DATA STRUCTURE

Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. Data structures provide a means to manage huge amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Examples of Data Structure

• An array data structure stores a number of elements of the same type in a specific order. They are accessed using an integer to specify which element is required (although the elements may be of almost any type). Arrays may be fixed-length or expandable.

• Record (also called tuple or struct) Records are among the simplest data structures. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called *fields* or *members*.

• A hash or dictionary or map is a more flexible variation on a record, in which name-value pairs can be added and deleted freely.

• Union. A union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g. "float or long integer". Contrast with a record, which could be defined to contain a float *and* an integer; whereas, in a union, there is only one value at a time.

• A tagged union (also called a variant, variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.

• A set is an abstract data structure that can store specific values, without any particular order, and no repeated values. Values themselves are not retrieved from sets, rather one tests a value for membership to obtain a Boolean "in" or "not in".

• An object contains a number of data fields, like a record, and also a number of program code fragments for accessing or modifying them. Data structures not containing code, like those above, are called plain old data structure.

Many others are possible, but they tend to be further variations and compounds of the above.

Basic principles

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by an address—a bit string that can be itself stored in memory and manipulated by the program. Thus the record and array data structures are based on computing the addresses of data items with arithmetic operations; while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking)

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

Language support

Most assembly languages and some low-level languages, such as BCPL(Basic Combined Programming Language), lack support for data structures. Many high-level programming languages, and some higher-level assembly languages, such as MASM, on the other hand, have special syntax or other built-in support for certain data structures, such as vectors (one-dimensional arrays) in the C language or multi-dimensional arrays in Pascal.

Most programming languages feature some sorts of library mechanism that allows data structure implementations to be reused by different programs. Modern languages usually come with standard libraries that implement the most common data structures. Examples are the C++ Standard Template Library, the Java Collections Framework, and Microsoft's .NET Framework.

Modern languages also generally support modular programming, the separation between the interface of a library module and its implementation. Some provide opaque data types that allow clients to hide implementation details. Object-oriented programming languages, such as C++, Java and .NET Framework may use classes for this purpose. Many known data structures have concurrent versions that allow multiple computing threads to access the data structure simultaneously.

What is Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

Analysis of algorithms

In computer science, the **analysis of algorithms** is the determination of the number of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional to the logarithm of the length of the list being searched, or in O(log(n)), colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a *hidden constant*.

Exact (not asymptotic) measures of efficiency can sometimes be computed but they usually require certain assumptions concerning the particular implementation of the algorithm, called model of computation. A model of computation may be defined in terms of an abstract computer, e.g., Turing machine, and/or by postulating that certain operations are executed in unit time. For example, if the sorted list to which we apply binary search has *n* elements, and we can guarantee that each lookup of an element in the list can be done in unit time, then at most log2*n* + 1 time units are needed to return an answer.

Cost models

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant.

One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

Two cost models are generally used:

• the **uniform cost model**, also called **uniform-cost measurement** (and similar variations), assigns a constant cost to every machine operation, regardless of the size of the numbers involved

• the **logarithmic cost model**, also called **logarithmic-cost measurement** (and variations thereof), assigns a cost to every machine operation proportional to the number of bits involved

The latter is more cumbersome to use, so it's only employed when necessary, for example in the analysis of arbitrary-precision arithmetic algorithms, like those used in cryptography.

A key point which is often overlooked is that published lower bounds for problems are often given for a model of computation that is more restricted than the set of operations that you could use in practice and therefore there are algorithms that are faster than what would naively be thought possible.

Orders of growth

Informally, an algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size n, the function $f(n)$ times a positive constant provides an upper bound or limit for the run-time of that algorithm. In other words, for a given input size n greater than some $n0$ and a constant c, the running time of that algorithm will never be larger than $c \times f(n)$. This concept is frequently expressed using Big O notation. For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order $O(n^2)$.

Big O notation is a convenient way to express the worst-case scenario for a given algorithm, although it can also be used to express the average-case — for example, the worst-case scenario for quicksort is $O(n^2)$, but the average-case run-time is $O(n \log n)$.

Efficiency of an algorithm

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space :** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space :** Its the space required to store all the constants and variables value.
- **Environment Space :** Its the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion.

Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
 statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
 for(j=0; j < N;j++)
 {
  statement;
 }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
  mid = (low + high) / 2;
  if (target < list[mid])
    high = mid - 1;
  else if (target > list[mid])
    low = mid + 1;
  else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field. Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
  int pivot = partition(list, left, right);
  quicksort(list, left, pivot - 1);
  quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort. Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(**

N). The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

- **O(expression)** is the set of functions that grow slower than or at the same rate as expression.
- **Omega(expression)** is the set of functions that grow faster than or at the same rate as expression.
- **Theta(expression)** consist of all the functions that lie in both O(expression) and Omega(expression).

Suppose you've calculated that an algorithm takes f(n) operations, where,

*f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n*

Since this polynomial grows at the same rate as **n^2**, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as **f(n)** grows by a factor of **n^2**, the time complexity can be best represented as **Theta(n^2)**.

How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we will mostly talk about time complexity (CPU usage).

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

The time required by a function/procedure is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, *).
- one assignment (e.g. x := 0)
- one test (e.g., x = 0)
- one read (of a primitive type: integer, float, character, boolean)
- one write (of a primitive type: integer, float, character, boolean)

Some functions/procedures perform the same number of operations every time they are called. For example, StackSize in the Stack implementation always returns the number of elements currently in the stack or states that the stack is empty, then we say that StackSize takes ***constant time***.

Other functions/ procedures may perform different numbers of operations, depending on the value of a parameter. For example, in the BubbleSort algorithm, the number of elements in the array, determines the number of operations performed by the algorithm. This parameter (number of elements) is called the ***problem size/ input size***.

When we are trying to find the complexity of the function/ procedure/ algorithm/ program, we are ***not*** interested in the ***exact*** number of operations that are being performed. Instead, we are interested in the relation of the ***number of operations*** to the ***problem size***.

Typically, we are usually interested in the ***worst case***: what is the ***maximum*** number of operations that might be performed for a given problem size. For example, inserting an element into an array, we have to move the current element and all of the elements that come after it one place to the right in the array. In the worst case, inserting at the beginning of the array, ***all*** of the elements in the array must be moved. Therefore, in the worst case, the time for insertion is proportional to the number of elements in the array, and we say that the worst-case time for the insertion operation is ***linear*** in the number of elements in the array. For a linear-time algorithm, if the problem size doubles, the number of operations also doubles.

Big-O notation

We express complexity using **big-O notation**.

For a problem of size N:

- a constant-time algorithm is "order 1": O(1)
- a linear-time algorithm is "order N": O(N)
- a quadratic-time algorithm is "order N squared": O(N^2)

Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time algorithm will be faster than a linear-time algorithm, which will be faster than a quadratic-time algorithm).

Formal definition:

A function T(N) is O(F(N)) if for some constant c and for values of N greater than some value n_0:

$$T(N) <= c * F(N)$$

The idea is that T(N) is the ***exact*** complexity of a procedure/function/algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e., the actual time/space or whatever for a problem of size N will be no worse than F(N)).

In practice, we want the smallest F(N) -- the ***least*** upper bound on the actual complexity. For example, consider:

$$T(N) = 3 * N^2 + 5.$$

We can show that T(N) is $O(N^2)$ by choosing $c = 4$ and $n_0 = 2$.

This is because for all values of N greater than 2:

$$3 * N^2 + 5 <= 4 * N^2$$

T(N) is **not** O(N), because whatever constant c and value n_0 you choose, There is always a value of $N > n_0$ such that $(3 * N^2 + 5) > (c * N)$

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).

If-Then-Else

```
if (cond) then
   block 1 (sequence of statements)
else
   block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

max(time(block 1), time(block 2))

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

Loops

```
for I in 1 .. N loop
   sequence of statements
end loop;
```

The loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Nested loops

```
for I in 1 .. N loop
   for J in 1 .. M loop
      sequence of statements
   end loop;
end loop;
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).

In a common special case where the stopping condition of the inner loop is $J < N$ instead of $J < M$ (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Statements with function/ procedure calls

When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/ procedure *f* takes constant time, and that function/procedure *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

f(k) has O(1)
g(k) has O(k)

When a loop is involved, the same rule applies. For example:

```
for J in 1 .. N loop
    g(J);
end loop;
```

has complexity (N^2). The loop executes N times and each function/procedure call g(N) is complexity O(N).

