CS302 - Data Structures

using C++

Topic: C++ Classes

Kostas Alexis

AUTONOMOUS ROBOTS LAB

Purpose

- Quick overview of basics of C++ classes and templates
- Understand role of Header and Source files
- Understand the importance of using templates

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};
```

Preventing C++ compiler errors – once declared ten

```
#endif
```

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};
```

All data fields declared **private**

```
#endif
```

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};
```

```
#endif
```

All data fields declared **private**

Keep all data fields hidden from client

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};
```

All data fields declared **private**

Accessor methods are declared **const**

```
#endif
```

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

pre-processor directives

```
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};
```

```
#endif
```

All data fields declared **private**

Accessor methods are declared **const**

Compiler will check the implementation that we are not changing any of the data fields

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

Parameters are passed by **constant reference**

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

Parameters are passed by **constant reference**

We don't want to allow clients to pass objects to our methods by reference as our method will then be able to change objects the client owns

PlainBox.h

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Parameters are passed by **constant reference**

We don't want to allow clients to pass objects to our methods by reference as our method will then be able to change objects the client owns

Passing by constant reference protects client

PlainBox Class

Basic class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "PlainBox.h"
```

Include header file

```
PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Basic class

Class namespace indicator

Default constructor

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Basic class

```
#include "PlainBox.h"
```

Class namespace indicator

```
PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Default constructor

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Basic class

Class namespace indicator

Goes directly before the name of the constructor or method

It does not go prior to the return type

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Basic class

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Initializer list (name + init value

Default constructor

PlainBox.cpp

PlainBox Class

Basic class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

Initializer list

Default constructor

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ }

Plai

{ }

void
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

```
PlainBox::PlainBox()
{
        item = 0.0;
} // end default constructor
```

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Parameterized constructor

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Method implementations

PlainBox.cpp

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "Pla

PlainBox::Pla
     : item(0.
{ } // end de

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Method implementations

PlainBox.cpp

Client Code

```
double dish = 8.5;
PlainBox firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

double bowl = 4.0;
PlainBox anotherBox = PlainBox(bowl);
anotherBox.setItem(dish);
std::cout << anotherBox.getItem() << std::endl;
```

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "Pla

PlainBox::Pla
     : item(0.
{ } // end de

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Method implementations

PlainBox.cpp

Client Code

```
double dish = 8.5;
PlainBox firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

double bowl = 4.0;
PlainBox anotherBox = PlainBox(bowl);
anotherBox.setItem(dish);
std::cout << anotherBox.getItem() << std::endl;
```

8.5

PlainBox Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const double& theItem);
     void setItem(const double& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

```
#include "Pla

PlainBox::Pla
     : item(0.
{ } // end de

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

Method implementations

PlainBox.cpp

Client Code

```
double dish = 8.5;
PlainBox firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

double bowl = 4.0;
PlainBox anotherBox = PlainBox(bowl);
anotherBox.setItem(dish);
std::cout << anotherBox.getItem() << std::endl;
```

8.5

PlainBox Class – v2 - Templates

What if we don't only want to store doubles?

PlainBox Class – v2 - Templates

What if we don't only want to store doubles?
Use typedef and replace all "double" with "ItemType"

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

typedef double ItemType;
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

typedef gives flexible typing decided by class implementer

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

typedef double ItemType;
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

typedef gives flexible typing decided by class implementer

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

void PlainBox::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

If next time we want the class to take "int" then we only have to change "double" to "int" next to "typedef" and recompile. Advances flexibility and code reusability.

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

typedef double ItemType;
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

typedef gives flexible typing decided by class implementer

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor
```

If next time we want the class to take "int" then we only have to change "double" to "int" next to "typedef" and recompile. Advances flexibility and code reusability.

But still we can't have a single implementation of the class that handles everything what the client wants to implement.

```
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

typedef double ItemType;
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

typedef gives flexible typing decided by class implementer

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor
```

If next time we want the class to take "int" then we only have to change "double" to "int" next to "typedef" and recompile. Advances flexibility and code reusability.

But still we can't have a single implementation of the class that handles everything what the client wants to implement.

```
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

typedef double ItemType;
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#endif
```

PlainBox.h

typedef gives flexible typing decided by class implementer

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

} // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

If next time we want the class to take "int" then we only have to change "double" to "int" next to "typedef" and recompile. Advances flexibility and code reusability.

But still we can't have a single implementation of the class that handles everything what the client wants to implement.

Use templates instead!

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

PlainBox.h

template allows client to decide type

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

We now need to include the implementation file PlainBox.cpp in the Header file

That is because template classes are entirely declarations – they are not compiled until the client instantiates and tells the compiler what is to be used (e.g., double)

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

```
#include "PlainBox.h"

PlainBox::PlainBox()
     : item(0.0)
{ } // end default constructor

PlainBox::PlainBox(const double& theItem)
     : item(theItem)
{ } // end constructor

void PlainBox::setItem(const double& theItem)
{
     item = theItem;
}    // end setItem

double PlainBox::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

```
#include "PlainBox.h"

template<class ItemType>
PlainBox<ItemType>::PlainBox()
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

Each of the methods and constructors needs to be preceded by the template declaration statement!

