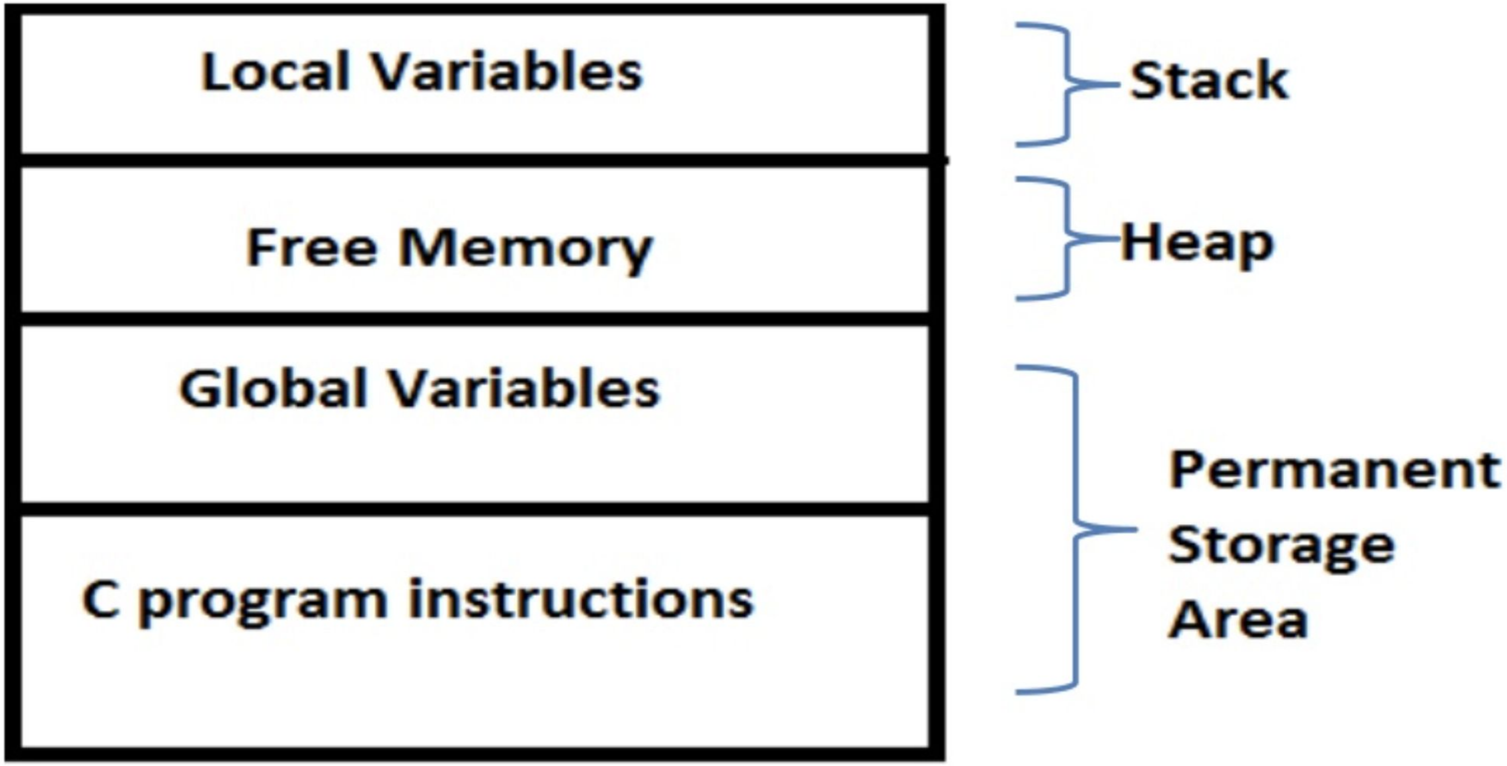
Memory segments

HEAP AND STACK

- Our program when loaded in to main memory is divided in to 4 segments : CODE,DATA,STACK,HEAP
- A data segment contains the global variables and static variables.
- A code(text) segment contains the executable instructions.
- A Stack segment store all the auto variables. Also each function call involves passing arguments from the caller to the callee. The callee may also declare variables. Function parameters ,return address and *automatic* local variables are accommodated in the stack.
- Hence a *stack* is an area of memory for storing data temporarily.

