Preprocessor

```
                        // Comment to end of line
                        /* Multi-line comment */
#include  <stdio.h>     // Insert standard header file
#include "myfile.h"     // Insert file in current directory
#define X some text     // Replace X with some text
#define F(a,b) a+b      // Replace F(1,2) with 1+2
#define X \
 some text              // Multiline definition
#undef X                // Remove definition
#if defined(X)          // Conditional compilation (#ifdef X)
#else                   // Optional (#ifndef X or #if !defined(X))
#endif                  // Required after #if, #ifdef
```

Literals

```
255, 0377, 0xff         // Integers (decimal, octal, hex)
2147483647L, 0x7fffffffl // Long (32-bit) integers
123.0, 1.23e2           // double (real) numbers
'a', '\141', '\x61'     // Character (literal, octal, hex)
'\n', '\\', '\'', '\"'  // Newline, backslash, single quote, double quote
"string\n"              // Array of characters ending with newline and \0
"hello" "world"         // Concatenated strings
true, false             // bool constants 1 and 0
nullptr                 // Pointer type with the address of 0
```

Declarations

```
int x;                  // Declare x to be an integer (value undefined)
int x=255;              // Declare and initialize x to 255
short s; long l;        // Usually 16 or 32 bit integer (int may be either)
char c='a';             // Usually 8 bit character
unsigned char u=255;
signed char s=-1;       // char might be either
unsigned long x =
  0xffffffffL;          // short, int, long are signed
float f; double d;      // Single or double precision real (never unsigned)
bool b=true;            // true or false, may also use int (1 or 0)
int a, b, c;            // Multiple declarations
int a[10];              // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};        // Initialized array (or a[3]={0,1,2}; )
int a[2][2]={{1,2},{4,5}}; // Array of array of ints
char s[]="hello";       // String (6 elements including '\0')
std::string s = "Hello"  // Creates string object with value "Hello"
std::string s = R"(Hello
World)";                // Creates string object with value "Hello\nWorld"
int* p;                 // p is a pointer to (address of) int
char* s="hello";        // s points to unnamed array containing "hello"
void* p=nullptr;        // Address of untyped memory (nullptr is 0)
int& r=x;               // r is a reference to (alias of) int x
enum weekend {SAT,SUN}; // weekend is a type with values SAT and SUN
enum weekend day;       // day is a variable of type weekend
enum weekend{SAT=0,SUN=1}; // Explicit representation as int
enum {SAT,SUN} day;     // Anonymous enum
enum class Color {Red,Blue};// Color is a strict type with values Red and Blue
Color x = Color::Red;   // Assign Color x to red
typedef String char*;   // String s; means char* s;
const int c=3;          // Constants must be initialized, cannot assign to
const int* p=a;         // Contents of p (elements of a) are constant
int* const p=a;         // p (but not contents) are constant
const int* const p=a;   // Both p and its contents are constant
const int& cr=x;        // cr cannot be assigned to change x
int8_t,uint8_t,int16_t,
uint16_t,int32_t,uint32_t,
int64_t,uint64_t        // Fixed length standard types
auto it = m.begin();    // Declares it to the result of m.begin()
auto const param = config["param"];
                        // Declares it to the const result
auto& s = singleton::instance();
                        // Declares it to a reference of the result
```

Templates

```
template <class T> T f(T t);// Overload f for all types
template <class T> class X {// Class with type parameter T
  X(T t); };            // A constructor
template <class T> X<T>::X(T t) {}
                        // Definition of constructor
X<int> x(3);            // An object of type "X of int"
template <class T, class U=T, int n=0>
                        // Template with default parameters
```

Namespaces

```
namespace N {class T {};}  // Hide name T
N::T t;                 // Use name T in namespace N
using namespace N;      // Make T visible without N::
```

STORAGE Classes

```
int x;                  // Auto (memory exists only while in scope)
static int x;           // Global lifetime even if local scope
extern int x;           // Information only, declared elsewhere
```

