Inheritance **(+*new* keyword, destructors, +Valgrind)**

When we want to dynamically create an object (instead of just declaring at the top of our program) we can use the *new* keyword. *(Remember from 1320 that dynamically allocating memory means WE are in charge of releasing memory when we are done with it. Next class we will learn about smart pointers to help us with this)*

Stack vs heap (example drawn the board)

```
#include <iostream>
#include <memory>

using namespace std;

class Bird
{
  public:
    string color;

    Bird(string color)
    {
      this->color=color;
    }

};

int main(int argc, char **argv)
{
  string answer;
  Bird *b1; //pointer, not an object
  cout<<"Do you see a blue bird?"<<endl;
  cin>>answer;

  //new (like malloc in C) returns the address of a newly created object

  if(answer=="yes")
  {
    b1=new Bird("blue"); //I just created a bird object
  }

  else
  {
    cout<<"Ok. What color is it?"<<endl;
    cin>>answer;

    b1=new Bird(answer);
  }

  cout<<"Bird color is: "<<b1->color<<endl;
  delete (b1);
}
```

Destructors:

Destroying an object when you're done with it and it goes out of scope (often used when you dynamically allocate memory and don't want to worry about calling delete each time).

```
computer$ g++ practice.cpp
computer$ ./a.out
Making A...

In the if statement...
Making B...
Destroying B...

Loop! i=0
Making D...
Destroying D...

Loop! i=1
Making D...
Destroying D...

Making C...
Destroying C...

Destroying A...
```

```
#include <iostream>

class A
{
public:
   A()
   {
      std::cout << "Making A...\n";
   }

    ~A()
    {
       std::cout << "Destroying A...\n\n";
    }
};

class B
{
public:
   B()
   {
      std::cout << "Making B...\n";
   }
```

```
    ~B()
    {
        std::cout << "Destroying B...\n\n";
    }
};

class C
{
public:
    C()
    {
        std::cout << "Making C...\n";
    }

    ~C()
    {
        std::cout << "Destroying C...\n\n";
    }
};

class D
{
public:
    D()
    {
        std::cout << "Making D...\n";
    }

    ~D()
    {
        std::cout << "Destroying D...\n\n";
    }
};


int main(int argc, char **argv) {

        A a_object; //creating an A object (using constructor)

        if(true)
        {
           std::cout<<"\nIn the if statement..."<<std::endl;
           B b_object; //creating a B object (using the constructor) in the scope of if.  When the if statement
is done, the B object will be destroyed (out of scope)-the destructor is called
        }

        //in the for loop scope (running twice) a D object is created, then destroyed twice (each iteration).
The constructor is called twice and destructor is called twice
        for(int i=0;i<2;i++)
        {
            std::cout<<"Loop! i="<<i<<std::endl;
```

