Module 5: Pointers, Abstract Classes and Virtual Functions

Lesson 1: Pointers and Dynamic Arrays

Learning Outcomes

- LO 5.1.1 Declare and initialize pointers.
- LO 5.1.2 Allocate memory for dynamic arrays and determine the operation of deep and shallow copy of pointers.

Introduction to Pointers

So far the memory items that are used are datatypes which can be set be a certain numerical value. Take for example, an `int` datatype can be set with any numerical value from –2147483648 up to 2147483647. To manipulate this numerical value, a memory item must be declared with an `int` datatype. The pointer data type however would not contain a numerical value but rather an address of a memory item.

Declaring pointers

As mentioned earlier, a pointer would only contain a value associated with a memory location of a certain variable. Obviously, this would raise the question how can they be declared? To declare a pointer variable, one must first specify the data type in which the pointer would point, followed by an asterisk (*), then finally the identifier of that pointer.

Syntax	**Example**

`dataType *identifier`	`int *pi;`
	`double *pd;`

The asterisk (*) character can appear anywhere between the space of the data type and identifier. A pointer of type `double` can store the address of memory size `double`.

Reference and dereference operator

The reference operator (i.e. returns the address of the variable) in C++ is a unary ampersand (&) operator. The syntax and example is shown below.

Syntax	**Example**

`pointer = &identifier`	`int *pi;`
	`int x = 10;`
	`pi = &x;`

The dereference operator on the other hand is a unary asterisk (*) operator. The dereference operator would refer to content of the memory in which the pointer points at. For example,

Example

`int *pi;`
`int x = 10;`
`pi = &x;`
`cout<<*pi;//output: 10`

The output operation would display the value that is being pointed to by pointer `p`. So on this case the value 10 would be displayed.

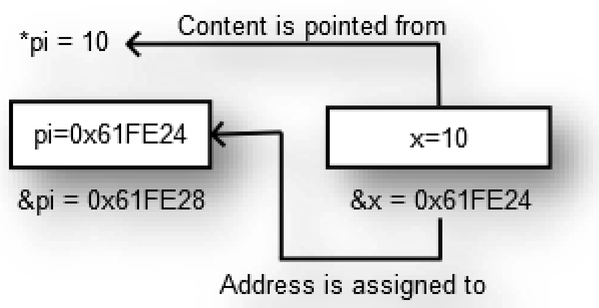

Figure 1: Visualization of pointer assignment and dereferencing

Pointers on classes and structures

The previous section implemented pointers to manipulate simple data types such as `int` and `float`. But pointers aren't limited to primitive datatypes, they can also be declared for other datatypes such as classes and structures.
`Consider the example below:`

Example

```
struct paperType
{
    char name[20];
    int widthMM;
    int lengthMM;
};
```

A pointer to a structure can be created by declaring a name then using it as preceded by a reference operator. To access a member variable of a structure, traditionally a mixture of dereference operator and dot operator is used, together with an open and close parenthesis. This is because the dot operator has a higher order of precedence than the dereferencing operator. Refer to the example below.

Example

```
paperType paper;
paperType *paperPtr;
paperPtr = &paper;
(*paperPtr).widthMM = 297;
```

However this kind of implementation can at some point be confusing because a misplacement of parenthesis would cause syntax error. So in order simplify, C++ provides an operator known as access operator arrow. This consist of a hyphen followed by a greater than symbol.

Syntax	Example
`pointerVariableName->MemberName`	`paperPtr->lengthMM = 210;`

By using the access operator arrow, eliminates the use of parentheses and dereferencing operator that could result to the abnormal termination of a program.

The `new` and `delete` operators

Previous section explained how pointers store the address of a variable and at the same time access the content. So, how are pointers any useful given that you can just simply access the content of a variable without using a pointer? This section would evidently show the power of using pointer. Particularly the means of allocating and deallocating memory during runtime. C++ provides two operators, the `new` and `delete` to allocate and deallocate memory respectively.

The new operator has two uses, to allocate single variables or allocate an array of variables.

