



Class - 4

Pointers



POINTERS

- The address of the first byte allocated to variable is known as address of variable.
- **'&' operator is the "address of" operator** in C++ which returns the address.
- The address operator cannot be used with a constant or an expression

Eg -

int a = 5;	-	&a	Valid
float f = 8.6;	-	&f	Valid
Constant	-	&26	Invalid
Expression	-	&(a+f)	Invalid

- Pointer variables are used to store the memory address. Used like normal variables.

POINTERS

- The general syntax of declaration of pointer variable is: *datatype *p_name;*
- The size allocated to all pointer variables is same as they all store integers.
- '*' is used to dereference the pointer
- In the program attached,
 - ptr can be used to access the address of variable a and *ptr can be used to access its value
 - Writing *(&a) and a is same.
- For referencing character pointer, we need to typecast it into something which is not C-style string. The reason for this unusual behavior is operator overloading of '<<'.

Example :

```
char c='S', *pc=&c;  
cout<<pc; //prints S  
cout<<(void*)pc; //prints address of char c
```

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int a=5;  
  
    int *ptr = &a; // address of 'a' is assigned to a pointer variable  
  
    cout<<"Value of ptr : "<<ptr<<"\n";  
    cout<<"Value stored at address "<<ptr<<" is "<<*ptr;  
    return 0;  
}
```

Output :

```
Value of ptr : 0x61ff08  
Value stored at address 0x61ff08 is 5
```

PREDICT THE OUTPUT

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

int main() {
    int a = 5;
    int *ptr;
    ptr = &a;
    *ptr = *ptr * 3;
    cout<<a;
}
```

PREDICT THE OUTPUT

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

int main() {
    int a = 5;
    int *ptr;
    ptr = &a;
    *ptr = *ptr * 3;
    cout<<a;
}
```

Output : 15

POINTER ARITHMETIC

- Addition and subtraction of an integer and a pointer variable is supported
- Pointer variables can also be used for increment and decrement operation
- **The increment/decrement operation changes the value of pointer variable by the size of data type that it points to.**
- Subtraction of two pointer variables of same base type returns the number of values present between them

Eg – `int *ptr1 = 2000, *ptr2 = 2020;`

`cout << ptr2 - ptr1;` Output : 5

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 5, *pi = &a;
6      double b = 6.5, *pd = &b;
7      char ch = 'P', *pc = &ch;
8
9      cout<<"Old value of pi : "<<pi<<"\n";
10     cout<<"Old value of pd : "<<pd<<"\n";
11     cout<<"Old value of pc : "<<(void*)pc<<"\n";
12
13     pi++;
14     pd = pd + 2;
15     pc++;
16
17     cout<<"New value of pi : "<<pi<<"\n";
18     cout<<"New value of pd : "<<pd<<"\n";
19     cout<<"New value of pc : "<<(void*)pc<<"\n";
20     return 0;
21 }
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\smart\OneDrive\Desktop> g++ Untitled-1.cpp
PS C:\Users\smart\OneDrive\Desktop> .\a.exe
Old value of pi : 0x61ff00
Old value of pd : 0x61fef8
Old value of pc : 0x61fef7
New value of pi : 0x61ff04
New value of pd : 0x61ff08
New value of pc : 0x61fef8
PS C:\Users\smart\OneDrive\Desktop>
```

COMBINATION OF DEREFERENCE AND INCREMENT/DECREMENT

- The dereference operator (*), address of operator (&) and increment/decrement have same precedence and are **Right to Left Associative**.

Expression	Evaluation	
<code>x = *ptr++</code>	<code>x = *ptr</code>	<code>ptr = ptr + 1</code>
<code>x = *++ptr</code>	<code>ptr = ptr + 1</code>	<code>x = *ptr</code>
<code>x = (*ptr)++</code>	<code>x = *ptr</code>	<code>*ptr = *ptr + 1</code>
<code>x = ++*ptr</code>	<code>*ptr = *ptr + 1</code>	<code>x = *ptr</code>

POINTER TO POINTER

- Pointer variable contains an address, and this variable takes space in memory so it itself has an address
- A pointer-to-pointer variable is used to store the address of a pointer variable
- The general syntax of declaration of pointer variable is:

