```
#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 )
```

High Level Programming

Exercises on C++ Libraries

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Exercise

- What does the keyword “const” do?
  1. It allows to keep search time within a container nearly constant
  2. It prevents objects to be casted
  3. It allows to keep execution time of a member function nearly constant
  4. It prevents variables to be copied (i.e., assigned) to other variables
  5. [✓ It prevents objects from being mutated]

Solution

- What does the keyword “const” do?
  1. It allows to keep search time within a container nearly constant
  2. It prevents objects to be casted
  3. It allows to keep execution time of a member function nearly constant
  4. It prevents variables to be copied (i.e., assigned) to other variables
  5. It prevents objects from being mutated

Exercise

- Which is the thing in common between passing parameters by address and by reference?
  - ✗ They both create a copy of the passed object
  - In both cases, deferencing the operator is necessary to access data
  - They both call a move constructor
  - ✓ In both cases it is possible to modify the data
  - They both free the resources of the passed object

Move constructors are called when transferring ownership of an object, typically in move semantics.

Dereferencing is the process of accessing the value stored at a memory address pointed to by a pointer. we use the dereference operator (*). So, *ptr gives us the value of num.

The responsibility for managing resources lies with the code that creates or allocates the object, not with how it's passed to a function.

2 is incorrect: In passing by address, you need to dereference the pointer to access the data, but in passing by reference, you don't need to dereference anything; you can access the data directly using the reference.
Modify by ref (int &ref) ref = 30;
Modify by address(int* ptr) *ptr = 20;

Solution

- Which is the thing in common between passing parameters by address and by reference?
  - They both create a copy of the passed object
  - In both cases, deferencing the operator is necessary to access data
  - They both call a move constructor
  - In both cases it is possible to modify the data
  - They both free the resources of the passed object

Exercise

- Analyze the following program
- Which is the output generated?

```
#include <functional>
#include <iostream>
using namespace std;
int main() {
  int i = 3;
  int j = 5;
  function<int (void)>
    f = [&i, j] { return i + j; };
  i = 22;
  j = 44;
  cout << f() << endl;
}
```

22(value changes cuz ref passed in capture list) + 5(doesn't change)
= 27

Solution

- Analyze the following program
- Which is the output generated?

```
#include <functional>
#include <iostream>
using namespace std;
int main() {
  int i = 3;
  int j = 5;
  function<int (void)>
    f = [&i, j] { return i + j; };
  i = 22;
  j = 44;
  cout << f() << endl;
}
```

```
27
```

Exercise

- Analyze the following program
- Which is the final value of the variable b?

```
#include <iostream>

int a = 10;    global variable a initialized to 10

int main() {
  int& b = a, a;   okay, so we have a reference &b = a
                   and a local variable a;
  a = 0; local variable a is set to 0
  b++;   the reference was to the global variable, so the value of the global variable is incremented.
}
```

Solution

- Analyze the following program
- Which is the final value of the variable b?

```
#include <iostream>

int a = 10;

int main() {
  int& b = a, a;
  a = 0;
  b++;
}
```

```
11
```

Exercise

- Analyze the following program
- Which is the final value of the variable b?

```
#include <iostream>

int main() {
  int a = 10;
  int& b = a;
  auto f = [=](){return a+b;};

  a = 0;
  b+=f();
  return 1;
}
```

= means all the variables are captured by value. So any changes that take place after will not affect the variables.

since b is referenced to this b = 0

b = b+ f()
b = 0 + (10+10) = 0 + 20 = 20

Solution

- Analyze the following program
- Which is the final value of the variable b?

```
#include <iostream>

int main() {
  int a = 10;
  int& b = a;
  auto f = [=](){return a+b;};

  a = 0;
  b+=f();
  return 1;
}
```

```
20
```

Exercise

The RAII (Resource Acquisition Is Initialization) paradigm in C++ is a technique where the lifetime of a resource (such as memory, file handles, locks, etc.) is tied to the lifetime of an object. Resources are acquired during object initialization, typically in the constructor, and released automatically when the object goes out of scope, typically in the destructor.

