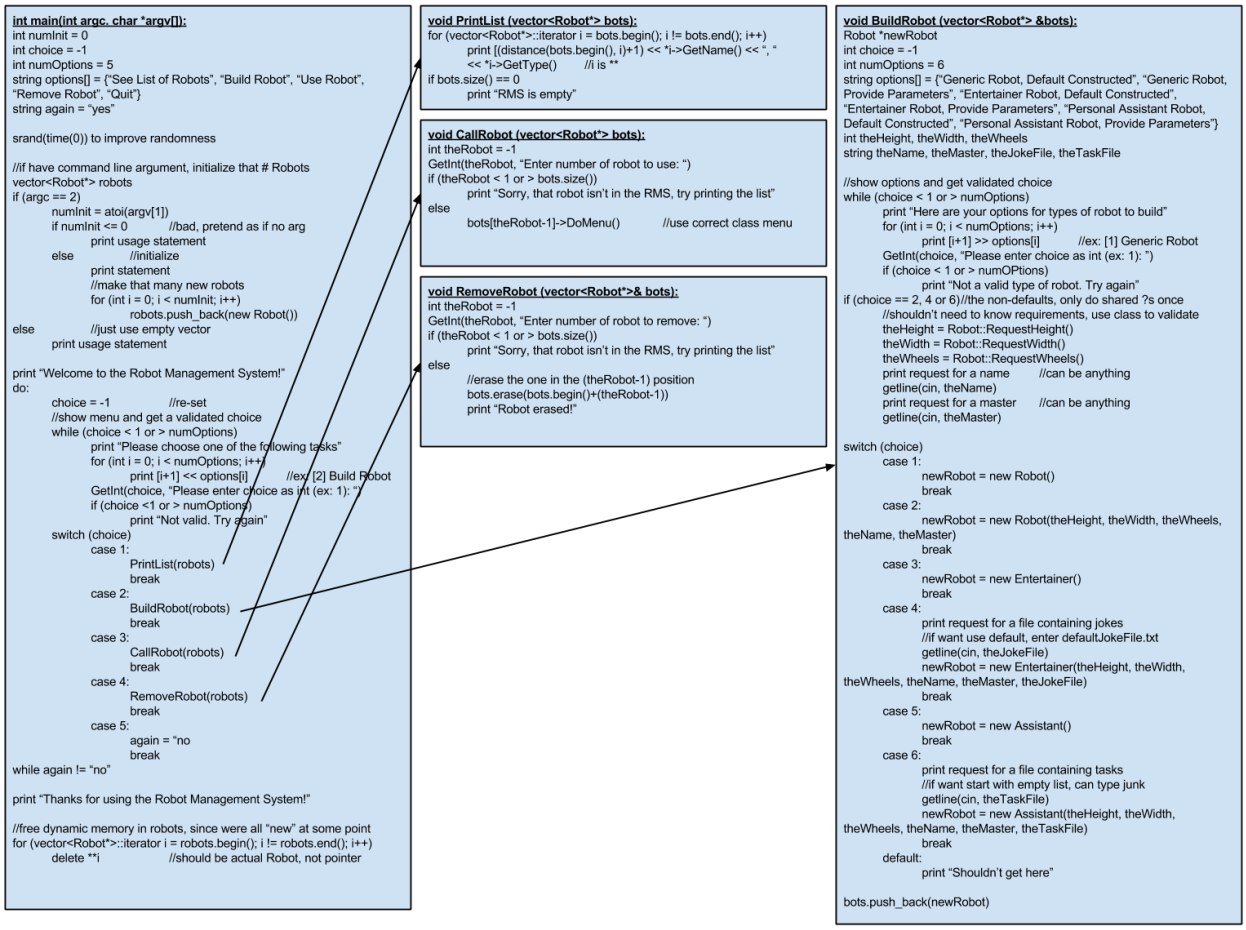
I added some things to the program specifically to meet particular requirements. For example, the entertainer robots can do the "neat bitwise trick" described in the assignment. I also added the calculation of a Greatest Common Factor that the personal assistant robots can do because it's a good use of recursion. Other such implementations were making the entertainer's list of jokes a vector of type Joke* to demonstrate pointers to structs, and having the function ReadJokes() throw an exception instead of just returning when the file failed to open. The representation of a robot's appearance in a dynamic 2-D array was partially to meet those requirements, but also because I was having fun with it. (I may have gotten a bit too into it, to be honest. Also, to give credit I started with the image from http://www.local-guru.net/blog/2008/10/11/ascii-art-robot and adapted it.)

A call to the default constructor of an entertainer robot requires availability of the file defaultJokeFile.txt, which includes some clever programming-related humor. I've also provided a testTaskFile.txt that can be used with the parameterized constructor of the Assistant class to make an initial task list.

To compile, simply run the included makefile. The executable it creates is called RMS.

