Fundamentals of Computer Programming

Chapter 5
Pointers (Part II)

Chere L. (M.Tech)
Lecturer, SWEG, AASTU

Outline

- Variables in a Memory
- Basics of Pointers
  - *What is pointer?*
  - *Why pointers*
  - *Pointer Declaration*
  - *Pointers Initialization*
- Pointer Operators (& and *)
- Types of Pointers
  - ➢ *NULL Pointer*
  - ➢ *Void pointers*
  - ➢ *Pointers of Pointer*
  - ➢ *Dangling Pointers*
  - ➢ *Wild Pointers*
- Pointers Expression
  - ➢ *Pointers Arithmetic*
  - ➢ *Pointers Comparison*
- Pointers and Constants
- Pointers and Arrays/Strings
- Pointers with Function
  - ➢ *Parameter pass-by-address*
  - ➢ *Pointer as return type/value*
- Dynamic Memory Management
  - ➢ *Memory Allocation (new)*
  - ➢ *Memory Allocation (delete)*
- Smart Pointers

Dynamic Memory Allocation and De -allocation

10.1 Memory Map – Memory is divided into four parts which are listed as follows

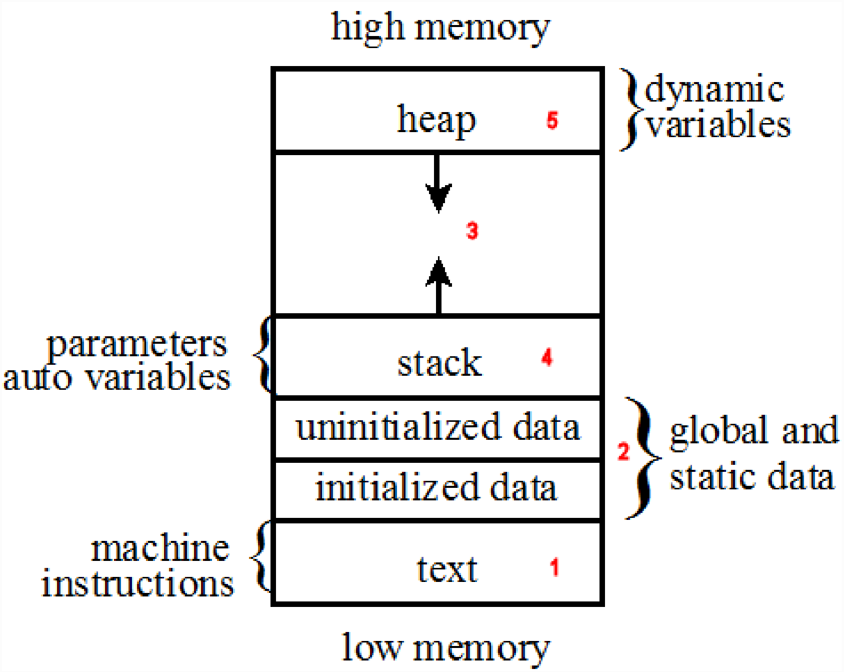
(1) Program Code: It holds the compiled code of the program (machine instructions).

(2) Global Variables: They remain in the as long as program continues.

(3) Free space: The space illustrated here was allocated but unused on a segmented-memory system and provided space where the stack and heap could grow.

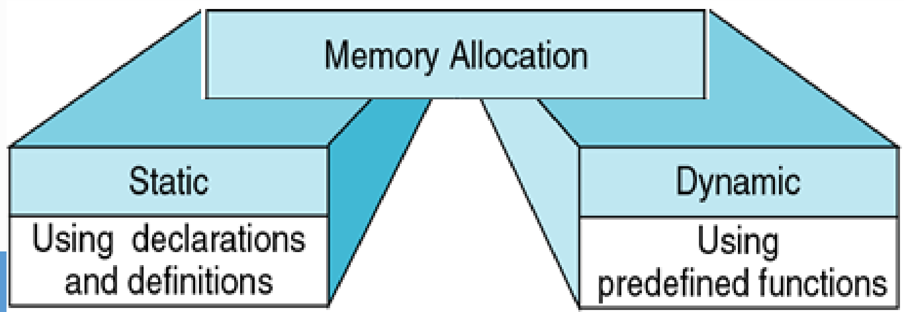
(4) Stack: It is used for holding return addresses at function calls, arguments passed to the functions, local variables for functions. It also stores the current state of the CPU.

(5) Heap: It is a region of free memory from which chunks of memory are allocated via dynamic memory allocation functions.



Dynamic Memory Management (2/12)

10.2 Memory Management




```
{
  int a[200];
  …
}
```

STATIC ALLOCATION

- The amount of memory to be allocated is known before hand.
- Memory is acquired automatically

Memory allocation is done during compilation.

For eg. int num;
This command will allocate two bytes of memory and name it 'num'.

