Lecture 2

POINTERS & DYNAMIC MEMORY MANAGEMENT

September 07, 2021
Tuesday

POINTERS | REVISITED

Pointers are just addresses.
OR
Variables leading to Variables.

- Variables can be considered as boxes
  - Content or Value *(Either assigned by Programmer or Operating System)*
  - Location or Address of the box in the memory.
- Pointer, is a data item
  - Value is the memory address.
  - Type tells us which type of data we will find at that memory address.
  - It provides us an indirect access to a variable it points to.

POINTERS | INITIALIZATION

- All pointers variable are defined using *.

```
int* p, *q
```

```
int* p, k     // pointer p and integer k;
```

- Can be initialized with address of a variable, or another pointer, or null.

```
p  = &k;             // & is known as the address of operator
p = q;               // p and q both are pointers.
p = NULL;            // known as null pointer, as it points to nothing. In C++ NULL = 0
```

- An uninitialized pointer is known as **wild pointer**.

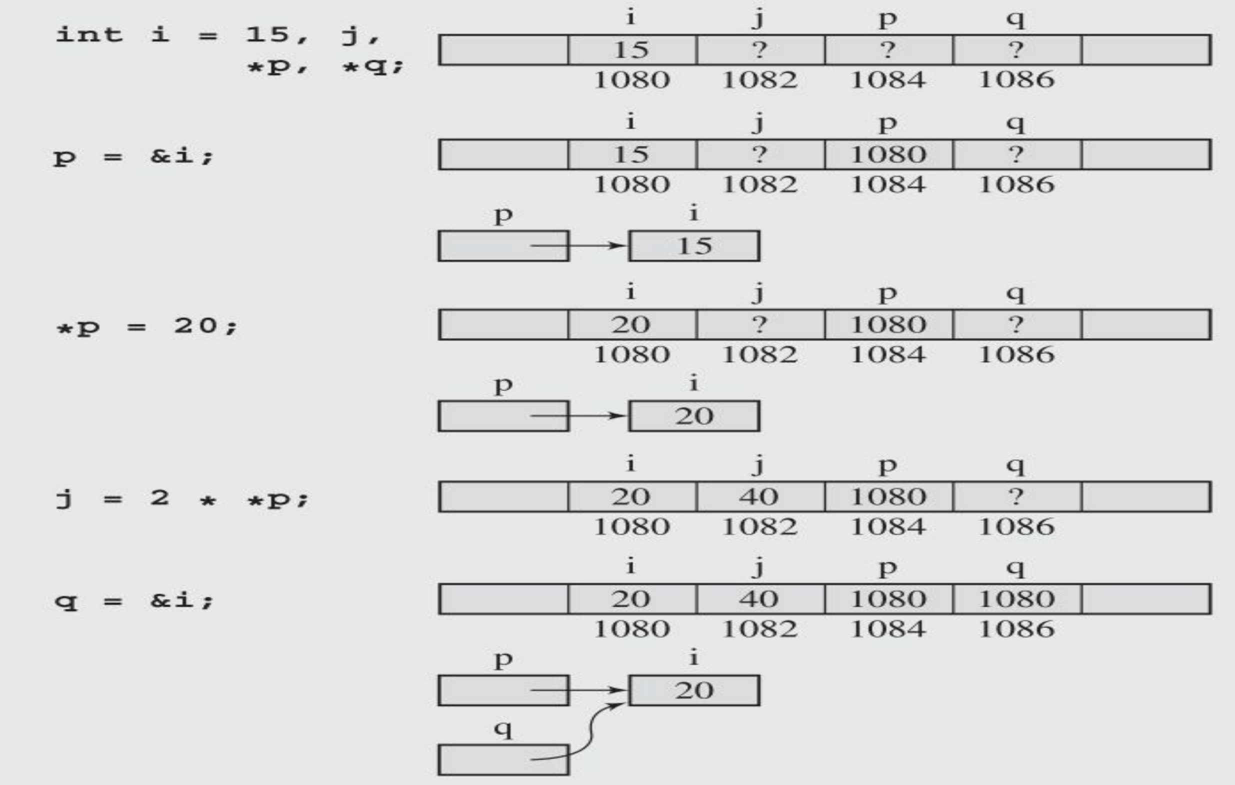
POINTERS | DEREFERENCING

- We can modify or inspect the location to which pointer is pointing.
- * operator known as indirection operator as well dereferencing operator

```
cout<<*p;          // inspecting the value using address referenced by pointer.
*p = 45;           // modifying the value using pointer.
```

- Yes, * operator is also used for pointer declaration.
- Yes, * operator is also used as multiplication operator.
- When we prepend a pointer with * it is used as dereferencing operator.
  - If * appears on the left side of the = sign, we are writing to the memory location.
  - else we are reading from the location.