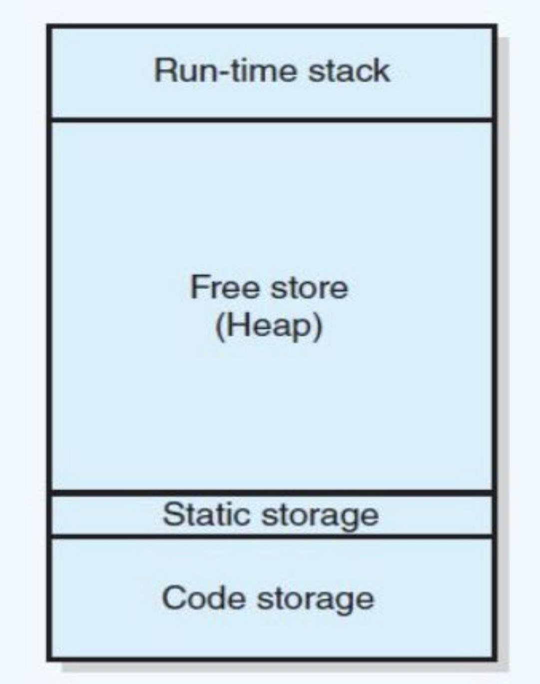
Memory segments



Memory segments: HEAP (free store)

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

https://www.youtube.com/watch?v=Hx6E2gPrpz8



Overview of Pointers

- A Pointer in C++ is variable whose value is a memory address.
- With pointers many memory locations can be referenced.
- Some data structures use pointers (e.g. linked list, tree).
- **<u>The * and & operators</u>**
  - & operator is the address operator
  - * operator is the dereferencing operator. It is used in
    pointers declaration

Overview of Pointers

- ❑ A pointer is a variable that holds the memory address of another variable of same type.
- ❑ This memory address is the location of another variable where it has been stored in the memory.
- ❑ It supports dynamic memory allocation routines.

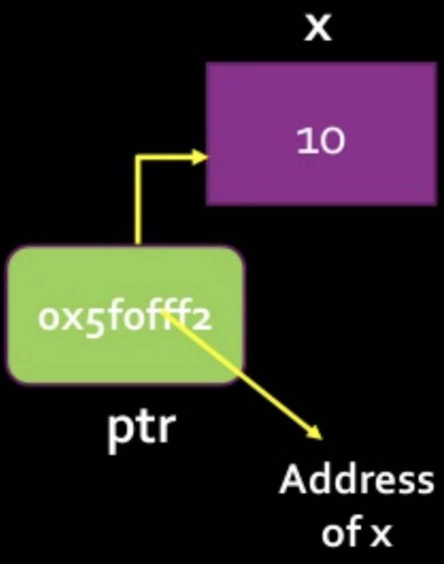
Address-of operator

Syntax : Datatype *variable_name;
eg. int *x; float *y; char *z;

Address of operator(&)- it is a unary operator that returns the memory address of its operand. Here the operand is a normal variable.

eg. int x = 10;
int *ptr = &x;

Now ptr will contain address where the variable x is stored in memory.