The memory is de-allocated (returned) automatically as soon as the variable (object goes out of scope.

```
int* ptr;
ptr = new int[20];
…
delete [] ptr;
```

Dynamic ALLOCATION

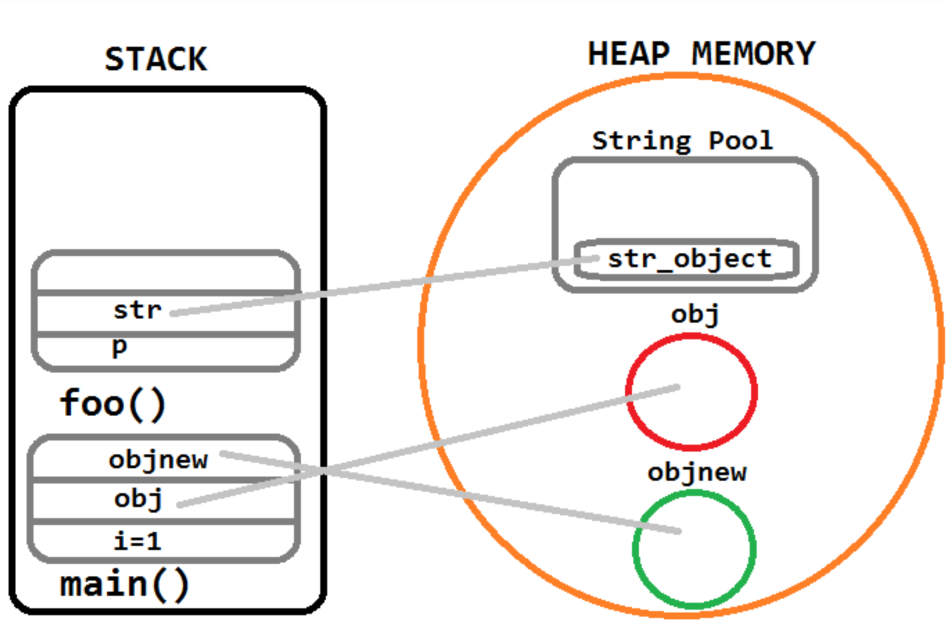
- The amount of memory to be allocated is not known before hand.
- It is allocated depending upon the requirements (acquired by program)

Memory allocation is done during run time. and pointers are crucial

Dynamic memory is allocated using the **new operator.** e.g. int*k=new int;

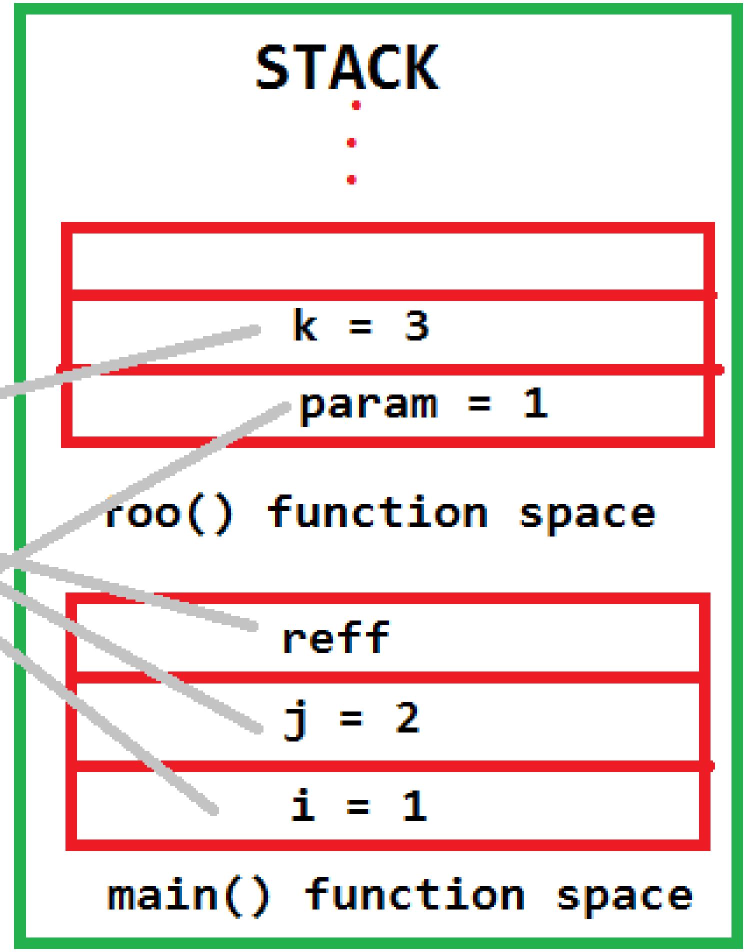
- Dynamic objects can exist beyond the function in which they were allocated
- To deallocate this type of memory **delete operator** is used. For eg. delete k;

Stack Vs. Heap



Stack based program execution

```
public class Stack_Test {

 public static void main(String[] args) {

        int i=1;
        int j=2;

        Stack_Test reff = new Stack_Test();
        reff.foo(i);
    }

    void foo(int param) {
        int k = 3;
        System.out.println(param);
    }

}
```



Dynamic Memory Management (5/12)

10.3 Dynamic Object Creation

- To allocate memory dynamically, use the unary operator **new**, followed by the **data type** being allocated.

```
new int;        // dynamically allocates an int
new double;     // dynamically allocates a double
```