`datatype **pp_name;`

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int a = 5;           // Integer variable
6      int *ptr = &a;       // Pointer to int
7      int *pptr = &ptr;    // Pointer to pointer to int
8      printf("Address of a   : %u\n", &a);
9      printf("Value of ptr   : %u\n", ptr);
10     printf("Address of ptr : %u\n", &ptr);
11     printf("Value of pptr  : %u\n", pptr);
12     printf("Address of pptr: %u\n", &pptr);
13     return 0;
14 }
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
$ gcc -w test.c
$ ./a.out
Address of a   : 3882647076
Value of ptr   : 3882647076
Address of ptr : 3882647080
Value of pptr  : 3882647080
Address of pptr: 3882647088
$
```


POINTER WITH 1D ARRAY

Consider an array

```
int arr[] = {1,2,3,4,5};
```

Here arr is a pointer to the first element aka arr is a pointer to int or (int*)

Remember

```
arr = &arr[0]
```

```
arr + 1 = &arr[1]
```

```
arr + 2 = &arr[2]
```

```
arr + 3 = &arr[3]
```

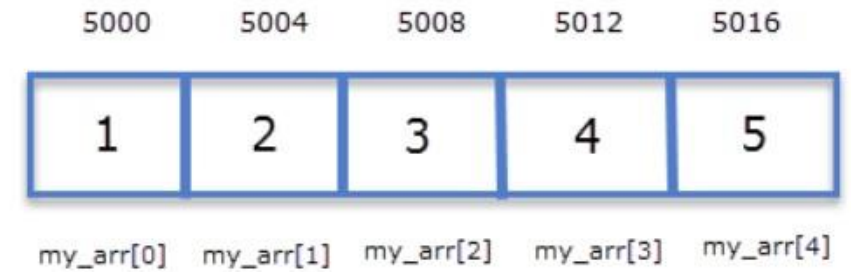
Thus

```
*(arr) = arr[0]
```

```
*(arr + 1) = arr[1]
```

```
*(arr + 2) = arr[2]
```

```
*(arr + 3) = arr[3]
```



```
int *p;
```

```
int arr[] = {11, 22, 33, 44, 55};
```

```
p = arr;
```

We **can do** p++, p--

but we **can not do** arr++, arr--

POINTER AND FUNCTIONS

- **Call by value**

```
void swapx(int x, int y) {  
    int t;  
    t = x;  
    x = y;  
    y = t;  
    cout<<"x= "<<x<<" y= "<<y;  
}
```

- **Call by reference**

```
void swapx(int* x, int* y) {  
    int t;  
    t = *x;  
    *x = *y;  
    *y = t;  
    cout<<"x= "<< *x <<" y= "<< *y;  
}
```

How can we return more than one value from function ?

PREDICT THE OUTPUT

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

int f(int x, int *py, int **ppz) {
    int y, z;
    **ppz += 1;
    z = **ppz;
    *py += 2;
    y = *py;
    x += 3;
    return x + y + z;
}

int main() {
    int c, *b, **a;
    c = 4;
    b = &c;
    a = &b;
    cout<<f(c,b,a);
}
```

PREDICT THE OUTPUT

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

int f(int x, int *py, int **ppz) {
    int y, z;
    **ppz += 1;
    z = **ppz;
    *py += 2;
    y = *py;
    x += 3;
    return x + y + z;
}

int main() {
    int c, *b, **a;
    c = 4;
    b = &c;
    a = &b;
    cout<<f(c,b,a);
}
```

Output : 19

Memory Layout in C++ program

1. Text Segment:

Also, known as Code segment, Contains executable instructions.

2. Data Segment

Divided into two parts

Initialized Data Segment

Contains Global and static variables that are initialized. It's not a read-only segment and hence the values can be modified.

Uninitialized Data Segment

Data in this segment are initialized by the kernel to 0 before the program starts execution.

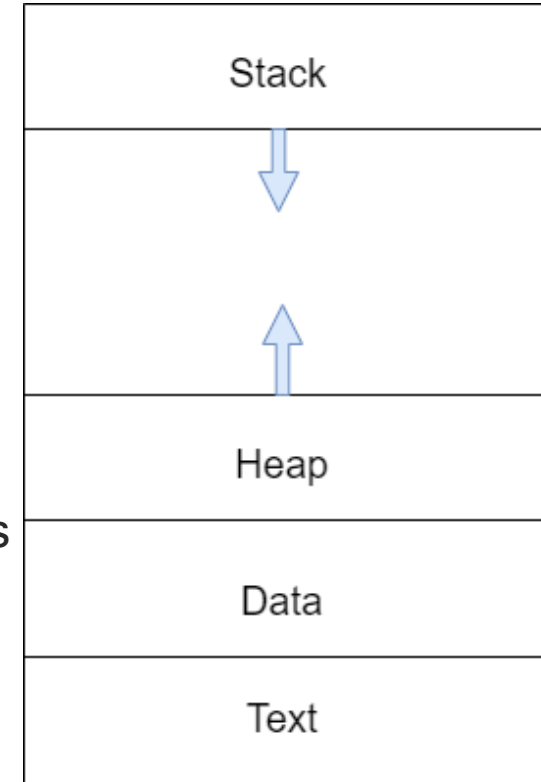
3. Stack:

Temporary variables are stored in this area. The virtual pointer is also stored here

Stack Frame: A set of values pushed for one function call is called Stack Frame.

4. Heap:

Here dynamic memory allocation takes place.



Memory allocation

Reserving or providing space to a variable is called memory allocation. For storing the data, memory allocation can be done in two ways –

- Static allocation or compile-time allocation** - Static memory allocation means providing space for the variable. The size and data type of the variable is known, and it remains constant throughout the program.

Example : `int arr[5]={1,5,10,15,25};`

- Dynamic allocation or run-time allocation** - The allocation in which memory is allocated dynamically. Pointers play a major role in dynamic memory allocation.

It is done using predefined functions like `new` and `delete` in C++ and `malloc()`, `calloc()` and `free` in C.

Note: Dynamically allocated memory is allocated on **Heap**, and non-static and local variables get memory allocated on **Stack**.

DYNAMIC MEMORY ALLOCATION

void pointer

- The void pointer in C++ is a pointer which is not associated with any data types.
- It is a general-purpose pointer.
- It can point to any data type.

