C++ : Pointers

pm_jat@daiict.ac.in

Pointer Type

- A pointer type is a one in which the variables can have memory address and special value NULL
- Null is not a valid address, this simply means “none”
- When a pointer variable has a memory address as its value, we say that this variable points to that memory
- One important use is; pointers provides a way to manage dynamic storage.
- A pointer can be used to access a location in the area where storage is dynamically allocated, which is usually called heap

Operations on Pointers

- Fundamental operations on pointers are assignment and de-referencing.
- Assignment operation sets a pointer variable’s value to some useful address.
- Dereferencing: Access variable associated with the address stored in the pointer variable

Creating Pointers in C/C++

- Declare

```
int *ptr;
```

- Creates a pointer variable ptr.

- C++ requires you to specify type of data which are stored at address stored in pointer variable.

- In another words, the pointer can point to address used to store that type only !!

Assignment to Pointers

- Assignment, can be done by three ways.
- First, Get memory from heap

  ptr = new int; //or malloc/calloc/ralloc function of C

  – ptr gets a value, that is address of new allocated memory from heap.
- Second, get address of any existing variable. For example-

  int *ptr, x;

  ptr = &x;

  – ptr gets a value, that is address of x
- Third, assign value of another pointer variable of same type

  int x, *ptr1 = &x, *ptr2;

  ptr2 = ptr1;

Dereferencing

- Purpose it to access variable associated with the address stored in the pointer variable
- Consider code below-

```
int *p, x = 50, y;
p = &x; //Assign to pointer variable, p now points to x
y = *p; //Take the r-value of variable that p points
cout << y; //would output 50
*p += 10; //Take the l-value of variable that p points
cout << x << y; //what would be output?
```

Example: Pointers

- Look at the code!!

```
int* p, x = 15;
p = &x;
```

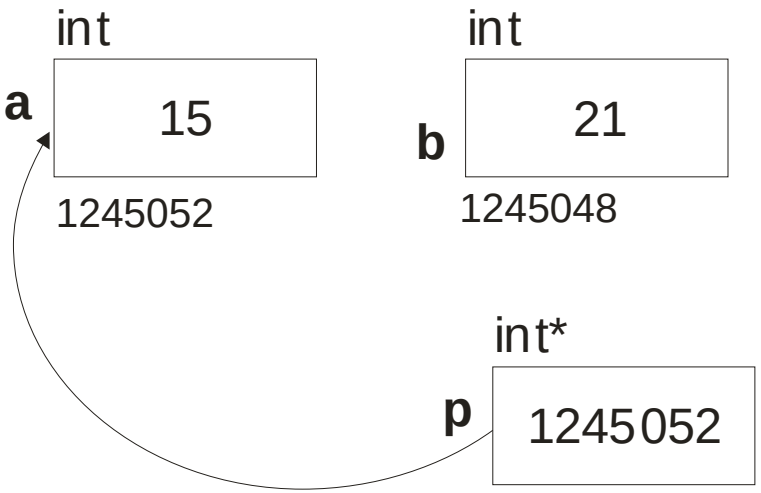
- This means?

What is output of this code?

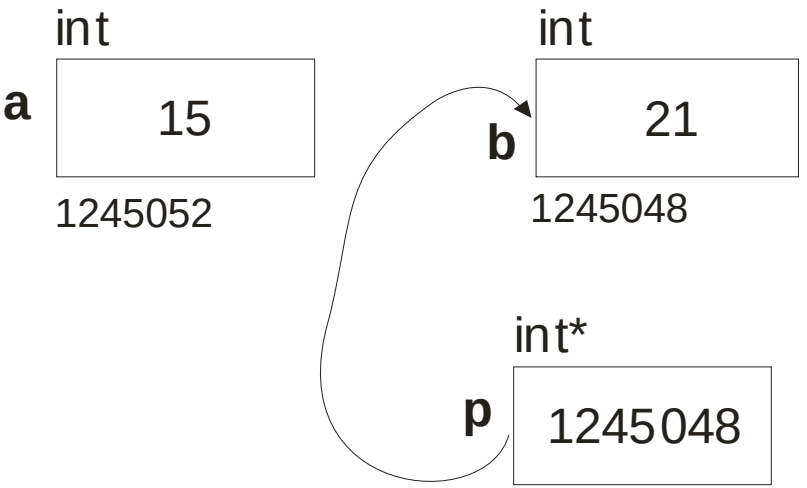
```
int main()
{
   int a=15, b=21;
   int* p;
   p = &a;
   cout << p << a << b << *p;
   p = &b;
   cout << p << a << b << *p;
}
```