Statements

```
x=y;                    // Every expression is a statement
int x;                  // Declarations are statements
;                       // Empty statement
{                       // A block is a single statement
    int x;              // Scope of x is from declaration to end of block
}
if (x) a;               // If x is true (not 0), evaluate a
else if (y) b;          // If not x and y (optional, may be repeated)
else c;                 // If not x and not y (optional)

while (x) a;            // Repeat 0 or more times while x is true

for (x; y; z) a;        // Equivalent to: x; while(y) {a; z;}

for (x : y) a;          // Range-based for loop e.g.
                        // for (auto& x in someList) x.y();

do a; while (x);        // Equivalent to: a; while(x) a;

switch (x) {            // x must be int
    case X1: a;         // If x == X1 (must be a const), jump here
    case X2: b;         // Else if x == X2, jump here
    default: c;         // Else jump here (optional)
}
break;                  // Jump out of while, do, or for loop, or switch
continue;               // Jump to bottom of while, do, or for loop
return x;               // Return x from function to caller
try { a; }
catch (T t) { b; }      // If a throws a T, then jump here
catch (...) { c; }      // If a throws something else, jump here
```

Functions

```
int f(int x, int y);    // f is a function taking 2 ints and returning int
void f();               // f is a procedure taking no arguments
void f(int a=0);        // f() is equivalent to f(0)
f();                    // Default return type is int
inline f();             // Optimize for speed
f() { statements; }     // Function definition (must be global)
T operator+(T x, T y);  // a+b (if type T) calls operator+(a, b)
T operator-(T x);       // -a calls function operator-(a)
T operator++(int);      // postfix ++ or -- (parameter ignored)
extern "C" {void f();}  // f() was compiled in C
```

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

```
int main()  { statements... }    // or
int main(int argc, char* argv[]) { statements... }
```

`argv` is an array of `argc` strings from the command line. By convention, `main` returns status `0` if successful, `1` or higher for errors.

Functions with different parameters may have the same name (overloading). Operators except `::` `.` `.*` `?:` may be overloaded. Precedence order is not affected. New operators may not be created.

Expressions

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation, which is undefined. There are no run time checks for arrays out of bounds, invalid pointers, etc.

```
T::X                    // Name X defined in class T
N::X                    // Name X defined in namespace N
::X                     // Global name X

t.x                     // Member x of struct or class t
p-> x                   // Member x of struct or class pointed to by p
a[i]                    // i'th element of array a
f(x,y)                  // Call to function f with arguments x and y
T(x,y)                  // Object of class T initialized with x and y
x++                     // Add 1 to x, evaluates to original x (postfix)
x--                     // Subtract 1 from x, evaluates to original x
typeid(x)               // Type of x
typeid(T)               // Equals typeid(x) if x is a T
dynamic_cast< T>(x)     // Converts x to a T, checked at run time.
static_cast< T>(x)      // Converts x to a T, not checked
reinterpret_cast< T>(x) // Interpret bits of x as a T
const_cast< T>(x)       // Converts x to same type T but not const
```

Classes

```
class T {               // A new type
private:                // Section accessible only to T's member functions
protected:              // Also accessible to classes derived from T
public:                 // Accessible to all
    int x;              // Member data
    void f();           // Member function
    void g() {return;}  // Inline member function
    void h() const;     // Does not modify any data members
    int operator+(int y);   // t+y means t.operator+(y)
    int operator-();    // -t means t.operator-()
    T(): x(1) {}        // Constructor with initialization list
    T(const T& t): x(t.x) {}// Copy constructor
    T& operator=(const T& t)
    {x=t.x; return *this; } // Assignment operator
    ~T();               // Destructor (automatic cleanup routine)
    explicit T(int a);  // Allow t=T(3) but not t=3
    T(float x): T((int)x) {}// Delegate constructor to T(int)
    operator int() const
    {return x;}         // Allows int(t)
    friend void i();    // Global function i() has private access
    friend class U;     // Members of class U have private access
    static int y;       // Data shared by all T objects
    static void l();    // Shared code.  May access y but not x
    class Z {};         // Nested class T::Z
    typedef int V;      // T::V means int
};
void T::f() {           // Code for member function f of class T
    this->x = x;}       // this is address of self (means x=x;)
int T::y = 2;           // Initialization of static member (required)
T::l();                 // Call to static member
T t;                    // Create object t implicit call constructor
t.f();                  // Call method f on object t

struct T {              // Equivalent to: class T { public:
  virtual void i();     // May be overridden at run time by derived class
  virtual void g()=0; };// Must be overridden (pure virtual)
class U: public T {     // Derived class U inherits all members of base T
  public:
  void g(int) override; };  // Override method g
class V: private T {};  // Inherited members of T become private
class W: public T, public U {};
                        // Multiple inheritance
class X: public virtual T {};
                        // Classes derived from X have base T directly
```