Syntax	Example

`new dataType`
`new dataType[intSize]`
	`int *ptr = new int;`
	`*ptr = 5;`
	
	`int *arr = new int[5];`
	`*arr = 1,2,3,4,5;`

The first example allocates a variable in the memory during program execution then the address is stored into `ptr`. The allocated memory can be accessed using the dereferencing operator. The other example shows allocating an integer array of 5 elements on the memory and can be accessed using the `arr` pointer.

Given that you would have the declarations below:

```
int *p = new int;
*p=42;
p = new int;
*p = 12;
```

Suppose that the location of the first integer allocated memory pointed by p is on memory address, 0x1b1a30 with a content 42. Then p pointed to another location with a supposed memory address of 0x6d1ac0 with a content of 12. Now what would happen to the first pointed location? The memory location would become inaccessible but is still remain allocated. This memory cannot be reallocated, thus it would cause what is known as a memory leak. A significant memory leak will happen if too many memory location would exist like this and may cause abnormal program termination.

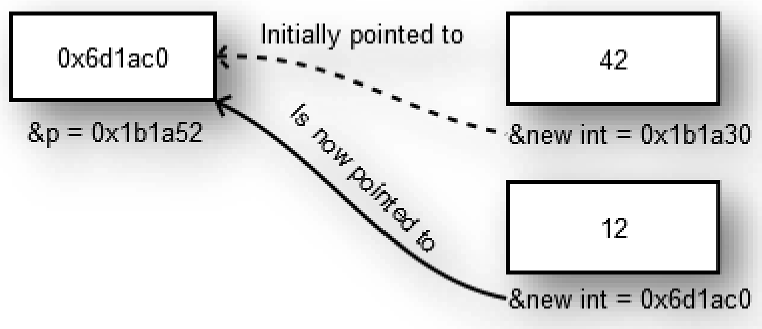

Figure 2: Memory leak caused by an allocated but unused memory space.

To avoid memory leak, C++ uses the `delete` operator to deallocate a memory location.

Syntax	**Example**
`delete pointerVariable`	`delete ptr;`
`delete [] pointerVariable`	`delete [] arr;`

Pointer operations

Pointers are allowed certain operation such as assignment, relational operation and some arithmetic operation. One pointer variable can be assigned to another, given that they are of the same type.

```
int *p = new int;
int *q;
p = q;
```

Pointers can also be compared for equality.

```
    p!=q;
p==q;
```

Arithmetic operation can be performed such as increment, and decrement. The pointer would be incremented or decremented according to their datatype that is an integer would be incremented by 4 bytes, a character by 1 byte an so on.

```
p++;
q--;
```

Dynamic arrays

Dynamic one-dimensional array

The arrays that are mentioned in Chapter 4, mentioned static arrays since the memory size of the array is allocated on compile time. So, every time the program would execute, the array would remain in size. To handle this limitation, one may allocate an array of considerable size during compile time, however this would result to a lot of wasted memory for the unused elements. This section would help you overcome this limitation by allocating memory of the array during runtime or also known as dynamic arrays.

Example

```
int *d_arr;
int arr_size;
cin>>arr_size;
d_arr = new int[arr_size];
for (int i=0;i<arr_size;i++)
{
    d_arr[i] = 0;
}
```

During runtime, this piece of code would allocate an array with a size that the use would input. It will then be initialized all with zeros.

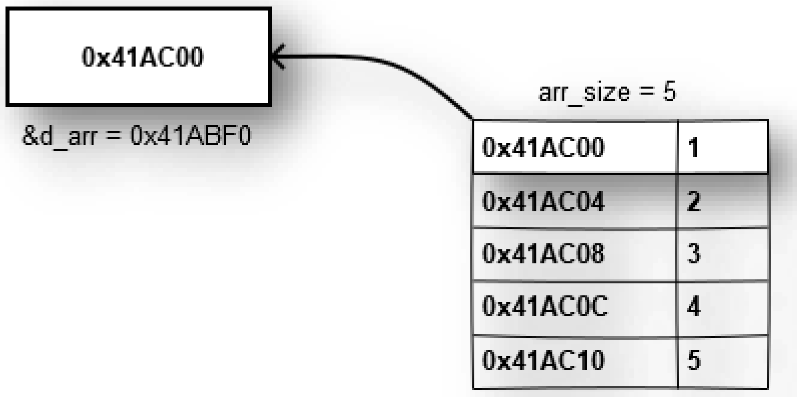

Figure 3: Dynamic array allocation and initialization

Pointers and function

You can pass a pointer to a function by means of pass by value and pass by address. The * to identify the variable as value parameter and * before the & to identify it as a reference parameter.