- To create an **array/string dynamically**, use the same form, but put brackets with a size after the data type:

```
new int[40];          //dynamically allocates an array of 40 int
new double[SIZE];     //dynamically allocates an array of size doubles
                      //  note that the SIZE can be a variable
```

- As **objects** (struct or class) are no different from simple data type, memory will allocated dynamically in the same way.

  e.g. struct Date { int idd, mm, yy}; new Date; new Date[5];

Dynamic Memory Management (6/12)

Note:

- The statements above (previous slide) are ***not very useful by themselves***, because the allocated spaces ***have no names*.**
- The ***new operator*** returns the ***starting address*** of the allocated space, and this address need to be stored in a pointer.
- Otherwise, the allocated memory will not be accessed.
- As a result, the syntax for dynamic memory creation updated as follow;

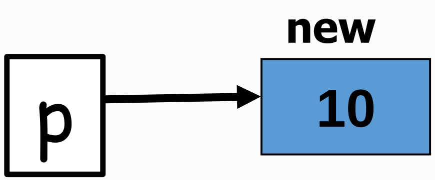
Dynamic Object Creation Syntax

- **DataType *ptr_Name = new DataType;** or
- **DataType *ptr_Name = new DataType[SIZE]**

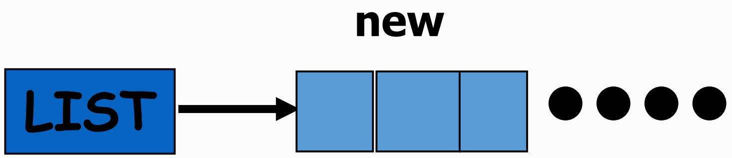
Where the **DataType** is either primitive or user defined.

Example 1: dynamic object creation

```
int * p;        // declare a pointer p
p = new int;    // dynamically allocate an int
                //and load address into p
*p = 10;        //store a value 10 on the allocated space
```




```
// we can also do these in single line statements
double * d = new double;   // declare a pointer d, dynamically
                           //allocate a double and load the address
int * list = new int[10];  //1D array dynamic memory allocation

int x = 40;
float * numbers = new float[x];
```



Dynamic Memory Management (8/12)

Example 2: dynamic string object creation

```
char *name ;                // declare a pointer name
name = new char[20];

 // we can also do these in with initialization
    char *city= "Addis Ababa";

char *name [] = {"Alemu", "Getachew", "Hailu", "Tena"};
```

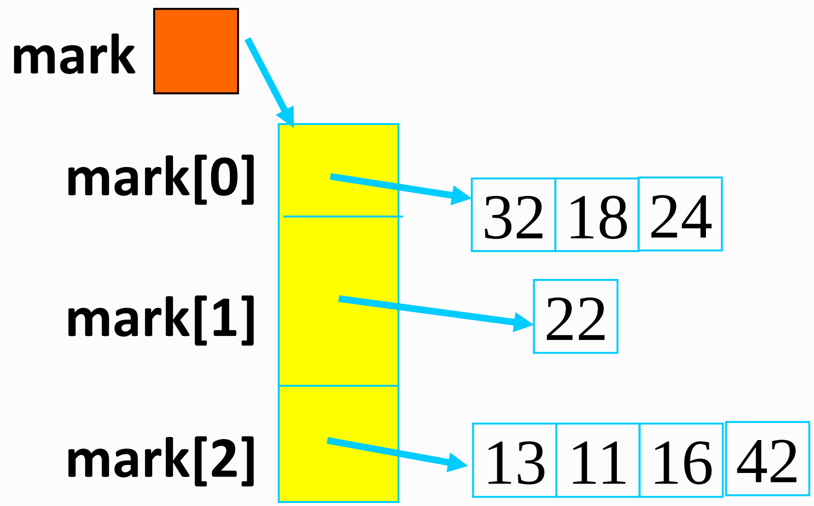
Example 3: dynamic 2D array creation

```
float *mark;
new float[3*5];
```

Dynamic Memory Management (9/11)

An other way of dynamic 2D array creation using array of pointer

```
float **mark = new float*[3];
for (int i=0; i<3; i++)
   mark[i] = new float [5];
```



Or

```
mark = new int*[5];
mark[0] = new int[4];
mark[1] = new int[7];
mark [2] = new int[2];
mark[3] = new int[3];
mark[5] = NULL;
```

Note:

- This kind of dynamic array creation is very useful to save space when all rows of the array are not full.

Dynamic Memory Management (10/12)

Example 4: dynamic struct object creation

```
struct Person{
        char fullName[20], gender;
        int age,
        float salary;
        };

Person *p = new Person;          //creation of dynamic space
Person *stud = new Person[20];   //creation of dynamic array of struct.
```

Note:

- Dynamic object creation request for **“unnamed”** memory from the Operating System.
- Assigning the address of dynamic memory space is an other way of initializing a pointer to a valid target (and the most important one).