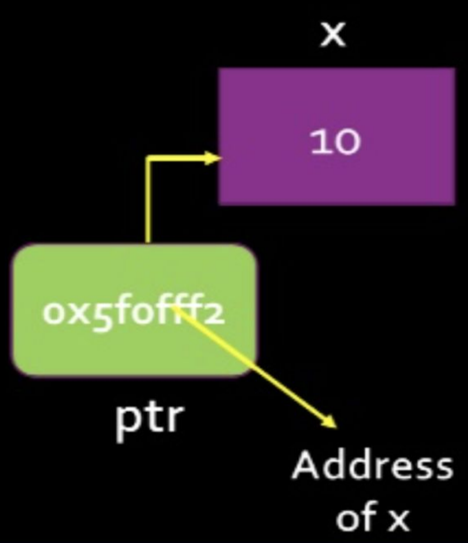


Dereference operator

- ❑ It is a unary operator that returns the value stored at the address pointed to by the pointer.
- ❑ Here the operand is a pointer variable.

eg.

```
int x = 10;
int *ptr = &x;
cout<< ptr;// address stored at ptr will be displayed
cout<<*ptr;// value pointed to by ptr will be displayed
```



OUTPUT

0x5f0fff2
10

Now ptr can also be used to change/display the value of x.

Fixed-size Arrays

Int main()

// 1 . Size fixed

{

Int A[5]; ⟶

1020	1022	1024	1026	1028
A[0]	A[1]	A[2]	A[3]	A[4]

Int B[n]; ⟶ // 2. Error..Size must be known at compile time.

}

Will not work for larger programs because of the limited size of the stack.

Static array vs dynamic array

Static Array	Dynamic Array
It is created in stack area of memory	It is created in heap area of memory
The size of the array is fixed.	The size of the array is decided during run time.
Memory allocation is done during compilation time.	Memory allocation is done during run time.
They remain in the memory as long as their scope is not over.	They need to be deallocated using delete operator.

Arrays

C++ treats the name of an array as constant pointer which contains base address i.e. address of first location of array.

For eg.

```
int x[10];
```

Here x is a constant pointer which contains the base address of the array x.

Dynamic arrays

- When we define an array variable, we specify a type , a name, and a dimension.

  int a[2]

- When we dynamically allocate an array, we specify the type and size but no name

- This new expression allocates an array of ten integers and returns a pointer to the first element in that array, which we use to initialize ptr.

```
int *ptr = new int[10];
// array of 10 uninitialized  integers (contain garbage value)
```

- Objects allocated on the free store are unnamed. We use objects on the heap only indirectly through their address.

New / delete / delete[]

- When we allocate memory, we must eventually free it.
- Otherwise, memory is gradually used up and may be exhausted
- We do so by applying the delete [] expression to a pointer that addresses the array we want to release:

  delete [ ] ptr; //de allocates the array pointed to by ptr

- The empty bracket pair between the delete keyword and the pointer is necessary
- It indicates to the compiler that the pointer addresses an array of elements
- It is essential to remember the bracket-pair when deleting pointers to arrays.
- For every call to new, there must be exactly one call to delete.

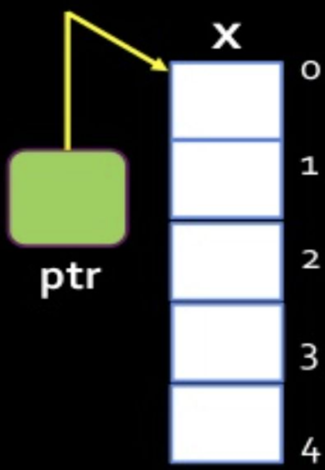
New / delete / delete[]

- When we allocate memory, we must eventually free it.
- Otherwise, memory is gradually used up and may be exhausted
- We do so by applying the delete [] expression to a pointer that addresses the array we want to release:

  delete [ ] ptr; //de allocates the array pointed to by ptr
- The empty bracket pair between the delete keyword and the pointer is necessary
- It indicates to the compiler that the pointer addresses an array of elements
- It is essential to remember the bracket-pair when deleting pointers to arrays.
- For every call to new, there must be exactly one call to delete.

Array decays into pointer

We can also store the base address of the array in a pointer variable. It can be used to access elements of array, because array is a continuous block of same memory locations.



