Part 10. Pointers & Dynamic Data Structures
Pointers & Dynamic Data Structures
Pointers & Dynamic Data Structures

- Static Data Structures
- The Address of a Variable: Pointers
- Dereferencing
- Pointers to Arrays
- Dynamic Data Structures
- Run-Time 1-D Arrays
- Run-Time 2-D Arrays
1. Static Data Structures

- The **usual variables** we declare in the program are **static** (we cannot get rid of them or change their size). For example:
  ```
  int k;  float x = 2.15;
  ```

- **Arrays** and **Structs** are also **static**, so once declared, we cannot erase them from memory or change their size. For example:
  ```
  float x [20];  int a[3] = { 2 , 2 , 4 };
  ```
Static Data Structures

- Static data are allocated memory at **Compile Time**, i.e. before the program is executed.
- Static data are allocated their memory space in a place called the **Data Segment**.
- Static data cannot change size during **Run Time**, i.e. while the program is running.
Where in Memory?

- The Memory Map:
  - one segment = 64 kbyte

- OS = Operating System Area
- CS = Code Segment (Main & Functions code)
- DS = Data Segment (Static Data)
- SS = Stack Segment (System Stack)
- Heap = Rest of Memory (for Dynamic Data)

LM

HM
2. The Address of a Variable: Pointers

- The & symbol is called the address operator.
- The purpose of & is to return the address of a variable in memory. For example, if x is a variable then &x is its address in memory.
- We can store the address of a variable in a special variable called a **pointer**.
- A **Pointer** is a variable whose value is a memory address of an item, *not its value*.
- A pointer knows about the type of the item it points to.
- All pointers have fixed size (typically 4 bytes).
Pointers

- A pointer variable must be declared before it is used. It must be bound to the same type as the variable it will point to.
- The asterisk operator * must precede each pointer name in the declaration.

```cpp
<vtype> variableName;
<pctype> * pointerName;

pointerName = & variableName;
```
Q: Why is it important to declare the type of the variable that a pointer points to?

A: For an operation like “p++” where “p” is a pointer variable, the compiler needs to know the data type of the variable “p” points to. If “p” is a character pointer then “p++” will increment “p” by one byte, if “p” were an integer pointer its value on “p++” would be incremented by 2 bytes.
Pointers

For Example:

```c
double x = 3.14;  // a variable of type double
double *p;       // a pointer to double
p = &x;          // p now stores the address of x
```

Starts at location 0012FF78
Pointers

- Pointers can only contain addresses.
  
  e.g. if we declare:
  
  ```c
  float *p, *q;
  ```
  
  then the following are **errors:**
  
  - `p = 2.55;`
  - `q = 15.5;`
The ++ and -- operators may be used to increment or decrement a pointer variable.

An integer may be added to or subtracted from a pointer variable.

A pointer may be subtracted from another pointer.
3. Dereferencing (Indirection)

(*) is the dereferencing (or indirection) operator. It can be used to access the value stored in a location.

For example, if (p) is a pointer, the value of (*p) is not the address stored in p but is instead the value stored in memory at that address (i.e. 3.14)
Indirection Operator

An asterisk has two uses with regard to pointers

- In a definition, it indicates that the object is a pointer
  \[ \text{char } *s; \quad \text{// } s \text{ is of type pointer to char} \]

- In expressions, when applied to a pointer it evaluates to the object to which the pointer points (indirection or dereferencing)
  \[ \text{int } k = 1; \]
  \[ \text{int } *p = \&k; \quad \text{// } p \text{ points to } k \]
  \[ *p = 2; \]
  \[ \text{cout } \ll k \ll \text{ endl; } \quad \text{// display a 2} \]
Example Program

```cpp
int main()
{
    double x = 3.14;   double *p;
    int k = 5; int *q;
    p = &x; q = &k;
    cout << "Address of x is " << p << " Value of x = " << *p << endl;
    cout << "Address of k is " << q << " Value of k = " << *q << endl;
    return 0;
}

Output:
Address of x is 0012FF78  Value of x = 3.14
Address of k is 0012FF70  Value of k = 5
Pointers as function parameters: Swap Function