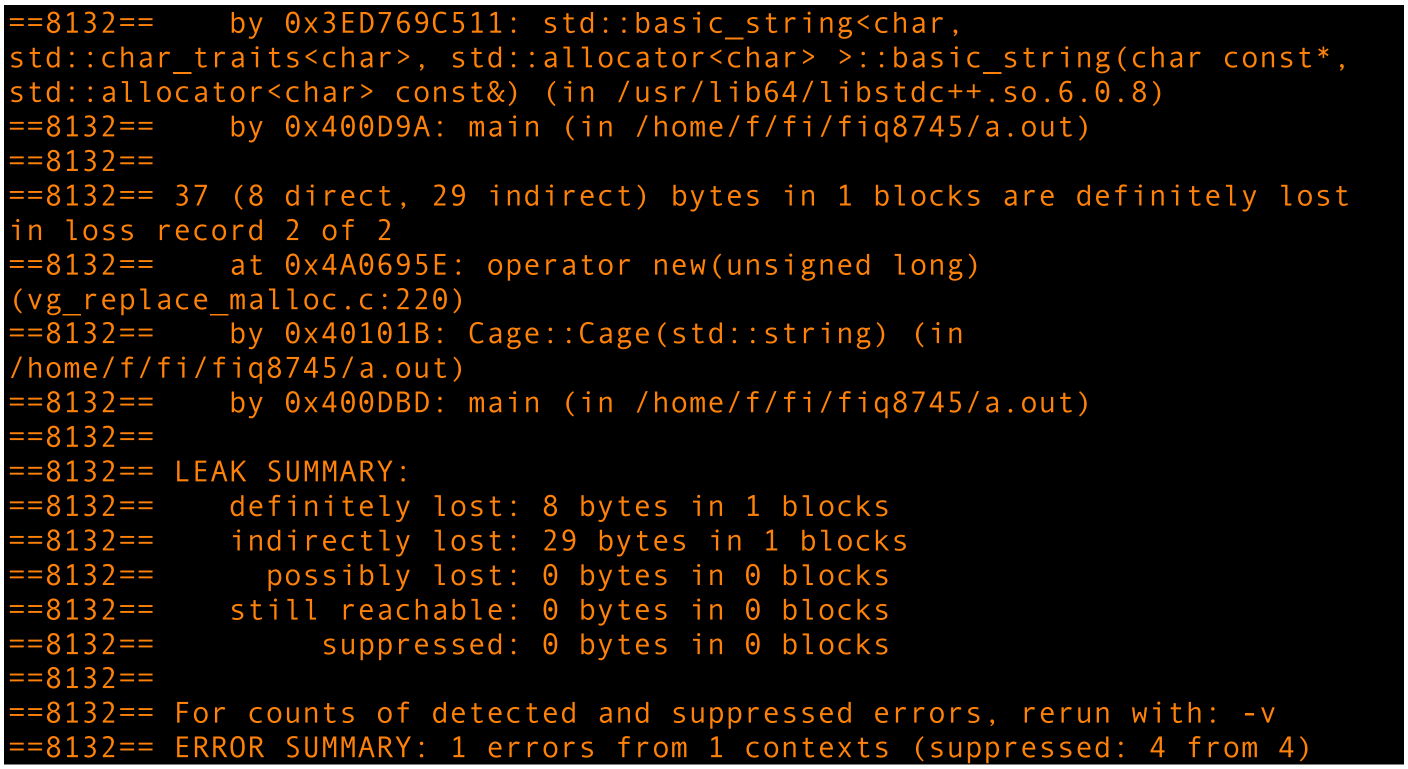
```
                D d_object;
          }

          //C object created here  (using constructor)
           C c_object;

          //C object is destroyed as they go out of scope (destructor called)
          //A object is destroyed as they go out of scope (destructor called)
}
```

Using Valgrind on Omega (a program)-if you do not include *delete(b1);* (in the constructor of the Cage), you get a leak. You could also manually free it (putting *delete(c1->b1);* in the *main* for example), but by putting it in the destructor, you don't forget to delete (since the destructor is called when the Cage goes out of scope).

Note that I also include *delete (c1);* in main to delete the Cage itself.

(This example run on Valgrind shows a memory leak-not using delete)

```
[fiq8745@omega ~]$ g++ stuff.cpp
[fiq8745@omega ~]$ valgrind --tool=memcheck --leak-check=yes --show-
reachable=yes --track-origins=yes --num-callers=20 --track-fds=yes ./a.out
==8132== Memcheck, a memory error detector
==8132== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==8132== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==8132== Command: ./a.out
==8132==
Do you have a cage with a blue bird?
yes
Bird color is: blue
==8132==
==8132== FILE DESCRIPTORS: 3 open at exit.
==8132== Open file descriptor 2: /dev/pts/4
==8132==    <inherited from parent>
==8132==
==8132== Open file descriptor 1: /dev/pts/4
==8132==    <inherited from parent>
==8132==
==8132== Open file descriptor 0: /dev/pts/4
==8132==    <inherited from parent>
==8132==
==8132==
==8132== HEAP SUMMARY:
==8132==     in use at exit: 37 bytes in 2 blocks
==8132==   total heap usage: 6 allocs, 4 frees, 127 bytes allocated
==8132==
==8132== 29 bytes in 1 blocks are indirectly lost in loss record 1 of 2
==8132==    at 0x4A0695E: operator new(unsigned long)
(vg_replace_malloc.c:220)
==8132==    by 0x3ED769B860: std::string::_Rep::_S_create(unsigned long,
unsigned long, std::allocator<char> const&) (in
/usr/lib64/libstdc++.so.6.0.8)
==8132==    by 0x3ED769C364: ??? (in /usr/lib64/libstdc++.so.6.0.8)
```



```
==8132==    by 0x3ED769C511: std::basic_string<char,
std::char_traits<char>, std::allocator<char> >::basic_string(char const*,
std::allocator<char> const&) (in /usr/lib64/libstdc++.so.6.0.8)
==8132==    by 0x400D9A: main (in /home/f/fi/fiq8745/a.out)
==8132==
==8132== 37 (8 direct, 29 indirect) bytes in 1 blocks are definitely lost
in loss record 2 of 2
==8132==    at 0x4A0695E: operator new(unsigned long)
(vg_replace_malloc.c:220)
==8132==    by 0x40101B: Cage::Cage(std::string) (in
/home/f/fi/fiq8745/a.out)
==8132==    by 0x400DBD: main (in /home/f/fi/fiq8745/a.out)
==8132==
==8132== LEAK SUMMARY:
==8132==    definitely lost: 8 bytes in 1 blocks
==8132==    indirectly lost: 29 bytes in 1 blocks
==8132==      possibly lost: 0 bytes in 0 blocks
==8132==    still reachable: 0 bytes in 0 blocks
==8132==         suppressed: 0 bytes in 0 blocks
==8132==
==8132== For counts of detected and suppressed errors, rerun with: -v
==8132== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