```
#include "PlainBox.h"

template<class ItemType>
PlainBox<ItemType>::PlainBox()
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

PlainBox.h

template allows client to decide type

```
#include "PlainBox.h"

template<class ItemType>
PlainBox<ItemType>::PlainBox()
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

Need to change the namespace indicator to indicate that we don't know what type the PlainBox will hold – we just know it is "ItemType" (template placeholder)

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

Client Code

```
double dish = 8.5;
PlainBox<double> firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

string animal = "Dog";
PlainBox<string> anotherBox = PlainBox<string>(animal);
anotherBox.setItem("Cat");
std::cout << anotherBox.getItem() << std::endl;
```

8.5

```
#include "Pla

template<clas
PlainBox<Item
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

```
#include "Pla

template<clas
PlainBox<Item
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

Client Code

```
double dish = 8.5;
PlainBox<double> firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

string animal = "Dog";
PlainBox<string> anotherBox = PlainBox<string>(animal);
anotherBox.setItem("Cat");
std::cout << anotherBox.getItem() << std::endl;
```

Now storing a string

PlainBox Class – v2 - Templates

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template<class ItemType>
class PlainBox
{
private:
     double item;

public:
     PlainBox();
     PlainBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
     double getItem() const;
};

#include "PlainBox.cpp"
#endif
```

template allows client to decide type

PlainBox.h

Client Code

```
double dish = 8.5;
PlainBox<double> firstBox(dish);
std::cout << firstBox.getItem() << std::endl;

string animal = "Dog";
PlainBox<string> anotherBox = PlainBox<string>(animal);
anotherBox.setItem("Cat");
std::cout << anotherBox.getItem() << std::endl;
```

cat

```
#include "Pla

template<clas
PlainBox<Item
     : item(0.0)
{ } // end default constructor

template<class ItemType>
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
     : item(theItem)
{ } // end constructor

template<class ItemType>
void PlainBox<ItemType>::setItem(const ItemType& theItem)
{
     item = theItem;
}    // end setItem

template<class ItemType>
ItemType PlainBox<ItemType>::getItem() const
{
     return item;
}    // end getItem
```

PlainBox.cpp

Tips to protect the integrity of a class

- Declare all data fields in the private section of the class declaration
- Declare as a const method any method that does not change the object's data fields (accessor methods)
- Precede the declaration of any parameter passed by reference with const, unless you are certain it must be modified by the method, in which case the method should be either protected or private.

Inheritance

- Base classes can be considered as parent classes.
- Derived classes can be consider as children that inherit the properties of their parents and may ad more or override previous functionality.

Inheritance

```
#ifndef TOY_BOX_
#define TOY_BOX_
#include "PlainBox.h"

enum Color {BLACK, RED, BLUE, GREEN, YELLOW,
WHITE};

template<class ItemType>
class ToyBox : public PlainBox<ItemType>
{
private:
     Color boxColor;

public:
     ToyBox();
     ToyBox(const Color& theColor);
     ToyBox(const ItemType& th
     Color& theColor);
     double getColor() const;
};

#include "ToyBox.cpp"
#endif
```

ToyBox.h

```
#include "ToyBox.h"

template<class ItemType>
ToyBox<ItemType>::ToyBox() : boxColor(BLACK)
{ } // end default constructor

template<class ItemType>
ToyBox<ItemType>::ToyBox(const Color& thecColor): boxColor(theColor)
{ } // end constructor

template<class ItemType>
void ToyBox<ItemType>::ToyBox(const ItemType& theItem, const Color&
     thecColor) : PlainBox<ItemType>(theItem), boxColor(theColor)
{
}    // end constructor

template<class ItemType>
Color ToyBox<ItemType>::getColor() const
{
     return boxColor;
}    // end getColor
```

ToyBox.cpp

Inheritance – Overriding Base Class Methods

```
#ifndef MAGIC_BOX_
#define MAGIC_BOX_
#include "PlainBox.h"

template<class ItemType>
class MagicBox : public PlainBox<ItemType>
{
private:
     bool firstItemStored;

public:
     MagicBox();
     MagicBox(const ItemType& theItem);
     void setItem(const ItemType& theItem);
};

#include "MagicBox.cpp"
#endif
```

MagicBox.h

```
#include "MagicBox.h"

template<class ItemType>
MagicBox<ItemType>::MagocBox() : firstItemStored(false)
{ } // end default constructor

template<class ItemType>
MagicBox<ItemType>::MagicBox(const ItemType& theItem):
     firstItemStored(false)
{
     setItem(theItem); // calls MagicBox version of setItem
} // end constructor

template<class ItemType>
void MagicBox<ItemTYpe>::setItem(const ItemType& theItem)
{
     if (!firstItemStored)
     {
          PlainBox<ItemType>::setItem(theItem);
          firstItemStored = true; // Box has magic now
     }
}    // end setItem
```

MagicBox.cpp

Virtual Methods

- Using the keyword virtual in front of the prototype, or header, of the methods informs the C++ compiler that the code this method executes is determined at runtime, not when the program is compiled. This is a virtual method.
- The rules of inheritance allow us to use a derived class anywhere that its base class is used.
- Virtual methods are therefore essential to allow the correct identification of the method to be used for the object at hand.

Thank you