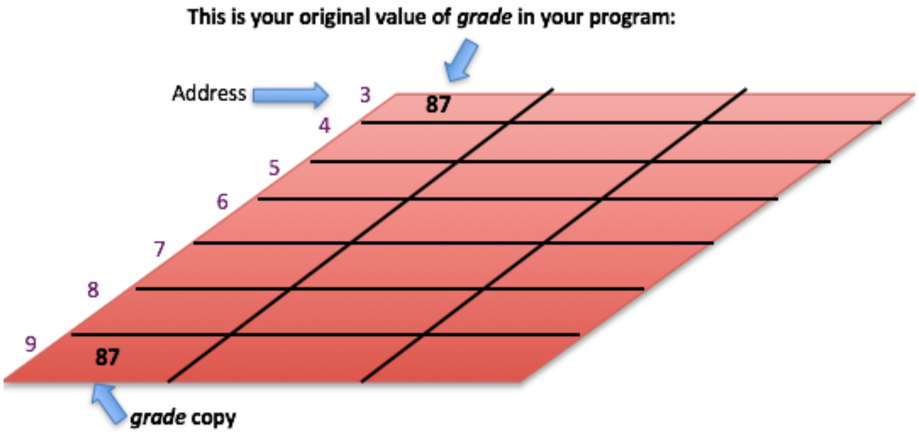
Inheritance (+references)

References:

Pass by Value vs Pass by Reference

- **Passing by value:**
  - The work you do in a function (on a variable for example) no longer exists when the function exits
  - This is because you are working on a copy and not the actual value itself (see below)
- **Passing by Reference:**
  - The work you do in a function (on a variable for example) DOES exist when the function exits
  - This is because you are working on the actual value itself (not a copy)

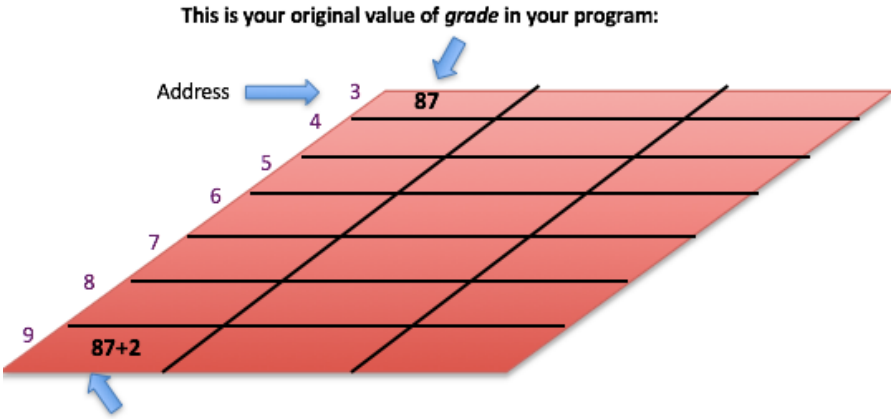
Pass by value example:



This is your original value of *grade* in your program:

***grade* copy**

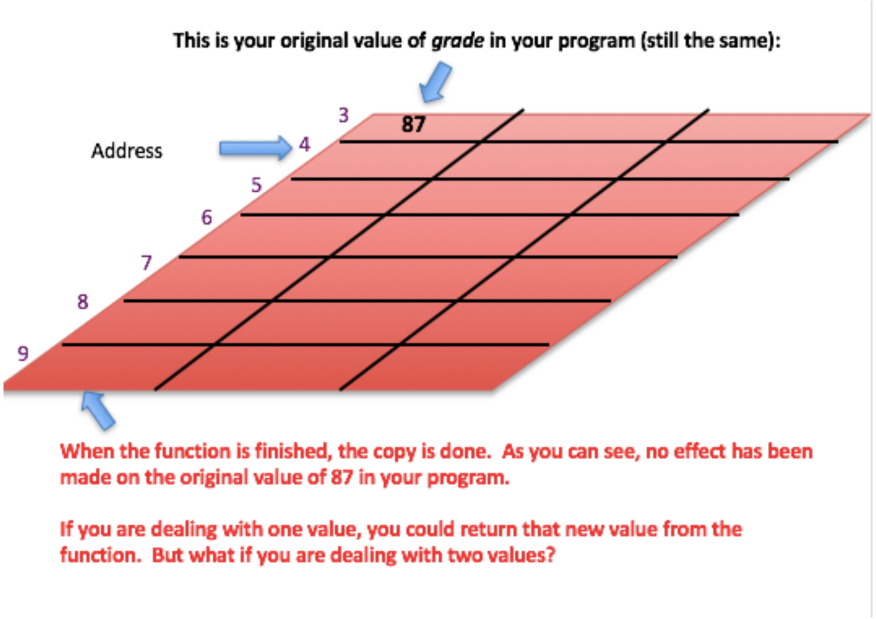
The function makes a copy of *grade (when you pass it as a parameter)* and does any changes to THAT COPY ALONE.