Example

```
void pointerParams(int* q, int* &p)
{

}
```

In the same way, pointer can also be the return value of a function.

Example

```
int *pointerReturnFunc()
{

}
```

Dynamic two-dimensional array

Dynamic two-dimensional arrays are created similar to a dynamic one-dimensional array. A pointer must point to a pointer in order to dynamically allocate the memory for the number of rows and the number of columns. For example,

Example

```
int **grid;
int row_size=10,col_size=15;
grid=new int* [row_size];
for (int row=0;row<row_size;row++)
{
   grid[row]=new int [col_size];
}
```

The `grid` is a pointer to a pointer. Meaning that the `grid` and `*grid` are pointers. The `grid` can store the address of the pointer or array of pointer. To dynamically allocate the rows, an array of pointers with 10 elements would be created. The next step to create columns by means of a loop to allocate the memory for each row pointer.

Deep-copy vs. shallow-copy pointers

Previous topics shows that when pointer would be used incorrectly, it could lead to accessing another data and would cause erroneous result. Consider the declaration below:

Example

```
int *source;
int *destination;
source = new int[5];
destination = source;
```

After allocating 5 elements of the `source` pointer the next statement copies the address value of the `source` is copied to the `destination` pointer. Now, if you would execute the statement

Example

```
delete [] destination;
```

This would leave the `source` pointer dangling since its content is now being deleted as referenced by the `destination` pointer. This type of copy is called a shallow copy since two or more pointer reference to just one memory allocation.

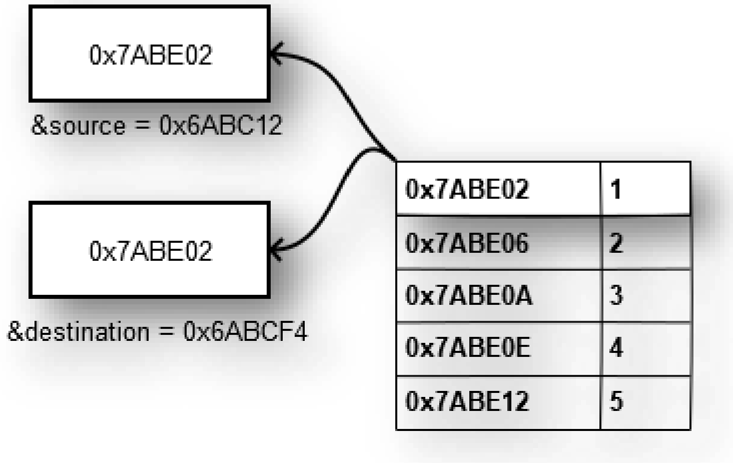

Figure 4: Shallow-copy pointer

However, if the following statements would be executed:

```
        Example
destination = new int[5];
         for (int i=0;i<5;i++)
         destination[i]=source[i];
```