Dynamic Memory Management (11/11)

10.4 Dynamic Object Destruction (deallocation)

- To **deallocate memory** that was created with **new operator**, we use the **unary operator delete**.
- The one operand should be a pointer that stores the address of the space to be deallocated.
- **Syntax:** **delete pointer_name;**

  or delete [] name_of_pointer; //deallacte dynamic array
- **Example**

  delete ptr; // deletes the space that **ptr** points to

  delete []arrPtr; //deallocate dynamic array **arrPtr**

Note:

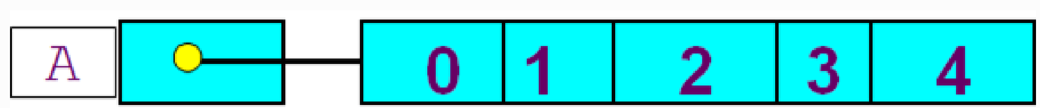
➢ *The pointer **ptr** and **arrPtr** still exists. That's a named variable subject to scope and extent determined at compile time. It can be reused:*

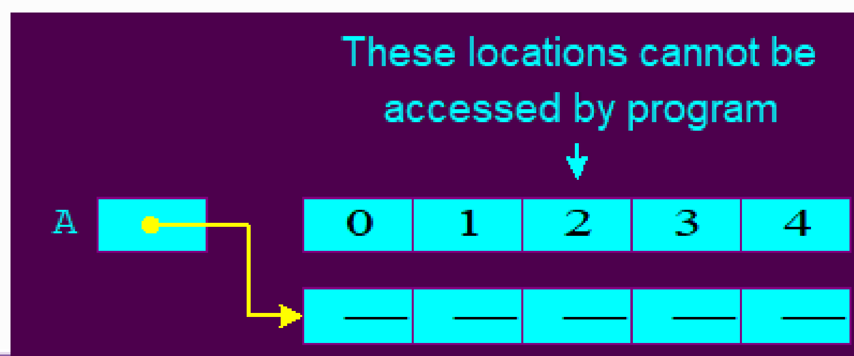
Dynamic Memory Management (12/12)

Why memory deallocation?

- To avoid **memory leak**
- A memory leak is serious bug that occurs when a piece (or pieces) of memory that was previously allocated by a programmer is *not properly deallocated* by the programmer.
- Even though that memory is no longer in use by the program, it is still “reserved”, and that piece of memory cannot be used by the program until it is properly deallocated by the programmer.
- **Example:**

```
int *A = new int [5];
for(int i=0; i<5; i++)
        A[i] = i;
A = new int [9];
```





Examples of Dynamic Memory Management

Example 1: Allocate and deallocate memory for scalar variable

```
#include <iostream>
using namespace std;
int main(){
    float *ptr = new float;

    cout <<"Please enter salary ";     cin>> *ptr;
    cout << "Your salary is: "<<*ptr<<endl;
    delete ptr;
    cout<<"\nValue:  "<<*ptr<<endl;            //print garbage data

    ptr = new float;
    cout<<"Enter mark: ";     cin>>*ptr;
cout << "Your mark is: "<<*ptr<<endl;
}
```

Examples of Dynamic Memory Management

Example 2: Work with an array of unknown size

```
#include <iostream>
using namespace std;
int main(){
    int n;
    cout <<  “How many students? “;     cin>> n;
    int *marks = new int[n];
    cout << “Input Grade for Student\n”;
    for(int i=0; i < n; i++){
            cout << (i+1)  << “ : ”;        cin >> marks[i];
    }
delete [n]marks;     //deallocation of memory pointed by marks
cout<<“1st Mark is: “<<*mark<<endl;          //print garbage data
Marks = 0;           //reset the point to NULL
}
```

Examples of Dynamic Memory Management

Example 3: Work with 2D array and strings

```
#include <iostream>
using namespace std;
int main(){
     int studNum, courseNum;
     cout<<"How many students? ";
   cin>>studNum;
   cout<<"How many Courses? ";
   cin>>courseNum;

     float *studMark;
   studMark = new float[studNum * courseNum];     //Dynamic 2D array creation

char **studName = new char*[studNum];
for(int i=0; i < studNum; i++)
          studName[i]= new char [30];
```

//Dynamic 2D string creation
//Also created using string class:
string studName[studNum];
//And input string as follow
getline(cin, studName[i]);

Examples of Dynamic Memory Management

```
cout <<"Input Name and marks for Students\n";
for(int i=0; i < studNum; i++){
 cout<<"Enter the "<<i+1<<" Student \n";
        cout<<"Name: ";          cin.ignore();
        gets(studName[i]);

 for(int j=0; j < courseNum; j++ ){
        cout<<"Mark "<<j+1<<": ";
        cin>>studMark[i+1*j];
} }
for(int i=0; i < studNum; i++){
        cout<<"\nName: "<<studName[i]<<endl;
        cout<<"Marks: ";
 for(int j=0; j < courseNum; j++ )
        cout<<studMark[i+1*j]<<", ";
 cout<<endl;
}
delete []studMark;   }
```

