Pointers and references in C++

References

1) References allow you to create an “alias” for the variable.

```
Ex:  int x = 5;
     int &y = x;  //now x and y both can be used interchangeably (if one is changed the other is also changed)
     y=10;    //now both x and y ae equal to 10
```

A reference must be initialized at the declaration, and once created it cannot be made to reference to a different object. This is one of the main differences compared to pointers.

References can’t be NULL but pointers can be.

Due to this when implementing complex data structures (like LinkedList, Tree,..), in C++ pointers will be used.

Note:

- references are safer, easier to use compared to pointers, but have limitations.
- pointers have less limitations, but are less safe (Ex: arrays are pointers, and in C++ you may go over the max index and create undesired side effects)

Uses of references:

- Modify the values of the parameters in a function and see the changes outside
  ```
  Ex: void myFct(int &x) { x=10}
  …
  int a = 5
  myFct(a); // after this function a will be 10!!
  ```
- Send big objects as reference to avoid the expensive cloning process of such big objects. Be careful if you wish to not allow changes to the content of the object to use const (see the topic const below)
- Use a for each loop to allow changes
  ```
  Ex:     vector<int> v{1,2,3,4};
          for(int &x : v)
                  x += 5; //will add 5 to each value in the vector
  ```
- To avoid expensive copies of objects even in for each loops
  ```
  Ex:     vector<string> v{“Bob”, “John”, “Mary”, “too loooong striiing to wriiteeee”};
          for(const string &str : v)          //without reference each string will be cloned (copied)
                  count << str << endl;
  ```

Returning a reference in a function

C++ allows you to return references, but not references to a local variable!!

Ex: (see https://www.tutorialspoint.com/cplusplus/returning_values_by_reference.htm)

```
#include <iostream>
#include <ctime>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i ) {
   return vals[i];   // return a reference to the ith element
}

// main function to call above defined function.
int main () {

   cout << "Value before change" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "vals[" << i << "] = ";
      cout << vals[i] << endl;
   }

   setValues(1) = 20.23; // change 2nd element
   setValues(3) = 70.8;  // change 4th element

   cout << "Value after change" << endl;
   for ( int i = 0; i < 5; i++ ) {
      cout << "vals[" << i << "] = ";
      cout << vals[i] << endl;
   }
   return 0;
}
```

When you return a reference make sure that the object being referred to does not go out of scope (like a local variable)

```
int& func() {
   int q;
   //! return q; // Compile time error
   static int x;
   return x;     // Safe, x lives outside this scope
}
```

Pointers

Pointers are variables whose value is the address of another variable or NULL (similar to references in Java); allows you to allocate dynamically objects (on the heap versus the stack).

Advantage: the heap in general is limited only by the resources of the computer you work on, while the stack in general is limited by the compiler version you have. So bigger sizes are ok.
Disadvantage: the heap may reside on hdd, not RAM so it may be slower than the stack.

To access the address memory of any variable use & operator (Ex: int x=3; cout<<&x;).
The output will be something like: 0xbfebd5c0.

Operations we can do with pointers are:

A) To declare a pointer you do:
```
type *myPointer; //so myPointer is the name of the variable, type represents the its type
type *myPointer = NULL; //a good practice to initialize a pointer with NULL
```
B) Assign the address of a variable to a pointer
```
Ex:     int i=5;
        int *myPointer;
        myPointer=&i;
```
C) Access the value at the address variable located at the address available in the pointer variable. In order to achieve this you must use the unary operator *
```
Ex:
        int i =5;
        int *myPointer = &i;    //now myPointer and &i will display the same address
        cout<<myPointer;        //displays the address stored in the pointer (or 0 for NULL)
        if(!myPointer)          //check if the pointer is not NULL
        {
                cout<<*myPointer;       //displays the value where the pointer points at

                *myPointer = 10;        //change the value to which my pointer point

        }                               //now i will be 10
```

D) Incrementing and decrementing a pointer using ++,--,+,-:

You may move the pointer by using ++ or –- by exactly one unit of the type of the pointer. (So for example if you have a pointer myP pointing to an array of integers, then myP+1 will point to the 2nd element in the array. (care myP-1 will point before the array and crash at runtime)

So if you have an array of integers and the pointer points at the start you may traverse the array by using the pointer like this:

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;
   // let us have array address in pointer.
   ptr = var;

   for (int i = 0; i < MAX; i++) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the next location
      ptr++;
   }

   return 0;
}
```

E) Compare Pointers using ==, >, <,…

== checks if 2 pointers point to the same place.

<, > compares if one pointer is after another one in memory. Makes sense in general when you work with pointers that point to elements in an array.

F) Pointer vs Arrays

Arrays are pointers that are constant. In other words you can't change them by using ++ or –

Ex: myArray++; //invalid since you are not allowed to change myArray

*(myArray+2) = 100; //valid since I change the value in position 3, not address of the array

G) Arrays of pointers

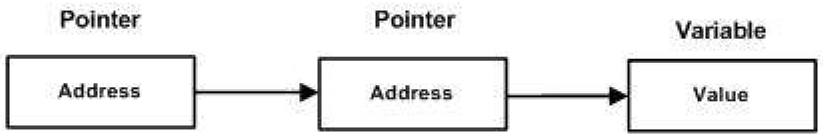
```
#include <iostream>

using namespace std;
const int MAX = 4;

int main () {
const char *names[MAX] = { "Zara Ali", "Hina Ali", "Nuha Ali", "Sara Ali" };

   for (int i = 0; i < MAX; i++) {
      cout << "Value of names[" << i << "] = ";
      cout << (names + i) << endl;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result (exact addresses may vary)–

```
Value of names[0] = 0x7ffd256683c0
Value of names[1] = 0x7ffd256683c8
Value of names[2] = 0x7ffd256683d0
Value of names[3] = 0x7ffd256683d8
```

H) Pointer to a pointer

You declare one using **.



Declare one using: int **var;

Accessing the value also require you to use **.

Ex:

```
  int myValue = 5;
  int *ptr = &myValue;
  int **pptr = &ptr; // pointer to a pointer
```

```
   cout << "Value of var :" << myValue << endl;
   cout << "Value available at ptr :" << pptr << endl;
   cout << "Value available at *ptr :" << *pptr << endl;
   cout << "Value available at **pptr :" << **pptr << endl;
```

Sample result of the code above is:

```
 Value of var :5
 Value available at ptr :0x7fff71d096d0
 Value available at *ptr :0x7fff71d096cc
 Value available at **pptr :5
```

This is useful for example to create a 2d array.

```
    int **p = new int*[10];
```

or

```
    int (*p)[10] = new int[10][10] ;
```

or

```
    int **p;

    p = new int*[10];                          //last one may make more sense but it's longer

    for(int i=0; i<10; i++) p[i] = new int[10];  //this will reserve space for each row
```

I) Constant Pointer vs Constant Pointed-to Data

From https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html#zz-2.2

5.3 Constant Pointer vs. Constant Pointed-to Data

1. Non-constant pointer to constant data: Data pointed to CANNOT be changed; but pointer CAN be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;

const int * iptr = &i1;  // non-constant pointer pointing to constant data
// *iptr = 9;   // error: assignment of read-only location
iptr = &i2;  // okay
```

2. Constant pointer to non-constant data: Data pointed to CAN be changed; but pointer CANNOT be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;

int * const iptr = &i1;  // constant pointer pointing to non-constant data
                         // constant pointer must be initialized during declaration
*iptr = 9;   // okay
// iptr = &i2;  // error: assignment of read-only variable
```

3. Constant pointer to constant data: Data pointed to CANNOT be changed; and pointer CANNOT be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;

const int * const iptr = &i1;  // constant pointer pointing to constant data
// *iptr = 9;   // error: assignment of read-only variable
// iptr = &i2;  // error: assignment of read-only variable
```

4. Non-constant pointer to non-constant data: Data pointed to CAN be changed; and pointer CAN be changed to point to another data. For example,

```
int i1 = 8, i2 = 9;

int * iptr = &i1;  // non-constant pointer pointing to non-constant data
*iptr = 9;   // okay
iptr = &i2;  // okay
```

Useful online compiler: https://www.onlinegdb.com/online_c++_compiler

Links:

https://www.geeksforgeeks.org/references-in-c/

http://www.cs.fsu.edu/~myers/cgs4406/notes/pointers.html

https://www.tutorialspoint.com/cplusplus/cpp_pointers.htm

https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html#zz-2.2