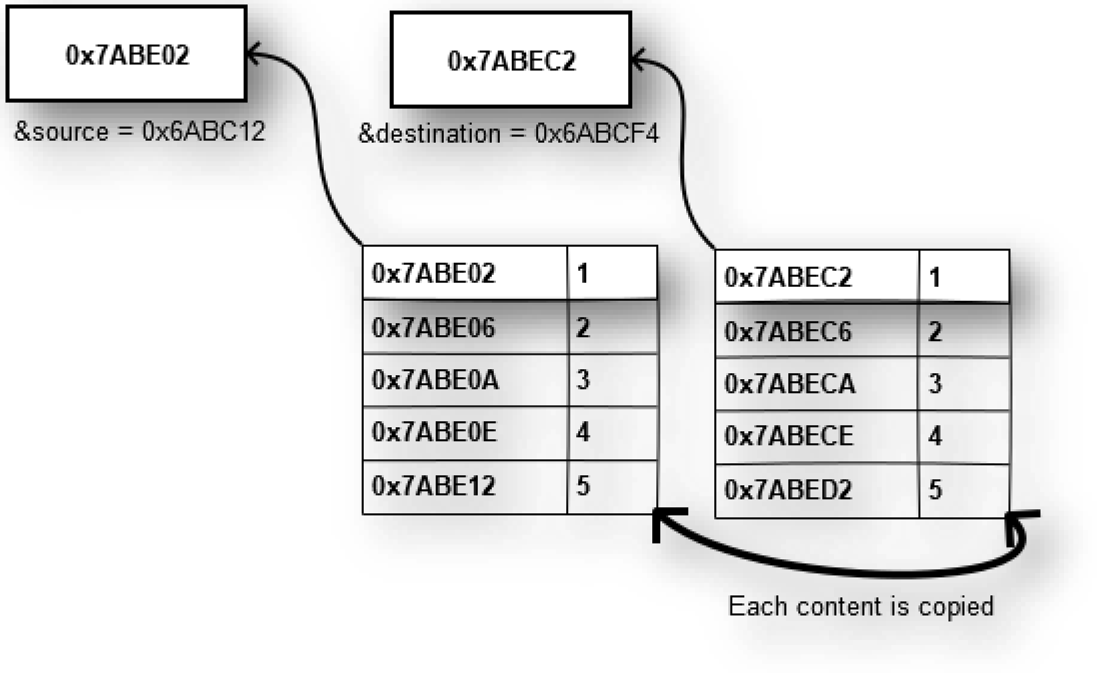

Figure 5: Deep copy pointer

The `destination` and the `source` would now have their own memory allocation. So, if the `destination` is deleted, it will not affect the `source` pointer. This copy is generally known as deep copy, since each pointer has its own data.

Lesson 2: Issues with classes and pointers

Learning Outcomes

- LO 5.2.1 Explore the unusual structure of classes with pointers.

Assignment operators on classes

The previous section covers the arrow notation which would access the pointer variable in a class. So since a class can have a pointer as member variable some peculiarities would emerge. Consider the class implementation below.

Example

```
class classWithPtr
{
    public:
        ~classWithPtr();
    private:
        int *data;
        int dataLen;
};
```

If two objects will be created from the class, then the first object (i.e. cwpObject1) will be assigned to the second (i.e. cwpObject2) as indicated below.

Example

```
classWithPtr cwpObject1, cwpObject2;
cwpObject1 = cwpObject2;
```

Since only the value `cwpObject1.data` is copied to `cwpObject2.data` then only shallow copy is accomplished. Now when `cwpObject1` would go out of scope then the data that is referenced by it would also be inaccessible to `cwpObject2`.