Examples of Dynamic Memory Management

Example 4: Work with 2D array and strings

```
#include <iostream>
using namespace std;
struct Employee{   string Name;    int Age;
};
int main(){
   Employee* DynArray;
   DynArray = new Employee[3];

   for (int i=0; i<3; i++){
        cout<<"\nEnter info "<<i+1<<" person\n";
        cout<<"Name: ";
        getline(cin, DynArray[i].Name);
        cout<<"Age: ";
        cin>>DynArray[i].Age;
        cin.ignore();
}
```

```
cout<<endl;
cout<<"\n\nDisplaying the Content;
cout"<<endl;
cout<<"Name: \t\tAge: "<<endl;
   for (int i = 0; i < 3; i++)
   {
      cout<<DynArray[i].Name<<"\t";
      cout<< DynArray[i].Age << endl;
   }
   delete[] DynArray;
   return 0;
}
```

Smart Pointers

Smart Pointers (1/17)

- A smart pointers are an **abstract interface** to actual pointers (also called raw pointers), but with the additional features.
- Smart pointers are designed for *automatic resource management and freeing a memory.*
- Doesn't need to be *freed explicitly*, and hence they are **"smart"** pointers and it know when they need to *free up the memory*.
- Primarily ***smart pointers*** are an object that stores a pointer (memory address) to a heap-allocated object.
- The design logic for a smart pointer is to implement it as **a class**
  - Class has a **destructor**, which will trigger automatically when an object is at the end of its scope.

Smart Pointers (2/17)

Why a smart pointers?

- Raw pointers are dangerous!
  - *Missing to free up memory (call delete) causes* ***memory leaks***
  - *After delete a pointer you may get a* ***dangling pointer***

```
 int * ptr = new int (25);
cout<<“Value: “<<*ptr<<endl;
delete p;
*p +=25;   //trying to access delete memory location
```

  - *Calling delete more than once yields* ***undefined behavior***

```
int * ptr1 = new int (25);
int * ptr2 = ptr1;
delete ptr1;
delete ptr2;      //undefined behavior: object already deleted!
```

Smart Pointers (3/17)

Smart pointers

- Overcome the drawback of raw pointers
- A wrapper of a raw pointers providing **same functionalities** with **more safety**
  - The **same syntax** as raw pointers for dereferencing:

    *p, p.val, p->val, p[idx], *(p+idx)

  - Auto **management** of dynamically allocated memory **lifetime**
    - ✓ Delete the pointed-to object *at the right time*
    - ✓ No longer needed to remember when to delete new'd memory!
  - Auto **set freed pointer to NULL**, avoiding dangling pointers
  - It defined in the standard C++ library, in the **std namespace** within the **<memory>** header file.

Smart Pointers (4/17)

Types of Smart pointers

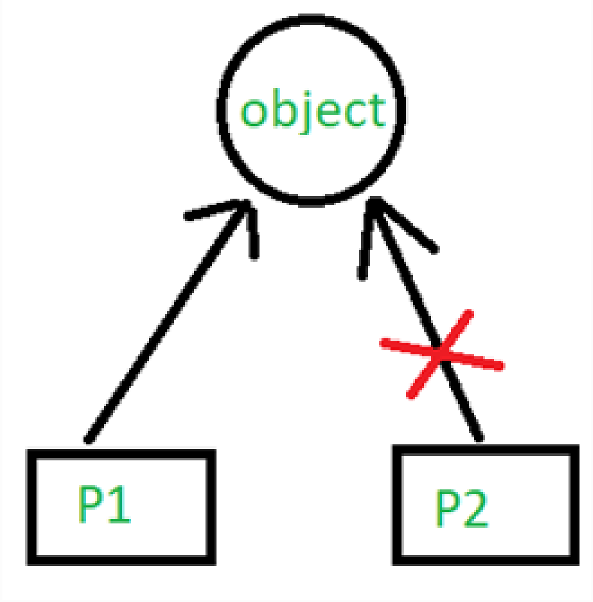
- **FOUR** kinds of smart pointers which all defined in the header **<memory>**

No	Smart Pointers	C++ Version	Description
1	auto_ptr	C++98	✓ A first naive attempt to implement a smart pointer with **exclusive-ownership.** ✓ **Deprecated** from C++11, and removed from the STL from C++14
2	unique_ptr	C++11	✓ Used for **exclusive-ownership** that can be copied only with move semantics
3	shared_ptr		✓ Used for **shared-ownership** with automatic garbage collection based on a reference count
4	weak_ptr		✓ Used for **observing** without **owning**. ✓ It is similar to ***shared_ptr***, but it does not contribute to the reference count

(a) Unique_ptr *(exclusive ownership)*

- Only permit one owner of the pointer
- unique_ptr can contain at maximum, only a **single raw pointer** ("unique" pointer) that points to a single memory location.

