

POINTERS

C++ Memory Map

Once a program is compiled, C++ creates four logically distinct regions of memory:

Code Area : Area to hold the compiled program code

Data Area : Area to hold global variables

Stack Area : Area to hold the return address of function calls, argument passed to the functions, local variables for functions and the current state of the CPU.

Heap : Area from which the memory is dynamically allocated to the program.

Accessing address of a variable

Computer's memory is organized as a linear collection of bytes. Every byte in the computer's memory has an address. Each variable in program is stored at a unique address. We can use address operator & to get address of a variable:

```
int num = 23;
cout << &num;    // prints address in hexadecimal
```

POINTER

A pointer is a variable that holds a memory address, usually the location of another variable in memory.

Defining a Pointer Variable

```
int *iptr;
```

iptr can hold the address of an int

Pointer Variables Assignment:

```
int num = 25;
int *iptr;
iptr = &num;
```

Memory layout



To access num using iptr and indirection operator *

```
cout << iptr;    // prints 0x4a00
cout << *iptr;   // prints 25
```

Similarly, following declaration shows:

```
char *cptr;
```

```
float *fptr;
```

cptr is a pointer to character and fptr is a pointer to float value.

Pointer Arithmetic

Some arithmetic operators can be used with pointers:

- Increment and decrement operators ++, --
- Integers can be added to or subtracted from pointers using the operators +, -, +=, and -=

Each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

If “p” is a character pointer then “p++” will increment “p” by 1 byte.

If “p” were an integer pointer its value on “p++” would be incremented by 2 bytes.

Pointers and Arrays

Array name is base address of array and is a constant pointer.

```
int vals[] = {4, 7, 11};
cout << vals;    // displays 0x4a00
cout << vals[0]; // displays 4
```

Lets takes an example:

```
int arr[]={4,7,11};
```

```
int *ptr = arr;
```

What is ptr + 1?

It means (address in ptr) + (1 * size of an int)

```
cout << *(ptr+1); // displays 7
```

```
cout << *(ptr+2); // displays 11
```

Array Access

Array notation arr[i] is equivalent to the pointer notation *(arr + i)

Assume the variable definitions

```
int arr[]={4,7,11};
```

```
int *ptr = arr;
```

Examples of use of ++ and --

```
ptr++; // points at 7
```

```
ptr--; // now points at 4
```

Character Pointers and Strings

Initialize to a character string.

```
char* a = "Hello";
```

a is pointer to the memory location where ‘H’ is stored. Here “a” can be viewed as a character array of size 6, the only difference being that a can be reassigned another memory location.

```
char a[5] = "Hello", *s=a;
```

```
a    Hello
```

*a gives 'H'
a[0] gives 'H'

S gives Hello , *s gives H

S++; *s='a'; now string a is Hallo, and S is allo

Pointers as Function Parameters

A pointer can be a parameter. It works like a reference parameter to allow change to argument from within function

Pointers as Function Parameters

```
void swap(int *x, int *y)
{
    int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
swap(&num1, &num2);
```

Pointers to Constants and Constant Pointers

Pointer to a constant: cannot change the value that is pointed at

Constant pointer: address in pointer cannot change once pointer is initialized

Pointers to Structures

We can create pointers to structure variables

```
struct Student {int rollno; float fees;};
Student stu1;
Student *stuPtr = &stu1;
(*stuPtr).rollno= 104;
```

-or-

Use the form ptr->member:

```
stuPtr->rollno = 104;
```

Static allocation of memory

In the static memory allocation, the amount of memory to be allocated is predicted and preknown. This memory is allocated during the compilation itself. All the declared variables declared normally, are allocated memory statically.

Dynamic allocation of memory

In the dynamic memory allocation, the amount of memory to be allocated is not known. This memory is allocated during run-time as and when required. The memory is dynamically allocated using new operator.

Free store

Free store is a pool of unallocated heap memory given to a program that is used by the program for dynamic allocation during execution.

Dynamic Memory Allocation

We can allocate storage for a variable while program is running by using new operator

To allocate memory of type integer

```
int *iptr=new int;
```

To allocate array

```
double *dptr = new double[25];
```

To allocate dynamic structure variables or objects

```
Student sptr = new Student; //Student is tag name of structure
```

Releasing Dynamic Memory

Use delete to free dynamic memory

```
delete iptr;
```

To free dynamic array memory

```
delete [] dptr;
```

To free dynamic structure

```
delete Student;
```

Memory Leak

If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks. These orphaned memory blocks when increase in number, bring adverse effect on the system. This situation is called memory leak

Self Referential Structure

The self referential structures are structures that include an element that is a pointer to another structure of the same type.

```
struct node
{
    int data;
    node* next;
}
```