```
#include <iostream>
#include <memory>

using namespace std;

class Bird
{
 public:
   string color;

   Bird(string color)
   {
     this->color=color;
   }

};

class Cage{

public:
Bird *b1;

 Cage(string color)
 {
   b1=new Bird(color);
 }

 ~Cage() //since we have the destructor, we don't have to worry about deleting the bird in the cage.  It will
delete when the Cage object goes out of scope.
 {
   delete (b1);
 }
};


int main(int argc, char **argv)
{
 string answer;
 Cage *c1; //pointer, not an object
 cout<<"Do you have a cage with a blue bird?"<<endl;
 cin>>answer;

 //new (like malloc in C) returns the address of a newly created object

 if(answer=="yes")
 {
   c1=new Cage("blue"); //I just created a cage object
 }

 else
 {
   cout<<"Ok. What color is the bird?"<<endl;
   cin>>answer;

   c1=new Cage(answer);
 }

 cout<<"Bird color is: "<<c1->b1->color<<endl;
 delete (c1);  //we still need to delete the actual cage object (the bird inside is deleted as stated above)
}
```

Two main aspects to consider with inheritance:

1. Types (situational) ***(show online-Google: types of inheritance)***
   - In 1325, the main types you will see me using periodically throughout the semester (note there are more):
     - Single inheritance (we saw this last class-a Student from a Person class)
     - Multiple inheritance (example below-Yoga studio)
     - Hierarchical inheritance
     - Multilevel inheritance (example below-Special Powers)

2. Access (using access specifiers):
   - public, private, protected
   - What is the base class passing along to derived classes (with access)?
   - How are derived classes inheriting (with access)?

Program 1:

(Simple example showing access with inheritance followed by a program-protected, private and public)

```
#include <iostream>
#include <vector>

using namespace std;

class Base_example
{
  protected: //only accessible to children/derived classes
       int y;

  private: //things only this class should have (base here)
       int z;

  public:
       int x;

      void set_y_value(int n)
      {
        y=n;
      }

     void set_z_value(int n)
     {
       z=n;
     }
};

class Derived_public: public Base_example //inherit all as same (except private-can't access)
{
       // x is public
       // y is protected (only accessible to children)
       // z is not accessible from Derived_public
};

class Derived_protected: protected Base_example //all inherited as protected
{

       // x is protected (only accessible in here and to derived classes)
       // y is protected
       // z is not accessible from Derived_protected
};


class Derived_private: private Base_example //all inherited as private
{
       // x is private, only accessible in here (not outside when I create a Derived_private obj).  Also not available
to any further derived classes-stops here
```

```
        // y is private (^^^)
        // z is not accessible from Derived_private

        //I made a public function to print out both:

public:
  void values(int n, int j)
  {
    x=n;
    y=j;
    //z=j;  can't do this-z is not accessible in here
    cout <<x<<" "<<y<<endl;
    set_y_value(4); //inherited this from the base class. It’s private so can't be used outside, but can be used
here
    cout <<x<<" "<<y<<endl;
  }
};


class Again_Derived_private: public Derived_private
{
  public:
  void values_again(int n, int j)
  {
        //You can't access these because they are private to Derived_private
        //   x=n;
        //   y=j;
  }