Detailed Design Drawings

The first picture is of RMS.cpp, the application file with a main function that runs everything. The other three are for the 3 robot classes, including both their .h and .cpp files (which also contain a few non-class functions).


int main(int argc, char *argv[]):
int numInit = 0
int choice = -1
int numOptions = 5
string options[] = {"See List of Robots", "Build Robot", "Use Robot",
"Remove Robot", "Quit"}
string again = "yes"
srand(time(0)) to improve randomness
//if have command line argument, initialize that # Robots
vector<Robot*> robots
if (argc == 2)
numInit = atoi(argv[1])
if numInit <= 0 //bad, pretend as if no arg
print usage statement
else //initialize
print statement
//make that many new robots
for (int i = 0; i < numInit; i++)
robots.push_back(new Robot())
else //just use empty vector
print usage statement
print "Welcome to the Robot Management System!"
do:
choice = -1 //re-set
//show menu and get a validated choice
while (choice < 1 or > numOptions)
print "Please choose one of the following tasks"
for (int i = 0; i < numOptions; i++)
print [i+1] << options[i] //ex: [2] Build Robot
GetInt(choice, "Please enter choice as int (ex: 1): ")
if (choice <1 or > numOptions)
print "Not valid. Try again"
switch (choice)
case 1:
PrintList(robots)
break
case 2:
BuildRobot(robots)
break
case 3:
CallRobot(robots)
break
case 4:
RemoveRobot(robots)
break
case 5:
again = "no
break
while again != "no"
print "Thanks for using the Robot Management System!"
//free dynamic memory in robots, since were all "new" at some point
for (vector<Robot*>::iterator i = robots.begin(); i != robots.end(); i++)
delete **i //should be actual Robot, not pointer
void PrintList (vector<Robot*> bots):
for (vector<Robot*>::iterator i = bots.begin(); i != bots.end(); i++)
print [(distance(bots.begin(), i)+1) << *i->GetName() << ", "
<< *i->GetType() //i is **
if bots.size() == 0
print "RMS is empty"
void CallRobot (vector<Robot*> bots):
int theRobot = -1
GetInt(theRobot, "Enter number of robot to use: ")
if (theRobot < 1 or > bots.size())
print "Sorry, that robot isn't in the RMS, try printing the list"
else
bots[theRobot-1]->DoMenu() //use correct class menu
void RemoveRobot (vector<Robot*>& bots):
int theRobot = -1
GetInt(theRobot, "Enter number of robot to remove: ")
if (theRobot < 1 or > bots.size())
print "Sorry, that robot isn't in the RMS, try printing the list"
else
//erase the one in the (theRobot-1) position
bots.erase(bots.begin()+(theRobot-1))
print "Robot erased!"
void BuildRobot (vector<Robot*> &bots):
Robot *newRobot
int choice = -1
int numOptions = 6
string options[] = {"Generic Robot, Default Constructed", "Generic Robot,
Provide Parameters", "Entertainer Robot, Default Constructed",
"Entertainer Robot, Provide Parameters", "Personal Assistant Robot,
Default Constructed", "Personal Assistant Robot, Provide Parameters"}
int theHeight, theWidth, theWheels
string theName, theMaster, theJokeFile, theTaskFile
//show options and get validated choice
while (choice < 1 or > numOptions)
print "Here are your options for types of robot to build"
for (int i = 0; i < numOptions; i++)
print [i+1] >> options[i] //ex: [1] Generic Robot
GetInt(choice, "Please enter choice as int (ex: 1): ")
if (choice < 1 or > numOPtions)
print "Not a valid type of robot. Try again"
if (choice == 2, 4 or 6)//the non-defaults, only do shared ?s once
//shouldn't need to know requirements, use class to validate
theHeight = Robot::RequestHeight()
theWidth = Robot::RequestWidth()
theWheels = Robot::RequestWheels()
print request for a name //can be anything
getline(cin, theName)
print request for a master //can be anything
getline(cin, theMaster)
switch (choice)
case 1:
newRobot = new Robot()
break
case 2:
newRobot = new Robot(theHeight, theWidth, theWheels,
theName, theMaster)
break
case 3:
newRobot = new Entertainer()
break
case 4:
print request for a file containing jokes
//if want use default, enter defaultJokeFile.txt
getline(cin, theJokeFile)
newRobot = new Entertainer(theHeight, theWidth,
theWheels, theName, theMaster, theJokeFile)
break
case 5:
newRobot = new Assistant()
break
case 6:
print request for a file containing tasks
//if want start with empty list, can type junk
getline(cin, theTaskFile)
newRobot = new Assistant(theHeight, theWidth,
theWheels, theName, theMaster, theTaskFile)
break
default:
print "Shouldn't get here"
bots.push_back(newRobot)

Robot Class:

namespace PERCIVAL_ROBOTS

robot.h:

```
class Robot
public:
        Robot()
        Robot(int h, int w, int wh, string n, string m)
        Robot(const Robot& r)
        virtual ~Robot()
        virtual void DoMenu()
        friend ostream& operator <<(ostream& outs, const Robot& r)
        string GetName()        //used in main file for list
        virtual string GetType()        //used in main file for list
        friend int RequestHeight()
        friend int RequestWidth()
        friend int RequestWheels()
protected:
        int height
        int width
        int wheels      //0 = y, 1 = n
        char **looks    //pts to 2-D dynamic array
        string name
        string master
        vector<string> menuOptions
        void ChangeName()
        void ChangeMaster()
private:
        void CreateLooks()
        void CreateOptions()
        void CopyLooks(const Robot& r)
        void CopyOptions(const Robot& r)
```

robot.cpp:

```
#include "robot.h"
```

Robot::Robot():

```
height = 17
width = 18
wheels = 0 //yes
name = "HAL"
master = "Dave"
CreateLooks()
CreateOptions()
```

