|  |  |
| --- | --- |
| M\_PI => get the . This can be converted to different type automatically in the assignment & directly output.  sin(x); / cos(x); => return the trig value, given x is a radian.  pow(2, 10); => return an INT value of 2 to the power of 10.  abs(x) (flexible return) => x can be int/long/float/double  fabs(x) (return float only) => x can be int/long/float/double.  floor(x) (return the floor int) (if x < 0, return the smaller closest INT) => x can be float/double). log(x) (log of e) => x can be float/double.  In a statement, the conversion will not be proceed withouit an explicit conversion (e.g. float y = 5/2 => y = 2 [in float]. ) | customise sort function: sort(vector.begin(), vector.end(), self-define-rule); self-define-rule should be bool function;  comparison function can add "const" and "&" in the parameters comparison: first > second: sort from greatest to smallest first < second: sort from smallest to greatest  define the operator in the class: in .h file: bool operator< (const class\_name& first); in class.cpp file: bool operator< (const class\_name& first, cons class\_name& second){  operating rule;}  run in the terminal:  int main(int argc, char\* argv[])  seg fault in c++: 1. Dereferencing a null pointer. 2. Dereferencing a uninitialized pointer 3. Access out-of-boundary memory on vector/list/etc. 4. Writing a read-only memory. 5. Stackoverflow. |
| STL sort: #include <algorithm>  default: sorts from least to greatest.  std::string string\_name; => create an empty string.  std::string string\_name(string2) => copy string 2 to string\_name.  std::string my\_string(10, ‘0’) => create a string with 10 \* ‘0’  STD string are mutable, python string is immutable. |
| STL vectors (dynamically-sized, 1-D array) #include <vector>  define: std::vector<int> scores;  >start empty unless specified => by contrast: int[] will contain garbage when unspecified defined.  constructions:  std::vector<int> a/ std::vector<double> b(100, 3.14)/ std::vector<int> c(1000)/ std::vector<double> d(b)  default: sorts from least to greatest.  begin/end can also be the \*q pointer, direct to the array in the heap.  std::string string\_name; => create an empty string.  std::string string\_name(string2) => copy string 2 to string\_name.  std::string my\_string(10, ‘0’) => create a string with 10 \* ‘0’  function of string: string.size() => get the size of the string (type: unsigned int)  C-style string: char h[] = “HELLO”;  STD string: std::string s1;  conversion:  C-style to STD: std::string s2(h);  STD to C-style: char h[] = s1.str();  STD string are mutable, python string is immutable.  seg fault in c++: 1. Dereferencing a null pointer 2. Dereferencing a uninitialized pointer 3. Access out-of-boundary memory on vector/list/etc. 4. Writing a read-only memory 5. Stackoverflow.  Iterator:  vector<string>::iterator p; vector<string>::const\_iterator q; : can change the iterator but cannot change the vector through the iterator (cannot in the LHS) Define iterator in template class: typedef T\* iterator; typedef const T\* const\_iterator; iterator-related functions: iterator erase(iterator p); 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;} (end() should not be dereference because it is a slot after the end of the vector, which is not the last element of the vector)  iterator version: erase\_from\_vector erase\_from\_vector(std::vector<std::string::iterator> itr, vector<string>& v){  std::vector<<std::string::iterator> itr2 = itr;  itr2++;  for ( ; itr2 != v.end(); itr++, itr2++){  (\*itr) = (\*itr2); // v[j] = v[j+1];}} template<class T> typename Vec<T>::iterator Vec<T>::erase(iterator p){  for (iterator q = p; q + 1 < m\_data + m\_size; ++q){  \*q = \*(q + 1)}  m\_size --;return p;} Vector has a member function ".erase(iterator)", which has a principle of operation above. bigO notation: O(n)  \*\* if we want to get the return value of .erase(), we cannot erase(.end()) because the moving pointer of erase function will point to nothing.  The iterator may be invalided after push\_back/resize/erase in vector, because the shifting/copying of arrays may lead to pointers are not matching the data we want.  Iterator in the list **cannot** "jump" (e.g. itr += 5  situations a iterator may be invalidated: - Iterator positioned on an STL vector, at after the point of an erase operation, are invalidated. - Iterators positioned anywhere on an STL vector may be invalid after insert (or push\_back or resize) operator. - Iterators attached to an STL list are not invalidated after an insert or push\_back/push\_front or erase/pop\_back/pop\_front (Except iterators attached to the erased element)  There is no "comparing operators" in list iterator while vector have.  insert function: v.insert(iterator p, element) iterator: all the element after p, including p, will "shift". return: the pointer of the element being inserted. reverse iterator: step through a list from back to the front std::list<int> a; unsigned int i; for ( i=1; i<10; ++i ) a.push\_back( i\*i ); std::list<int>::reverse\_iterator