PASSING ARGUMENTS

* Permanent change – pass by reference (so not copying it over) &
* Don’t want to change – pass by constant reference (const type & variable\_name)
* Passing a file to a function MUST be by reference (not const – outputting/reading changes it)

POINTERS

|  |  |  |
| --- | --- | --- |
| int x =5, y = 9;  int \*p = &x, \*q = &y;  \*p = 17;  \*q = \*p;  p = q;  \*q = 13; | //A  //B  //C  //D |  |

DYNAMIC MEMORY

* int \*p = new int; / int \*a = new int[n] -- use new to “reserve space” on the heap
* delete p; -- must delete the space on heap when you’re done – else it’s a memory leak
* delete [] a; -- to delete the array

CLASSES

|  |  |  |  |
| --- | --- | --- | --- |
| Class.h  class Class {  public:  Class();  Class(int x);  private:  int foo;  }; | //nonmembers can access  //constructor  //constructor w/ parameters  //nonmembers can’t access  //SEMICOLON | Class.cpp  #include “Class.h”  Class::Class() { }  void nonmember(a,b) {} | //include header  //default constructor  //must include types (const if necessary) |

* template <class T> class Vec --- all in same file for a template class (no separate cpp file)
* template <class T> void Vec<T>::create(const T& T…
* when dynamically allocated memory – must have copy constructor (Vec(const Vec& v)), assignment operator (v1 = v2), destructor (~Vec())
* bool operator< (arg1, arg2) { return true if less than; return false if greater than }
* THIS: this->variable == (\*this).variable (shows when an object class member is being used (refers to current class object)
* class Class1: public Class2 // if parts of Class2 are “protected” – only derived classes can access
* virtual functions – when redefined, the new definitions are used. If virtual func() = 0 – must be redefined in the derived class

REFERENCE/RETURN VALUES

|  |  |  |
| --- | --- | --- |
| string a = “Tom”;  string b = a;  string& c = a;  b[1] = ‘i’;  c[1] = ‘a’; | // copy constructor  //c is an alias to a  //Tom Tim Tom  //Tam Tim Tam | * if a function returns a const string& - same thing as in example (must be const ref rather than just by ref or can change internal workings of a class potentially) |

MAPS

* entries in maps are pairs: std::pair<const key\_type, value\_type>
* search, insert, and erase are O(logn)
* ordered by increasing value of the key (NEED < OPERATOR), can’t have duplicate keys
* if use [] the key is ALWAYS in the map afterwards – use find or insert
* m.find(key) returns iterator to the found pair/.end() if not in the map
* m.insert(make\_pair(key,value)) returns new pair – iterator to existing pair and false/iterator to newly added pair and true
* m.erase(iterator), erase(iterator first, iterator last) – deletes [first, last), erase(key) – first two return an iterator to the next valid element

SETS

* a map with only keys, same run time, keys sorted/can’t duplicate them
* pair<iterator,bool> set<Key>::insert(key)
* same erase/find versions as map

TREES

* depth first (don’t backtrack until hit a leaf) vs breadth first (go one level at a time)
* in order (L V R), pre order (V L R), post order (L R V)

OPERATORS/FRIENDS

* If you change an existing obj inside an operator and need to return that obj, RETURN REF
* Friends must be declared in public of the class – allows the other class to access private members

EXCEPTION MECHANISM (TRY/THROW/CATCH)

* try { something and throw if it doesn’t work } catch (type var) { do something, can use var }
* catch(…) will catch all types of exceptions, but you can’t use the value that’s thrown
* will find the closest enclosing try/catch block once thrown. Rest of the code is abandoned

GARBAGE COLLECTION

* reference counting: + fast and incremental, - can’t handle cyclical data structures, ? requires ##% extra memory (1 integer per node)
* stop & copy: + can handle cyclical data structures / runs fast if most of the memory is garbage / data is clustered together and memory is “defragmented”, - requires 100% extra memory / requires a long pause in execution
* mark-sweep: + can handle cyclical data structures / requires only 1% extra memory, - requires a long pause in execution / runs the same speed regardless of how much memory is garbage (must touch all nodes)

HASH TABLES

* separate chaining to solve collisions: a list of keys/values hashed at the location, works well when collisions are infrequent
* open addressing: linear probing, quadratic probing (i+n\*n)%N, secondary hashing (use of a second hash function), - slows dramatically when table is close to full / fails when it is full / cost of computing new hashes

PRIORITY QUEUES / STACKS

* parent at (i-1)/2, children at 2i+1, 2i+2 - values popped based on priority
* percolate down for each index from (n-1)/2 down to 0 to create a valid heap
* stacks allow access and deletion only from the top, no access to middle, all operations are O(1)
* queues allow insertion at back and removal from the front, generally made from a list, O(1)