Dynamic memory allocation with new and delete

Memory in your C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use**delete** operator, which de-allocates memory previously allocated by new operator.

The need for dynamic memory allocation

C++ supports three basic types of memory allocation,

- **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
- **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
- **Dynamic memory allocation**

Both static and automatic allocation have two things in common:

- The size of the variable / array must be known at compile time.
- Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

Most of the time, this is just fine. However, you will come across situations where one or both of these constraints cause problems, usually when dealing with external (user or file) input.

For example, we may want to use a string to hold someone’s name, but we do not know how long their name is until they enter it. Or we may want to read in a number of records from disk, but we don’t know in advance how many records there are. Or we may be creating a game, with a variable number of monsters (that changes over time as some monsters die and new ones are spawned) trying to kill the player.

If we have to declare the size of everything at compile time, the best we can do is try to make a guess the maximum size of variables we’ll need and hope that’s enough:

char name[25]; // let's hope their name is less than 25 chars!

Record record[500]; // let's hope there are less than 500 records!

Monster monster[40]; // 40 monsters maximum

Polygon rendering[30000]; // this 3d rendering better not have more than 30,000 polygons!

This is a poor solution for at least four reasons:

First, it leads to wasted memory if the variables aren’t actually used. For example, if we allocate 25 chars for every name, but names on average are only 12 chars long, we’re using over twice what we really need. Or consider the rendering array above: if a rendering only uses 10,000 polygons, we have 20,000 Polygons worth of memory not being used!

Second, how do we tell which bits of memory are actually used? For strings, it’s easy: a string that starts with a \0 is clearly not being used. But what about monster[24]? Is it alive or dead right now? That necessitates having some way to tell active from inactive items, which adds complexity and can use up additional memory.

Third, most normal variables (including fixed arrays) are allocated in a portion of memory called the **stack**. The amount of stack memory for a program is generally quite small -- Visual Studio defaults the stack size to 1MB. If you exceed this number, stack overflow will result, and the operating system will probably close down the program.

On Visual Studio, you can see this happen when running this program:

```
int main()
{
   int array[1000000]; // allocate 1 million integers (probably 4MB of memory)
}
```

Being limited to just 1MB of memory would be problematic for many programs, especially those that deal with graphics.

Fourth, and most importantly, it can lead to artificial limitations and/or array overflows. What happens when the user tries to read in 600 records from disk, but we've only allocated memory for a maximum of 500 records? Either we have to give the user an error, only read the 500 records, or (in the worst case where we don't handle this case at all) overflow the record array and watching something bad happen.

Fortunately, these problems are easily addressed via dynamic memory allocation. **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed. This memory does not come from the program's limited stack memory -- instead, it is allocated from a much larger pool of memory managed by the operating system called the **heap**. On modern machines, the heap can be gigabytes in size.

Dynamically allocating single variables

To allocate a *single* variable dynamically, we use the scalar (non-array) form of the **new** operator:

*int *ptr = new int; // dynamically allocate an integer and assign the address to ptr*

In the above case, we're requesting an integer's worth of memory from the operating system. The new operator returns the *address* of the variable that has been allocated. Because we want to remember this address so we can use it later, it's common to store this address in a pointer. We can then dereference the pointer to access the memory:

*int *ptr = new int; // dynamically allocate an integer and assign the address to ptr*

**ptr = 7; // assign value of 7 to allocated memory*

If it wasn't before, it should now be clear at least one case in which pointers are useful. Without a pointer to hold the memory address of the memory that was just allocated, we'd have no way to access the memory that was just allocated for us!

How does dynamic memory allocation work?

Your computer has memory (probably lots of it) that is available for applications to use. When you run an application, your operating system loads the application into some of that memory. Your application reserves some additional memory for normal operations (keeping track of which functions were called, creating and destroying global and local variables, etc…). However, much of the memory just sits there, waiting to be handed out to programs that request it.

When you dynamically allocate memory, you're asking the operating system to reserve some memory for your program's use. If it can fulfill this request, it will return the address of that memory to your application. From that point forward, your application can use this memory as it wishes. When your application is done with the memory, it can return the memory back to the operating system to be given to another program.

Unlike static or automatic memory, the program itself is responsible for requesting and disposing of dynamically allocated memory.

