DHA SUFFA UNIVERSITY
Department of Computer Science
Object Oriented Programming
CS-1002L
Spring 2020

Lab 11

Smart Pointers and casting

Objectives

To learn about smart pointers and casting in C++.

Pointers

A pointer provides a way of accessing a variable without referring to the variable directly. The address of the variable is used.The declaration of the pointer ip,

```
int *ip;
```

means that the expression *ip is an int. This definition set aside two bytes in which to store the address of an integer variable and gives this storage space the name ip

Smart Pointers

Smart pointers is a feature of c++11 they are used to deal with automatic deallocation of memory. It is defined in `std` namespace and `<memory>` header file.

Types of Smart pointers

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

unique_ptr

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer. It allows single owner at a time. It can be moved to another pointer, but it cannot be copied or shared. The object is deleted in following situations.

- When `std::unique_ptr` object is destroyed
- When `std::unique_ptr` object is assigned another pointer via `operator=` or `reset()`

Syntax

```
//Syntax for declaring and assigning values to a unique_ptr

std::unique_ptr<int> p1(new int(5));
```

Example 11.1

```
/*Example of unique_ptr*/
#include <iostream>
#include <memory>
int main()
{
	/*Declaring a unique pointers to a variables of type int */
	std::unique_ptr<int> ptr1(new int(10));
	std::unique_ptr<int> ptr2(new int(20));
	std::unique_ptr<int> ptr3(new int(30));

	/*Printing values contained by pointers using dereferencing operator*/
	std::cout<<"====Result ===="<<std::endl;
	std::cout<<"Value of ptr 1="<<*ptr1<<std::endl;
	std::cout<<"Value of ptr 2="<<*ptr2<<std::endl;
	std::cout<<"Value of ptr 3="<<*ptr3<<std::endl;
	/*ptr1=ptr3;
	Error:unable to assign one pointer to another because its a unique_ptr
	ptr3=std::move(ptr2);//Moving ownership of ptr2 to ptr3
	std::cout<<"====Result ===="<<std::endl;
	std::cout<<"Value of ptr 1="<<*ptr1<<std::endl;
	std::cout<<"Value of ptr 3="<<*ptr3<<std::endl;
	std::cout<<"Value of ptr 2="<<*ptr2<<std::endl;*/
	return 0;
}
```

Output

```
 ====Result ====
Value of ptr 1=10
Value of ptr 2=20
Value of ptr 3=30
```

shared_ptr

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. It is used when you want to assign one raw pointer to multiple owners. To delete shared pointer you need to first delete all the `shared_ptr` owners.

Syntax

```
//Syntax for declaring and assigning values to a shared_ptr

std::shared_ptr<int> p0(new int(5));
```

Example 11.2

```
/*Example of shared_ptr*/
#include <iostream>
#include <memory>
int main()
{
      /*Declaring a shared pointers to a variables of type int */
       std::shared_ptr<int> ptr1(new int(10));
       std::shared_ptr<int> ptr2(new int(20));
       std::shared_ptr<int> ptr3(new int(30));
      /*Printing values contained by pointers using dereferencing operator*/
      std::cout<<"====Result ===="<<std::endl;
      std::cout<<"Value of ptr 1="<<*ptr1<<std::endl;
      std::cout<<"Value of ptr 2="<<*ptr2<<std::endl;
     std::cout<<"Value of ptr 3="<<*ptr3<<std::endl;
     ptr1=ptr3;/*It will not produce error as it was producing in unique
     _ptr because here in shared_ptr you can share ownership*/
     std::cout<<"====Result ===="<<std::endl;
     std::cout<<"Value of ptr 1="<<*ptr1<<std::endl;
     std::cout<<"Value of ptr 2="<<*ptr2<<std::endl;
     std::cout<<"Value of ptr 3="<<*ptr3<<std::endl;
     ptr1.reset();
     ptr2.reset();
     ptr3.reset();
     return 0;
}
```

Output

```
====Result ====
Value of ptr 1=10
Value of ptr 2=20
Value of ptr 3=30
====Result ====
Value of ptr 1=30
Value of ptr 2=20
Value of ptr 3=30
```

weak_ptr

`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`

Syntax

```
//Syntax for declaring weak_ptr
std::weak_ptr<int> ptr;
```

Casting

Casting means to convert one data type of one operand to the date type of the other operand.There are two types of casting

- Implicit casting
- Explicit casting

Implicit casting

Casting which is done automatically by compiler is known as Implicit cast.