All classes have a default copy constructor, assignment operator, and destructor, which perform the corresponding operations on each data member and each base class as shown above. There is also a default no-argument constructor (required to create arrays) if the class has no constructors. Constructors, assignment, and destructors do not inherit.

```
 sizeof x               // Number of bytes used to represent object x
 sizeof(T)              // Number of bytes to represent type T
 ++x                    // Add 1 to x, evaluates to new value (prefix)
 --x                    // Subtract 1 from x, evaluates to new value
 ~x                     // Bitwise complement of x
 !x                     // true if x is 0, else false (1 or 0 in C)
 -x                     // Unary minus
 +x                     // Unary plus (default)
 &x                     // Address of x
 *p                     // Contents of address p (*&x equals x)
 new T                  // Address of newly allocated T object
 new T(x, y)            // Address of a T initialized with x, y
 new T[x]               // Address of allocated n-element array of T
 delete p               // Destroy and free object at address p
 delete[] p             // Destroy and free array of objects at p
 (T) x                  // Convert x to T (obsolete, use ..._cast<T>(x))

 x * y                  // Multiply
 x / y                  // Divide (integers round toward 0)
 x % y                  // Modulo (result has sign of x)

 x + y                  // Add, or \&x[y]
 x - y                  // Subtract, or number of elements from *x to *y
 x << y                 // x shifted y bits to left (x * pow(2, y))
 x >> y                 // x shifted y bits to right (x / pow(2, y))

 x < y                  // Less than
 x <= y                 // Less than or equal to
 x > y                  // Greater than
 x >= y                 // Greater than or equal to

 x & y                  // Bitwise and (3 & 6 is 2)
 x ^ y                  // Bitwise exclusive or (3 ^ 6 is 5)
 x | y                  // Bitwise or (3 | 6 is 7)
 x && y                 // x and then y (evaluates y only if x (not 0))
 x || y                 // x or else y (evaluates y only if x is false (0))
 x = y                  // Assign y to x, returns new value of x
 x += y                 // x = x + y, also -= *= /= <<= >>= &= |= ^=
 x ? y : z              // y if x is true (nonzero), else z
 throw x                // Throw exception, aborts if not caught
 x , y                  // evaluates x and y, returns y (seldom used)
```

`math.h`, `cmath` (floating point math)

```
#include <cmath>            // Include cmath (std namespace)
sin(x); cos(x); tan(x);     // Trig functions, x (double) is in radians
asin(x); acos(x); atan(x);  // Inverses
atan2(y, x);                // atan(y/x)
sinh(x); cosh(x); tanh(x);  // Hyperbolic sin, cos, tan functions
exp(x); log(x); log10(x);   // e to the x, log base e, log base 10
pow(x, y); sqrt(x);         // x to the y, square root
ceil(x); floor(x);          // Round up or down (as a double)
fabs(x); fmod(x, y);        // Absolute value, x mod y
```

`assert.h`, `cassert` (Debugging Aid)

```
#include <cassert>        // Include iostream (std namespace)
assert(e);                // If e is false, print message and abort
#define NDEBUG            // (before #include <assert.h>), turn off assert
```