};


int main(int argc, char **argv) {

  Derived_public d_pub;
  Derived_protected d_prot;
  Derived_private d_priv;

  d_pub.x=3;
  d_pub.set_y_value(45);
  d_pub.set_z_value(3);

  //d_prot.set_y_value(45); can’t do this-function inherited as protected

  d_priv.values(4,5);
  //d_priv.x=4; can't do this-x inherited as private
  //d_priv.set_y_value(4); can't do this- set_y_value() inherited as private

return 0;
}
```

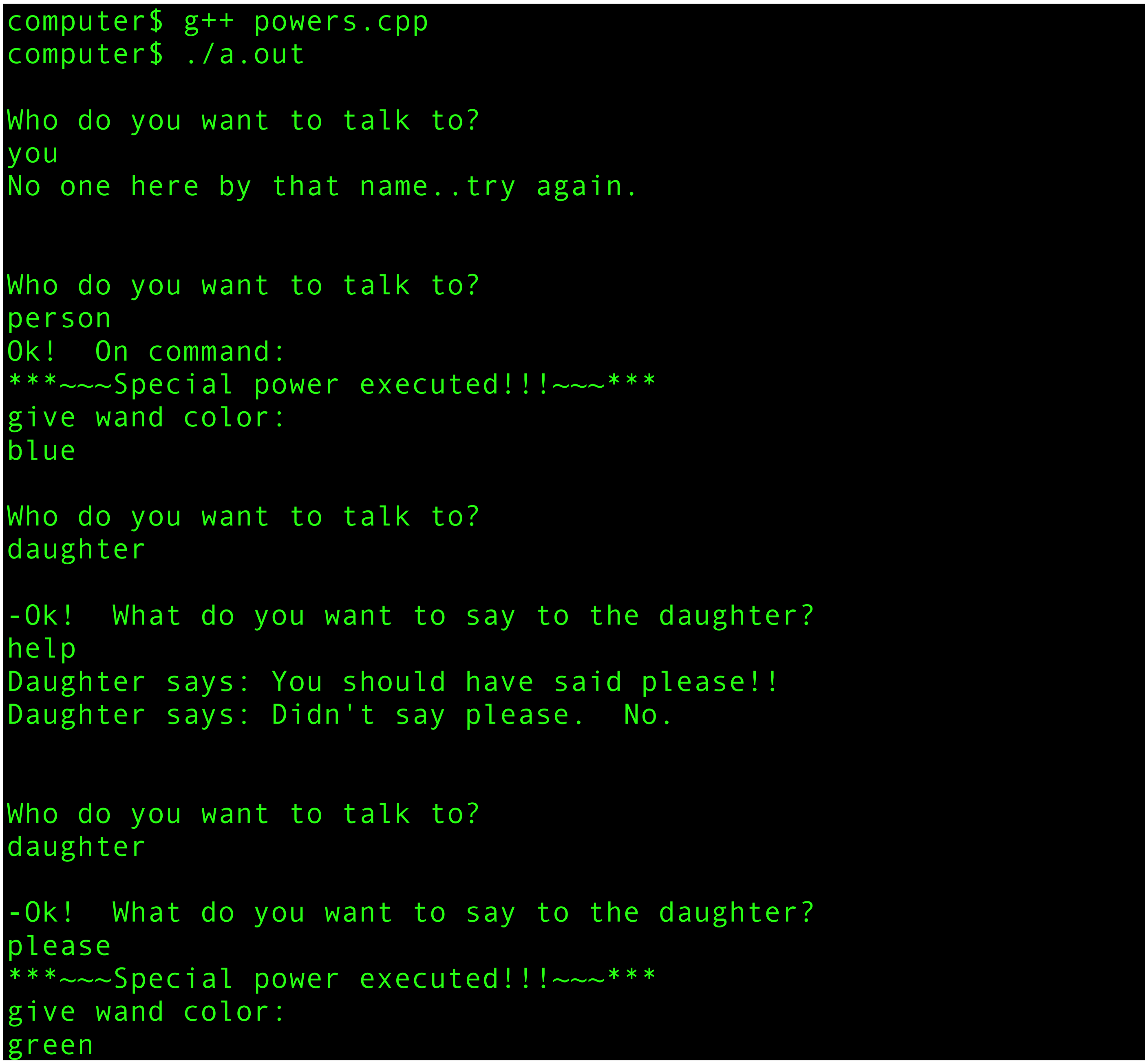
We did a public inheritance example last class (with the shapes).

Protected example:

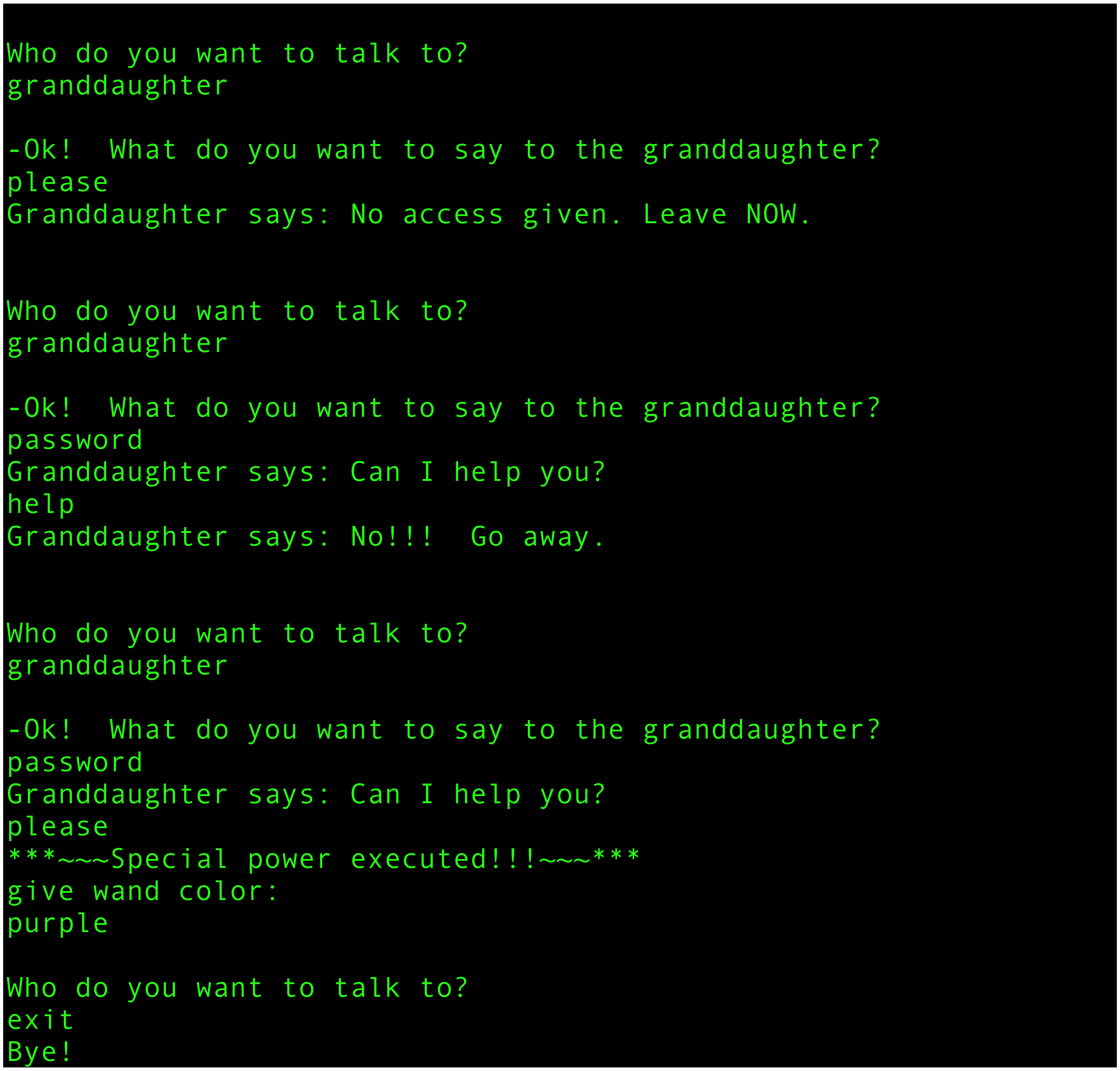
A special person has special powers and a wand. Anyone can have this person do the special powers on command and give a color to the wand.

This special person has passed down these special powers (and a wand) to her daughters. The daughter is more powerful and does not have to do these special powers on command-a person must ask “please” before the daughter will do anything.

The daughters also passed down the special powers to their daughters (granddaughters of the original). The granddaughters are the most powerful-someone must gain access to even talk to them (using the word “password”). Once access is gained, that person must then ask “please” before the granddaughter will do anything. *(Just make one of each person for this program)*



```
computer$ g++ powers.cpp
computer$ ./a.out

Who do you want to talk to?
you
No one here by that name..try again.