For eg.

```
int x[5];
int * ptr=x; // ptr will contain the base address of x
```

we can also write

```
int * ptr= &x[0]; //ptr will contain the base address of x
```

Constant pointer vs pointer to constant

- **Constant Pointer-** It means that the address stored in the pointer is constant i.e. the address stored in it cannot be changed. It will always point to the same address. The value stored at this address can be changed.

- **Pointer to a Constant-** It means that the pointer is pointing to a constant i.e. the address stored in the pointer can be changed but the pointer will always point to a constant value.

Constant pointer vs pointer to constant

For e.g.

Case 1:

```
int x=10, y=20;
 int * p1=&x;  //non-const pointer to non-const int
 *p1=20;//valid i.e. value can be changed
 p1=&y;  //valid i.e. address in p1 can be changed. Now it will point to y.
```

Case 2:

```
 const int x=10;
 int y=20;
 const  int * p2=&x; // non-const pointer to const int
*p2=50; //invalid i.e. value can not be changed
  p2=&y; // valid i.e. address stored can be changed
 *p2=100;// invalid  as p2 is pointing to a constant integer
```

Constant pointer vs pointer to constant

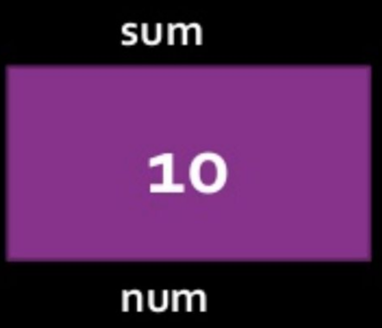
```
Case 3:
 int x=10,y=20;
int * const  p3= &x; //const pointer to non-const int
*p3=60; //valid i.e. value can be changed
p3=&y; //invalid as it is a constant pointer, thus address can not be changed

Case 4:
int x=10,y=20;
const int * const  p4=&x; // const pointer to const int
 p4=&y;// invalid
 *p4=90;// invalid
```

“References” in C++

A reference variable is a name that acts as an alias or an alternative name, for an already existing variable.

SYNTAX:
Data type & variable name = already existing variable;



EXAMPLE:

```
int num=10;
int & sum = num; // sum is a reference variable or alias name for num
```

NOTE: Both num and sum refer to the same memory location. Any changes made to the sum will also be reflected in num.