Robot::Robot(int h, int w, int wh, string n, string m):

```
height = h      //must be between 17 and 22
width = w       //must be between 18 and 28
wheels = wh
name = n
master = m
CreateLooks()
CreateOptions()
```

Robot::Robot(const Robot& r):

```
height = r.height
width = r.width
wheels = r.wheels
name = r.name
master = r.master
CopyLooks(r)
CopyOptions(r)
```

void Robot::CreateLooks():

```
//make 2-D array
looks = new char*[height]
for (int i = 0; i < height; i++)
        looks[i] = new char[width]
first put a space in every element to get rid of any junk values

//fill accordingly, starting with head
string line= "<width/3 spaces>()<3 spaces>()"
for (int i = 0; i < line.length(); i++)
        looks[0][i] = line.at(i)
line = "<width/3spaces>__\ /__"
for (int i = 0; i < line.length(); i++)
        looks[1][i] = line.at(i)
line = "<width/3-1 spaces>/<width/3+1 spaces> \"
for (int i = 0; i < line.length(); i++)
        looks[2][i] = line.at(i)
line = "<width/3-2 spaces>|/ \  / \|"
for (int i = 0; i < line.length(); i++)
        looks[3][i] = line.at(i)
line = "<width/3-2 spaces>|| H||H ||"
for (int i = 0; i < line.length(); i++)
        looks[4][i] = line.at(i)
line = "<width/3-2 spaces>|\_/  \_/|"
for (int i = 0; i < line.length(); i++)
        looks[5][i] = line.at(i)
line = "<width/3-2 spaces>| \____/ |"
for (int i = 0; i < line.length(); i++)
        looks[6][i] = line.at(i)
line = "<width/3-1 spaces>\_______/"
for (int i = 0; i < line.length(); i++)
        looks[7][i] = line.at(i)
line = "<width/6 spaces><width/6+1 _>|__|<width/6+1 _"
for (int i = 0; i < line.length(); i++)
        looks[8][i] = line.at(i)
line = " / |<width/2+1 spaces>| \"
for (int i = 0; i < line.length(); i++)
        looks[9][i] = line.at(i)
line = " | |<width/6+1 spaces>O O O | |"
for (int i = 0; i < line.length(); i++)
        looks[10][i] = line.at(i)
line = " / |<width/6+1 spaces>O O O | \"
for (int i = 0; i < line.length(); i++)
        looks[11][i] = line.at(i)
line = "| /|<width/2+1 spaces>|\ |"
for (int i = 0; i < line.length(); i++)
        looks[12][i] = line.at(i)
if height == 17     //smallest robot arms end at bottom
        line = " \||<width/2+1 _>||/"
        for (int i = 0; i < line.length(); i++)
                looks[13][i] = line.at(i)
else    //other robots ends arms and continue body
        line = " \||<width/2+1 spaces>||/"
        for (int i = 0; i < line.length(); i++)
                looks[13][i] = line.at(i)
        //make empty additional body, except bottom
        line = "<width/6 spaces>|<width/2+1 spaces>|"
        for (int j = 14; j <= (height-5); i++)
                for (int i = 0; i < line.length(); i++)
                        looks[j][i] = line.at(i)
        //add bottom
        line = "<width/6 spaces>|<widht/2+1 _>|"
        for (int i = 0; i < line.length(); i++)
                looks[height-4][i] = line.at(i)
//legs
line = "<width/6+1 spaces>|__|<width/9 spaces>|__|"
for (int i = 0; i < line.length(); i++)
        looks[height-3][i] = line.at(i)
line = "<width/6 spaces>/ __ \<width/9-2 spaces>/ __ \"
for (int i = 0; i < line.length(); i++)
        looks[height-2][i] = line.at(i)
if wheels == 0          //draw wheels
        line = "<width/9 spaces>OOOOOO<width/9 spaces>OOOOOO"
        for (int i = 0; i < line.length(); i++)
                looks[height-1][i] = line.at(i)
else            //no wheels
        line = "<width/9 spaces>|-----|<width/9-2 spaces>|-----|"
        for (int i = 0; i < line.length(); i++)
                looks[height-1][i] = line.at(i)
```

void Robot::CreateOptions():

```
menuOptions.push_back("Change Name")
menuOptions.push_back("Change Master's Name")
menuOptions.push_back("Show Name")
menuOptions.push_back("Show Master's Name")
menuOptions.push_back("Return to Main Menu")
```

void Robot::CopyLooks(const Robot& r):

```
//make 2-D array
looks = new char*[height]
for (int i = 0; i < height; i++)
        looks[i] = new char[width]
//fill with values from r
for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
                looks[i][j] = r.looks[i][j]
```

void Robot::CopyOptions(const Robot& r):

```
for(vector<string>::const_iterator i=r.menuOptions.begin();
i != r.menuOptions.end(); i++)
        menuOptions.push_back(*i)
```

void Robot::DoMenu():

```
int choice = -1
string again = "yes"
cout << *this               //serves as an intro
do:
        choice = -1     //re-set
        while (choice < 1 or > menuOptions.size())
                for iterator i = menuOptions.begin()...
                        print [(distance(menuOptions.begin(), i)+1)]
                        << *i << endl   //ex: [1] Change Name
                GetInt(choice, "Please enter # choice (ex: 1): ")
                if (choice < 1 or > menuOptions.size())
                        print "Not valid. Try again"
        switch (choice)
                case 1:
                        ChangeName()
                        break;
                case 2:
                        ChangeMaster()
                        break
                case 3:
                        print "My name is " << name
                        break
                case 4:
                        print "My master's name is " << master
                        break
                case 5:
                        again = "no"
                        break
while again != "no"
```