p first points to **a** then it points to **b**

Pointers Example



p = &a; sets p points to a



p = &b; sets p points to b

Revise : What is a Pointer

- A variable like any other variable … and stores memory address … have type associated to which it can point
- It points to address that is stored as its value
- Pointer variable gets its value, either by new allocation, or by extracting address of another variable, or by simply assigning another pointer variable of same type
- By dereferencing you get r-value/l-value of the variable to which it points

Pointer Pitfalls: Understand : `int *p`

- Suppose, you have following declaration

```
int *p;
```

- Have you allocated storage to p?
- Yes. Storage binding of p is done here, i.e. could be static or stack-dynamic depending upon its context of declaration
- Caution: Sometimes student get confusion between storage binding of p, and value binding of p?
- In above case storage binding of p has been done but not value binding, and it is the value binding makes p to point some memory area

pointer needs to point to some valid memory before using

- Before you can meaningfully use pointers, you need to make it to point to some memory area (that is assign some value to it)
- In C/C++, any un-initialized (auto) variable store junk (static are filled with zero)
- Following statement may crash you program?

```
int *p;
*p = 100;
```

- Because p is not pointing to valid memory; therefore dereferencing is illegal

Pointer Pitfalls:
Context determines meaning of * with a pointer variable

- Consider following two code fragment

```
int *p = new int; //Line-1
*p = 100; //Line-2
```

- See what appears assigning to *p in above two statements, may be confusing
- In fact, ***** in declaration (Line-1) has different meaning than ***** in assignment (Line-2)
- In declaration ***** indicates that p is a pointer variable, while anywhere else ***** indicates dereferencing

Pointer Pitfalls:
Context determines meaning of * with a pointer variable

- Probably you would have less confusion, if you re-write the code as following-

```
int* p = new int; //Line-1
*p = 100; //Line-2
```

- And that is what we mean by previous version of code too

- Unfortunately for C++, both mean same, whereas you might read these differently

Pointer Pitfalls: Understand : pointer type

- So, second approach, appears better. And it is possibly because we assumed int* as separate data type; which is OK as far as your understanding goes, but truly C/C++ has no such data type defined
- For example if you write following code to create three pointers to int (int*) variables-

```
int* pa, pb, pc;
```

- However this makes pa pointer to int, while others int; actually it pushes * to the variable name, and interprets it as *pa; making it equivalent to-

```
int *pa, pb, pc;
```

Pointer Pitfalls:
Understand : pointer type

- Therefore you need to correct it, as -

```
int *pa, *pb, *pc;
```

- So, you have come back to original style of declaring pointer variables, that is placing * closer to the variables, even though it is source of confusion

Pointer Pitfalls:
Understand : pointer type

- In fact, most of confusion should go away if you correctly understand-
  - What is data type of p, and what you can assign to it
  - What is data type of *p, and so forth

- For this, let us assume a hypothetical data type T*; where T is any valid Abstract Data Type

Pointer Pitfalls: Understand : pointer type

- Consider following declaration-

```
int a, *pa, b, *pb;
```

- Below is how you may read it-
  - Type of a, and b are int
  - pa, and pb are pointer variables, because these are prefixed with *, and
  - int is the type you have when you dereference pa, and pb