`iostream.h`, `iostream` (Replaces `stdio.h`)

```
#include <iostream>         // Include iostream (std namespace)
cin >> x >> y;              // Read words x and y (any type) from stdin
cout << "x=" << 3 << endl;  // Write line to stdout
cerr << x << y << flush;    // Write to stderr and flush
c = cin.get();              // c = getchar();
cin.get(c);                 // Read char
cin.getline(s, n, '\n');    // Read line into char s[n] to '\n' (default)
if (cin)                    // Good state (not EOF)?
                            // To read/write any type T:
istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;}
```

`fstream.h`, `fstream` (File I/O works like `cin`, `cout` as above)

```
#include <fstream>          // Include filestream (std namespace)
ifstream f1("filename");    // Open text file for reading
if (f1)                     // Test if open and input available
    f1 >> x;                // Read object from file
f1.get(s);                  // Read char or line
f1.getline(s, n);           // Read line into string s[n]
ofstream f2("filename");    // Open file for writing
if (f2) f2 << x;            // Write to file
```

`memory` (dynamic memory management)

```
#include <memory>           // Include memory (std namespace)
shared_ptr<int> x;          // Empty shared_ptr to a integer on heap. Uses reference counting for c
x = make_shared<int>(12);   // Allocate value 12 on heap
shared_ptr<int> y = x;      // Copy shared_ptr, implicit changes reference count to 2.
cout << *y;                 // Dereference y to print '12'
if (y.get() == x.get()) {   // Raw pointers (here x == y)
    cout << "Same";
}
y.reset();                  // Eliminate one owner of object
if (y.get() != x.get()) {
    cout << "Different";
}
if (y == nullptr) {         // Can compare against nullptr (here returns true)
    cout << "Empty";
}
y = make_shared<int>(15);   // Assign new value
cout << *y;                 // Dereference x to print '15'
cout << *x;                 // Dereference x to print '12'
weak_ptr<int> w;            // Create empty weak pointer
w = y;                      // w has weak reference to y.
if (shared_ptr<int> s = w.lock()) { // Has to be copied into a shared_ptr before usage
    cout << *s;
}
unique_ptr<int> z;          // Create empty unique pointers
unique_ptr<int> q;
z = make_unique<int>(16);   // Allocate int (16) on heap. Only one reference allowed.
q = move(z);                // Move reference from z to q.
if (z == nullptr){
    cout << "Z null";
}
cout << *q;
shared_ptr<B> r;
r = dynamic_pointer_cast<B>(t); // Converts t to a shared_ptr<B>
```

`string` (Variable sized character array)

```
#include <string>         // Include string (std namespace)
string s1, s2="hello";    // Create strings
s1.size(), s2.size();     // Number of characters: 0, 5
s1 += s2 + ' ' + "world"; // Concatenation
s1 == "hello world"       // Comparison, also <, >, !=, etc.
s1[0];                    // 'h'
s1.substr(m, n);          // Substring of size n starting at s1[m]
s1.c_str();               // Convert to const char*
s1 = to_string(12.05);    // Converts number to string
getline(cin, s);          // Read line ending in '\n'
```

`vector` (Variable sized array/stack with built in memory allocation)

```
#include <vector>         // Include vector (std namespace)
vector<int> a(10);        // a[0]..a[9] are int (default size is 0)
vector<int> b{1,2,3};       // Create vector with values 1,2,3
a.size();                 // Number of elements (10)
a.push_back(3);           // Increase size to 11, a[10]=3
a.back()=4;               // a[10]=4;
a.pop_back();             // Decrease size by 1
a.front();                // a[0];
a[20]=1;                  // Crash: not bounds checked
a.at(20)=1;               // Like a[20] but throws out_of_range()
for (int& p : a)
  p=0;                    // C++11: Set all elements of a to 0
for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p)
  *p=0;                   // C++03: Set all elements of a to 0
vector<int> b(a.begin(), a.end());  // b is copy of a
vector<T> c(n, x);        // c[0]..c[n-1] init to x
T d[10]; vector<T> e(d, d+10);      // e is initialized from d
```