This is your original value of *grade* in your program:

The function makes a copy of *grade* and does any changes to THAT COPY ALONE (example: add 2 to the value. Two is being added to the copy, not the actual value of grade)

This is your original value of *grade* in your program (still the same):



When the function is finished, the copy is done. As you can see, no effect has been made on the original value of 87 in your program.

If you are dealing with one value, you could return that new value from the function. But what if you are dealing with two values?

In C++, we can have something called a *reference*

- An alias to an existing variable ***(also see Stroustrup's Glossary definition)***
  - Like an "address nickname" for a variable-still talking about the same variable but calling it by a different name (the address)
- Main purpose: support pass by reference
- Similar to a pointer but doesn't change what it is pointing at
  - With pointers, we can change what it is pointing at (it is a variable meant to hold any address)
  - By using a reference, we are always talking about the same thing

Examples:

```
computer$ g++ -std=c++14 practice.cpp
computer$ ./a.out
2
3
```

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

using namespace std;

void change(vector <int> &v) //we use & in the function parameter to indicate we are passing by reference (so we can modify the vector).
{
  v.push_back(4);

}

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

  vector<int> stuff={3,4};
  vector <int> &s=stuff; //s is a reference for stuff
```

```
   cout <<stuff.size()<<endl;
   change(s); //we can now use s to pass stuff by reference (meaning any changes in the function will be
 reflected on the actual vector stuff-not just a copy (it will last past the function call)
   cout <<stuff.size()<<endl; //notice the size is larger by 1 now
 }
```

Sample run: *(size of vector changes by one since we added an element in the function)*
***Note: I am using a different standard of C++ to allow initialization of the vector with 3 and 4.*

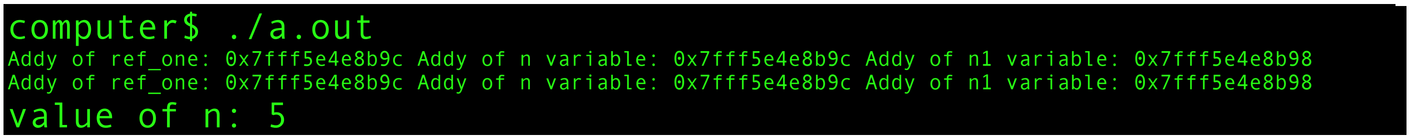
We can also just pass in the vector (or whatever variable we are using) directly:

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

using namespace std;

void change(vector <int> &v)
{
  v.push_back(4);

}

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

  vector<int> stuff={3,4};
  cout <<stuff.size()<<endl;
  change(stuff); //notice I am just passing the vector stuff directly in-it knows it is by reference because the
parameter of the function uses the &
  cout <<stuff.size()<<endl;
}
```

Notice you can change the value of a variable through its reference:

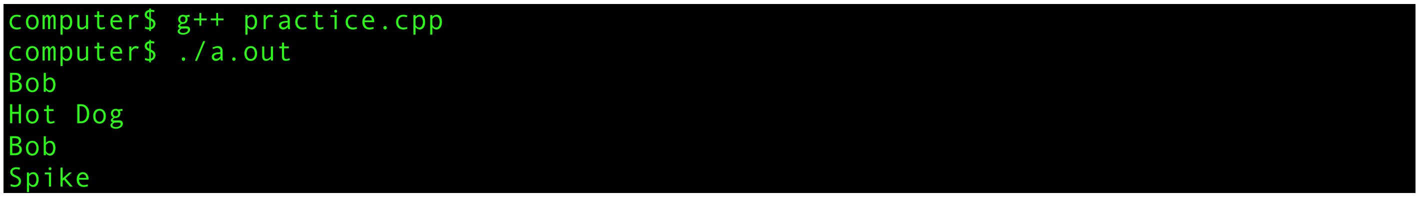
```
#include <iostream>
using namespace std;


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

  int n=3;
  int n1=5;

  int &ref_one=n; //ref_one is now a reference to the variable n (has the same addy)
  cout << "Addy of ref_one: "<<&ref_one <<" Addy of n variable: "<<&n<<" Addy of n1 variable:
"<<&n1<<endl; //same addy
```

```
  ref_one=n1; //changing the value of n (through reference) to the value of n1 (5)-not reassigning the reference (it is still a reference to n)
  cout << "Addy of ref_one: "<<&ref_one <<" Addy of n variable: "<<&n<<" Addy of n1 variable: "<<&n1<<endl; //same addy...

  cout << "value of n: "<<n<<endl; //but diff value (we changed the value of the variable referred to by ref_one)

}
```



```
computer$ ./a.out
Addy of ref_one: 0x7fff5e4e8b9c Addy of n variable: 0x7fff5e4e8b9c Addy of n1 variable: 0x7fff5e4e8b98
Addy of ref_one: 0x7fff5e4e8b9c Addy of n variable: 0x7fff5e4e8b9c Addy of n1 variable: 0x7fff5e4e8b98
value of n: 5
```

Using classes:

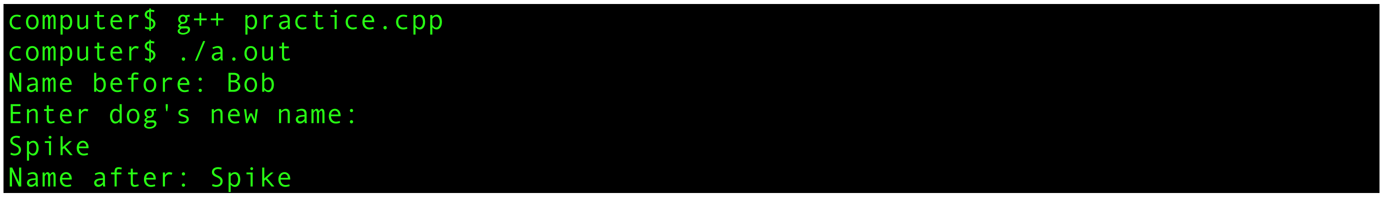


```
computer$ g++ practice.cpp
computer$ ./a.out
Bob
Hot Dog
Bob
Spike
```

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

using namespace std;

class Dog{

public:
  string name;

};

//name won't change because passing by value
void change_name_no(Dog dg)
{
  dg.name="Dogz";
}

//name will change because passing in by reference
void change_name_yes(Dog &dg)
{
  dg.name="Hot Dog";
}
```

```
    //can also pass by pointer (like in C).  Notice the parameter is a pointer here
    void change_name_yes(Dog *dg)
    {
      dg->name="Spike"; //Remember that -> really means: (*dg).name
    }

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

      Dog d;
      d.name="Bob";
      change_name_no(d);
      cout<<d.name<<endl; //name is still Bob-no change

      change_name_yes(d);
      cout<<d.name<<endl; //name changed in function

      d.name="Bob";
      cout<<d.name<<endl; //change name back to Bob

      Dog *ptr=&d;

      change_name_yes(ptr); //using pointer to change
      cout<<d.name<<endl;

}
```

Now that we can pass by reference, we can really start having our classes interact with each other (and not relying on our main to be the middle man):