Pointer Pitfalls: Understand : pointer type

- You can say that-
  - a, b are int
  - pa, pb are int*
  - *pa, *pb are int

- Therefore following statements have no type errors

```
int a, *pa = &a, b, *pb = &b;
*pa = 100; *pb = 50;
pb = &a; pa = &b;
cout << *pa << *pb << endl;
```

An Example

- Example: int* as data type

```
typedef int* IntPtr;
int main()
{
    int a = 100;
    IntPtr p1, p2;
    p1 = &a;
    p2 = &a;
    cout << *p1 << *p2 << endl;
    return 0;
}
```

Pointer Pitfalls:
Understand : pointer type

- Our this understanding of having hypothetical data type T*, must greatly help in understanding multi-level dereferencing.
- If you have declaration

```
int a, *b, **c, ***d;
```

- You have following data types for your all type checking purpose-

```
a – int
b – int*, *b - int
c – int**, *c – int*, **c - int
d - int***, *d – int**, **d – int*, ***d - int
```

Pointer Arithmetic

- You can add to and subtract from a pointer variable!
- Adding 1 to a pointer variable, means advancing it to “next memory cell”
- Suppose p is a pointer to data type T pointing to address a; then address of next memory cell would be a + sizeof(T)
- Now if we add n to p; i.e. p + n, would be a + n * sizeof(T)
- Same applies to subtraction too; and p - n, would be a - n * sizeof(T)

Pointer Arithmetic

- Following are valid C/C++ expressions, where p is a pointer variable and i be an *integer*

```
p++; p--; p += i; p -= i;
```

- Pointer arithmetic is often used when you have discrete list of objects, and are stored in contiguous memory, like arrays

Pointer Problems

- Dangling Pointer or Reference
- Lost Heap-Dynamic Variables or Garbage
- In C/C++, Freedom of pointers make memory vulnerable to un-authorized access
  - … pointer arithmetic and

Dangling Pointer or Reference

- A Dangling Pointer or Dangling Reference, is a pointer that contains the address of a “heap-dynamic variable” that has been de-allocated.
- … pointer still points to that memory
- Dangling pointers are dangerous for several reason-
  - The location being pointed may have been re-allocated to some new “heap-dynamic variable”
  - You may get wrong data if dereference dangling pointer, because it may have data of new “heap-dynamic variable”
  - You may overwrite data of new “heap-dynamic variable”

Some of the instances leaving dangling pointers

- You use object after destroying it

```
int* p = new int;
*p = 100;
cout << *p << endl;
delete p;
//hereafter p should not be used

cout << *p << endl; //may show up wrong info
cin >> *p; //may overwrite another object's data
```

- Good practice is always set such pointers to NULL immediately after delete

```
p = NULL;
```

Some of the instances leaving dangling pointers

- Return address of some auto variable from a function

```
int* funcA() {
  int array[10], x;
  ...
  //would cause a dangling pointer
  return array;
  //or
  return &x;
}
```

- Therefore NEVER return address of a auto variable

Lost Heap-Dynamic Variables or Garbage

- A lost heap-dynamic variable is an allocated heap-dynamic variable that is no more accessible to the user program.

- This often called garbage, because this the allocated memory which your program lost reference of, and also can not be re-allocated to some other variable, as still considered to be in use

Lost Heap-Dynamic Variables or Garbage

- You have garbage when
  - you allocate memory from heap by new, malloc, or so, and do not free it
  - you write a function which returns a pointer to dynamically allocated memory within the function, and caller may not know that it need to be free the memory …not a good function design …

- In C/C++, if you do not explicitly allocate any memory, then possibly garbage is not the concern
- Some languages like java provide automatic garbage collection based on if there are any active references to such memory

void pointer in C/C++

- Which can point to any type

```
void *p;
```

- They are generic pointers
- Can not dereference a void pointer, and can not do any pointer arithmetic
- You need to cast it to a type to dereference
- Normally used to pass generic pointers as parameters to functions and return generic pointers from function
- … where caller casts returned pointer to a particular pointer type before dereferencing

Thanks