`deque` (Array stack queue)

`deque<T>` is like `vector<T>`, but also supports:

```
#include <deque>          // Include deque (std namespace)
a.push_front(x);          // Puts x at a[0], shifts elements toward back
a.pop_front();            // Removes a[0], shifts toward front
```

`utility` (pair)

```
#include <utility>        // Include utility (std namespace)
pair<string, int> a("hello", 3);  // A 2-element struct
a.first;                  // "hello"
a.second;                 // 3
```

`chrono` (Time related library)

```
#include <chrono>         // Include chrono
using namespace std::chrono; // Use namespace
auto from =               // Get current time_point
  high_resolution_clock::now();
// ... do some work
auto to =                 // Get current time_point
  high_resolution_clock::now();
using ms =                // Define ms as floating point duration
  duration<float, milliseconds::period>;
                          // Compute duration in milliseconds
cout << duration_cast<ms>(to - from)
  .count() << "ms";
```

`future` (thread support library)

```
#include <future>         // Include future
function<int(int)> fib =  // Create lambda function
  [&](int i){
    if (i <= 1){
      return 1;
    }
    return fib(i-1)
         + fib(i-2);
  };
future<int> fut =         // result of async function
  async(launch::async, fib, 4); // start async function in other thread
// do some other work
cout << fut.get();        // get result of async function. Wait if needed.
```

`map` (associative array - usually implemented as binary search trees - avg. time complexity: O(log n))

```
#include <map>            // Include map (std namespace)
map<string, int> a;       // Map from string to int
a["hello"] = 3;           // Add or replace element a["hello"]
for (auto& p:a)
    cout << p.first << p.second;  // Prints hello, 3
a.size();                 // 1
```

`unordered_map` (associative array - usually implemented as hash table - avg. time complexity: O(1))

```
#include <unordered_map>  // Include map (std namespace)
unordered_map<string, int> a; // Map from string to int
a["hello"] = 3;           // Add or replace element a["hello"]
for (auto& p:a)
    cout << p.first << p.second;  // Prints hello, 3
a.size();                 // 1
```

`set` (store unique elements - usually implemented as binary search trees - avg. time complexity: O(log n))

```
#include <set>            // Include set (std namespace)
set<int> s;               // Set of integers
s.insert(123);            // Add element to set
if (s.find(123) != s.end()) // Search for an element
    s.erase(123);
cout << s.size();         // Number of elements in set
```

`unordered_set` (store unique elements - usually implemented as a hash set - avg. time complexity: O(1))

```
#include <unordered_set>  // Include set (std namespace)
unordered_set<int> s;     // Set of integers
s.insert(123);            // Add element to set
if (s.find(123) != s.end()) // Search for an element
    s.erase(123);
cout << s.size();         // Number of elements in set
```

`algorithm` (A collection of 60 algorithms on sequences with iterators)

```
#include <algorithm>      // Include algorithm (std namespace)
min(x, y); max(x, y);     // Smaller/larger of x, y (any type defining <)
swap(x, y);               // Exchange values of variables x and y
sort(a, a+n);             // Sort array a[0]..a[n-1] by <
sort(a.begin(), a.end()); // Sort vector or deque
reverse(a.begin(), a.end()); // Reverse vector or deque
```