```
int a = 5;  
float b = 7.6;  
void *p;
```

```
p=&a;  
cout<<"Integer variable is "<<*((int* )p); // prints 5
```

```
p=&b;  
cout<<"Float variable is "<<*((float* )p); // prints 7.6
```

DYNAMIC MEMORY ALLOCATION

- **Malloc**

```
ptr = (cast-type*) malloc(byte-size)
```

- **Calloc**

```
ptr = (cast-type*) calloc(n, element-size);
```

Malloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```


ptr =  ← 20 bytes of memory →

Annotations: A bracket above the expression `5 * sizeof(int)` points to the text "4 bytes". An arrow points from the text "A large 20 bytes memory block is dynamically allocated to ptr" to the rectangle.



Calloc()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

ptr =  → 5 blocks of 4 bytes each is dynamically allocated to ptr

Annotations: A bracket above the expression `5, sizeof(int)` points to the text "4 bytes". A bracket below the row of rectangles is labeled "4b". A bracket below the entire row is labeled "20 bytes of memory". An arrow points from the text "5 blocks of 4 bytes each is dynamically allocated to ptr" to the row of rectangles.



DYNAMIC MEMORY ALLOCATION

- **new operator**

To allocate the space dynamically, the operator new is used.

Syntax

Pointer_variable = new data_type;

The pointer_variable is of pointer data_type. The data type can be int, float, string, char, etc.

Example:

```
int *m = new int;    //new integer allocated
```

```
Float *d = new float(21.01);  // new float variable allocated with initial value=21.01
```

```
int *arr = new int[10];    //10 blocks each of sizeof(int) allocated
```

DYNAMIC MEMORY ALLOCATION

- **delete operator**

We delete the allocated space in C++ using the **delete** operator.

Syntax

delete pointer_variable_name;

Example

delete m; // free m that is a variable

delete [] arr; // Release a block of memory

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

int main() {
    // Below variables are allocated memory dynamically.
    int *ptr1 = new int;
    int *ptr2 = new int[10];

    // Dynamically allocated memory is deallocated
    delete ptr1;
    delete [] ptr2;
    return 0;
}
```

NOTE: While allocating memory on heap we need to delete the memory manually as memory is not freed(deallocated) by the compiler itself even if the scope of allocated memory finishes(as in case of stack).

RECURSION

- A technique where a function calls **itself** in loop.
- A powerful construct to solve complex problems easily
- Before writing a recursive function for a problem we should consider these points:
 - We should be able to define the solution of the problem in terms of a similar type of smaller problem.
 - At each step we get closer to the final solution of our original solution
 - There should be a terminating condition to stop recursion.

```
1  #include<stdio.h>
2
3  void func()
4  {
5      //Statement Before Calling
6      //Exit Condition
7      func();
8      //Statement before return
9  }
10
11 int main()
12 {
13     func();
14 }
```

PROBLEM: PRINT 1 TO N

Here are 2 variants of the problem.