Who do you want to talk to?
person
Ok!  On command:
***~~~Special power executed!!!~~~***
give wand color:
blue

Who do you want to talk to?
daughter

-Ok!  What do you want to say to the daughter?
help
Daughter says: You should have said please!!
Daughter says: Didn't say please.  No.


Who do you want to talk to?
daughter

-Ok!  What do you want to say to the daughter?
please
***~~~Special power executed!!!~~~***
give wand color:
green
```



```
Who do you want to talk to?
granddaughter

-Ok!  What do you want to say to the granddaughter?
please
Granddaughter says: No access given. Leave NOW.


Who do you want to talk to?
granddaughter

-Ok!  What do you want to say to the granddaughter?
password
Granddaughter says: Can I help you?
help
Granddaughter says: No!!!  Go away.


Who do you want to talk to?
granddaughter

-Ok!  What do you want to say to the granddaughter?
password
Granddaughter says: Can I help you?
please
***~~~Special power executed!!!~~~***
give wand color:
purple

Who do you want to talk to?
exit
Bye!
```

```
#include <iostream>

using namespace std;

class Special_person
{

 protected://only children can access
   string wand_color;
   bool can_fly;

      public:
   void special_power() //others can call on you to use your special power cuz public
   {
```

```
      cout << "***~~~Special power executed!!!~~~***"<<endl;

    }

    void give_info()
    {
      cout<<"give wand color: "<<endl;
      cin>> wand_color;

      if(wand_color=="blue")//can only fly if blue
      {
        can_fly=true;
      }

      else
      {
        can_fly=false;
      }
    }

};



class Daughter_special_person: protected Special_person //inherit all from above, but only visible to daughter
{ //can pass on to her daughter


  public:
    void ask_use_power(string request)
    {
      if(request=="please")
      {
        special_power(); //can use it internally, not accessible directly to outside
      }
      else
      {
        cout<<"Daughter says: You should have said please!!"<<endl;
      }



    }

    void ask_give_info(string request)
    {
      if(request=="please")
      {
        give_info(); //can use it internally, not accessible directly to outside
      }

      else
```

```
      {
        cout<<"Daughter says: Didn't say please.  No.\n"<<endl;
      }

    }

};


class Granddaughter_special_person: protected Daughter_special_person
{


  public:
    void gain_access(string request) //password (w/exclamation mark)
    {
      string answer;

      if(request=="password")
      {
        cout<<"Granddaughter says: Can I help you?"<<endl;
        cin >>answer;

        if(answer=="please")
        {
          ask_use_power(answer); //from Daughter_special_person
          give_info(); //from Special_person class
        }

        else
        {
          cout<<"Granddaughter says: No!!!  Go away.\n"<<endl;
        }
      }

      else
      {
          cout<< "Granddaughter says: No access given. Leave NOW.\n"<<endl;
      }
    }


};


int main(void) {

  string answer;//

  Special_person s1;
  Daughter_special_person d1;
```

```
   //d1.special_power(); can't do this.  protected inheritance
   Granddaughter_special_person g1;


   while(answer!="exit")
   {
    cout << "\nWho do you want to talk to?"<<endl;
    cin >> answer;

    if(answer=="exit")
    {
     cout << "Bye!"<<endl;
    }

    else if(answer=="person") //s1
    {
     cout << "Ok!  On command: "<<endl;
     s1.special_power();
     s1.give_info();
    }

    else if(answer=="daughter")//d1
    {
     cout << "\n-Ok!  What do you want to say to the daughter?"<<endl;
     cin>>answer;

     d1.ask_use_power(answer);
     d1.ask_give_info(answer);
    }

    else if(answer=="granddaughter")//g1
    {
     cout << "\n-Ok!  What do you want to say to the granddaughter?"<<endl;
     cin>>answer;