```
computer$ g++ practice.cpp
computer$ ./a.out
Name before: Bob
Enter dog's new name:
Spike
Name after: Spike
```

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

    using namespace std;

    class Dog{

    private:
      string name;
```

```
public:
  void set_name(string n)
  {
    name=n;
  }

  string get_name()
  {
    return name;
  }

};


class Owner{

public:
  string name;

  void change_name_yes(Dog &dg) //now an Owner object can actually change a value in a Dog object-true interaction between objects (we don't have to change the Dog object in the main-we can actually change it with the Owner).  It would be like in the real world where an owner adopts a dog and gives it a new name
  {
      string answer;
      cout <<"Enter dog's new name: "<<endl;
      cin >> answer;
      dg.set_name(answer);
  }

};


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

  Dog d;
  Owner o;

  d.set_name("Bob");
  cout << "Name before: "<<d.get_name() <<endl;
  o.change_name_yes(d); //the Owner is actually changing the Dog's name
  cout << "Name after: "<<d.get_name() << endl;
```

Returning references:

Notice the size of our vector *stuff* in the Bag class doesn’t change (not using any references):

```
computer$ g++ practice.cpp
computer$ ./a.out
Size: 0
Size: 0
```

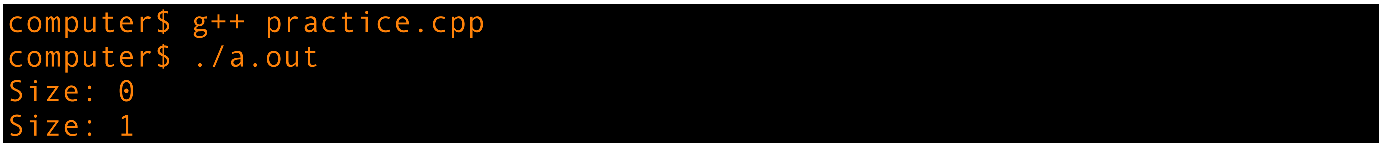
```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class Bag{
    vector<string> stuff;

public:
    vector<string> get_stuff()
    {
        return stuff;
    }

    void size_stuff()
    {
        cout<<"Size: "<<stuff.size()<<endl;
    }
};

class Person{

public:
    void add_bag(Bag b)
    {
        b.get_stuff().push_back("item");
    }

};


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

    Person p1;
    Bag b1;

    b1.size_stuff();
    p1.add_bag(b1);
    b1.size_stuff();

}
```

The size of our vector *stuff* in the Bag class <u>still</u> doesn't change (even though we are passing in a reference to change the vector by adding an item):

```
computer$ g++ practice.cpp
computer$ ./a.out
Size: 0
Size: 0
```

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

using namespace std;

class Bag{
        vector<string> stuff;

public:
        vector<string> get_stuff() //returning a copy
        {
                return stuff;
        }

        void size_stuff()
        {
                cout<<"Size: "<<stuff.size()<<endl;
        }
};

class Person{

public:
        void add_bag(Bag& b) //we are passing in the actual Bag object but…
        {
                b.get_stuff().push_back("item"); //we are using get_stuff() to return a the vector, but it is
not the actual vector-it is a copy.  We are modifying that copy, not the actual vector
        }

};


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

        Person p1;
        Bag b1;

        b1.size_stuff();
        p1.add_bag(b1);
        b1.size_stuff();

}
```

NOW it changes (when we return a reference to the vector AND pass in reference to the function using Bag):



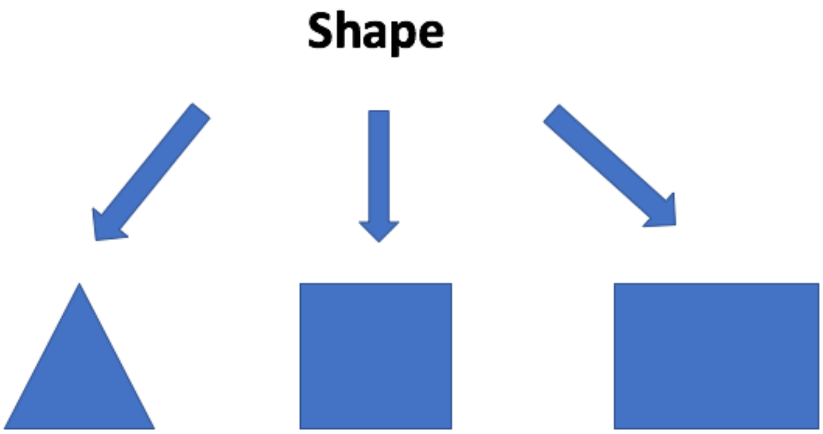
```
computer$ g++ practice.cpp
computer$ ./a.out
Size: 0
Size: 1
```

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Bag{
        vector<string> stuff;

public:
        vector<string>& get_stuff() //returning a reference to the actual vector stuff, not a copy (so we can modify it)
        {
                return stuff;
        }

        void size_stuff()
        {
                cout<<"Size: "<<stuff.size()<<endl;
        }
};

class Person{

public:
        void add_bag(Bag& b) //passing in the actual Bag object (using a reference)
        {
                b.get_stuff().push_back("item");
        }

};


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

        Person p1;
        Bag b1;
        b1.size_stuff();
        p1.add_bag(b1);
        b1.size_stuff();

}
```