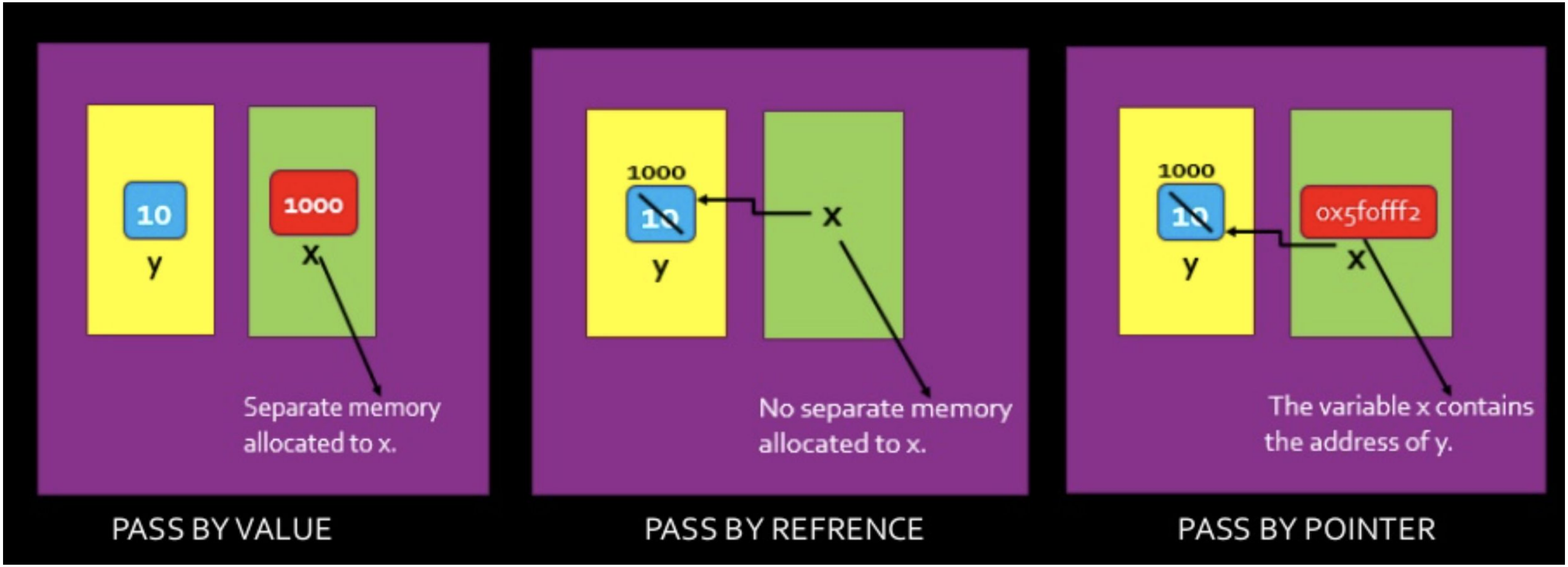
“References” in C++

- This method helps in returning more than one value from the function back to the calling program.
- When dealing with large objects reference arguments speed up a program because instead of passing an entire large object, only reference needs to be passed.

Kinds of parameters passing

PASS BY VALUE	PASS BY REFERENCE	PASS BY POINTER
Separate memory is allocated to formal parameters.	Formal and actual parameters share the same memory space.	Formal parameters contain the address of actual parameters.
Changes done in formal parameters are not reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.
For eg. **void cube(int x)** **{ x= x*x*x; }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(y); cout<<y<<endl;}** output: 1010	For eg. **void cube(int &x)** **{ x= x*x*x; }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(y); cout<<y<<endl;}** output: 101000	For eg. **void cube(int *x)** **{ *x= (*x)*(*x)*(*x); }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(&y); cout<<y<<endl;}** output: 101000

Kinds of parameters passing



Dynamic memory management errors

Some common program errors are associated with dynamic memory allocation:

- Failing to delete a pointer to dynamically allocated memory, thus preventing the memory from being returned to the free store. Failure to delete dynamically allocated memory is spoken of as a "memory leak."
- Applying a delete expression to the same memory location twice. This error can happen when two pointers address the same dynamically allocated object. If delete is applied to one of the pointers, then the object's memory is returned to the free store. If we subsequently delete the second pointer, then the free store may be corrupted.

Dynamic memory management errors

- A memory leak 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.

- That's why it's called a memory *leak*– because it's like a leaky faucet in which water is being wasted, only in this case it's computer memory.

References & more in-depth study

Pointers & references

https://www.geeksforgeeks.org/pointers-vs-references-cpp/

https://stackoverflow.com/questions/57483/what-are-the-differences-between-a-pointer-variable-and-a-reference-variable-in

Pointers & arrays

https://www.learncpp.com/cpp-tutorial/6-8-pointers-and-arrays/

References & more in-depth study

Pointers & strings

http://www.math.bas.bg/~nkirov/2005/oop/deitel/cpphtp4_05.pdf

https://www.prismnet.com/~mcmahon/Notes/strings.html

https://www.codesdope.com/cpp-string/

https://stackoverflow.com/questions/20794832/pointers-and-strings-c

Function pointers

https://en.wikipedia.org/wiki/Function_pointer

https://www.learncpp.com/cpp-tutorial/78-function-pointers/

http://www.dev-hq.net/c++/20--function-pointers