Robot::~Robot():

```
for (int i = 0; i < height; i++)
        delete [] looks[i]
delete [] looks
```

string Robot::GetName():

```
return name
```

string Robot::GetType():

```
return "generic robot"
```

int RequestHeight():

```
int theHeight = -1
while (theHeight < 17 or > 22)
        GetInt(theHeight, "Enter height 17-22:")
        if (theHeight < 17 or > 22)
                print "Out of range. Try again."
return theHeight
```

int RequestWidth():

```
int theWidth = -1
while (theWidth < 18 or > 28)
        GetInt(theWidth, "Enter width 18-28:")
        if (theWidth < 18 or > 28)
                print "Out of range. Try again."
return theWidth
```

int RequestWheels():

```
int theWheels = -1
while (theWiheels  != 0 or 1)
        GetInt(theWheels, "Enter 0 (y) or 1 for wheels:")
        if (theWiheels  != 0 or 1)
                print "Out of range. Try again."
return theWheels
```

ostream& operator<<(ostream& outs, const Robot& r):

```
outs << "Hi! My name is " << r.name << ", and my master's name is"
      << r.master << "." << endl
//print physical representation with looks
for (int i = 0; i < r.height; i++)
        for (int j = 0; j < r.width(); j++)
                outs << r.looks[i][j]
        outs << endl            //for row break
outs << endl
outs << "I am " << r.height << " tall and " << r.width << " wide."
outs << "What can I do for you today?" << endl

return outs
```

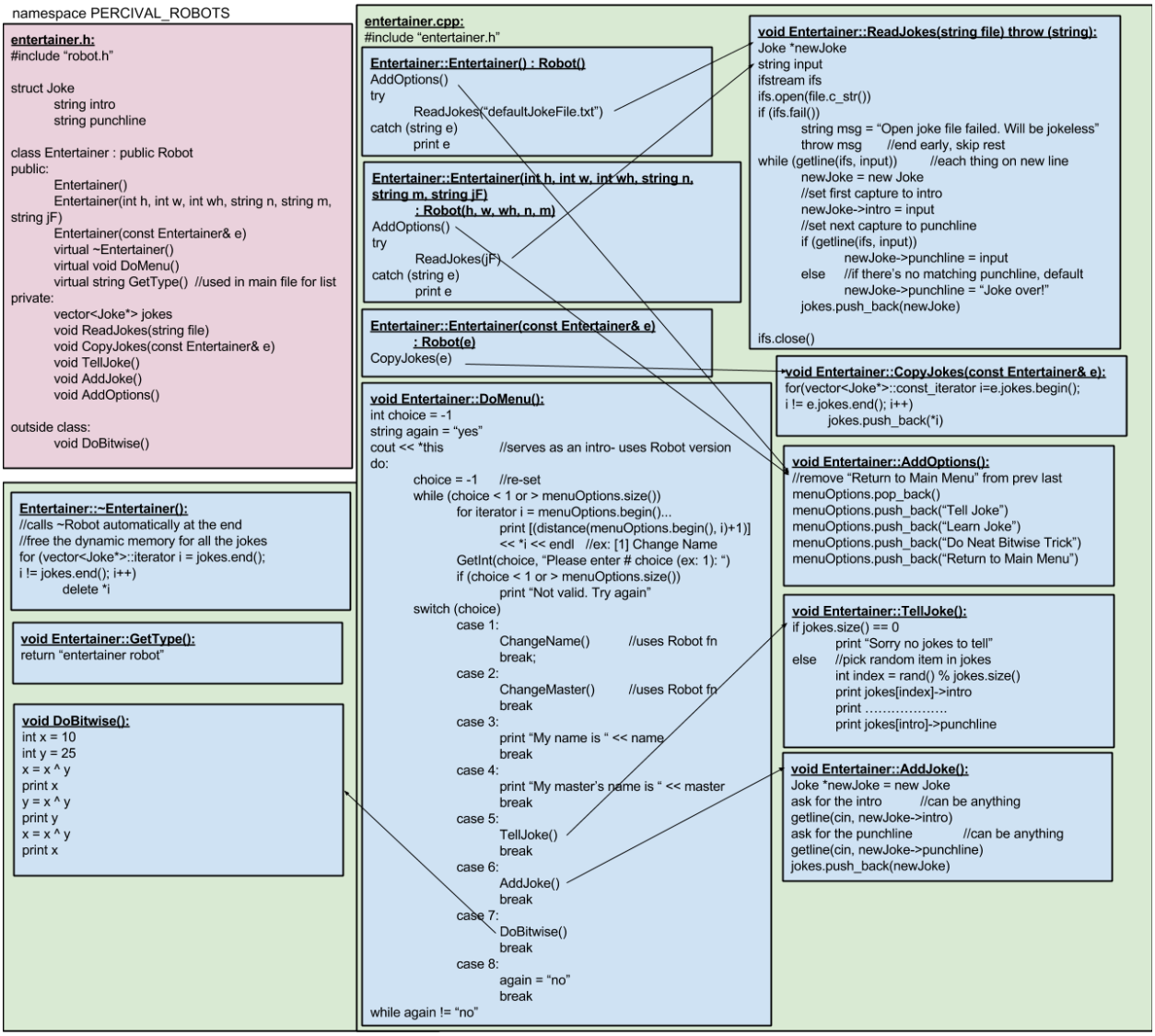
void Robot::ChangeName():

```
string newName
print request for name          //can be anything
getline(cin, newName)
name = newName
```