- *When an object is created (allocated memory) and pointer P1 is pointing to it, then only one pointer (P1) can point this one at one time.*



- *So it can't shared with another pointer,*
- *However, the control can be transferred to P2 by removing P1.*

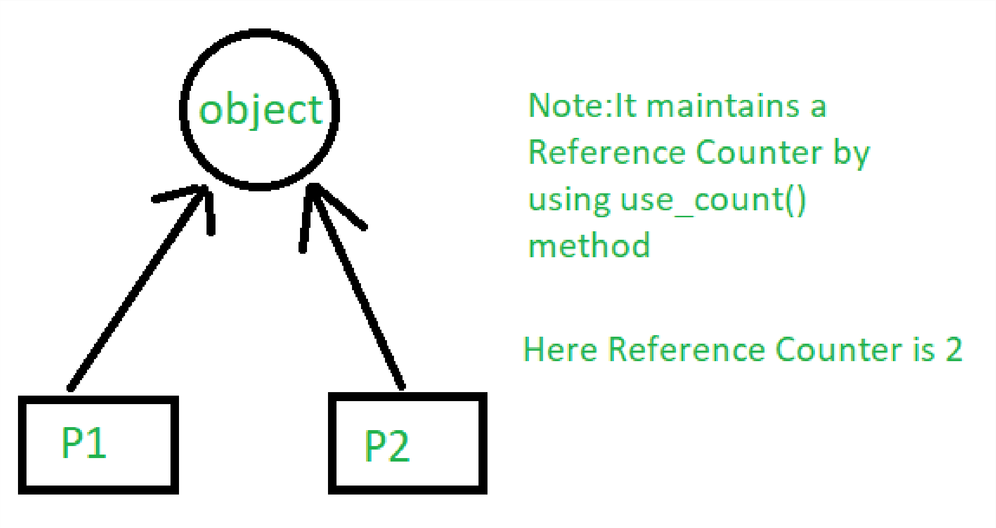
Unique_ptr features

- Provides **exclusive ownership** for the pointer
- Unlike **auto_ptr** *copying* and *assignment* from lvalue is not allowed.
- However, it provide *moving* and *assignment* from rvalue which is implemented using move semantics
- Useful in template specialization to handle **dynamic arrays**.
- No *overhead* in memory
- *Custom deleter* can be provided if needed
- *Easy conversion* to shared_ptr

Smart Pointers (7/17)

(b) shared_ptr *(shared ownership)*

- Provides ***shared ownership*** - many pointers may own the same object
- Also provide ***garbage collection*** mechanism that based on ***a reference counter*** contained in a control block.
- When multiple object share the same resource (memory location), each ***new shared owner copies the pointer*** to the control block and increases the count by 1.



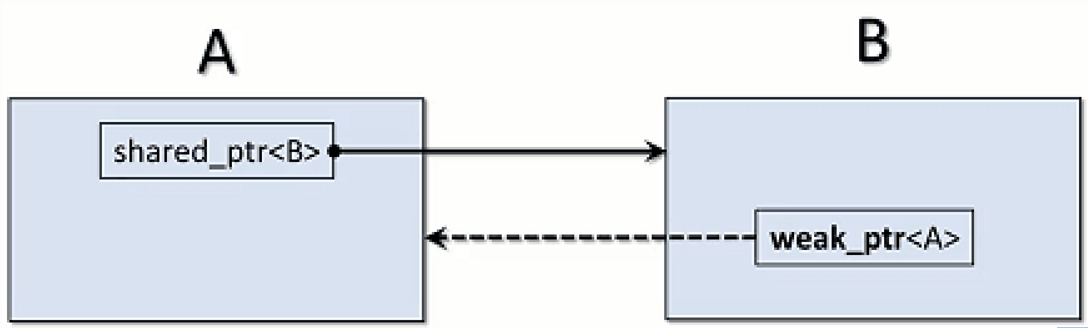
Smart Pointers (8/17)

shared_ptr features

- Provides **shared ownership** for the pointer
- Provide custom **deleter** and custom allocator, if needed.
- **Memory overhead** (typically 2 pointers: for the object itself + control block)
- The **copy/assign** operators are not disabled
- Require *to increment* or *decrement* reference counts as needed.
- Many operations require **atomic update of ref_counter**, which may be result in **slow.**
- Until C++17 no **template specialization shared_prt<T[]>** to handle dynamic arrays.
- An object referenced by the contained raw pointer will not be destroyed until reference count is greater than zero.

(c) weak_ptr

- It is similar to a shared_ptr but doesn't affect the reference count.
- It implements **weak ownership**: the object needs to be accessed only if still exists, and it may be **deleted by others.**
- Not *really* a pointer – can't actually dereference unless you "get" its associated shared_ptr.
- Used as **observer** to determine if the pointer is **dangling** before converting to shared_ptr and using it.
- Typical use cases are **implementing a cache** and **breaking cycles**



Smart Pointers (10/17)

Some useful methods of smart pointers

- **get() –** returns a pointer (memory address) to the managed object
- **move() –** ***transfer ownership*** between *unique_ptr*
- **release()**
  - ✓ returns a reference to the ***deleter object*** used for the disposal of the managed object
- **reset()**
  - ✓ ***releases ownership*** by returning the managed object
  - ✓ Also set the internal pointer to **NULL**
- **use_count() –** returns the ***#of shared pointers*** managing current object
- **lock() –** Creates a new shared_ptr owning the managed object
- **unique() –** returns whether ***use_count()==1***
- **expired() –** check if ***use_count()==0***