Deleting single variables

When we are done with a dynamically allocated variable, we need to explicitly tell C++ to free the memory for reuse. For single variables, this is done via the scalar (non-array) form of the **delete** operator:

// assume ptr has previously been allocated with operator new

delete ptr; // return the memory pointed to by ptr to the operating system

ptr = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)

What does it mean to delete memory?

The delete operator does not *actually* delete anything. It simply returns the memory being pointed to back to the operating system. The operating system is then free to reassign that memory to another application (or to this application again later).

Although it looks like we're deleting a *variable*, this is not the case! The pointer variable still has the same scope as before, and can be assigned a new value just like any other variable.

Note that deleting a pointer that is not pointing to dynamically allocated memory may cause bad things to happen.

Dangling pointers

C++ does not make any guarantees about what will happen to the contents of deallocated memory, or to the value of the pointer being deleted. In most cases, the memory returned to the operating system will contain the same values it had before it was returned, and the pointer will be left pointing to the now deallocated memory.

Pointers that are pointing to deallocated memory are called **dangling pointer**. Dereferencing or deleting a dangling pointer will lead to undefined behavior.

```
#include <iostream>
int main()
{
    int *ptr = new int; // dynamically allocate an integer
    *ptr = 7; // put a value in that memory location
    delete ptr; // return the memory to the operating system.  ptr is now a dangling
pointer.
    std::cout << *ptr; // Dereferencing a dangling pointer will cause undefined
behavior
    delete ptr; // trying to deallocate the memory again will also lead to undefined
behavior.
    return 0;
}
```

In the above program, the value of 7 that was previously assigned to the allocated memory will probably still be there, but it's possible that the value at that memory address could have changed. It's also possible the memory could be allocated to another application (or for the operating system's own usage), and trying to access that memory will cause the operating system to shut the program down.

Deallocating memory may create multiple dangling pointers. Consider the following example:

```
#include <iostream>
int main()
{
    int *ptr = new int; // dynamically allocate an integer
    int *otherPtr = ptr; // otherPtr is now pointed at that same memory location
```

delete ptr; // return the memory to the operating system. ptr and otherPtr are now dangling pointers.

ptr = 0; // ptr is now a nullptr

// however, otherPtr is still a dangling pointer!

return 0;

}

Rule: To avoid dangling pointers, after deleting memory, set all pointers pointing to the deleted memory to 0 (or nullptr in C++11).

Operator new can fail

When requesting memory from the operating system, in rare circumstances, the operating system may not have any memory to grant the request with.

By default, if new fails, a *bad_alloc* exception is thrown. If this exception isn't properly handled (and it won't be, since we haven't covered exceptions or exception handling yet), the program will simply terminate (crash) with an unhandled exception error.

In many cases, having new throw an exception (or having your program crash) is undesirable, so there's an alternate form of new that can be used instead to tell new to return a null pointer if memory can't be allocated. This is done by adding the constant std::nothrow between the new keyword and the allocation type:

*int *value = new (std::nothrow) int; // value will be set to a null pointer if the integer allocation fails*

In the above example, if new fails to allocate memory, it will return a null pointer instead of the address of the allocated memory.

Note that if you then attempt to dereference this memory, your program will crash. Consequently, the best practice is to check all memory requests to ensure they actually succeeded before using the allocated memory.

*int *value = new (std::nothrow) int; // ask for an integer's worth of memory*

```
if (!value) // handle case where new returned null
{
    std::cout << "Could not allocate memory";
    exit(1);
}
```

Because asking new for memory only fails rarely (and almost never in a dev environment), it’s common to forget to do this check!

Null pointers and dynamic memory allocation

Null pointers (pointers set to address 0 or nullptr) are particularly useful when dealing with dynamic memory allocation. In the context of dynamic memory allocation, a null pointer basically says “no memory has been allocated to this pointer”. This allows us to do things like conditionally allocate memory:

```
// If ptr isn't already allocated, allocate it
if (!ptr)
    ptr = new int;
```

Deleting a null pointer has no effect. Thus, there is no need for the following:

```
if (ptr)
    delete ptr;
```

Instead, you can just write:

```
delete ptr;
```

If ptr is non-null, the dynamically allocated variable will be deleted. If it is null, nothing will happen.