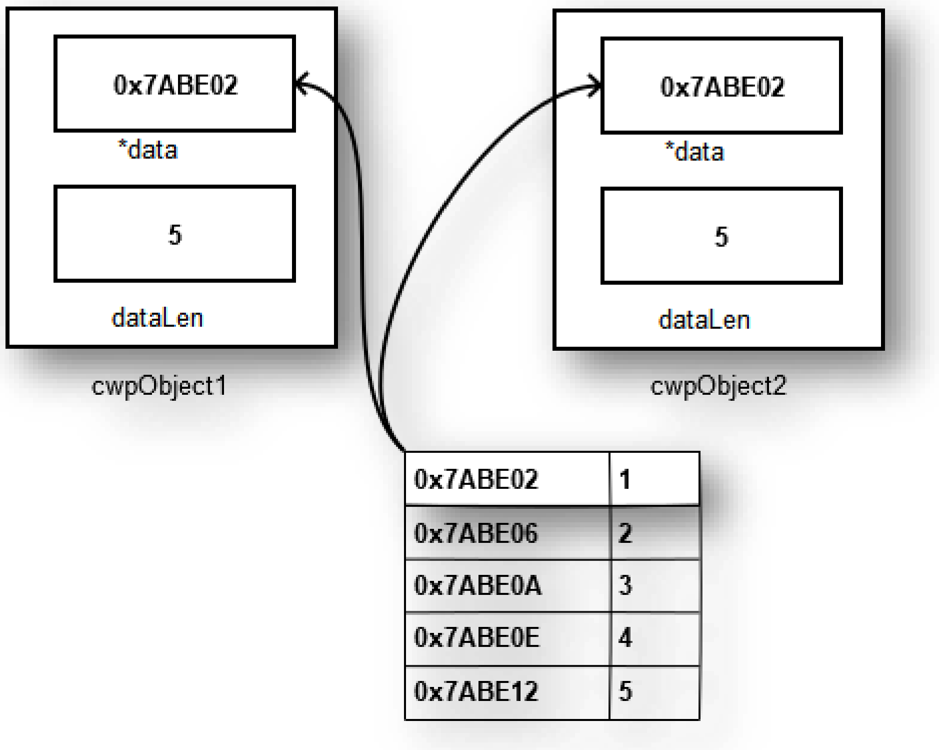

Figure 6: Shallow copy when using the default assignment operator on classes

To avoid this scenario, extending the functionality to the assignment operator so that it would not copy the value of the pointer instead, the data that is referenced by it. This process is referred to as operator overloading. In a way giving both `cwpObject1` and `cwpObject2` with their own data set.

Destructors and delete operation

Suppose that the data pointer creates a dynamic array during execution. If the destructor shown below would destroy `cwpObject1` when it will go out of scope then only the pointer value is deleted.

Example

```
classWithPtr::~classWithPtr()
{
}
```

But the space which contains the dynamic array itself would remained marked as used, so to deallocate the memory for the dynamic array then use the delete operator inside the destructor

Example

```
classWithPtr::~classWithPtr()
{
    delete [] data;
}
```

Deep copy constructors

Constructors by default would perform member-wise initialization. However this would lead to shallow copying of data allocated. To perform deep copy of a pointer, overriding the default constructor must be done by using a copy constructor. It must execute the following operation

- Initializing an object from the value of the other object.
- When an object is passed by value
- When an object is the return value.

This would ensure that one object has its own data and not just share it with another object. The general syntax is as follows:

Syntax

```
className(const className& otherObjectdelete [] );
```

For example:

Example

```
class classWithPtr
{
    public:
        ~classWithPtr();

        //fill with zeros the allocated data
        void zeroFill();

        //fill the allocated data as specified by n
        void customFill(int n);

        //print the content of the allocated data
        void print();

        classWithPtr(int s=10);

        //copy constructor
        classWithPtr(const classWithPtr &otherPtrObj);

    private:
        int *data;
        int dataLen;
};


//implementation of class classWithPtr
#include "classWithPtr.h"
#include <iostream>
#include <cassert>
using namespace std;

classWithPtr::classWithPtr(int s)
{
    if (s<=0)
    {
       cout<<"No   negative   sizes   allowed.   Default:
10"<<endl;
       dataLen = 10;
    }

    else
        dataLen = s;

    data = new int[dataLen];

}


classWithPtr::classWithPtr(const           classWithPtr
&otherPtrObj)
{
    dataLen = otherPtrObj.dataLen;
    data = new int[dataLen];

    for (int i=0;i<dataLen;i++)
        data[i] = otherPtrObj.data[i];
}

void classWithPtr::zeroFill()
{
    assert(dataLen>0);
    if (dataLen>0)
        for (int i=0;i<dataLen;i++)
           data[i] = 0;

}

void classWithPtr::customFill(int n)
{
    assert (dataLen>0);
    if (dataLen>0)
        for (int i=0;i<dataLen;i++)
           data[i] = n;

}

classWithPtr::~classWithPtr()
{
    delete [] data;
}

void classWithPtr::print()
{
    assert(dataLen>0);
    if (dataLen>0)
       for (int i=0;i<dataLen;i++)
          cout<<data[i]<<",";
    cout<<endl;
}

#include <iostream>
#include "classWithPtr.h"
int main()
{
    classWithPtr cwtPtr(5);
```

```
           cwtPtr.zeroFill();

           classWithPtr cwtPtr1 (cwtPtr);
           cwtPtr1.print();//output: 0,0,0,0,0,
           cwtPtr.customFill(5);
           cwtPtr.print();//output: 5,5,5,5,5,
           return 0;
 }
```

