|  |  |
| --- | --- |
| **Destructor Prototype:** ~Table();  **Function:**  template <class T> void Table<T>::~Table() {  for (unsigned int j = 0; j < rows; j++) {  delete [] values[j];  }  delete [] values;  } | **Copy Constructor Prototype:**  Table(const Table& t) { copy(t); }  **Assignment Operator Prototype:**  const Table& operator=(const Table& t);  **Functions:**  //Assgn 1 Tble 2 another, avoid self-assgnment  template <class T> const Table<T>& Table<T>::operator=(const Table<T>& v) {  if (this != &v) {  destroy();  this->copy(v); //Copy is below  }  return \*this;  }  //Create the Table as a copy of the given Table  template <class T> void Table<T>::copy(const Table<T>& v) {  this->create(v.rows,v.cols);  for (unsigned int i = 0; i < rows; i++) {  for (unsigned int j = 0; j < cols; j++) {  values[i][j] = v.values[i][j];  }  }  } |
| **For Loop Example:**  for (int i = 0; i < 10; ++i) {  //code goes here;  }  **While Loop Example:**  while (condition) {  condition = false;  }  **Stream Manipulators:**  #include <iostream>, std::cout, std::cin  std::cout << std::endl; // ends line in output stream, clears buffer  std::cin >> var\_name >> var\_name2;  std::setprecision(); //requires std::fixed |
| **Const**  -- Const objects can only be used by const member functions  -- In classes, if const at end of member function prototype then it does not change any member variables. | **Standard Library Types**  **Char:** Designated by single quotes, just a character. |
| **Order Notation:**  -- O(1), a.k.a. CONSTANT: The number of operations is independent of the size of the problem. e.g., compute quadratic root.  -- O(log n), a.k.a. LOGARITHMIC. e.g., dictionary lookup, binary search.  -- O(n), a.k.a. LINEAR. e.g., sum up a list.  -- O(n log n), e.g., sorting.  -- O(n^(1/2)), O(n^3), O(n^k), a.k.a. POLYNOMIAL, find the closest pair  -- O(2^n), O(kn), a.k.a. EXPONENTIAL. e.g., Fibonacci, playing chess.  -- O(N \* M), nested for loops. | **Iterators /Reverse Iterators :**  -- use dereference operator to access value at iterator (\*)  -- use select/dereference operator to access member functions ( itr->member() ).  -- reverse\_iterator increments backwards, find beginning reverse itr with .rbegin() and the .rend().  --\*itr for value  -- itr->func() is the same as (\*itr).func() |
| **Sort**  #include <algorithm>  //function prototype for sorting & sort call example  bool by\_total\_snowfall(const Snow &a, const Snow &b);  sort(container.begin(), container.end(),by\_total\_snowfall); |
| **STD::FIND:**  #include <algorithm>  std::find(container.begin(), container.end(), value); |
| **Standard Library Containers:**  **Arrays:** Can be dynamically created, fixed size, has [], created by type[size], int t[] = {4,5,3,2,2}, has size, iterator stuff, etc.  **std::string:** Container of chars, has iterator stuff, size(), [], can append with +=, push\_back/pop\_back, insert, erase.  **std::vector:** Has [], push/pop\_back, insert, eras, and iterator stuff. Can access iterator with v.begin() + int.  **std::list:** Has iterator stuff, push/pop \_ back/front, .front() and .back() for element access, no []! Not connected  **Erase & Insert:**  var.erase(iterator position);  //erases the object at position, returns next  var.insert(iterator position, val);  //inserts val in container before position  container<type>::iterator for itr | **Recursion Example:**  int intpow(int n, int p) {  if (p == 0) {  return 1;  } else {  return n \* intpow(n, p-1);  }  }  void countdown(int n) {  std::cout << n << std::endl;  if (n == 0) return;  else countdown(n-1);  } |
| **Operators:**  **+,-,\*, /, %, >, <, !=, ==, +=, -=,\*=, /=, %=**  **(pre)++i (post)i++.**  **Assignment Operator Special: (:)**  TrainCar(char t, int w) : type(t), weight(w), prev(NULL){  //other function stuff can go here  } |
| string a = "Tommy"; string& c = a; //c is alias | std::list::iterator p = s.begin();  ++p; p = s.erase(p); |