Inheritance:

- Derived (child/sub) classes can inherit qualities/characteristics from base (parent/super) classes
    - Create new classes using existing classes
- Reusability of code
- We can change something once (in our base class) instead of for each derived class
- Different ways to inherit (we will talk about this next lecture)

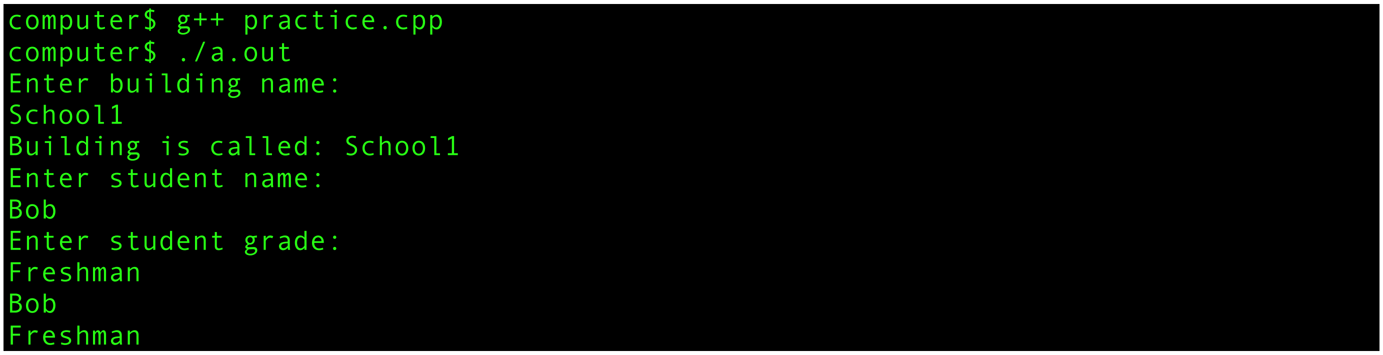
Inheritance – Reuse and extension of fields and method implementations from another class **(also see Stroustrup's glossary for his definition).**



There are things that all shapes have in common. We can put that in our base class (Shape)

Now all our derived classes also have those things.

Program 1:



```
computer$ g++ practice.cpp
computer$ ./a.out
Enter building name:
School1
Building is called: School1
Enter student name:
Bob
Enter student grade:
Freshman
Bob
Freshman
```

```
#include <iostream>
#include <string>
```

```
class Person{
public:
  std::string name;

};

class Student: public Person{
public:
  std::string grade;

};


class Building{
  std::string name;

public:

  void print_name()
  {
    std::cout <<"Building is called: "<<name << endl;
  }

  void set_name()
  {
    std::cout << "Enter building name: " << endl;
    std::cin >>name;
  }

};

class School: public Building{

public:
  void enroll_student(Student &s) //if we don’t have this reference, then the name changes will not be around after the function call is over (it will still be Pat and Senior)
  {
    std::cout<<"Enter student name:"<<endl;
    std::cin>>s.name;

    std::cout<<"Enter student grade:"<<endl;
    std::cin>>s.grade;
  }

  void see_student(Student s1)
  {
    std::cout<<s1.name<<endl;
    std::cout<<s1.grade<<endl;
  }
};

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

  School s1;
  s1.set_name();
  s1.print_name();

  Student stud1;
  stud1.name="Pat";
  stud1.grade="Senior";

  s1.enroll_student(stud1);
  s1.see_student(stud1);

}
```