- The RAII paradigm guarantees that ...
  - The resources are deleted before initialization and then new ones allocated
  - The resources are copied at object initialization so that objects can have independent lives
  - The resources are acquired at object initialization and released when its lifetime ends
  - The resources are allocated when a new operator is used
  - The resources are copied when objects are passed as parameters, to keep original data safe

Solution

- The RAII paradigm guarantees that ...
  - The resources are deleted before initialization and then new ones allocated
  - The resources are copied at object initialization so that objects can have independent lives.
  - The resources are acquired at object initialization and released when its lifetime ends
  - The resources are allocated when a new operator is used
  - The resources are copied when objects are passed as parameters, to keep original data safe

Exercise

- Analyze the following program
- When (and why) the default constructor and assignment operators are called

Exercise

```
class Y {
private: int i;
public:
  Y () { ... }                          // Constructor
  ~Y() { ... }                          // Destructor
  Y (const Y &n) { ... }                // Copy Constructor
  Y &operator=(const Y &n) {            // Copy Assignement Operator
    ... return *this;
  }
  Y (Y&& n) noexcept { ... }            // Move Constructor
  Y &operator=(Y&&n) noexcept {        // Move Assignment Operator
    ... return *this;
  }
  void set(int n) {i = n;};
  int get () {return i;}
};
```

Exercise

```
void f1(Y y) { y.set(5); }
void f2(Y &y) { int n = y.get(); }

   int main() {
1.     Y y1;   Default constructor is called here
2.     Y y2=y1;   Copy constructor is called to create y2 from y1
3.     f1(y1);  y1 is passed by value which means a copy is made, so copy constructor is called again
4.     f1(std::move(y1));   Casts y1 to an rvalue reference, which triggers the move constructor instead of a copy constructor.
5.     return 0;
   }
```

Y y2 = y1; uses copy initialization and invokes the copy constructor.
y2 = y1; uses copy assignment and invokes the copy assignment operator.

Exercise

```
void f1(Y y) { y.set(5); }
void f2(Y &y) { int n = y.get(); }

   int main() {
1.    Y y1;
2.    Y y2=y1;
3.    f1(y1);
4.    f1(std::move(y1));
5.    return 0;
   }
```

```
{1}  [Constructor]
{2}  [Copy Constructor]
{3}  [Copy Constructor]
{f1} [Destructor]
{4}  [Move Constructor]
{f1} [Destructor]
{5}  [Destructor]
     [Destructor]
```

I don't understand why destructor is called 4 times. Check vid or ask prof!!

Exercise

- Analyze the following program
- When (and why) the default constructor and assignment operators are called

Exercise

```
class Y {
private: int i;
public:
  Y () { ... }                            // Constructor
  ~Y() { ... }                            // Destructor
  Y (const Y &n) { ... }                  // Copy Constructor
  Y &operator=(const Y &n) {              // Copy Assignement Operator
    ... return *this;
  }
  Y (Y&& n) noexcept { ... }              // Move Constructor
  Y &operator=(Y&&n) noexcept {          // Move Assignment Operator
    ... return *this;
  }
  void set(int n) {i = n;};
  int get () {return i;}
};
```

Exercise

```
void f1(Y y) { }
Y f2(Y &y) { Y ay; return ay; }
void f3(Y y1, Y &y2){ }

   int main() {
1.    Y y1, y2, y3;
2.    y1=y2;
3.    f3(y1, y3);
4.    Y y4 = f2(y1);
5.    return 0;
   }
```

default constructor is called three times

copy assignment operator is called

Since, y is being passed by value copy constructor is called once.
Since, y2 is being passed by reference nothing is called

Default constructor is called for y4,
since y1 is being passed by ref copy constructor is not called.
In F2:
Copy constructor is called (ay) (this is not indicated in the ans? why
Copy constructor is called for Y y4 = ay(returned by function)

side note: After a copy assignment operation, the two objects are independent of each other.