     g1.gain_access(answer);
    }

    else
    {
     cout<<"No one here by that name..try again.\n"<<endl;
    }
   }

    return 0;
  }
```

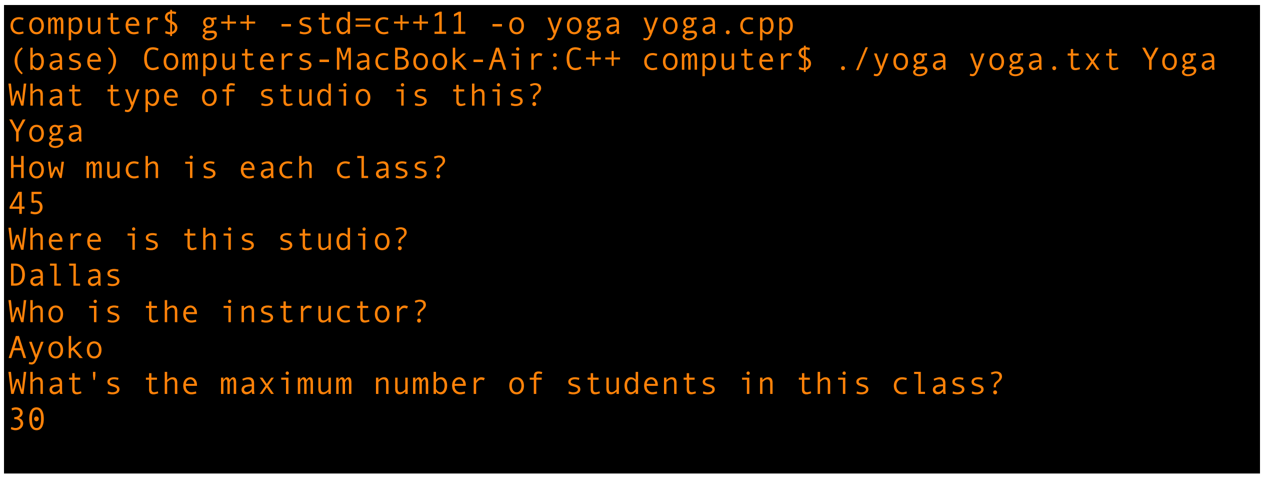
Program 2:

(Simple example showing multiple inheritance followed by a program)

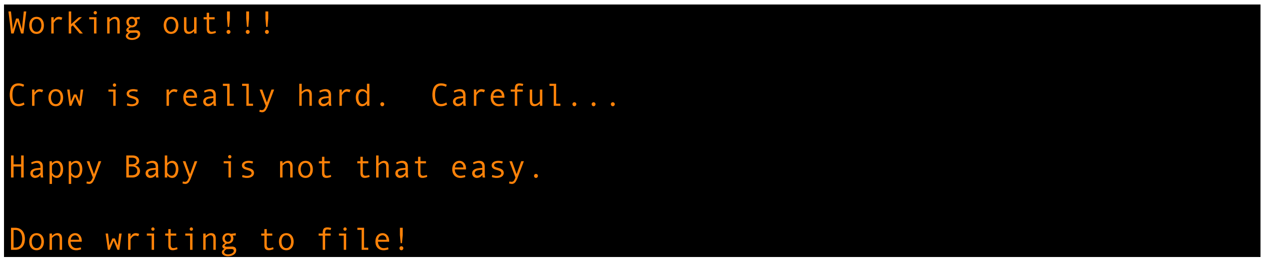
```
computer$ g++ practice.cpp
computer$ ./a.out word
3 66
```

```
#include <iostream>

using namespace std;

class A{

protected: //only accessible to children/derived classes
        int y;

  private: //only visible in this class, not children
        int z;

  public:
        int x;

       void set_y_value(int n)
       {
         y=n;
       }

      void set_z_value(int n)
      {
        z=n;
      }
};

class B{
        protected: //only accessible to children/derived classes
                     int a;

          private: //only visible in this class, not children
                     int b;

          public:
                     int c;

               void set_a_value(int n)
               {
                 a=n;
               }

              void set_b_value(int n)
              {
                b=n;
```

```
            }
};

//public inheritance-inheriting from both classes (A and B)
class C: public A, public B{
        protected:  //only accessible to children/derived classes
                int q;

         private: //things only this class should have (base here)
                int r;

         public:
                int s;

             void print_from_A()
             {
                    cout<<y<<" "<<x<<" "<<endl;
                   //cout<<z<<endl; //can't do z from A (because private, so can't access here)
             }

            void set_r_value(int n)
            {
              r=n;
            }
};



int main(int argc, char **argv) {

        C c_class;
        c_class.set_y_value(3);  //using a function from A (inherited)
        c_class.x=66;  //using a variable from A class (inherited)
        c_class.c=77; //using a variable from B class (inherited)
        c_class.print_from_A(); //this function is in the C class

