CS250 - Data Structures & Algorithms

Review C++ (Pointers & Dynamic Memory Allocation)

NUST
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Introduction

- Data Structure is a systematic way to organize data in order to use it efficiently

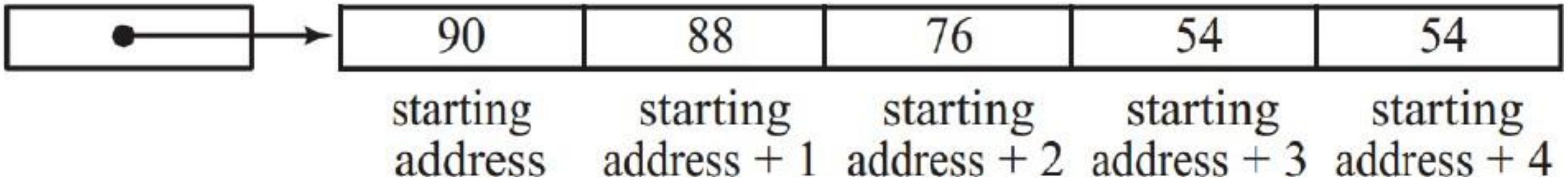
- An algorithm is an effective method for solving a problem using a finite sequence of instructions
- Data types:
  - Primitive Data Types
    - Bool, Int, float etc.
  - User-Defined Data Types (UDTs)
    - Aggregate data types e.g., structures, array-of-integers etc.
  - Abstract Data Types (ADTs)
    - Does not specify how the data type is implemented
    - In an object-oriented language such as C++, an ADT and its implementation together make up a class

Today's lecture

- Pointers
- Referencing vs pointers
- Arrays & dynamic memory allocation
- The **new** operator & memory leaks
- Concept of shalow/deep copying

Pointers

A pointer is a variable that contains the address of a variable.



Pointers

The unary operator & assigns the address of a variable to a pointer,

```
pc = &c;
```

Here, assigns the physical address of variable c to the variable pc.

Pointers

Two uses of the asterisk *:

- **Create pointer:** the declaration of the pointer p appears as

```
int* p;
```

  p is a pointer to an integer variable.

- **Access what the pointer points to:** the unary operator *p is the indirection or dereferencing operator.

```
std::cout << *p;
```

Pointers

Use of * and & unary operators.

- using variables

```
int a = 5;
int b = 10;
int c;
c = a + b;
```

- using pointers to variables

```
int* pa; pa = &a;
int* pb; pb = &b;
int* pc; pc = &c;
*pc = *pa + *pb;
```

Pointers

How to pronounced this in plain English?

```
int* p;
```

The pointer declaration int* ptr; can be vocalized backwards:

```
p;   "p is..."
*    "...a pointer to..."
int  "...an integer."
```

Pointers

Pointers are variables

- they can be used without dereferencing:

```
pb = pa;
pc = pa;
```

Pointers are constrained to point to a particular kind of object;

- every pointer points to a specific data type.

Pointer example

Implementing swap() function
Consider a function to swap values of two integer variables.

```
void swap( int x, int y ) {...}
```

Here, the parameters are passed by value,

- actual arguments remain unaffected, as the function receives copies of the original values.

More realistic solution is to use pointers:

```
void swap( int* px, int* py ) {...}
```

the parameters are passed by reference.

- the function can access and change objects within the function

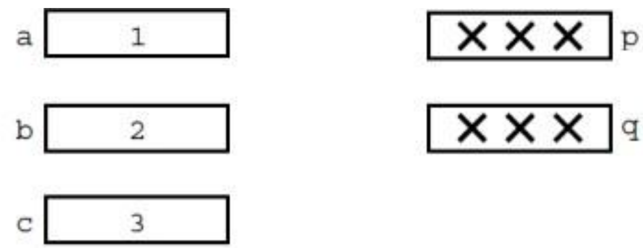
Pointer example

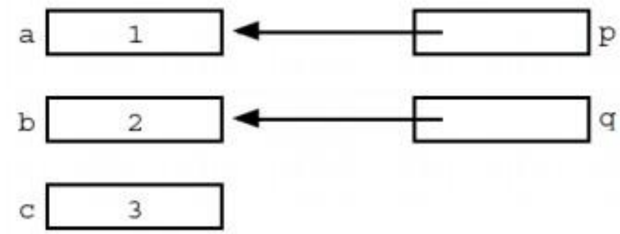
A complete example might look like this:

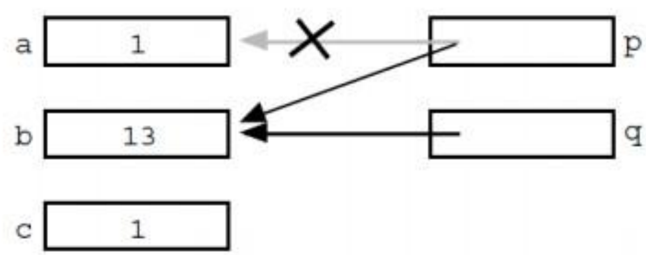
```
void swap( int* px, int* py )
{
    int temp = *px;
    *px = *py;
    *py = temp;
}

int main()
{
    int a = 5;
    int b = 10;
    swap( a, b ); // wrong! address of the variable must be taken
    swap( &a, &b ); // correct
    assert( a == 10 && b == 5 );
    return 0;
}
```

