Templating:

Form of abstraction. It is a way to have more generically programmed functions.

\*Key idea: It enables you to parameterize the data type.

We define a set of overloaded functions, which are functions different from the usual functions in that they all share the same name.

\*Need distinct signatures: or will not compile

These two functions have the same signature:

int Divide (int n, int m) ;

double Divide (int n, int m) ;

template <class T1, class T2> T2 FindMaxIndex(const T1 data[], T2 size)

#include "sequence.template" //include implementation, inside header file

We want to parameterize the data type of size instead of using straight int because some compilers don't know how to convert from size\_t to int

Template Class

The concept of data type parameterization has also been to class extended to apply to **class**

When we have the situation where there is a need for

a related group of **classes** that differ in their construction only in some or all of the component data types

Can define one generic **class template** instead of defining several specific **classes**

Similar to concept of STL: class templates + algorithms + iterators for various containers

Include template in header file

template

template <class T>

OrderedPair<T>::OrderedPair(T xx, T yy) : x(xx), y(yy) { }

template <class T>

void OrderedPair<T>::SetPair(T xx, T yy)

{ x = xx; y = yy; }

|  |
| --- |
| How is use of class template better than **typedef**.  A declaration of typedef:  typedef int int\_t; // declares int\_t to be an alias for the type int  Have to specify type with typedef in the declaration whereas with templating you don’t, you just specify your template variable and it naturally supports the primitive types. |

Cant do:

Template <class T>

T GetValue()

{

T value;

}

It does not work because you will not be able to pass the value to the function, which was the point in the template function: having many data type choices for one function

To use this function template we use the following format for the function call:

function\_name <type> (parameters);

Big O:

Useful for determining the best possible algorithm for the specific problem.

|  |
| --- |
| **Definition:** an upper-bound order for the growth rate (of no worse than type) of the resource requirements of an algorithm as input size becomes sufficiently large. |

Resource requirements: are time and space requirements

* Information about how quickly resource requirement will grow as input size increases
* How well will it scale as input size increases? Will requirements become really high? (think worst case)

**Misconceptions:**

(everything is stated as a correction or said “false” at the end)

1. Involves growth rate, not amount
2. *It does not* give absolute information about the algorithms resource requirements

* Information such as a specific amount of resource needed for a certain input size (or how much the needed resources will increase)

1. Captures long term, not short term behavior (it is asymptotic)
2. Most non-math people would prefer an upper-bound order (on growth rate) specified in terms of something that the *actual growth rate of the algorithm will not exceed*

Not what is specified in Big-O notation

1. “Growth rate order specified in conventional Big-O notation gives the algorithm’s actual growth rate in the worst-case scenario.” false

**Note:** *Specified growth rate order can be any that are as fast as or faster than the actual growth rate order*

*Ex: 9n^2 + 5n  is* ***O(n^2)****, R(n) = 9n^2 + 5n is* ***O(n^3)****, R(n) = 9n^2 + 5n is* ***O (n^4)*** *all correct*

But what matters the most is O(n^2) in this case, because the others are saying that the worst case is no better than O(n^3)... but the worst case is still closer to O(n^2), therefore we should focus on **O(n^2)** when dealing with the worst case since it is more useful. Want to arrive at a growth rate that is as *tight as possible*.

Worst, best, and average cases examples.

1. Worst case: in a search function, the element is not found.
2. Best-case: found immediately
3. average -case: make assumption of input. Can be difficult to determine

Multiple ways to solve a problem, efficiency is key to a better way to solving the problem so we analyze that finding the speed and time.

Temporal. time  efficiency

Spacial efficiency: Space

Same technique to analyze time can be used for space

Worst case: most useful because the best case might not happen so instead we want to minimize the worst case scenario for the program.

In worst case you need to find the upper-bound: cannot be worse than that

Fastest to slowest:

***O(1) < O(log n) < O( Sqrt(n) ) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n)***

**Linked List**

**Linked List vs Array**

***Strength/Weaknesses:***

|  |  |  |
| --- | --- | --- |
| **Container Type** | **Strength** | **Weakness** |
| **Array** | Random access | **insert /delete anomalies. Resize woe. expensive** |
| **Linked List** | insert/deletion made easier than arrays (no moving entire list over) | Random access impossible |

*Key Similarities/Differences:*

|  |  |  |
| --- | --- | --- |
| Container Type | Similarities | Differences |
| Array | Data structure container for storing and collection of data items | Consists of a single block of contiguous memory locations |
| Linked List | Also data structure container for storing and collection of data items | \* Consists of as many blocks of separate memory locations as there are data items  \*\* Nodes form a logical       group by being linked in some way through their addresses |

Linked List

Each node holds a data item (in a data field) and a pointer to the next node ( in link field)

Special end marker value for last node’s link field