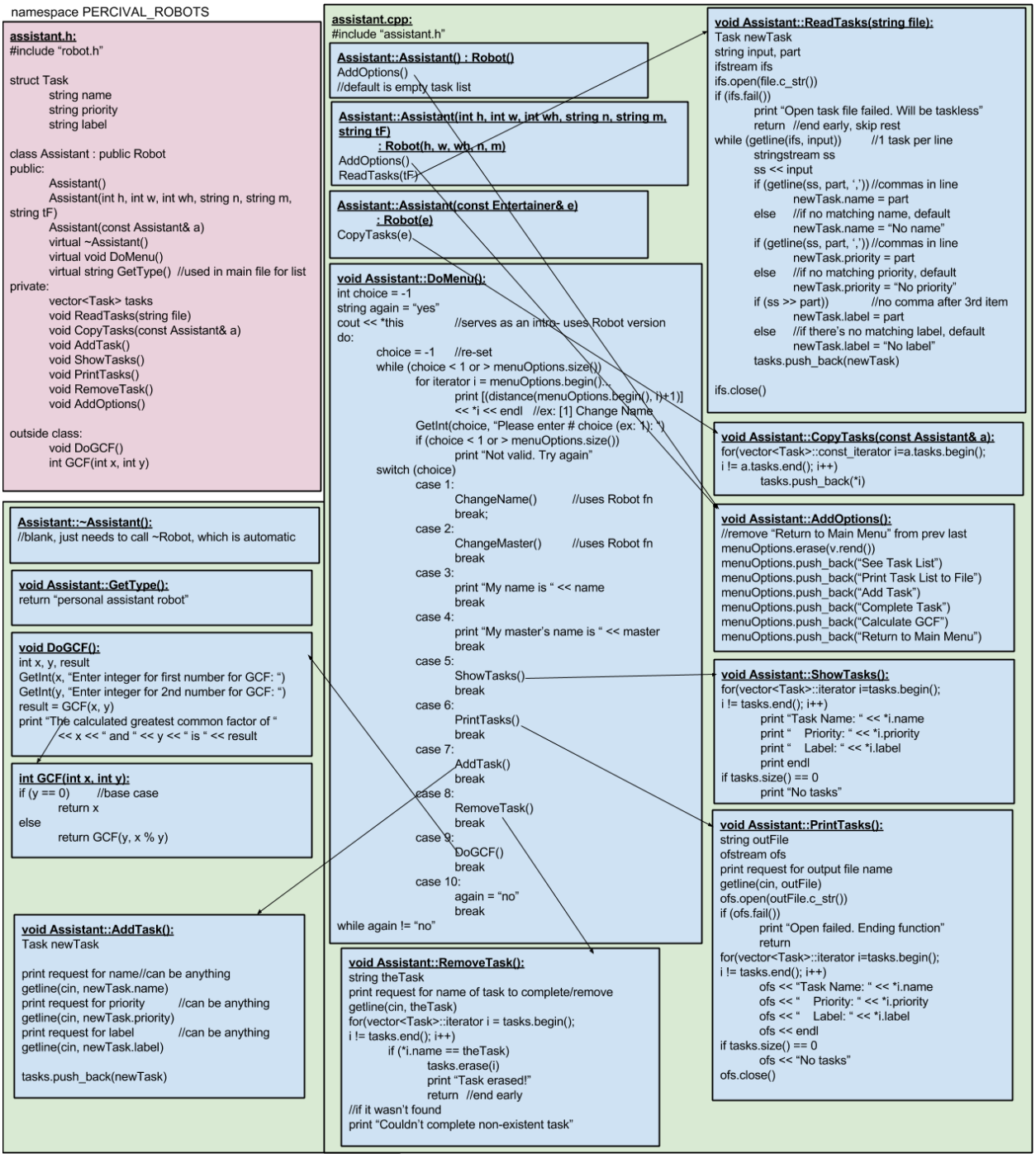
void Robot::ChangeMaster():

```
string newMaster
print request for master's name         //can be anything
getline(cin, newMaster)
master = newMaster
```

Entertainer Class:



Assistant Class:



Testing

(This isn't a requirement of the report, which surprised me, but I find it helpful to write out at least a brief plan before I start, so I thought I'd go ahead and include it. Maybe an extra credit point?)

Try running the program without a command line argument, with an invalid command line argument (perhaps "a" and 0), and with a valid integer command line argument. Check that the appropriate

statements are printed and the right number of generic robots is initialized when the argument is valid (print the list to see them).

Check the input validation on the main menu in the RMS, as well as the menu in BuildRobot(). Ensure that choosing "Quit" in the main menu causes the loop to end and the program to finish.