Explicit casting

Casting in which one data type is converted into other forcefully is known as Explicit cast.Types of casting in c++ are as follows:

1. `static_cast`
2. `const_cast`
3. `reinterpret_cast`

4. dynamic_cast

static_cast

Static casting is casting of operands during compile time while in dynamic casting casting is done when the programs starts to execute.The outcome of the cast is to convert the value to the type that you specify between the angle brackets

Example 11.3

```
//Example of static_cast
//fileName:temp.cpp
#include <iostream>
#include <iomanip>

int main()
{
    //Declaring and initializing variable in c++11 format
     int tf{0};

     //Asking user to input temperature
     std::cout<<"Enter Temperature in Fahrenheit"<<std::endl;
     std::cin>>tf;

     //Using static_cast operator to convert fahrenheit into celsius
     double tc=(static_cast<double>(tf)-32)*5/9;
     std::cout<<std::setprecision(3)<<tc<<" Celsius";
     return 0;
}
```

Output

```
Enter Temperature in Fahrenheit
56
13.3 Celsuis
```

const_cast

const_cast used to remove constant-ness from pointer or references that are declared const.

Syntax

```
//Syntax of casting
const_cast<type to convert to>(expression)
```

Example 11.4

```
//Example of const_cast
#include <iostream>

int main()
{
      //Declaring and initializing variable in c++11
      int x{0};

      //Declaring and initializing a pointer
      const int *p=&x;
      std::cout<<"Enter Number"<<std::endl;
      std::cin>>x;
      std::cout<<"Value of x = "<<*p<<std::endl;

      //*p=7; //Error: assignment of read-only location
      *const_cast<int*>(p)=7; //Solution of above Error
      std::cout<<"Value of x after modification  = "<<*p;
      return 0;
}
```

Output

```
Enter Number
20
Value of x = 20
Value of x after modification  = 7
```

reinterpret_cast

reinterpret_cast is used to convert one pointer to any other type of pointer.It allows you to cast any built in type to any derived or user-defined type.They are used in following scenarios

- When you want to convert integer to pointer
- when you want to convert pointer to pointer
- when you want to convert function pointers to function pointer

Syntax

```
reinterpret_cast<data type>(pointer name);
```

Example 11.5

```
//FileName:Oreo.h

#ifndef OREO_H
#define OREO_H
#include <iostream>
class Oreo
{
      //member variables
      protected:
                  std::string name;
      //member functions
      public:
      Oreo();
            ~Oreo();
                  void unlock();
                  void AppStore();

};
#endif
//FileName:Oreo.cpp

#include "Oreo.h"
```

```
//defining member functions
Oreo::Oreo():name(""){}

Oreo::~Oreo(){}

void Oreo :: AppStore()
{
      std::cout << "Accessing App Store from Android 8.0 Oreo" <<
std::endl;
}
void Oreo :: unlock()
{
      std::cout << "Unlocking Android 8.0 (Oreo)" << std::endl;
}


//FileName:Pie.h

#ifndef PIE_H
#define PIE_H
#include <iostream>

class Pie
{
      //member variables
      protected:
                  std::string name;
      //member functions
      public:
            Pie();
            ~Pie();
                  void unlock();
                  void AppStore();

};
#endif

//FileName:Pie.cpp
```

```
#include "Pie.h"
//defining member functions
Pie::Pie():name(""){}

Pie::~Pie(){}

void Pie :: AppStore()
{
      std::cout << "Accessing App Store from Android 9.0 (Pie)" <<
std::endl;
}
void Pie :: unlock()
{
      std::cout << "Unlocking Android 9.0 (Pie)" << std::endl;
}

//FileName:main.cpp
#include "Oreo.h"
#include "Pie.h"
int main()
{
       Oreo* o1 = new Oreo();
    // converting the pointer to object
    // referenced of class Oreo to class Pie
    Pie* p1 = reinterpret_cast<Pie*>(o1);
      //function calls
    // Accessing the function of class Pie
      p1->AppStore();
    p1->unlock();
    return 0;
}
```