Smart Pointers (11/17)

Example 1: unique_ptr

```
#include<iostream>
#include<memory>
using namespace std;
int main() {
            unique_ptr<int> ptr1(new int);
            *ptr1 = 100;
            //cout<<ptr1<<endl;
            cout<<"Address 1: "<<ptr1.get()<< endl;
            cout<<"Value : "<< *ptr1<<endl;

            //int *ptr2 = ptr1;           //cannot copy
            unique_ptr<int> ptr2;
             ptr2 = move(ptr1);   //transfer ownership
            ptr1.reset();                 //reset pointer to null
            cout<<"\nAddress 2: "<<ptr2.get()<< endl;
            cout<<"Value : "<< *ptr2<<endl;
```

Smart Pointers (12/17)

Example 1: unique_ptr (cont'd)

```
unique_ptr<int> ptr3(ptr2.release());        //transfer ownership
cout<<"\nAddress 3: "<<ptr3.get()<< endl;
cout<<"Value: "<< *ptr3<<endl;

ptr2.reset(ptr3.release());             //transfer ownership and destroy pointer

unique_ptr<int[]> ptr4(new int[5]);          //unique_ptr to array object
cout<<endl;
for (int i =0; i<5; i++){
        cout<<"Enter "<<i+1<<" array element: ";    cin>>ptr4[i];
        }

 cout<<"\n\nArray Elements are: ";
 for (int i =0; i<5; i++){    cout<<ptr4[i]<<", ";  }
 cout<<endl;
}
```

Smart Pointers (13/17)

Example 2: shared_ptr

```
#include<iostream>
#include<memory>
using namespace std;
int main() {
    shared_ptr<int> ptr1(new int(7));
   shared_ptr<int> ptr2(new int(6));
   cout<<"Address 1: "<<ptr1<<"\nAddress 2: "<<ptr2<< endl;

   // Returns the number of shared_ptr objects referring to the same managed object.
   cout<<"Shared Objects(ptr1): " << ptr1.use_count() << endl;
   cout<<"Shared Objects(ptr2): "<< ptr2.use_count() << endl;

   // Relinquishes ownership of ptr1 on the object and pointer becomes NULL
   ptr1.reset();
   cout<<"\nAddress 3: "<< ptr1.get() << endl;
   cout<<"Shared Objects: "<< ptr1.use_count() << endl;
   cout<<"Address 4: "<< ptr2.get() << endl;
   cout<<"Shared Objects: "<<ptr2.use_count()<<endl;
```

Smart Pointers (14/17)

Example 2: shared_ptr (cont'd)

```
shared_ptr<int> ptr3 (ptr2);            //initialize ptr3 with the shared object ptr2
  cout<<"\nAddress 5: "<< ptr2.get() << endl;
  cout<<"Address 6: "<< ptr3.get() << endl;
  cout<<"Shared Objects(ptr3): "<< ptr3.use_count() << endl;
  *ptr3 = 110;
   cout<<"Value: "<< *ptr2<< endl;

    ptr2.reset();
   cout<<"\nAddress 7: " << ptr2.get() <<"\nAddress 8: " << ptr3.get() << endl;
   cout<<"Shared Objects: "<< ptr3.use_count() << endl;

   //invalid: accessing a pointer that release an object
   //*ptr2 = 10;     cout << *ptr2<< endl;

     ptr3.reset();
     cout<<"\nShared Objects: "<< ptr3.use_count() << endl;
}
```

Smart Pointers (15/17)

Example 3: weak_ptr

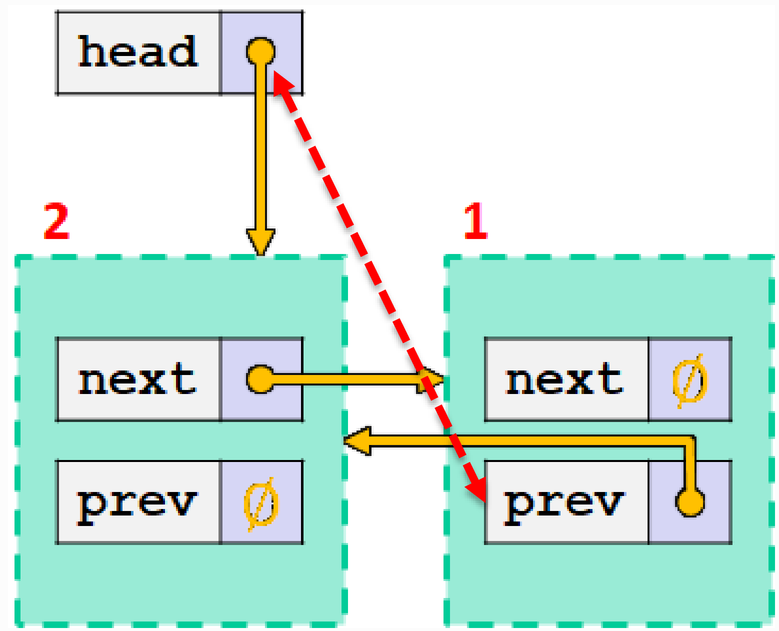
```
#include<iostream>
#include<memory>
using namespace std;
int main() {
   weak_ptr<int> wp;
    if(1) {
        shared_ptr<int> sp = make_shared<int>(53);
        wp = sp;

        auto p = wp.lock();
        if (p) {          cout << "Pointer is alive" << endl;       }
        else {            cout << "Pointing to Null" << endl;       }
    }

     auto p = wp.lock();
    if (p) {       cout << "Pointer is alive" << endl;    }
    else {       cout << "Pointing to Null" << endl;   }
}
```