POINTERS | EXAMPLE



```
int i = 15, j,
        *p, *q;
```

i	j	p	q
15	?	?	?
1080	1082	1084	1086

```
p = &i;
```

i	j	p	q
15	?	1080	?
1080	1082	1084	1086

```
*p = 20;
```

i	j	p	q
20	?	1080	?
1080	1082	1084	1086

```
j = 2 * *p;
```

i	j	p	q
20	40	1080	?
1080	1082	1084	1086

```
q = &i;
```

i	j	p	q
20	40	1080	1080
1080	1082	1084	1086

POINTERS & REFERENCE VARIABLES

- **r** is an integer reference variable.
  - must be initialized in its declaration as a reference to a particular variable.
  - this reference cannot be changed.
  - different name for a variable **n,** as **n** changes **r** changes.
- What a pointer accomplishes with dereferencing, a reference variable can accomplish without dereferencing.

```
int n = 123, *p = &n, &r = n;
```

Can a reference variable be null?

```
cout<<n<<'    '<<*p<<'  '<<r;
123   123   123
r = 17;
cout<<n<<'    '<<*p<<'  '<<r;
17    17    17
```

ARGUMENTS PASSING TO A FUNCTION

- How many ways we can pass an argument to a function?
  - Pass by Value (By Default)
    - Sending a copy of the variable.
  - Pass by Reference
    - The address of the argument is passed usually using **&** operator.
    - The receiving function may use * dereferencing operator to modify the value of the argument

When pass by reference
will be more appropriate?

POINTERS | TYPES

- NULL Pointer
- Void Pointer
- Constant Pointer
- Pointer to Function

What is the difference between?

1. int* const ptr
2. const int* ptr
3. const int* const ptr;

POINTERS | NULL POINTER

A null pointer points to nothing or null value.

```
int*  ptr = NULL;        // NULL = 0
```

- What happens when we dereference a null pointer?
  - **Segmentation Fault**
- Different from uninitialized and dangling pointers.
- Have a special purpose can be checked if a pointer is NULL before dereferencing.

Is it a good practice to crash your program?

POINTERS | VOID POINTER

Pointers which are type independent or generic.

- Pointers to some memory that the compiler doesn’t know the type of.
- Can not be dereferenced. C++ forbids void pointer arithmetics.
- static_cast operator can be used to convert the void type to respective data type.

```
int i = 10;
float f = 11.0;
double d = 12.0;
void* ptr;

ptr = &i;
ptr = &f;
ptr = &d;
cout<< *(static_cast<int*>(ptr));
```

POINTERS | CONSTANT POINTER

Pointer pointing to fixed memory location

- Constant Pointers
  - Cannot change the address pointer is pointing to

```
int* const   ptr;
```

- Pointers to Constants
  - Cannot update the value at the memory address.

```
const  int*  ptr;
```

- Constant Pointers to constant
  - Cannot update the address or value

```
const  int* const ptr;
```

POINTERS TO FUNCTIONS

Pointer, pointing to the memory address of the executable code of the function.

- We can call a function using pointer.
- We can pass a function as argument to other function.
- A function which takes other functions as arguments is called Functional.

POINTERS TO FUNCTIONS

```
int multiply(int a, int b){ return a * b; }

// display is Functional receiving a function as first argument
void display(int (*ptrMul) (int, int), char* message){
        cout<<”Message is “<<message;
        cout<<”Product is “<<ptrMul(5, 3);
}

int main ( ) {
        // defining and initializing a pointer to the function display.
        void (*pDisplay) (int (*ptrMul)(int, int), char*) = display;
        (*pDisplay)(multiply, “Pointers are Fun”);       //calling a function by pointer
}
```

MEMORY MANAGEMENT

STATIC MEMORY ALLOCATION

- We have used pointers to point a variable that already exists in the system.
    - All the memory program requires is already setup when we begin.
    - All the memory is allocated from a pool of memory known as **stack.**
    - Compiler performs all the allocation and deallocation.
    - Stack is usually allocated from bottom to the top.
    - Stack is faster just one statement for memory allocation.

What if we don’t know how much memory we would need?

DYNAMIC MEMORY ALLOCATION

- We can use pointers to get the address dynamically allocated memory at runtime.
- Programmer has to perform the allocation and deallocation of memory in a program for all the required objects/arrays/built-in data types.
- Dynamically allocated memory is coming from the pool of memory known as **heap.**
- Heap is usually allocated from top to bottom.
- Heap memory is slow.

POINTERS | DYNAMIC MEMORY ALLOCATION

- **new** keyword is used to dynamically allocate memory
  - Takes required memory from the memory manager.
  - Required memory is specified by the type following the new keyword.
  - If new cannot allocate required memory, it throws an exception **bad_alloc**.

```
int  *p = new int;          // allocates an uninitialized int
int *q = new int(4);        // allocates an int and initialized to 9
```

- **delete** keyword is used to return the allocated memory back to the available memory pool.
  - do not release the memory not allocated by new, as it results in undefined behavior

```
delete p;
```

POINTERS | ISSUES

- **Dangling Reference Problem**
    - The delete keyword only frees the memory location, but the pointer still holds the address. Such pointer is known as **dangling pointer**. Accessing the nonexistent location might crash the program.
    - After using delete, pointer must be assigned a new address or null value.
        - Deleting a null pointer twice is harmless.
    - Deleting a regular pointer twice may lead to
        - The memory manager might not allow executing the delete statement on this memory location as it is not owned by the program any more.
        - The memory may now belongs to another object and would result in unwanted data loss.

POINTERS | ISSUES

- **Memory Leak**
    - Assigning multiple memory locations to one pointer without releasing the first memory before next assignment. First allocated memory becomes inaccessible

```
p = new int;
p = new int;
```

```
p = new int;
delete p;
p = new int;
```