Output

```
Accessing App Store from Android 9.0 (Pie)
Unlocking Android 9.0 (Pie)
```

dynamic_cast

dynamic_cast is used when you want to convert a base class pointer into a derived class pointer.This operator answers the question of whether we can safely assign the address of an object to a particular data type.It involves a run-time check If the object bound to the pointer is not an object of target type,it fails and object value is 0

Syntax

```
dynamic_cast<data type>(pointer name);
```

Example 11.6

```
//FileName:Android.h

#ifndef ANDROID_H
#define ANDROID_H
#include<iostream>
//base class of mobile
class Android
{
      //data members and functions
      protected:
            std::string type;
      public:
            Android();
            ~Android();
            virtual void details();
            void call();
};
#endif


//FileName:Android.cpp

#include"Android.h"
//function definition
Android::Android():type(""){}
```

```
Android::~Android(){}
void Android::call()
{
	std::cout << "Make Call" << std::endl;
}
void Android::details()
{
	std::cout << " Type :" <<type<<std::endl;

}

//FileName:Samsung.h

#ifndef SAMSUNG_H
#define SAMSUNG_H
#include "Android.h"

//inherited from Mobile class

class Samsung : public Android
{
	public:
		Samsung();
		Samsung(std::string ,float);
		~Samsung();
		void openAppStore();
		void details();
	//data member and member functions
	protected:
		std::string os;
		float megapixelCamera;
};
#endif


//FileName:Samsung.cpp

#include"Samsung.h"
```

```
//function definition
Samsung::Samsung():os(""), megapixelCamera(0.0){}
Samsung::Samsung(std::string n,float val):os(n),
megapixelCamera(val){}
Samsung::~Samsung(){}
void Samsung::openAppStore()
{
      std::cout << "Open App Store" <<std::endl;
}
void Samsung::details()
{
      std::cout << " Os :" <<os<<std::endl;
      std::cout << " Mega Pixel :" <<megapixelCamera<<std::endl;
}

//FileName:main.cpp

#include"Samsung.h"
int main()
{
      //object creation
      Android *m1 = new Samsung("Marshmellow",14.00f);

      Samsung *s1 = dynamic_cast<Samsung*>(m1);
      if(s1!=0)
      {
            s1->details();
      }
      else
      {
            std::cout<<"Invalid casting."<<std::endl;
      }

      return 0;
}
```

Output

```
Os :Marshmellow
Mega Pixel :14
```

Task (Home Assignment)

Write a program in C++ to

1. Create a unique pointer named “uni_ptr1” with an integer value “1”.
2. Create another unique pointer named “uni_ptr2” with integer value “2”.
3. Print value contained by uni_ptr2.
4. Transfer the ownership of uni_ptr1 to uni_ptr2 setting uni_ptr1 to nullptr.
5. Similarly create two different shared pointers sh_ptr1 and sh_ptr2, pointing to same location in memory.
6. Create another pointer sh_ptr3 to point to same location as created in step 5.
7. Print the value contained by above three pointers on console.
8. Count the member objects sharing same location as sh_ptr1.
9. Reset the sh_ptr1 to nullptr.
10. Count and display member objects sharing same location as sh_ptr1.
11. Count and display member objects sharing same location as sh_ptr2.