}
```



```
computer$ g++ -std=c++11 -o yoga yoga.cpp
(base) Computers-MacBook-Air:C++ computer$ ./yoga yoga.txt Yoga
What type of studio is this?
Yoga
How much is each class?
45
Where is this studio?
Dallas
Who is the instructor?
Ayoko
What's the maximum number of students in this class?
30
```



```
Working out!!!

Crow is really hard.  Careful...

Happy Baby is not that easy.

Done writing to file!
```

```
#include <iostream>
#include <map>
#include <fstream>
#include <sstream>

class Class_studio{

protected:
        std::string location;
        int max_size;
        std::string instructor;

public:

        int get_size()
        {
                return max_size;
        }

        std::string get_location()
        {
                return location;
        }


};

class Exercise{

protected:
        std::string type;
        std::map<std::string,int> moves; //move, level of diff

public:
        void do_exercise()
        {
                std::cout<<"\nWorking out!!!\n"<<std::endl;
        }

        void make_exercise_list(std::string filename)
        {
                std::ifstream inFile;
                std::string line_from_file, move, intermediate;
                int level;
                inFile.open(filename);

                if (!inFile.is_open()) {

                                std::cout << "Unable to open file";
                                exit(1); // terminate with error-we will learn about exceptions later

                }

                while (!inFile.eof())
                {
                                getline(inFile, line_from_file);
                                std::stringstream delimt_line(line_from_file);
                                getline(delimt_line,intermediate,',');
                                move=intermediate;

                                getline(delimt_line,intermediate);
                                level=stoi(intermediate);
                                moves.insert({move,level});
                }
        }

};

class Exercise_class: public Class_studio, public Exercise{
        float price_per_class;
        std::string class_type;

        public:
                void register_studio(std::string studio_type)
                {
                        std::cout<<"What type of studio is this?\n"<<studio_type<<std::endl;
                        class_type=studio_type;

                        std::cout<<"How much is each class?"<<std::endl;
                        std::cin>>price_per_class;

                        std::cout<<"Where is this studio?"<<std::endl;
                        std::cin>>location; //from the base class Class_studio

                        std::cout<<"Who is the instructor?"<<std::endl;
                        std::cin>>instructor; //from the base class Class_studio

                        std::cout<<"What's the maximum number of students in this class?"<<std::endl;
                        std::cin>>max_size; //from the base class Class_studio

                }

                float get_price_per_class()
                {
```

```
            return price_per_class;
        }

        void show_level(std::string move)
        {
                switch(moves.at(move))
                {
                    case 0: std::cout<<move<<" is very easy!\n"<<std::endl;
                                    break;

                    case 1: std::cout<<move<<" is not that easy.\n"<<std::endl;
                                    break;

                    case 2: std::cout<<move<<" is really hard.  Careful...\n"<<std::endl;
                                    break;

                    default: std::cout<<"Unknown...\n"<<std::endl;

                }
        }

        void output_info(std::string output_filename)
        {
            std::ofstream outf(output_filename);

             // If we couldn't open the output file stream for writing
            if (!outf)
            {
                // Print an error and exit
                 std::cout << "File write unsuccessful!  Exiting..." << std::endl;
                 exit(1);
             }

             outf << "--Class Info:--" << std::endl;
            outf << "Type: "<<class_type<<std::endl;
            outf << "Price: "<<price_per_class<<std::endl;
            outf << "Location: "<<location<<std::endl;
            outf << "Instructor:"<<instructor<<std::endl;
            outf << "Max size: "<<max_size<<std::endl;

            outf.close();
            std::cout<<"Done writing to file!"<<std::endl;
        }


};



int main(int argc, char **argv) {
	Exercise_class e1;
	e1.register_studio(argv[2]);
	e1.make_exercise_list(argv[1]);
	e1.do_exercise();
	e1.show_level("Crow");
	e1.show_level("Happy Baby");
	e1.output_info("/Users/computer/Desktop/class_info.txt"); //Saving it to my desktop in a file
named class_info.txt (giving path name)
}
```