Pointer example


```
void PointerTest() {
   int a = 1;
   int b = 2;
   int c = 3;
   int* p;
   int* q;

   p = &a; // set p to refer to a
   q = &b; // set q to refer to b

   c = *p; //retrieve p's pointee value (1) and put it in c
   p = q; //change p to share with q (p's pointee is now b)
   *p = 13; // dereference p to set its pointee (b) to 13 (*q is now 13)

}
```







Difference between constant pointer, pointers to constant, and constant pointers to constants

Pointers are the variables that hold the address of some other variables, constants, or functions.

There are several ways to qualify pointers using **const**.

- Pointers to constant.
- Constant pointers.
- Constant pointers to constant.

Pointers to constant

In the pointers to constant, the data pointed by the pointer is constant and cannot be changed.

Although, the pointer itself can change and points somewhere else (as the pointer itself is a variable).

Pointers to constant cont'd

```
// C++ program to illustrate concept of the pointers to constant
int main(){
    int high =100 ;
    int low = 66 ;
    const int* score = &high;
    // Pointer variable are read from the right to left
    cout << *score << "\n"; //100
    // Score is a pointer to integer which is constant *score = 78
    // It will give you an Error: assignment of read-only location
    //'* score' because value stored in constant cannot be changed
    score = &low;
    // This can be done here as we are changing the location
    //where the score points now it points to low
    cout << *score << "\n"; //66
    return 0;
}
```

Constant pointers

In constant pointers, the pointer points to a fixed memory location, and the value at that location can be changed because it is a variable, but the pointer will always point to the same location because it is made constant here.

Below is an example to understand the constant pointers with respect to references. It can be assumed references as constant pointers which are automatically dereferenced. The value passed in the actual parameter can be changed but the reference points to the same variable.


```
int *const ptr = &a; // ptr points to a
*ptr = 56; // ok we can change value of a which is *ptr by dereferencing
ptr = & b; // Error because the value of ptr cannot be changed as it is a constant
```

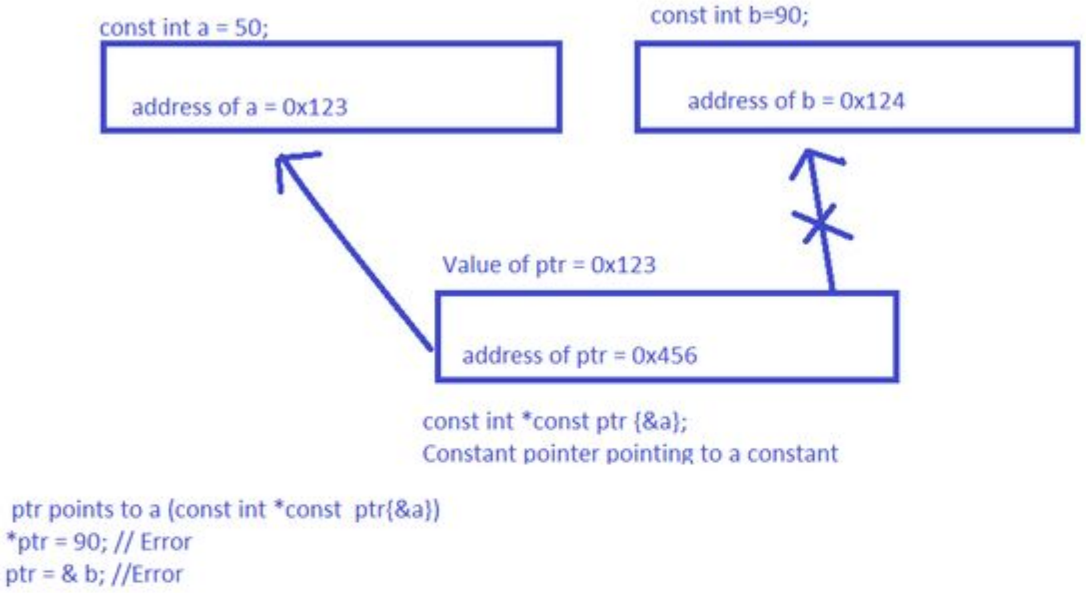
Constant pointers cont'd

Below is the program to illustrate the same:

```
int main(){

        int a = 90;
        int b = 50;
        int* const ptr = &a ;
        cout << *ptr << "\n";  //90
        cout << ptr << "\n"; //0x7ffc641845a8
        // Address what it points to
        *ptr = 56;
        // Acceptable to change the value of a
        // Error: assignment of read-only variable ‘ptr’
        // ptr = &b;
        cout << *ptr << "\n"; //56
        cout << ptr << "\n"; //0x7ffc641845a8
        return 0;
}
```