Memory leaks

Dynamically allocated memory effectively has no scope. That is, it stays allocated until it is explicitly deallocated or until the program ends (and the operating system cleans it up, assuming your operating system does that). However, the pointers used to hold dynamically allocated memory addresses follow the scoping rules of normal variables. This mismatch can create interesting problems.

Consider the following function:

```
void doSomething()
{
   int *ptr = new int;
}
```

This function allocates an integer dynamically, but never frees it using delete. Because pointers follow all of the same rules as normal variables, when the function ends, ptr will go out of scope. Because ptr is the only variable holding the address of the dynamically allocated integer, when ptr is destroyed there are no more references to the dynamically allocated memory. This is called a **memory leak**. As a result, the dynamically allocated integer can not be deleted, and thus can not be reallocated or reused while the program is running. Memory leaks eat up free memory while the program is running, making less memory available not only to this program, but to other programs as well. Programs with severe memory leak problems can eat all the available memory, causing the entire machine to run slowly or even crash.

Memory leaks can also result if the pointer holding the address of the dynamically allocated memory is reassigned to another value:

```
int value = 5;
int *ptr = new int; // allocate memory
ptr = &value; // old address lost, memory leak results
```

It is also possible to get a memory leak via double-allocation:

```
int *ptr = new int;
ptr = new int; // old address lost, memory leak results
```

The address returned from the second allocation overwrites the address of the first allocation. Consequently, the first allocation becomes a memory leak!

new and delete

Arrays can be used to store multiple homogenous data but there are serious drawbacks of using arrays. Programmer should allocate the memory of an array when they declare it but most of time, the exact memory needed cannot be determined until runtime. The best thing to do in this situation is to declare the array with maximum possible memory required (declare array with maximum possible size expected) but this wastes memory.

To avoid wastage of memory, you can dynamically allocate the memory required during runtime using new and delete operator.

C++ Memory Management

C++ Program to store GPA of n number of students and display it where n is the number of students entered by user.

```
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    int n;
    cout << "Enter total number of students: ";
    cin >> n;
    float* ptr;
    ptr = new float[n];       // memory allocation for n number of floats
    cout << "Enter GPA of students." <<endl;
    for (int i = 0; i < n; ++i) {
        cout << "Student" << i+1 << ": ";
        cin >> *(ptr + i);
    }
    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < n; ++i) {
        cout << "Student" << i+1 << " :" << *(ptr + i) << endl;
    }
    delete [] ptr;    // ptr memory is released
    return 0;
}
```

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9
Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
Student4 :2.9
```

In this program, only memory required to store n(entered by user) number of floating-point data is declared dynamically.

The new Operator

ptr = new float[n];

This expression in the above program returns a pointer to a section of memory just large enough to hold the n number of floating-point data.

The delete Operator

Once the memory is allocated using new operator, it should released to the operating system. If the program uses large amount of memory using new, system may crash because there will be no memory available for operating system. The following expression returns memory to the operating system.

delete [] ptr;

The brackets [] indicates that, array is deleted. If you need to delete a single object then, you don't need to use brackets.

delete ptr;

Example 2: C++ Memory Management

Object-oriented approach to handle above program in C++.

```
#include <iostream>

using namespace std;

class Test {

private:

   int n;

   float *ptr;

public:

   Test() {

      cout << "Enter total number of students: ";

      cin >> n;

      ptr = new float[n];

      cout << "Enter GPA of students." <<endl;

      for (int i = 0; i < n; ++i) {

         cout << "Student" << i+1 << ": ";

         cin >> *(ptr + i);

      }
```

```
    }
    ~Test() {
        delete[] ptr;
    }
    void Display() {
        cout << "\nDisplaying GPA of students." << endl;
        for (int i = 0; i < n; ++i) {
            cout << "Student" << i+1 << " :" << *(ptr + i) << endl;
        }
    }
};
int main() {
    Test s;
    s.Display();
    return 0;
}
```

The output of this program is same as above program. When the object s is created, the constructor is called which allocates the memory for n floating-point data.

When the object is destroyed, that is, object goes out of scope then, destructor is automatically called.

```
    ~Test() {
        delete[] ptr;
    }
```

This destructor executes delete[] ptr; and returns memory to the operating system.