ri; for( ri = a.rbegin(); ri != a.rend(); ++ri ) cout << \*ri << endl;  List:  sort: (is a member function in list): my\_lst.sort(opt\_condition)bigO: O(nlogn)  insert function: template <class T> void insert(Node<T>\* &head, Node<T>\* &pnt, const T& value){  Node<T>\* temp = new Node<T>;  temp->value = value;  temp->pnt = pnt;  if (head == pnt){ // insert in the front  head = temp;  }else{  while (head->pnt != pnt){  head = head->pnt;}}  Node<T> lastNode = (\*head);  lastNode->pnt = temp;} erase function: template <class T> void erase(Node<T>\* &head, Node<T>\* &pnt){  if (head == pnt){ // erase from the front  head = pnt->pnt;}  while (head->pnt != pnt){head = head->pnt;}  head->pnt = pnt->pnt;}  **in doubly-linked list:**  template <class T> void erase(Node<T>\* &p, Node<T>\* &head, Node<T>\* &tail){  node<T>\* prevNode = p->prev;  node<T>\* nextNode = p->next;  if (head == p && nextNode != NULL){  // delete the first element and >1 elements  head = nextNode;  nextNode->prev = NULL;  }if (p == head){ // delete the only element  head = NULL;  tail = NULL;  }if (p == tail){ // delete the last element  prevNode = NULL;  tail = prevNode;  }else{ // general case  prevNode->next = nextNode; nextNode->prev = prevNode;}  delete p;} |
| Vector implementation:  template <Class T>  class Vec{  public:  typedef unsigned int size\_type;  Vec() {this->create();} // default constructor;  Vec(const Vec& v) {this->copy(v);}; // copy constructor  Vec(const int a, const int b) {this->create(a, b);} // constructor  ~Vec() {destroy();} // destructor;  void push\_back(const T& t);  Vec& operator=(const Vec& v);  T& operator[] (size\_type i) {return m\_data[i]}; => = vector[i] = vector.operator[](7); (read-and-write function)  const T& operator[] (size\_type i) const {return m\_data[i]} => (read-only get function) (const in return type refer that the return value is not allowed to modify either) void push\_back(const T& t); => as shown;  private:  void create();  void create(int a, int b);  void destroy();  T\* m\_data;  size\_type m\_size;  size\_type m\_alloc; }  template <class T> void Vec<T>::push\_back(const T& val){  if (m\_size == m\_alloc){  // copy the current array to the new one with doubled size  // step 1. create a temp pointer, to the newly-created double-sized array  T\* temp = new T[m\_alloc \* 2];  // step 2. copy the old array to the new array  for (size\_type i = 0; i < m\_alloc; i++){  temp[i] = m\_data[i]; }  m\_alloc \*= 2;  // step 3. delete the old array  delete[] m\_data;  // direct the pointer to the new array  m\_data = temp;}  // add the new variable to the array  m\_data[m\_size] = val;  ++ m\_size;}  template <Class T> void Vec<T>::copy(const Vec<T>& v){  m\_data = new T[v.m\_alloc];  for (size\_type i=0; i < v.m\_size; i++){  m\_data[i] = v.m\_data[i];}  m\_size = v.m\_size;  m\_alloc = v.m\_alloc;} template <Class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v){  if(this != &v){ // check if they are not self-assignment (v1 = v1)  this->destroy();  this->copy(v);}  return \*this;}  ---------------------------------------  pop\_back: remove the last element in the vector, size -1. best / avg / worst cases are all O(1)  erase\_from\_vector(unsigned int i, vector<std::string>& v){ // remove an element from a specific location i  for (unsigned int j = i; j < v.size() - 1; j++){  v[j] = v[j+1];}  v.pop\_back();}  Recursion - Usually have same bigO notation with the iterative version Binary search template <class T> bool binsearch(const std::vector<T> &v, int low, int high, const T &x){  if (high == low){  return x == v[low];}  int mid = (low + high) / 2;  if (x <= v[mid]){  return binsearch(v, low, mid, x);  }else{  return binsearch(v, mid+1, high, x);}}  // driver function to initial call the binary search ------- Merge sort // driver function template <class T> void mergesort(std::vector<T>& values){  std::vector<T> scratch(values.size());  mergesort(0, int(values.size()-1), values, scratch);} // recursive function template <class T> void mergesort(int low, int high, std::vector<T>& values, std::vector<T>& scratch){  std::cout << "mergesort: low = " << low << ", high = " << high << std::endl;  if (low >= high) {return;}  int mid = (low + high) / 2;  mergesort(low, mid, values, scratch);  mergesort(mid+1, high, values, scratch);  merge(low, mid, high, values, scratch);} // helper function of the recursive function template <class T> void merge(int low, int mid, int high, int value, std::vector<T> &scratch){  int i = low;  int j = mid + 1;  k = low;   // while there's still something left in one of the sorted sub-intervals:  while (i <= mid && j <= high){  // look at the top values, grab the smaller one, store it in the scratch vector  if (values[i] < values[j]){  scratch[k] = values[i]; i++;  }else{  scratch[k] = values[j]; j++;}k++;}  while (i <= mid){  scratch[k] = values[i];i++;k++;}  while (j <= high){  scratch[k] = values[j];j++;k++;}  // copy the scratch back to values  for (l = low; l <= high; l++){  values[l] = scratch[l];}} |