Create at least one of each type of robot (test BuildRobot()), using both default construction and parameterized construction for each. Check the input validation in RequestHeight(), RequestWidth() and RequestWheels(). See what happens when you enter an invalid filename for the jokes file and tasks file (they should both result in a message and an empty list of jokes/tasks).

Once made, call on each of the different types of robots to use. Make sure the call to DoMenu() calls the appropriate version of the virtual function depending on the true type of the robot. Also try using a robot that isn't in the list and make sure you get the print statement saying it wasn't there.

Test removal of a robot. Make sure the statement about erasure prints and the robot is no longer in the list. Also try removing a robot that isn't in the list and make sure you get the print statement.

Print the list of robots in the RMS at various points and make sure it updates correctly. Include right at the beginning and after a build or removal. Check that both the name and the type (virtual function) are correct.

Within the call to DoMenu() for each robot type:

- Check out the introduction that is printed using <<. Make sure all the variables printed are correct based on the construction. Look at the physical representation and see how it changes for different values of width, height, and wheels.
- Verify input validation for the menu.
- Try changing the name and changing the master's name, then show each to confirm that they changed.
- Ensure that choosing "Return to Main Menu" causes the loop to end, the function to end, and then you see the main menu in main of RMS again.
- Additional for Entertainer:
    - Have it tell a joke, make sure it prints properly (no guarantee it's funny). Repeat a few times to check the randomness. Try with an empty joke list, ensure it says "Sorry, no jokes to tell".
    - Add a Joke (both to an existing list of jokes and an empty list). Have it tell jokes again to see that it gets used.
    - Do the bitwise trick and check out the results.
- Additional for Assistant:
    - Both show and print the list of tasks. They should be the same except one is in a file. Check format and values for correctness. Try with an empty list of files (default constructed) and see that it prints correct statement.
    - Add a Task (both to an existing list and an empty list). Show list again to check success.

- Remove a Task and print list to check success. Also try removing a task that doesn't exist and verify prints appropriate message.
- Test printing tasks to an output file.
- Find a couple Greatest Common Factors to test. Have x > y, x < y, x = y, x divisible by y, and y divisible by x. Also try 0s and negative numbers.

Code Style & Use of Program

Note: I saw on Piazza that this in unnecessary, but I had already written it so it felt like a waste to delete it, even if it is a bit silly.

I try to use lots of indentation and whitespace. I make all of my {} code blocks with the first { on its own line (instead of with the class, function header, if statement, etc.). For my identifiers I work to be descriptive, and my variable names are camel case while my function names have each word capitalized (including the first). I have comment blocks at the top of each file, and before each function (besides main). (Here also for each requirement.) Variable declarations have descriptions of what that variable is used for. Chunks of code are generally separated by a blank line and frequently preceded by a descriptive comment. I add comments when I got something from a source, and to add more details on either the algorithm or the reason why I do something. I think about the order of my code and try to make it as logical and readable as possible.

All input is validated, at least when it needs to be. I make heavy use of the GetInt() function that I designed, both for capturing actual integers that will be used as integers and to simplify the user's interaction with my menu by having them select an integer instead of typing a string. There are things where I don't restrict what the user can enter, such as the strings for name and master's name, because they can be flexible and the input won't break the program.

When the program runs, there are a lot of descriptive print statements about the progress of the program, what the user needs to do, and what the results of their actions are. Requests for user input aim to be as clear and easy to work with as possible. Output such as details on a robot is informative, clear, and sometimes tries to be clever.

Reflection

This project was a good review of everything we have learned in the class, and it was also instructive to put it all together.

One thing I learned during the course of doing the project was that declaring friend functions in a class header file that are inside a namespace does not make those functions available outside the namespace. I had never encountered this issue in previous assignments because I hadn't combined the topics of namespaces, classes, friend functions, and separate compilation. It took me a while to figure out, but I discovered that the solution was to declare the functions again inside the namespace but outside the class. (helpful sources: http://stackoverflow.com/questions/1749311/friend-function-within-a-namespace, http://stackoverflow.com/questions/8207633/whats-the-scope-of-inline-friend-functions, and Piazza)

Another detail I learned about was that in order to iterate over a vector that was passed to a function through a const argument, you need a special kind of iterator called a const_iterator. When I tried using a regular iterator I got a compiler error, but quickly found the solution at https://www.daniweb.com/software-development/cpp/threads/125172/vector-iterator.

A neat technique I discovered was to manipulate a string using the insert function (http://www.cplusplus.com/reference/string/string/insert/), which allows you to quickly and easily add a certain number of a certain character. It was really useful for building the lines/rows that went into my 2-D array looks for a robot's physical representation. (Although, similarly, I learned that ASCII art can be tough and takes a lot of code, particularly when you want it to be dynamic.)

At first I designed the CallRobot() and RemoveRobot() functions in the RMS to have the user enter the name of the robot they wanted to use or remove and find that name in the vector. I liked this approach because I thought it would be the most intuitive for the user. However, I then realized that if there were multiple robots with the same name (say default generic robots created by the command line argument) then it would only ever be possible to use or remove the first robot with that name, until its name was changed. Therefore, I decided it would be best to instead ask the user to provide a corresponding number to indicate which robot to use or remove. As a result, I also tweaked my PrintList() function to show those numbers.

I was surprised that vectors weren't listed as a requirement; especially since I used a lot of them. Another useful technique I learned about while doing this project was the pop_back function to remove the last element from a vector (http://www.cplusplus.com/reference/vector/vector/pop_back/).