This example would first allocate 15 integer elements to `cwtPtr` object then fill it with zero using the `zeroFill()` function. The copy constructor would then allocate the same amount of memory (deep copy) for `cwtPtr1` object. By calling the `customFill(5)` function for `cwtPtr` object, the content of `cwtPtr` will not be the same as `cwtPtr1`.
In general if a class has a pointer as member variable you must:

- Overload the assignment operator
- Include a pointer destructor on the class
- On top of a regular constructor, include a copy constructor.

Lesson 3: Virtual Functions and Abstract Classes

Learning Outcomes

- LO 5.3.1 Examine the operation of virtual functions and implement of abstract classes.

Virtual Functions

Previous chapters talked about the necessary match for the formal parameter and actual parameter, which is datatype of the formal parameter must be the same with the datatype of the actual parameter. However in C++, in the case of classes the object from a derived class can be the actual parameter of a formal parameter which is its base class. Consider the following class declaration.

Example

```
//from player.h
#ifndef PLAYER_H_INCLUDED
#define PLAYER_H_INCLUDED
#include <iostream>
#include <string>
using namespace std;

class player
{
    public:
        player(string    p="
");
        void print();

    private:
        string name;

};
#endif // PLAYER_H_INCLUDED
```

With the implementation details as follows:

Example

```
//from playerImp.cpp
#include "player.h"

player::player(string p)
{
   name = p;
}

void player::print()
{
cout<<name;
}
```

The player class is then inherited to character class as indicated below.

Example

```
#include "player.h"
#ifndef CHARACTERTYPE_H_INCLUDED
#define CHARACTERTYPE_H_INCLUDED
```

```
class character:public player
{
    public:
        character(string p ="",string c=" ");
        void print();
    private:
        string name;
};
```

Now, the implementation would be:

Example

```
#include "characterType.h"
character::character(string p, string c):player(p)
{
    name = c;
}

void character::print()
{
    player::print();
    cout<<"--"<<name<<" ";
}
```

If the client code would have a function that would pass by reference the player object, then it would work fine if the base class would be passed as actual parameter.

Example

```
void funcRefPrint(player &p)
{
    p.print();
}
…
player p1("PlayerOne");
funcRefPrint(p1); //output: PlayerOne
```

However if the derived class would be passed as actual parameter then the print function that is referenced would now be from the base class.

Example

```
character c1("PlayerTwo","Palladin");
funcRefPrint(c1); //output: PlayerTwo
```

This is because binding of the function happens during compile time. That is `funcRefPrint(player &p`) binds the `print()` function from the base class since it is type used on the formal parameter. This compile-time binding is otherwise known as ***early binding*** or ***static binding***.

The logical operation for this scenario must be that the function in print() on the derived class must be bind to the `funcRefPrint(player &p)`. C++ provides ***virtual function*** as feature to address this problem. The binding would occur on runtime rather than on compile time, otherwise known as ***late binding*** or ***dynamic binding***.

By adding the reserved word, virtual to the print() function on the base class as indicated below.

Example

```
class player
{
    public:
        player(string p=" ");
        virtual void print();

    private:
        string name;

};
```

By executing the same code as before, specifically for c1 object. The output would be the `print()` function of the derived class.