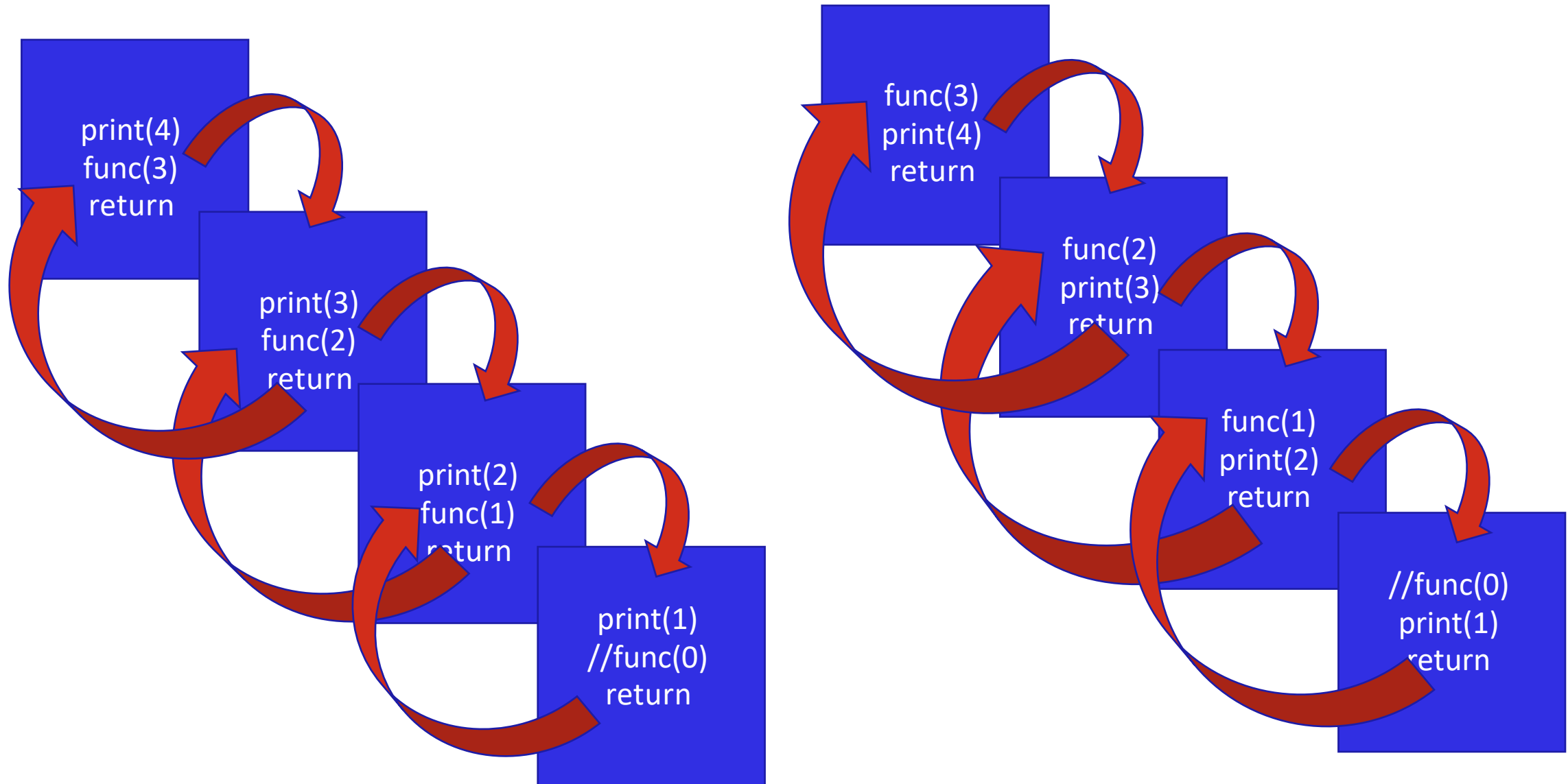
One where numbers are printed in ascending order and other with descending order.

- Step 1: Determine the subproblem
- Step 2: Identify Exit Condition
- Step 3: Determine where to solve the current iteration (before recursive call or after recursive call)

```
void printNumbers(int n) {  
    if(n>1)  
        printNumbers(n-1);  
    cout<<n<<" ";  
}
```

```
void printNumbers(int n) {  
    cout<<n<<" ";  
    if(n>1)  
        printNumbers(n-1);  
}
```

HOW IT'S WORKING... ?



PROBLEM: FACTORIAL OF A NUMBER

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

int fact(int n) {
    if(n==1) return 1;
    return n*fact(n-1);
}

int main() {
    int n;
    cin>>n;

    cout<<fact(n);
    return 0;
}
```