`thread` (Multi-threading library)

```
#include <thread>         // Include thread
unsigned c =
  hardware_concurrency(); // Hardware threads (or 0 for unknown)
auto lambdaFn = [](){     // Lambda function used for thread body
    cout << "Hello multithreading";
};
thread t(lambdaFn);       // Create and run thread with lambda
t.join();                 // Wait for t finishes

// --- shared resource example ---
mutex mut;                          // Mutex for synchronization
condition_variable cond;            // Shared condition variable
const char* sharedMes               // Shared resource
  = nullptr;
auto pingPongFn =                   // thread body (lambda). Print someone else's message
  [&](const char* mes){
    while (true){
      unique_lock<mutex> lock(mut);// locks the mutex
      do {
        cond.wait(lock, [&](){     // wait for condition to be true (unlocks while waiting which al
          return sharedMes != mes; // statement for when to continue
        });
      } while (sharedMes == mes);  // prevents spurious wakeup
      cout << sharedMes << endl;
      sharedMes = mes;
      lock.unlock();               // no need to have lock on notify
      cond.notify_all();           // notify all condition has changed
    }
  };
sharedMes = "ping";
thread t1(pingPongFn, sharedMes);  // start example with 3 concurrent threads
thread t2(pingPongFn, "pong");
thread t3(pingPongFn, "boing");
```

FIND functions:
#include <algorithm>

- std::vector<int>::iterator low = std::lower_bound(v.begin(), v.end(), 4); int index = low - v.begin();
  - first element >= than 4, need sorted container
  - [1,2,4,4,5] -> index = 2
  - O(log N) binary search
- std::vector<int>::iterator up = std::upper_bound(v.begin(), v.end(), 4); int index = up - v.begin();
  - first element > than 4, need sorted container
  - [1,2,4,4,5] -> index = 4
  - O(log N) binary search
- bool exists = std::binary_search(v.begin(), v.end(), 4);
  - [1,2,4,4,5] -> exists = true
  - O(log N) binary search

SORT functions:
#include <algorithm>

- merge sort / quick sort
  - std::sort(v.begin(), v.end()); # increasing order
  - std::sort(v.begin(), v.end(), greater<int>()); # decreasing order
  - O(N log N) on average

DATA structure:

1. Array
   - Use: Simple, fixed-size, contiguous data structure.
   - Average time complexity: Insertion O(N); Deletion O(N); Search O(N); Index O(1).
   - int arr[] = {1, 2, 3}; int a = arr[1];
2. Vector
   - Use: Dynamic array with flexible size.
   - Average time complexity: Insertion O(N) [Amortized O(1) for push_back]; Deletion O(N); Search O(N); Index O(1).
   - #include <vector> std::vector<int> vec = {1, 2, 3}; int s = vec.size(); bool e = vec.empty(); int b = vec.back(); int f = vec.front(); int value = vec[2]; vec.push_back(4); vec.pop_back(); vec.clear();
3. Deque
   1. Double-Ended Queue
   2. Use: Efficient insertion and deletion at both ends with indexing.
   3. Average time complexity: Insertion O(1) at both ends (amortized); Deletion O(1) at both ends (amortized); Search O(N); Index O(1).
   4. #include <deque> std::deque<int> deq = {1, 2, 3}; int s = deq.size(); bool e = deq.empty(); int b = deq.back(); int f = deq.front(); int value = deq[2]; deq.push_back(4); deq.push_front(4); deq.pop_back(); deq.pop_front(); deq.clear();
4. List
   - Doubly-Linked List
   - Use: Frequent insertions and deletions in a sequence in the middle of the container.
   - Average time complexity: Insertion O(1); Deletion O(1); Search O(N); Index not applicable.
   - #include <list> std::list<int> ml = {1, 2, 3}; int s = ml.size(); bool e = ml.empty(); int b = ml.back(); int f = ml.front(); int value = ml[2]; ml.push_back(4); ml.push_front(4); ml.pop_back(); ml.pop_front(); ml.clear();
5. Stack
   - Use: Managing data in a last-in, first-out (LIFO) manner.
   - Average time complexity: Insertion O(1); Deletion O(1); Search O(N); Index not applicable.
   - #include <stack> std::stack<int> st; int s = st.size(); bool b = st.empty(); int e = st.top(); st.push(1); st.pop(); std::swap(st,emptyStack);
6. Queue
   - Use: Managing data in a first-in, first-out (FIFO) manner.
   - Average time complexity: Insertion O(1); Deletion O(1); Search O(N); Index not applicable.
   - #include <queue> std::queue<int> qu; int s = qu.size(); bool b = qu.empty(); int e = qu.top(); qu.push(1); qu.pop(); std::swap(st,emptyQueue);
7. Priority Queue
   - Binary Heap (binary tree): Max Heap or Min Heap (priority_queue <int, vector<int>, greater<int>> gq;)
   - Use: Maintaining a collection of elements with priorities.
   - Average time complexity: Insertion O(log N); Deletion O(log N); Search O(N); Index not applicable.
   - #include <queue> std::priority_queue<int> qu; int s = qu.size(); bool b = qu.empty(); int e = qu.top(); qu.push(1); qu.pop(); std::swap(st,emptyQueue);
8. Map
   - Red-Black Tree (self balancing binary search tree), only the keys are used for ordering
   - Use: Associative storage of key-value pairs with ordered keys.
   - Average time complexity: Insertion O(log N); Deletion O(log N); Search O(log N); Index O(log N).
   - #include <map> std::map<std::string, int> myMap; int s = myMap.size(); bool b = myMap.empty(); int a = myMap["A"]; myMap["A"] = 25; bool b = myMap.count("A"); myMap.clear();
9. Set
   - Red-Black Tree (self balancing binary search tree), only the keys are used for ordering
   - Increasing order (set<int> s;), decreasing order (set<int, greater<int>> s;)
   - Use: Storing a collection of unique elements in sorted order.
   - Average time complexity: Insertion O(log N); Deletion O(log N); Search O(log N); Index not applicable.
   - #include <set> std::set<int> mySet; int s = mySet(); bool b = mySet.empty(); mySet.insert(1); bool b = mySet.count(1); for(auto a:mySet){a;}; auto it = mySet.find(1); bool e = it != mySet.end(); auto it = mySet.lower_bound(2); auto it = mySet.upper_bound(2); mySet.clear();
10. Unordered Map
    - Hash Table, Hash Map, Hash based data structure.
    - Use: Associative storage of key-value pairs with fast average-time operations.
    - Average time complexity: Insertion O(1) average (amortized); Deletion O(1) average (amortized); Search O(1) average (amortized); Access O(1). Hash collisions, load factor, hash function quality, amortized analysis.
    - #include <unordered_map> std::unordered_map<std::string, int> myMap; int s = myMap.size(); bool b = myMap.empty(); int a = myMap["A"]; myMap["A"] = 25; bool b = myMap.count("A"); myMap.clear();
11. Unordered Set
    - Hash Table, Hash Map, Hash based data structure.
    - Use: Storing a collection of unique elements with fast average-time operations.
    - Average time complexity: Insertion O(1) average (amortized); Deletion O(1) average (amortized); Search O(1) average (amortized); Index not applicable. Hash collisions, load factor, hash function quality, amortized analysis.
    - #include <unordered_set> std::unordered_set<int> mySet; int s = mySet(); bool b = mySet.empty(); mySet.insert(1); bool b = mySet.count(1); for(auto a:mySet){a;}; auto it = mySet.find(1); bool e = it != mySet.end(); auto it = mySet.lower_bound(2); auto it = mySet.upper_bound(2); mySet.clear();

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

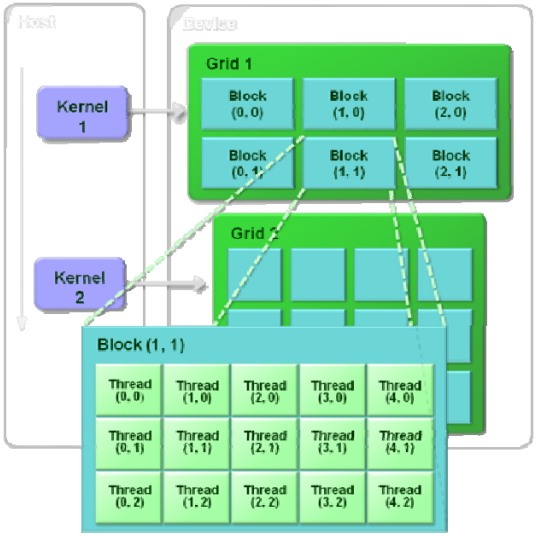
CUDA C Quick Reference

Kernels

kernel <<< dim3 Dg, dim3 Db, size_t Ns, cudaStream_t S >>> (arguments);