This project was a good opportunity for me to become more familiar with GDB. I hadn't used it much in the past, but especially with this large and complex program I found it was really useful, particularly when I got some strange errors that didn't make sense right away.

I absolutely approach problems differently now that I have gone through all the materials for this class. There are so many things that I can do now that I would have had no idea where to start on at the beginning of the quarter. I did know a fair number of the basic programming concepts, but not the details of how to implement them in C++. I was also getting beyond my realm of knowledge by about week 3, and definitely there by week 5. The toughest topic for me was probably dynamic 2-D arrays, and I'm glad to say I feel like I'm mostly comfortable with them by now. I've also struggled with recursion; not so much the actual implementation, but more coming up with problems or solutions to problems that are a good fit for it. That one I admittedly might need more time and practice with, but if someone tells me to use recursion for a particular task I can manage. I'm also looking forward to future opportunities to see more examples and the true power of some of the more advanced topics we've been learning at the end here, such as polymorphism. Although this project really helped cement those concepts for me more than they were before.

My process has also evolved a fair bit over the course of the term. I've always understood why design is important, but in the beginning it was a bit challenging for me sometimes. The way I do it has developed and I have found good ways to make it logical and effective. I've also really learned the value of

incremental programming; testing one part at a time instead of coding a giant mess and trying to work through it. I struggled at first because I would get impatient or it seemed tedious, but as the programs got more complex I began to really value that work process suggestion. It's helped me learn to debug more efficiently and have a lot fewer duplicate errors because I catch them before writing similar code. For example, on this project I got the Robot class completely functional (piece by piece), before beginning to code the Entertainer class, and I got the Entertainer class fully correct and tested before beginning to code the Assistant class. The errors I found in earlier stages were then avoided in later ones.

Requirements Summary

I have marked the requirements in my code with comments as requested, but also thought it might be helpful to have a summary of them all together in one place here.

1. Demonstrates simple input and output (cout , cin, getline)
    a. I marked the first use of cout in RMS.cpp, but it is pervasive throughout.
    b. For getline I marked the requests for name and master in BuildRobot() in RMS.cpp, but it is also used heavily throughout.
    c. For cin I marked its use in GetInt() (in robot.cpp), because I mostly relied on that function for (validated) input whenever getline didn't make sense.
2. Demonstrates explicit type casting
    a. Use of string.c_str() when opening files using a name that was captured from the user as a string. I marked the use in ReadJokes() in entertainer.cpp, but there are a few others.
3. Demonstrates logical operators and bitwise operators (specifically &&, ||, !, or == along with &, |, ^)
    a. For logical, I marked == in the check for command line arguments in RMS.cpp, || in the main menu display/choice capture loop in RMS.cpp, ! at the end of that same do-while loop, and && in RequestWheels() in robot.cpp. However, most of these are used many times, especially || and ==.
    b. For bitwise I used the trick from the project description with ^ in the DoBitwise() function in entertainer.cpp.
4. Demonstrates at least one loop (preferably a for, while, or do-while loop)
    a. Marked my do-while loop with the main menu in RMS.cpp, but there're others.
    b. The while loop I marked is the one for the menu options in BuildRobot() in RMS.cpp, but there are many more.
    c. I marked my for loop in the same place, although like the others there are lots.
5. Demonstrates at least one random number
    a. Entertainment robot randomly chooses a joke in TellJoke() in entertainer.cpp. (Note the use of srand(time(0)) in RMS.cpp to improve randomness.)
6. Demonstrates understanding of the three general error categories we talked about (syntax, logic, and run-time)
    a. A good example of a syntax error (which breaks the rules of the language) was when I originally tried to use a normal iterator on a vector that was passed as part of a const

argument. I marked this in CopyOptions() in robot.cpp, because that was the first place I encountered it. I had to change it to a const_iterator to avoid compiler errors.

   b. A logic error causes code to behave differently than was intended. A good example of this is using a < instead of a >, and I think I may have even done that at one point. I chose to mark the if statement in CallRobot() in RMS.cpp that checks whether the chosen robot is valid.
   c. A run-time error only appears when the program is running, and is often indicated by the program crashing (although not necessarily; most logic errors can also be considered run-time errors). One good example of this is when you try to access an item in a vector that is outside the valid range of items in that vector. I chose to use as an example what would happen if I removed "i != bots.end()" from the for loop in PrintList() in RMS.cpp.