Constant Pointers to constants

In the constant pointers to constants, the data pointed to by the pointer is constant and cannot be changed. The pointer itself is constant and cannot change and point somewhere else. Below is the image to illustrate the same:



Constant Pointers to constants

Below is the program to illustrate the same:

```
int main(){
        const int a = 50 ;
        const int b = 90;
        // ptr points to a
        const int* const ptr = &a;
        // *ptr = 90;
        // Error: assignment of read-only
        // location '*(const int*)ptr'
        // ptr = &b;
        // Error: assignment of read-only
        // variable 'ptr'
        // Address of a
        cout << ptr << "\n"; //0x7ffea7e22d68
        // Value of a
        cout << *ptr << "\n"; //50
        return 0;
}
```

Differences between references & pointers

NUST SEECS

- Consider the following declarations:

  int n=5, *p = &n, &r = n;

- A reference variable must be initialized in its declaration as a reference to a particular variable, and this reference cannot be changed, meaning that a reference variable CANNOT be null
- A reference variable r can be considered a different name for a variable n, in other words it is an "alias" for n.
  - If n changes then r changes as well.
- cout << n << ' ' << *p << ' ' << r << endl;

Output: 5 5 5

```
//Use of reference variables:
void increment(int& x) {
   x++;
}
int num = 5;
increment(num);
cout << num << endl;  // Output: 6
```

Constant pointer vs pointer constant

- It is important to note the difference between the

  type int *const and the type const int *

where const int * is a type of pointer to a **constant integer**:

const int * s = &m;

after which the assignment s = &m; where m is an integer (whether constant or not) is admissible, but the assignment *s = 2; is **erroneous**, even if m is not a constant

Array names as pointers

NUST SEECS

```
#include <iostream>

void main() {
	const SIZE = 5

	int i, arr[SIZE] = {98, 87, 92, 79, 85};

	for(i=0; i<SIZE; i++)
	  cout << arr[i] << *(arr + i) << endl;
}
```

String Literals

Consider,

```
std::cout << "Hello";
```

"Hello" is stored in memory as a sequence of bytes, terminated with the null character ′\0′.

```
H  e  l  l  o  \0
```

Pointer to store the address of the first byte, or first character ′H′.

```
char *pchar = "Hello";
```

Pointers & Arrays

Pointers and arrays are closely related.

```
char amessage[] = "hello"; // array
char* pmessage = "hello";  // pointer
```

Construction of the arrays of pointers is also allowed.

- use of arrays of pointers is to store character strings of variable lengths.

```
char* months[] = { "Illegal month", "Jan", "Feb", "Mar" };
```

Pointers: function arguments

Pointers as function arguments

```
void print( char* message )
{
   std::cout << message;
}

int main()
{
   print( "Hello" );
   return 0;
}
```

Pointers: function arguments

Another example:

```
int main()
{
   char* pmessage;
   pmessage = "Hello";
   print( pmessage );
   return 0;
}
```

Content

- Pointers
- Referencing vs pointers
- Arrays & dynamic memory allocation
- The **new** operator & memory leaks

Dynamic memory allocation

- Arrays are useful, however we must know in advance about the amount of memory required
- In many situations, we do not know exact size required until runtime
- Reserving maximum wastes memory
- Here comes the concept of **dynamic memory allocation**

Dynamic memory allocation

- To avoid wasting memory, array allocation or deallocation can take place at run time

- To allocate memory, we need to use the **new** operator
  - Reserves the number of bytes requested by the declaration
  - Returns the address of the **first reserved location** or NULL if sufficient memory is not available

- To deallocate memory (which has previously been allocated using the **new** operator) we need to use the **delete** operator
  - Releases a block of bytes previously reserved. The address of the first reserved location is passed as an argument to **delete**.

The **new** operator

- **new** keyword obtains memory from OS and returns a pointer to starting location

```
int x = 10;
int *ptr;

ptr = new int;
*ptr = x;
cout<<*ptr;
```

Memory leaks – delete operator

NUST SEECS

- If your program reserves many chunks of memory using **new**, eventually all the available memory will be reserved and system will crash
- To ensure safe and efficient use of memory, **new** is matched by a corresponding **delete**
- If the program terminates the memory is released automatically, however, if a function allocates memory using **new** and doesn't release it then the pointer is destroyed but not the memory which causes waste of memory
- E.g., consider two lines of code

```
p = new int;

p = new int;
```

where after allocating one cell for an integer, the same pointer p is used to allocate another cell

Correct should be

```
p = new int;
delete p;
p = new int;
```

Case of 1D array

```
cout << "Enter array size: ";
cin >> SIZE;

int *arr;
arr = new int[SIZE]; // allocation

delete [] arr; // deallocation
```