Example

```
 funcRefPrint(c1); //output: PlayerTwo--Palladin
```

Virtual functions would also work on when parameter that would be used is a pointer. Take for example:

Example

```
 void funcPtrPrint(player *p)
        {
              p->print();
        }
       …

        player *p_ptr = new player ("PlayerPtrOne");
       character *c_ptr = new character ("PlayerPtrTwo","Mage");
       funcPtrPrint(p_ptr);//output: PlayerPtrOne
       cout<<endl;
       funcPtrPrint(c_ptr);//output: PlayerPtrTwo—Mage
```

However, virtual functions would not work if pass by value would be implemented. This is because an actual copy of the object is made by the function, not just a reference. Thus the formal parameter as base class permanently binds to actual parameter. For instance:

Example

```
void funcValPrint (player p)
{
    p.print();
}

      funcValPrint(p1);//output: PlayerOne
cout<<endl;
funcValPrint(c1);//output: PlayerTwo
```

For a base class that have a virtual constructor, it is necessary that its destructor should also be virtual. This is because when a derived class passed as reference parameter would go out of scope it would destroy both the variables of the derived class and base class (i.e. member variables inherited from the base class).

Abstract Classes

The previous section introduced virtual function that would enforce run-time binding. By inheritance a classes would not be made from scratch instead are derived from a base class. The derived class can redefine, override, public or protected members of the base class. However, there are functions that are available on base class that seem meaningless to definition in the base class. Take for example the class `space` shown below:

Example

```
class space
{
    public:
        space(int dims);
        double fieldOfCoverage();
        int getDimensions();
    private:
        int dimensions;
};
```

A derived class which is either `OneDimension` or `TwoDimension` class could derive the space class and implement its own content of function `fieldOfCoverage()`. However, this base class requires you to write an implementation of functions even without an existing dimension yet. You may opt to write blank content of the specified function(s) but still an object is still created from this class. The necessary matter is to include it on base class definition but prevent the user from creating object out of it. Since we want to do this on `fieldOfCoverage()`, it must be converted to pure virtual functions as shown below.

Example

```
virtual int fieldOfCoverage()=0;
```

Classes with one or more pure virtual functions are called ***abstract classes***. The example below indicates the complete implementation and utilization of an abstract class.

Example

```
//from space.h
#ifndef SPACE_H_INCLUDED
```

```
#define SPACE_H_INCLUDED

class space
{
    public:
        //default constructor for space class
        space(int dims);
        //field of coverage, returns the coverage depending
        //on dimension size
        virtual double fieldOfCoverage()=0;
        //returns the number of dimensions
        int getDimensions();
    private:
        int dimensions;
};
#endif // SPACE_H_INCLUDED

//from spaceImp.cpp - implements the functions on space.h
#include "space.h"
space::space(int dims)
 {
      dimensions=dims;
 }

int space::getDimensions()
{
    return dimensions;
}

//from OneDimension.h
#ifndef ONEDIMENSION_H_INCLUDED
#define ONEDIMENSION_H_INCLUDED
#include "space.h"

class OneDimension:public space
{
    public:
        //default constructor for one dimension
        OneDimension(double segment=0);
        double fieldOfCoverage();

    private:
        double segmentLength;

};
#endif // ONEDIMENSION_H_INCLUDED

//from OneDimsImp.cpp - implements the functions on OneDimension.h

#include "OneDimension.h"
OneDimension::OneDimension(double segment):space(1)
{
    segmentLength = segment;
}

double OneDimension::fieldOfCoverage()
{
    return segmentLength;
}

//from TwoDimension.h
#ifndef TWODIMENSION_H_INCLUDED
#define TWODIMENSION_H_INCLUDED
#include "space.h"

class TwoDimension:public space
{
    public:
        //default constructor for two dimension
        TwoDimension(double x=0, double y=0);
        double fieldOfCoverage();
    private:
        double width;
        double height;
};
#endif // TWODIMENSION_H_INCLUDED

//from TwoDimsImp - implements the functions on TwoDimension.h
#include "TwoDimension.h"
TwoDimension::TwoDimension(double x, double y):space(2)
{
    width = x;
    height = y;
}
double TwoDimension::fieldOfCoverage()
{
    return width*height;
}
//Clients code
#include <iostream>
#include "OneDimension.h"
#include "TwoDimension.h"

using namespace std;
int main()
{
    OneDimension Dim1(5);
    TwoDimension Dim2 (6,7);
    cout<<Dim1.fieldOfCoverage()<<endl;//output: 5
    cout<<Dim2.fieldOfCoverage();//output: 42
}
```

In general the output is self-explanatory, since each derived class has its own implementation of the pure virtual function of the base class.