System headers

```
#include <iostream>           //For cin & cout
#include <vector>             //For vector
#include <list>               //For list
#include <memory>             //For shared_ptr
#include <thread>             //For thread
#include <mutex>              //For mutex and unique_lock
#include <condition_variable> //For condition_variable
#include <functional>         //For bind/function

using namespace std; //Standard classes are in namspace std
```

Memory allocation

```
void allocTest() {
	int *intOnTheHeap=new int;
	delete intOnTheHeap;

	int *intArrayOnTheHeap=new int[10];
	delete[] intArrayOnTheHeap;
}
```

Classes

```
//Class definition, ususally in a .h header file

class TestClass
{
public:
	TestClass();
	TestClass(const TestClass& other);
	TestClass& operator= (const TestClass& other);
	~TestClass();

	void set(int i) {
		//Short member functions can be declared inside class as a convenience
		this->i=i; //Note that this is a pointer
	}

	int get() const;
private:
	int i;
}; //Note the ; at the end

//Class implementation, usually in the corresponding .cpp source file

TestClass::TestClass() : i(0) {
	cout<<"Constructor called"<<endl;
}

TestClass::TestClass(const TestClass& other) {
	//Copy constructor, the default implementation would have done the same thing
	i=other.i;
}

TestClass& TestClass::operator= (const TestClass& other) {
	//Operator=, the default implementation would have done the same thing
	i=other.i;
	return *this; //By convention operator= always return a reference to this
}

TestClass::~TestClass() {
	cout<<"Destructor called"<<endl;
}

int TestClass::get() const { //Note that const is part of the function signature
	return i;
}
```

Templates

```
//Template function, need to go in header file, as the compiler needs to see the entire
//function body to specialize the template
template<typename T>
T duplicate(T param) {
	return param*2;
}

//Calling template function
void testTemplate() {
	cout<<duplicate(2)<<endl;   //T=int
	cout<<duplicate(2.5)<<endl; //T=double
}

//Template class, goes in .h header file
template<typename T>
class TemplateClass
{
public:
	void set(const T& data);
	T get() const {
		//Short member functions can be declared inside class as a convenience
		return data;
	}
private:
	T data;
};

//Non inline template member functions, still need to go in header file
template<typename T>
void TemplateClass<T>::set(const T& data) {
	this->data=data;
}

//Calling template class
void testTemplate2() {
	TemplateClass<string> ts;
	TemplateClass<int> ti;
	ts.set("hello");
	ti.set(2);
}
```

STL

```
void testVector() {
	vector<int> vi1;         //Empty vector
	vector<int> vi2={1,2,3}; //Pre-initialized vector

	vi2.push_back(4);        //Adding elements (to the end)

	cout<<vi2[2]<<endl;      //Random access (without baound checking)
	cout<<vi2.at(10)<<endl;  //Random access (bound checked), throws exception

	vector<int>::iterator it; //Iterating with iterator
	//Note the != in the end condition, NEVER use < with iterators
	for(it=vi2.begin();it!=vi2.end();++it) {
		int number=*it; //Dereference iterator to get the element
		cout<<number<<endl;
	}

	//Iterating with random access
	for(int i=0;i<vi2.size();i++) cout<<vi2[i]<<endl;

	cout<<vi2.front()<<endl; //Quick access to first element
	cout<<vi2.back()<<endl;  //Quick access to last element

	vi2.pop_back(); //Erasing elements (from the end), element is NOT returned
}
```

```
void testList()
{
	list<int> li1;		//Empty list
	list<int> li2={1,3,4}; //Pre-initialized list

	li2.push_back(5);  //Adding elements (to the end)
	li2.push_front(0); //Adding elements (at the beginning)

	list<int>::iterator it=li2.begin(); //it points to 0
	it++;				    //it points to 1
	it++;				    //it points to 3
	li2.insert(it,2); //insert element before current iterator position

	//List has NO random access, only iterators
	//Note the != in the end condition, NEVER use < with iterators
	for(it=li2.begin();it!=li2.end();++it) {
		int number=*it; //Dereference iterator to get the element
		cout<<number<<endl;
	}

	cout<<li2.front()<<endl; //Quick access to first element
	cout<<li2.back()<<endl;  //Quick access to last element

	li2.pop_front(); //Erasing elements (from the beginning), element is NOT returned
	li2.pop_back();  //Erasing elements (from the end), element is NOT returned
}
```

Smart pointers

```
void testSharedPtr() {
	//Works, but long to write
	shared_ptr<string> ptr1=make_shared<string>("hello world");

	//using C++11 auto keyword
	auto ptr2=make_shared<string>("hello world");
}
```

Threads

```
void threadFunc(int i) {
	cout<<"Inside a new thread "<<i<<endl;
}

void createThread() {
	thread t(threadFunc,10);
	t.join();
}
```

Mutex

```
mutex myMutex;

void criticalSection() {
	//Here the mutex is not locked
	{
		unique_lock<mutex> lock(myMutex);
		//Here the mutex is locked
	}
	//Here the mutex is not locked
}
```

Condition variable

```
condition_variable myCv;

void testConditionVariableWait() {
	unique_lock<mutex> lock(myMutex);
	myCv.wait(lock);
}
```

```
void testConditionVariableNotify() {
	unique_lock<mutex> lock(myMutex);
	myCv.notify_one(); //Wakes only one of the waiting threads (if any)
	myCv.notify_all(); //Wakes all the waiting threads (if any)
}
```

Bind/function

```
void printNumber(int i, int j, int k) {
	cout<<"i="<<i<<" j="<<j<<" k="<<k<<endl;
}

void testBind() {
	//Binding all parameters of original function leaves a function without arguments
	function<void ()> fn1=bind(&printNumber,1,2,3);
	fn1(); //prints i=1 j=2 k=3

	//Binding all parameters of original function but the last, which is taken
	//from the first parameter of the resulting function
	function<void (int)> fn2=bind(&printNumber,1,2,placeholders::_1);
	fn2(4); //prints i=1 j=2 k=4

	//Binding:
	//1 to first parameter of original function,
	//the first and second parameter of the resulting function, swapped, to the
	//other two parameters of the original function
	//ignoring the third parameter of the resulting function
	function<void (int,int,int)>
	fn3=bind(&printNumber,1,placeholders::_2,placeholders::_1);
	fn3(2,3,4); //prints i=1 j=3 k=2
}
```

Processes

```
#include <sys/types.h>        //For pid_t
#include <unistd.h>           //For fork, exec*, pipe
#include <signal.h>           //For kill
#include <sys/wait.h>         //For wait, waitpid
#include <sys/stat.h>         //For mkfifo
#include <sys/mman.h>         //For shm_open, mmap
#include <semaphore.h>        //For sem_* functions

pid_t fork(void);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int kill(pid_t pid, int sig);
struct sigaction {
	void     (*sa_handler)(int);
	//[...]
};
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int pipe(int pipefd[2]);
int mkfifo(const char *pathname, mode_t mode);
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```