Smart Pointers (16/17)

Example 4: weak_ptr …..

```
#include<iostream>
#include<memory>od
using namespace std;

struct student;
using shPtr = shared_ptr<student>;

struct student {
            shPtr next;
            shPtr prev;     };

int main(){
  shPtr head(new student());
  cout<<"Header: "<<head.get()<<endl;
  head->next = shPtr(new student());
  cout<<"Next: "<<head->next.get()<<endl;
  head->next->prev = head;
  cout<<"Previous: "<<head->next->prev.get()<<endl;
}
```

Cycle of shared_ptr



Changing to a weak_ptr

Smart Pointers (17/17)

How to convert weak_ptr to shared_ptr?

```
#include<iostream>
#include<memory>od
using namespace std;

int main()
{
      weak_ptr<int> ptr1;
     {
        shared_ptr<int> ptr2  = make_shared<int>(53);
         ptr1 = ptr2;
      }
       shared_ptr<int> ptr3 = ptr1.lock();
      cout<<"Address: "<<ptr3.get()<<endl;
      cout<<"Value: "<<*ptr3<<endl;

}
```

Note:

- ***make_shared* – *unlike* **shared_ptr** *initialization it avoid double memory allocation, one for the object itself, one for the control block (reference count).***
- *In general it is a good idea to use **make_shared and make_unique** whenever possible in order to avoid memory leaks and avoid double memory allocation*

Summary (1/2)

- Memory allocation
  - ***Static allocation –*** compilation time where amount of memory to be allocated is known before hand
  - ***Dynamic allocation –*** runtime time where amount of memory to be allocated is not known before hand
- new operator
  - used allocate memory dynamically for objects
  - *Syntax: dataType <object> = **new** datatype;*

    *dataType <object> = **new** datatype[SIZE]; //dynamic array*

    *dataType <object> = **new** datatype(value); //initialization*
- delete operator
  - *Used to deallocated a memory allocated by new operator*
  - *Syntax: delete <object>;*

    delete []<object>;

Summary (2/2)

- unique_ptr
  - ***takes ownership*** of a pointer
  - *Cannot be copied, but can be moved*
  - ***get***(*) returns a copy of the pointer, but is dangerous to use; better to use* ***release***(*) instead*
  - ***reset***(*) - deletes old pointer value and stores a new one (NULL)*
  - ***move()*** *– transfer unique_ptr ownership*
- shared_ptr
  - allows shared objects to have multiple owners by doing *reference counting*
  - *deletes an object once its reference count reaches zero*
- weak_ptr
  - *works with a shared object but doesn't affect the reference count*
  - *Can't actually be dereferenced, but can check if the object still exists and can get a shared_ptr from the weak_ptr if it does*

FAQ

After de-allocation of a heap allocated memory, why still the data is get accessed?

- In some cases, yes.
- Calling **delete** operator may or may not zero (deallocate) the memory.
  - In C/C++ this behavior is not defined.
- In some compile using certain compilers, the memory may be zeroed (fully de-allocated).
- In some cases when you ***call delete operator***, what happens is that the memory is marked as available, so the next time someone does new, the memory may be used .
  - *If you think about it, it's logical - when you tell the compiler that you are no longer interested in the memory (using delete), why should the computer spend time on zeroing it.*

Exercise

1. Compare dynamic array and static array. And also discuss the advantages of dynamic array over static array.
2. Provide an example of dangling pointer
3. Write a program that demonstrate resizing of dynamic array.
4. Consider the declaration: **int** matrix;**
   a) Create a dynamic 3x5 two dimensional array
   b) Write a function **matrixAllocate** that takes two integers, **M** and **N** and allocate a block of heap memory for ***NxM*** 2D array.
5. Write a program that demonstrate how to pass smart pointer to a function.
6. Modify your solution of previous chapter exercises using pointer (both raw pointer and smart pointer).

Exercise

7. What is/are the constraint(s) of using smart pointer with function.
8. Demonstrate how to convert ***shared_ptr*** to ***unique_ptr***?
9. Discuss the following concepts briefly by providing an example
   - *Undefined behavior*
   - *Unspecified behavior*
   - *Implementation-defined behavior*
10. Discuss the main features introduced in C++ versions (C++11 – C++20).

Reading Resources/Materials

Chapter 9:

- ✓ Walter Savitch; Problem Solving With C++ [10th edition, University of California, San Diego, 2018

Chapter 24:

- ✓ P. Deitel , H. Deitel; C++ how to program, 10th edition, Global Edition (2017)

Thank You
For Your Attention!!