Dg.x*Dg.y = number of blocks, Dg.z = 1.
Db.x*Db.y*Db.z = number threads per block.
Ns = dynamically allocated shared memory, optional, default=0.
S = associated stream, optional, default=0.

Thread Hierarchy



Memory Hierarchy

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	Thread	Thread
Local	Off-chip	No	R/W	Thread	Thread
Shared	On-chip	N/A	R/W	Block	Block
Global	Off-chip	No	R/W	Global	Application
Constant	Off-chip	Yes	R	Global	Application
Texture	Off-chip	Yes	R	Global	Application

Device Memory

Linear Memory

cudaMalloc(void ** devptr, size_t size)

cudaFree(void * dptr)

cudaMemcpy(void *dst, const void *src, size_t size, enum cudaMemcpyKind kind)
kind = cudaMemcpyHostToHost or cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice

CUDA Arrays

See Programming Guide for description of CUDA arrays and texture references.

Page-locked Host Memory

cudaMallocHost(void ** ptr, size_t size)

cudaFreeHost(void * ptr)

Shared Memory

Static allocation
__shared__ int a[128]

Dynamic allocation at kernel launch
extern __shared_ float b[]

Error Handling

cudaError_t cudaGetLastError(void)

const char * cudaGetErrorString(cudaError_t error)

CUDA Compilation

nvcc *flags* file.cu

A few common flags
-o output file name
-g host debugging information
-G device debugging
-deviceemu emulate on host
-use_fast_math use fast math library
-arch compile for specific GPU architecture
-X pass option to host compiler

#pragma unroll *n* unroll loop *n* times.

Language Extensions

Function Qualifiers

__global__ call host, execute device.
__device__ call device, execute device.
__host__ call host, execute host (default).
__noinline__ if possible, do not inline

__host__ and __device__ may be combined to generate code for both host and device.

Variable Qualifiers

__device__ variable on device
__constant__ variable in constant memory
__shared__ variable in shared memory

Vector Types

[u]char1, [u]char2, [u]char3, [u]char4
[u]short1, [u]short2, [u]short3, [u]short4
[u]int1, [u]int2, [u]int3, [u]int4
[u]long1, [u]long2, [u]long3, [u]long4
longlong1, longlong2
float1, float2, float3, float4
double1, double2

Execution configuration

kernel <<< dim3 Dg, dim3 Db, size_t Ns, cudaStream_t S >>> (arguments)

Grids are 1D or 2D so Dg.z = 1 always
Ns optional, default 0
S optional, default 0

Built-in Variables

dim3 gridDim size of grid (1D, 2D).
dim3 blockDim size of block (1D, 2D, 3D).
dim3 blockIdx location in grid.
dim3 threadIdx location in block.
int warpSize threads in warp.

Memory Fence Functions

__threadfence(), __threadfence_block()

Synchronisation Function

__syncthreads()

Fast Mathematical Functions

__fdividef(x,y), __sinf(x), __cosf(x), __tanf(x), __sincosf(x,sinptr,cosptr), __logf(x), __log2f(x), __log10f(x), __expf(x), __exp10f(x), __powf(x,y)

Texture Functions

tex1Dfetch(), tex1D(), tex2D(), tex3D()

Timing

clock_t clock(void)

Atomic Operations

atomicAdd(), atomicSub(), atomicExch(), atomicMin(), atomicMax(), atomicInc,(), atomicDec(), atomicCAS(), atomicAnd(), atomicOr(), atomicXor().

Warp Voting Functions

int __all(int predicate)
int __any(int predicate)