CS2303 In Class Exercises

February 16, 2020

Abstract

Today's goal is that you can learn some aspects of using C++, namely "this", and template classes.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 What's "this"?

The keyword "this" is a pointer, and it refers to the instance of the class executing.

We can imagine a class "Room.cpp", having attributes of length, width and height, etc. One instance of that class is a 10 foot by 10 foot by 10 foot cube.

We can cause this to take place like this:

```
#ifndef ROOM_H_
#define ROOM_H_

class Room {
public:
Room();
virtual ~Room();

private:
double height;
double length;
double depth;
};

#endif /* ROOM_H_ */
```

There is a pointer, "this". We can use the pointer implicitly, which involves no typing. Implicit use looks like:

```
#ifndef ROOM_H_
#define ROOM_H_

class Room {
public:
Room();
virtual ~Room();
void setHeight(double h);

private:
double height;
double length;
double depth;
};

#endif /* ROOM_H_ */
```

and

```
#include "Room.h"

Room::Room()
:height(0), length(0), depth(0)
// TODO Auto-generated constructor stub
{
}

Room::~Room() {
// TODO Auto-generated destructor stub
}

void Room::setHeight(double h)
{
height=h;
}
```

Note that in the method setHeight, no typing of "this" occurred. This is called implicit use.

By contrast, we can make explicit use of "this", in two ways. The first explicit way makes use of the pointer's field selector.

```
void Room::setHeight(double h)
{
height=h;
}
void Room::setLength(double ln)
```

```
{
this->length=ln;
}
```

The second explicit way dereferences the pointer, then uses the period field selector.

```
void Room::setHeight(double h)
{
height=h;
}
void Room::setLength(double ln)
{
this->length=ln;
}
void Room::setDepth(double d)
{
(*this).depth=d;
}
```

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Create a class's .h file. Provide three (private) attributes.

```
class Sam{
private:
int height;
int weight;
char* favorite_color[8];
}
```

2. Create a class's .cpp file. Provide setter functions, one each for your attributes. Use the implicit method for "this", and use the two explicit method's for "this".

```
void Sam::setHeight(int h){
this->height = h;
}
void Sam::setWeight(int w){
weight = w;
}
void Sam::setFavorite_Color(char* color){
facorite_color = color;
}
```

Note that we can use (preferably private) attributes of the class where in C we used a struct, and passed a pointer to it.

We can still make user-defined datatypes the typedef struct way.

```
#ifndef ROOM_H_
#define ROOM_H_

typedef struct
{
int junk;
double stuff;
float moreStuff;
char forFun;

}SomeDataType;

class Room {
public:
Room();
virtual ~Room();
void setHeight(double h);
void setLength(double ln);
void setDepth(double d);

private:
double height;
double length;
double depth;
SomeDataType x;
};

#endif /* ROOM_H_ */
```

and

```
Room::Room()
:height(0), length(0), depth(0),x({1, 2.1, 3.2,'w'})
// TODO Auto-generated constructor stub
{
}
```

We can set fields within the attribute that is defined by typedef if we wish.

```
void Room::setChar(char c)
{
    x.forFun=c;
}
```

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Is this example above a use of "this"? If so, which one?

implicit

We can test this method, by invoking a companion method.

```
bool Tests::testSetChar()
{
bool ans = true;
cout <<"In testSetChar" << endl;
Room* rP = new Room();
rP->setChar('v');
if(rP->getChar()!='v')
{
ans = false;
cout <<"testSetChar failed" << endl;

}
delete rP;
return ans;
}
```

When the code invokes delete, the destructor method is called. We can write into the destructor.

```
Room::~Room() {
// TODO Auto-generated destructor stub
cout << "The destructor for Room was called." << endl;
}
```

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Why does the example code for testSetChar print this?

   ```
   In testSetChar

   The destructor for Room was called.
   ```

Because the function prints out "In testSetChar" and
then at the end of the function the destructor is called which prints out "the destructor for Room was called."