| **Recursive Print Data:**  void PrintData(Node \*head) {  if (head == NULL) return; //(!head) works  std::cout << head->value << " ";  PrintData(head->next);  }  **Vector Push Front:**  template <class T>  void Vec<T>::push\_front(const T& val) {  // if it’s the first element, use push\_back  if (m\_alloc == 0) { push\_back(val); return; }  assert (m\_alloc > 0);  if (m\_first == 0) {  // Calculate the new allocation.  m\_alloc \*= 2;  assert (m\_alloc > 1);  // Allocate the new array  T\* new\_data = new T[ m\_alloc ];  // put the existing data in the array  m\_first = m\_alloc / 2;  // copy the data  for (unsigned int i=0; i<m\_size; ++i) {  new\_data[m\_first+i] = m\_data[i]; }  // delete the old array and reset  delete [] m\_data;  m\_data = new\_data;  }  // move the first index back one spot  m\_first--;  //Add the value at the end and increment  m\_data[m\_first] = val;  ++m\_size;  }  **Order Notation pt 2:**  **int foo(int n) {**  **if (n == 1 || n == 0) return 1;**  **return foo(n-1) + foo(n-2);**  **}**  **ans for above: O(2^n)**  **-----------------------------------------------**  **for (int i = 0; i < n; i++) {**  **my\_vector.erase(my\_vector.begin());**  **}**  **ans for above: O(n^2) (erase loops through too)** | **Template Class Example:**  #ifndef Vec\_h\_  #define Vec\_h\_  template <class T> class Vec {  public:  // TYPEDEFS (two redacted)  typedef unsigned int size\_type;  // CONSTRUCTORS, ASSIGNMNENT OPERATOR, & DESTRUCTOR  Vec() { this->create(); }  Vec(size\_type n, const T& t = T()) { this->create(n, t); }  Vec(const Vec& v) { copy(v); }  Vec& operator=(const Vec& v);  ~Vec() { delete [] m\_data; }  // MEMBER FUNCTIONS AND OTHER OPERATORS  T& operator[] (size\_type i) { return m\_data[i]; }  const T& operator[] (size\_type i) const { return m\_data[i]; }  void push\_back(const T& t);  iterator erase(iterator p);  void resize(size\_type n, const T& fill\_in\_value = T());  void clear() { delete [] m\_data; create(); }  bool empty() const { return m\_size == 0; }  size\_type size() const { return m\_size; }  // ITERATOR OPERATIONS  iterator begin() { return m\_data; }  const\_iterator begin() const { return m\_data; }  iterator end() { return m\_data + m\_size; }  const\_iterator end() const { return m\_data + m\_size; }  private:  // PRIVATE MEMBER FUNCTIONS  void create();  void create(size\_type n, const T& val);  void copy(const Vec<T>& v);  // REPRESENTATION  T\* m\_data; // Pointer to first location in the allocated array  size\_type m\_size; // Number of elements stored in the vector  size\_type m\_alloc; // Number of array locations allocated, m\_size <= m\_alloc  };  // Create an empty vector (null pointers everywhere).  template <class T> void Vec<T>::create() {  m\_data = NULL;  m\_size = m\_alloc = 0; // No memory allocated yet  }  // Create a vector with size n, each location having the given value  template <class T> void Vec<T>::create(size\_type n, const T& val) {  m\_data = new T[n];  m\_size = m\_alloc = n;  for (T\* p = m\_data; p != m\_data + m\_size; ++p)  \*p = val;  }  // Shift each entry of the array after the iterator. Return the iterator,  // which will have the same value, but point to a different element.  template <class T> typename Vec<T>::iterator Vec<T>::erase(iterator p) {  // remember iterator and T\* are equivalent  for (iterator q = p; q < m\_data+m\_size-1; ++q)  \*q = \*(q+1);  m\_size --;  return p;  }  // If n is less than or equal to the current size, just change the size. If n is  // greater than the current size, the new slots must be filled in with the given value.  template <class T> void Vec<T>::resize(size\_type n, const T& fill\_in\_value) {  if (n <= m\_size)  m\_size = n;  else {  // If necessary, allocate new space and copy the old values  if (n > m\_alloc) {  m\_alloc = n;  T\* new\_data = new T[m\_alloc];  for (size\_type i=0; i<m\_size; ++i)  new\_data[i] = m\_data[i];  delete [] m\_data;  m\_data = new\_data;  }  // Now fill in the remaining values and assign the final size.  for (size\_type i = m\_size; i<n; ++i)  m\_data[i] = fill\_in\_value;  m\_size = n;  }  }  #endif |