```cpp
void Pswap ( int *p, int *q)
{
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main ()
{
    int a = 5;    int b = 7;
    Pswap (& a, & b);
    cout << a << ' ' << b << ' ';
    return 0;
}
```
A pointer can be made to point to another pointer.

Example:

```cpp
int c = 23;
int *p;
int **q;

p = &a;  q = &p;

cout << a << ' ' << *p << ' ' << **q << ' ';
```

Output: 23 23 23
4. Pointers to Arrays

- C++ regards the name of the array as the address (pointer) of the first element in the array
- Example:

```c++
int a[3] = {12, 55, 93};
int *r = a;
cout << *r;  // equivalent to a[0], gives 12
cout << *r+1;  // equivalent to a[0]+1, gives 13
cout << *(r+1);  // equivalent to a[1], gives 55
cout << *(r+2) + 7;  // equivalent to a[2]+7, gives 100
cout << *(++r);  // now r points to a[1], gives 55
r--;  // brings back r to point to a[0]
```
// This program uses a pointer to display the contents of an integer array.
#include <iostream.h>

void main(void)
{
    int set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
    int *nums, index;
    nums = set;
    cout << "The numbers in set are:\n";
    for (index = 0; index < 8; index++)
    {
        cout << *nums << " ";
        nums++;
    }
}
5. Dynamic Data Structures

- The Heap (free memory)
- A Dynamic Data Structure is allocated memory at run-time. Consists of nodes to store data and pointers to these nodes to access the data.
- Nodes are created (allocated) and destroyed (de-allocated) at run-time.
- Using *dynamic allocation* allows your programs to create data structures with sizes that can be defined while the program is running and to expand the sizes when needed.
Nodes & Pointers

- A node is an anonymous variable (has no name)
- No name is needed because there is always a pointer pointing to the node.
Creating Nodes: the "new" Operator

- The `new` operator allocates memory from the heap to a node of specified type at Run Time. It returns the address of that node.
- The statements:
  ```
  int *p  ;
  ...................
  p = new int;
  ```

create a new node of type `int` and let a pointer `p` point to it. **No data is put into the node**
- The node created has no name, it is called an **Anonymous Variable**. It can only be accessed via its pointer using the indirection operator, i.e. by using `(*p)`
Accessing Data with Pointers

- indirection operator

\[ *p = 15.5; \]

\[ // \text{*p reads as: contents of node pointed to by } p \]

Stores floating value 15.5 in the node pointed to by p

![Diagram of a pointer and its indirection](image)
Example:

```cpp
float *p;
p = new float;
*p = 15.5;
cout << "The contents of the node pointed to by p is " << *p << endl;
```

Output

```
The contents of the node pointed to by p is 15.5
```
## Pointer Operations

**Assignment:** Only if the two pointers are bound to the same type

\[
*p = 3.3; \\
*q = 1.5; \\
q = p;
\]

**Comparison:** only when both are bound to the same type

Only equality or inequality, e.g. \( p == q \) or \( p != q \)

if \( (p == q) \) .....
**Pointer Operations**

- **Copy contents of one node into another node**
  
  ```
  *p = 3.3;  *q = 1.5;
  *q = *p ;
  ```

- **The Null Pointer:**
  
  ```
  p = NULL;  // Now p points to nothing
  ```

  e.g. ```if ( p == NULL ) .... ```
Returning Nodes to the Heap

- Operation:
  - `delete <pointer variable>;`
  - Returns space of node pointed to by pointer back to heap for re-use
- When finished with a node delete it
- Pointer is not destroyed but undefined:
- Example: `delete p;`
6. Run-Time 1-D Arrays

**Drawbacks of static arrays:**

- Capacity is fixed at *compile time*
- If size > number of elements, memory is wasted
- If size < number of elements, we suffer array overflow

**Solution: Dynamic (Run-Time) Arrays:**

- Capacity specified *during program execution*.
- Acquire additional memory as needed.
- Release memory locations when they are not needed.
The operator `new` can be used in an expression of the form:

```
new <Type> [n]
```

`n` is an integer expression (could be a variable). This allocates a 1-D array with `n` elements, each of type `<Type>`; it returns the base address of that array.

The address returned by `new` must be assigned to a pointer of type `Type`. 
Example

```cpp
int n;

cout << "Enter size of array: ";
cin >> n;    // size is entered at run-time

if (n > 0)
{
    int *A = new int [n]; // A is now the base address
    // process A
    for (int i = 0; i < n; i++) cin >> A[i];

    ...
}
```
Run-Time 1-D Arrays

Because run-time arrays can take a lot of memory from the heap, we must de-allocate that space after we finish with it.

To return memory allocated to array pointed to by $A$, use the `delete` operator in the form:

```
delete [] A;
```
Example

```cpp
int n;
cout << "Enter size of array: ";
cin >> n; // size is entered at run-time
if (n > 0)
{
    int *A = new int [n]; // A is now the base address
    // process A
    for (int i = 0; i < n; i++) cin >> A[i];
    ........
delete [] A; // Release memory locations
}
```
7. Run-Time 2-D Arrays

- Run-Time arrays are always 1-D arrays with a pointer giving its base address.
- Physically, 2-D arrays are 1-D arrays of arrays.
- We can work with a “virtual” 2-D array that is physically 1-D array by “mapping” elements at a given row and column to the 1-D array.
Suppose we want to process a 2-D dynamic array of integers with $N$ rows and $M$ columns (size is $N*M$).

Dynamic allocation:

```cpp
int *B = new int[N*M];
```

The element at row (i) and column (j) is physically the element $B[k]$ where

$$k = j + i * M; \text{ with } (i = 0...N-1 \text{ and } j = 0...M-1)$$
Run-Time 2-D Arrays

- Example: a 3x4 array

\[ N = 3; \ M = 4; \ \text{int} \ *B = \text{new int}[N*M]; \]

\[ \text{for} \ (i = 0; \ i < N; \ i++) \]

\[ \text{for} \ (j = 0; \ j < M; \ j++) \ { \]

\[ k = j + i * M; \ \text{cout} << k << " "; \]

\[ \}

Output is the index in the physical 1-D array:

0 1 2 3 4 5 6 7 8 9 10 11
Run-Time 2-D Arrays

- **Inverse mapping:**
  
  To convert from an index (k) in the physical 1-D array to a row (i) and a column (j) in the virtual 2-D array:

  $$k = 0,1,\ldots, N\times M - 1$$

  $$i = k / M; \quad (a\ number\ 0\ldots N - 1)$$

  $$j = k \% M; \quad (a\ number\ 0\ldots M - 1)$$

  For example with $N = 3$ and $M = 4$, $k = 6$ gives

  $$i = 1\ and\ j = 2$$
It is possible to create PHYSICAL 2-D run-time arrays using an array of pointers to 1-d arrays. The following function receives the number of rows \( N \) and the number of columns \( M \) and returns a pointer to a 2-D array of integers of size \( N \times M \)

```cpp
int ** Matrix (int N, int M) {
    int ** m;    m = new int * [N];
    for (int i = 0; i < N; i++)  m[i] = new int [M];
    return m;
}
```
The function can now be used in a main program as follows:

```cpp
int main ()
{
    int N = 20;  int M = 30;
    int ** B;
    B = Matrix (N , M);
    // B can now be used as a 2-D array
    // B[i][j] is the element at row i and column j
```
Physical Run-Time 2-D Arrays

To delete this array, each row must be deleted individually before deleting the B pointer:

```cpp
for (int i = 0; i < N; i++)
    delete [] B[i]; // deleting row i

delete [] B;
```