2. Improve the code for setting the length of the room, so that it only sets positive values.

   ```
   if(ln<0){
   this->height = ln*-1;}
   ```

3. Write a method for getting the length of the room.

```
double Room::getLength(){
return length;
}
```

4. Write a test method for accessing the length of the room. Don't forget to delete any instances of objects that you are finished with after the test.

```
bool Tests::testLength()
{
bool ans = true;
cout <<"In testLength" << endl;
Room* rP = new Room();
rP->setLength(5.5);
if(rP->getLength()!=5.5)
{
ans = false;
cout <<"testLength failed" << endl;
}
delete rP;
return
```

2 Templates

2.1 Template Functions

If user-defined operations are identical for several datatypes, developers can express the operation compactly and conveniently using function templates.

Example:

```
bool Tests::testArrayPrint()
{
bool ans = true;
int theInts[5] = {1,2,3,4,5};
```

```
double theDoubles[6]={1.1,2.2,3.3,4.4,5.5,6.6};
char theChars[6] = {'a','b','c','d','e'};
printArray(theInts, 5);
printArray(theDoubles, 6);
printArray(theChars,5);
return ans;
}
```

The code for printArray uses the notation for being able to accept a datatype as a parameter.

```
template<typename T>
void printArray(T* array, int count)
{
for(int i = 0; i < count; i++)
{
cout << array[i] << " ";
}
cout << endl;
}
```

This template function is used by the compiler when the compiler encounters an invocation, such as printArray(theInts). Because the type used in the invocation is int, the compiler uses the template function to compile a version of the function for the datatype int. What we provide, the template, is not executable code per se, it is material for the compiler. The result is, we provide the template function in a .h file.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Write a template function that adds up an array of whatever type is used when the function is invoked. (Recall that characters can be added, and do not worry about the result.)

```
template<typename T>
        T addUp(T* stuff, int count){
        T sum = stuff[0];
                for(i = 1; i < count; i++){
                        sum+=stuff[i];
                }
        return sum;
}
```

2.2 Template Classes

It is also possible to create classes that use a datatype as a parameter.

Here is an example of a stack. We can express the stack without choosing a particular datatype to be occupying the stack.

Example: Template Class for Stack of Datatype T

```
/*
* Stack.h
*
*  Created on: Feb 16, 2020
*/

#ifndef STACK_H_
#define STACK_H_
#include <stdbool.h>

template<typename T>
class Stack
{
    public:
        Stack( int = 10);
        ~Stack()
       {
         delete [] stackPtr;
       }

       bool push(T&);
       bool pop(T&);
       bool isEmpty();
       bool isFull()
       {
          return top == size-1;
       }
    private:
       int size;
       int top;
       T* stackPtr;

};

template<typename T>
Stack<T>::Stack(int s)
    :size(s>0?s : 10),
     top(-1),
     stackPtr(new T[size])
```

if(s>0){
size = s;
}else{
size =10;}

```
{
//empty body
}

template<typename T>
bool Stack<T>::push(T& pushValue)
{
    bool ans = true;
    if(!isFull())
    {
       stackPtr[ ++top] = pushValue;
    }
    else
    {
        ans = false;
    }
    return ans;
}

template<typename T>
bool Stack<T>::pop(T& popValue)
{
     bool ans = true;
     if (!isEmpty())
     {
        popValue = stackPtr[top--];
     }
     else
     {
         ans = false;
     }
     return ans;
}
#endif /* STACK_H_ */
```

3 Call by Reference

We have already been using call by reference. In function declarations, we have declared parameters to be a pointer to some datatype. We have written into the memory designated by the pointer. So we are familiar with this idea, and it is good to associate this idea with the name "call by reference". When values being passed to functions are of primitive datatypes, the compiler will pass by value. That means, a place on the stack will be filled in with the value that appeared as an argument in the function invocation. A function may write to that location on the stack. The value of the variable in the calling function will

not be changed. Sometimes we want that value to be changeable. Then we use the ampersand (&). In the template function above, the value popped from the stack is provided to the calling environment through that reference. The returned value, by contrast, is boolean.

Exercises: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Write a function definition that has primitive datatypes for parameters. Use the notation that allows the function to write on the variable found in the parameter list, effecting a change in the calling environment.