```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
   int freq[MAXPAROLA] ; /* vettore di contatori
   delle frequenze delle lunghezze delle parole */
   char riga[MAXRIGA] ;
   int i, inizio, lunghezza ;
   FILE * f ;

   for(i=0; i<MAXPAROLA; i++)
      freq[i]=0 ;

   if(argc != 2)
   {
      fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
      exit(1);
   }
   f = fopen(argv[1], "rt") ;
   if(f==NULL)
   {
      fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
      exit(1);
   }

   while( fgets( riga, MAXRIGA, f ) != NULL )
```

Multi-Threading

Multi-Threading in C++

Stefano Quer and Alessandro Savino

Dipartimento di Automatica e Informatica

Politecnico di Torino

Introduction

- The header thread defines the class std::thread that can be used to start new threads
  - Using this class is the best way to use platform-independent threads

The std::thread class encapsulates many of the details of thread management, making it easier to work with threads compared to using platform-specific APIs like POSIX threads.

```
#include <thread>
```

  - Using it may require additional compiler flags
    - For gcc and clang, use -pthread

```
set(CMAKE_CXX_FLAGS
  "${CMAKE_CXX_FLAGS} -std=c++14 -pthread")
```

Introduction

- The library is based on objects of type std::thread
  - The operator std::thread works with any callable object like a function, an instance of a class, a lambda expression

```
void threadFunction(int id) {
   cout << "Thread id: " << id << endl;
}


int main() {
   // Create a thread and start it with threadFunction
   thread t1(threadFunction,
           1);// The two arguments are the function to be called and the arguments to be passed to that function.

   // join the thread with the main thread
   t1.join(); // The main thread waits for the thread to finish its execution.
   cout << "Print this after the main thread has joined the thread t1" << endl;
```

Main thread primitives

- The library covers all main functionalities
  - But … there is no way to automatically capture the data computed by a thread

Type	Main characteristics
std::thread t;	Creates a thread object t.
std::thread t(f);	Creates a thread object t associated with the thread function f.

std::thread t(f,p1,p2,…);
std::thread t{f(p1,p2,…)};
	Creates a thread object t associated with the thread function f which receives the parameters p1, p2, etc.

t2=std::move(t1);	Move the thread associated to the thread object t1 to object t2. Moves the thread associated with the thread object t1 to object t2. After this operation, t1 no longer owns the thread, and t2 becomes the owner.
t.detach()	Makes the thread t as detached. Detaches the thread associated with t. This means that the thread can continue to execute independently of the thread object. After detaching, the thread is no longer joinable.
t.join()	Waits the thread t for joining. Waits for the thread associated with t to finish its execution. This blocks the current thread until the thread associated with t completes its execution. After joining, the thread object t is no longer associated with any thread.

Other primitives

- The thread library also contains useful functions related to starting and stopping threads

Type	Main characteristics
std::this_thread::sleep_for	Stop the current thread for a given amount of time.
std::this_thread::sleep_until	Stop the current thread until a given point in time.
std::this_thread::yield	Let the operating system schedule another thread. The term "yield" in the context of std::this_thread::yield refers to a mechanism that allows a thread to voluntarily give up its current time slice or CPU execution time to allow another thread to run.
std::this_thread::get_id	Get the (operating-system-specific) id of the current thread. This function returns an object representing the identifier of the current thread. This identifier is specific to the operating system and can be used to uniquely identify the thread
std::thread::hardware_concurrency	Reports the actual max number of threads based on your architecture.

This function returns an estimate of the maximum number of concurrent threads that can be executed simultaneously on the current hardware architecture.

Examples

```
void f1() {
   ...
}

void f2(int a, int b) {
   ...
}
```

Function definitions (without and with parameters)

```
std::thread t1;

std::thread t2(f1);

std::thread t3(f2, 123, 456);

std::thread t4([] { f2(123, 456); });
```

Creates an object that does not refer to a thread

Starts an object thread that calls f1()

Starts a thread that calls f2(123, 456)

Works also with lambda functions

creates a thread object t4 that is associated with the execution of a lambda function. The lambda function itself calls the function f2 with parameters 123 and 456. This is a way to start a thread with a function call and parameters without explicitly defining a separate function.

Join and Move

- The member function **join** can be used to wait for a thread to finish
  - Function **join** must be called exactly once for each thread
- Standard threads are not copyable, but movable, so they can be used in containers
  - Moving an **std::thread** transfers all resources associated with the running thread
  - Only the new thread can be joined

Move semantics: Standard threads in C++ are not copyable, meaning you can't make a direct copy of a thread. However, they are movable. This means you can transfer ownership of a thread from one variable to another. When you move a thread, all resources associated with the running thread are transferred to the new owner. The original thread object no longer represents a running thread of execution, and thus can't be joined. Only the new owner of the thread can call join() on it.

Examples

```
void f1() {
   ...
}

void f2(int a, int b) {
   ...
}
```

Function definitions
(without and with parameters)

```
std::thread t1;
std::thread t2(f2, 123, 456);
// Alternative syntax
std::thread t3{f2(123, 456)};

t1 = std::move(t2);
t1.detach();
t3.join();
```

Starts a thread that
calls f2(123, 456)

std::thread is moveable not copyable
Associate f2 to thread object t1

Examples

```
void f(int &result) {
   ...
   result = ...;
}
```

Function definition
(with a paramter by reference)

```
int main() {
   ...
   std::thread t (f, std::ref(i));
   ...
   t.join();
}
```

Parameter by reference

Variable i will assume the value once the execution is terminated

Examples

```
std::thread t1([] { std::cout << "Hi\n"; });

std::thread t2 = std::move(t1);

t2.join();
```

t1 is now empty

The thread originally started in t1 is joined

```
std::thread t1(some_function);
std::thread t2 = std::move(t1);
std::thread t1(some_other_function);
std::thread t3;
t3 = std::move(t2);
t1 = std::move(t3);
```

Error: t1 is running some_other_function
The assignment terminates the program

Examples

The output operation shoud be protected

```
void my_thread () {
  std::cout << "TID" <<
    std::this_thread_::get_id() << endl;
}

main() {
  std::thread t1{mythread};
  std::thread t2{mythread};

  t1.join();
  t2.join();

  return 1;
}
```

Examples

```
void safe_print (int i) {
  ... Enter critical section ...
  std::cout << i;
  ... Exit critical section ...
  return;
}
```

For example, use a mutex (see, unit 06)

```
std::vector<std::thread> threadPool;

for (int i = 1; i <= 9; ++i) {
  threadPool.emplace_back([i] { safe_print(i); });
}

for (auto& t : threadPool) {
  t.join();
}
```

Digits 1 to 9 are printed (unordered)