CS 162C++
Virtual Methods

One benefit of inheritance is virtual methods and polymorphism.

Example

Consider the following example

```
#include <iostream>

class NPC {
public:
    std::string action() { return "Do nothing."; }
};

class ShopKeeper : public NPC {
public:
    std::string action() { return "Sell something."; }
};

class Guide : public NPC {
public:
    std::string action() { return "Give directions."; }
};

int main()
{
    const int SIZE = 3;
    NPC * actors[SIZE];

    actors[0] = new NPC;
    actors[1] = new ShopKeeper;
    actors[2] = new Guide;

    for(int i = 0; i < SIZE; i++ )
        std::cout << "Actor " << i << " action: " << actors[i]->action() << std::endl;

    return 0;
}
```

output:

```
Actor 0 action: Do nothing.
Actor 1 action: Do nothing.
Actor 2 action: Do nothing.
Program ended with exit code: 0
```

Since the array of actors is of type **NPC**, then when the `action()` method is called for each element of the array it returns the method for type `NPC`, not the `action()` method for `ShopKeeper` or `Guide`. If we instead create a virtual method for action as shown below, the compiler adds code that checks for what type of object is instantiated in the array at run time rather than defining the interface at compile time. This increases the overhead of the function calls, but adds functionality as shown below:

```
#include <iostream>

class NPC {
public:
    virtual std::string action() { return "Do nothing."; }
};

class ShopKeeper : public NPC {
public:
    std::string action() { return "Sell something."; }
};

class Guide : public NPC {
public:
    std::string action() { return "Give directions."; }
};

int main()
{
    const int SIZE = 3;
    NPC * actors[SIZE];

    actors[0] = new NPC;
    actors[1] = new ShopKeeper;
    actors[2] = new Guide;

    for(int i = 0; i < SIZE; i++ )
        std::cout << "Actor " << i << " action: " << actors[i]->action() << std::endl;

    return 0;
}
```

output:

```
Actor 0 action: Do nothing.
Actor 1 action: Sell something.
Actor 2 action: Give directions.
Program ended with exit code: 0
```

Notice that now we are getting the correct action for each of the child types, not the type of the parent. We could define the methods in the children to be virtual also and their children could be differentiated.

As you have noticed during this term, when Code::Blocks sets up a new class for you, it always makes the destructor virtual – this is so that if you delete an object via a pointer to a parent type, the correct destructor is always called.

One thing to remember is that when you are accessing a child object through a parent pointer, you can **only** access those things that they **have in common**. Obviously, the parent class does not know about or have methods to access things that were defined in the child.

Pure Virtual Methods and Abstract Classes

If you think of the above example, you would never create a generic NPC in a game, you would always create a child element. Just like you never see a mammal in nature, you always see a horse, cow, whale, or human.

When we are defining a generic parent that will never be instantiated, we can make the virtual method a **pure virtual method**. This causes the parent class to be an **abstract class**.

In the following example, **NPC is now an abstract class**, so we had to comment out the line that defined it and change the loop to not access that element of the array.

```
#include <iostream>

    // abstract class
class NPC {
public:
        // pure virtual method
    virtual std::string action() = 0;
};

    // non-abstract class, has definition for action
class ShopKeeper : public NPC {
public:
    std::string action() { return "Sell something."; }
};

    // non-abstract class, has definition for action
class Guide : public NPC {
public:
    std::string action() { return "Give directions."; }
};

int main()
{
    const int SIZE = 3;
    NPC * actors[SIZE];

    //actors[0] = new NPC;  // commented out, can not instantiate a member of an abstract
class
    actors[1] = new ShopKeeper;
    actors[2] = new Guide;

    // loop starts at 1 to avoid undefined location
    for(int i = 1; i < SIZE; i++ )
        std::cout << "Actor " << i << " action: " << actors[i]->action() << std::endl;

    return 0;
}
```

output:

```
Actor 1 action: Sell something.
Actor 2 action: Give directions.
Program ended with exit code: 0
```