7. Demonstrates at least one debugging "trick" that we have learned throughout the class
   a. One is the use of numbers instead of strings as the choice for menus, which makes it less likely that typos will interfere with functionality and allows easier input validation. This is marked inside the main menu in RMS.cpp, but is in all of my menus. I also ended up doing it for capturing which robot to use or remove.
   b. Another is input verification/validation, which is done many times using my function GetInt(), as well as in the RequestHeight(), RequestWidth(), and RequestWheels() functions. This ensures that user input is appropriate and won't cause issues. I chose to mark GetInt() (in robot.cpp) because it's really prevalent.
   c. Some of the data in a robot's introduction (through the << operator in robot.cpp) could serve as print statements to help debug certain issues. For example, I used the printed values of height and width to help troubleshoot when the robots didn't look how I wanted, so I marked that.
   d. One case where I checked for bad conditions was with opening files for input and output. This is important because if a file doesn't open properly the rest of the code won't work. I marked this in ReadTasks() in assistant.cpp.
8. Demonstrates at least one function that you define and at least one overloaded function that you define
   a. There are many examples of functions I defined, so I just marked PrintList() in RMS.cpp.
   b. The constructors of all 3 classes are overloaded, because each has a default constructor, a parameterized constructor, and a copy constructor. I marked them in robot.cpp.
9. Demonstrates general functional decomposition to reduce how large a single section of code is or to make the plan or algorithm obvious for your program
   a. This is really spread throughout, but I marked the switch statement in the main menu in RMS.cpp because it's a good example of how I chose to put each of the options in its own function instead of making the switch statement big and ugly.
10. Demonstrates how scope of variables works (how a specific variable only is accessible from within its defined block of code)
   a. I added a note in CreateLooks() in robot.cpp pointing out that I re-use the variable "i" in a series of for loops and they don't interfere with each other because the scope of each is limited to that for loop. Also noting that they cannot be accessed outside the for loop.

    b. I also added a note in the switch statement in the main menu in RMS.cpp pointing out that I have to pass the variable robots to the functions because it was declared in main and would otherwise not be accessible outside of it.
11. Demonstrates the different passing mechanisms
    a. I marked pass-by-reference on BuildRobot() and pass-by-value on CallRobot(), both in RMS.cpp. Of course there are plenty of other examples as well.
12. Demonstrates at least one std::string variable and at least one c-style string
    a. There are many std::string variables (C++-style strings), so I just marked the first one in RMS.cpp.
    b. My C-style string was covered by the command line argument in RMS.cpp. Also, all of the statements in quotations used throughout are actually C-style strings.
13. Demonstrates some form of recursion
    a. The Personal Assistant Robot can give you a Greatest Common Factor recursively with GCF() in assistant.cpp.
14. Demonstrates at least one multi-dimensional array and one dynamically allocated array
    a. The dynamic 2-D array for the robot's appearance (looks in robot.h) covers both aspects of this requirement. It is instantiated and filled in CreateLooks() and deleted in the destructor.
15. Demonstrates at least one command line argument
    a. The user can decide to start with a number of generic robots. Usage statements are printed if they don't provide an argument or provide an invalid one (also tell them when using a valid argument). This is in the main function of RMS.cpp.
16. Demonstrates definition and use of at least one from each of: class, struct, and object
    a. Using a class means using an object. I define three classes and use them all with objects. Marked the Robot class definition in robot.h and a Robot creation in BuildRobot() in RMS.cpp.
    b. The Task and Joke structs are used in the Assistant and Entertainer classes, respectively. I marked the Task definition and use (in the vector variable) in assistant.h and its further use in the function ShowTasks() in assistant.cpp.
17. Demonstrate at least one of each of: pointer to an array, a struct, and an object
    a. In the Robot class (robot.h), looks is a pointer to an array.
    b. The vector in RMS.cpp is full of pointers to Robot objects. A pointer to a Robot object is also used in BuildRobot().
    c. Made the joke list a vector of type Joke* (in entertainer.h) instead of Joke just to show a pointer to a struct. As part of this, a pointer to a Joke struct is also used in ReadJokes() and AddJoke() in entertainer.cpp.
18. Demonstrates at least one custom namespace
    a. The 3 robot classes and some associated functions are all in a PERCIVAL_ROBOTS namespace, which spans the majority of the .h and .cpp files outside of RMS.cpp. I marked it in robot.h. I also marked the using declarations in RMS.cpp.
19. Demonstrates at least one header file you write

    a. There is a .h (and .cpp) for each robot class. I marked the top of robot.h, as well as the #include in RMS.cpp.
20. Demonstrates at least one makefile that we use to compile the project on flip
    a. This is just the file called makefile. I marked it at the top (with different comment characters, of course).
21. Demonstrates at least one default constructor, a copy constructor, and a destructor for a class
    a. All 3 Robot classes have all 3 of these. I marked them in assistant.cpp.
22. Demonstrates at least one overloaded operator
    a. I overloaded << to print a Robot's intro (including name etc. and physical appearance) as a friend function, marked in robot.cpp.
23. Demonstrates some form of file IO
    a. The jokes list in the Entertainer class is built using file input. The task list in the Assistant class can be created using file input if the user wants. I marked ReadTasks() in assistant.cpp.
    b. The function PrintTasks() in assistant.cpp can output the current task list to an output file.
24. Demonstrates inheritance and polymorphism
    a. The three classes I created are robots and different sub-types of robots. The Assistant and Entertainer classes are derived from Robot, and inherit its member variables and functions. I marked the statement that Entertainer inherits from Robot in entertainer.h, as well as a part in DoMenu() in entertainer.cpp that demonstrates that Entertainer can access Robot's protected member variables.
    b. DoMenu() in the three classes is a virtual function, as is GetType(). Marked the call to DoMenu() on a Robot pointer in CallRobot() in RMS.cpp, which calls the appropriate version of DoMenu() that matches the true type of the robot being pointed to. Also marked the declaration of DoMenu() in robot.h.
25. Demonstrates exceptions
    a. ReadJokes() throws a string message when the file open fails, which is caught and printed in the Entertainer constructors (both marked in entertainer.cpp).