Default constructor
Default constructor
Default constructor
Copy assignment operator
Copy constructor
Default constructor
Copy constructor (in f2 for ay not mentioned in ans why?)
Copy constructor

Exercise

```
void f1(Y y) { }
Y f2(Y &y) { Y ay; return ay; }
void f3(Y y1, Y &y2){ }

   int main() {
1.    Y y1, y2, y3;
2.    y1=y2;
3.    f3(y1, y3);
4.    Y y4 = f2(y1);
5.    return 0;
   }
```

```
{1}  [Constructor]
     [Constructor]
     [Constructor]
{2}  [Copy Assignment Op]
{3}  [Copy Constructor]
{f3}[Destructor]
{f2}[Constructor]
{5}  [Destructor]
     [Destructor]
     [Destructor]
     [Destructor]
```

Exercise

- Analyze the following program
- When (and why) the default constructor and assignment operators are called

Exercise

```
class Y {
private: int i;
public:
  Y () { ... }                          // Constructor
  ~Y() { ... }                          // Destructor
  Y (const Y &n) { ... }                // Copy Constructor
  Y &operator=(const Y &n) {            // Copy Assignement Operator
    ... return *this;
  }
  Y (Y&& n) noexcept { ... }            // Move Constructor
  Y &operator=(Y&&n) noexcept {        // Move Assignment Operator
    ... return *this;
  }
  void set(int n) {i = n;};
  int get () {return i;}
};
```

Exercise

```
void f1(Y y) { }
void f2(Y &y) { }

   int main() {
1.    Y y1;
2.    f1(y1);
3.    f2(y1);
4.    Y *y2 = new Y;
5.    Y y3;
6.    y3 = (std::move(y1));
7.    return 0;
}
```

Exercise

```
void f1(Y y) { }
void f2(Y &y) { }

   int main() {
1.     Y y1;
2.     f1(y1);
3.     f2(y1);
4.     Y *y2 = new Y;
5.     Y y3;
6.     y3 = (std::move(y1));
7.     return 0;
}
```

y1: Default constructor is called to initialize y1.

f1 takes its parameter by value, so the copy constructor of Y is called to copy y1 into the parameter y of f1.
The destructor of Y is called for the parameter y when f1 returns.

f2 takes its parameter by reference, so no constructors, destructors, or copy/move operations are called.

new Y: Allocates a new Y object on the heap, calling the default constructor

Default constructor is called to initialize y3.

y3: Default constructor is called to initialize y3.
Move assignment operator is called to move the contents of y1 into y3.

At the end of main, destructors are called for all local objects (y1 and y3).
The dynamically allocated object (y2) is not automatically destroyed. To avoid memory leaks, you should delete the dynamically allocated object:

```
{1}  [Constructor]
{2}  [Copy Constructor]
{f1} [Destructor]
{4}  [Constructor]
{5}  [Constructor]
{6}  [Move Assigment Op]
{7}  [Destructor]
     [Destructor]
```

Exercise

- Analyze the following program
- Which is the output generated?

```
#include <iostream>
#include <vector>
class A {
public:
  ~A() { std::cout << "*"; }
  A() { std::cout << "a"; }
  A(const A&) { std::cout << "&"; }
};
int main() {
  int i=3;
  std::vector<A> v(i);
}
```

since i = 3, the default constructor is called three times for class A (so prints a three times) aaa
When main ends the destructor is called three times so ***
Hence -> aaa***

Solution

- Analyze the following program
- Which is the output generated?

```
#include <iostream>
#include <vector>
class A {
public:
  ~A() { std::cout << "*"; }
  A() { std::cout << "a"; }
  A(const A&) { std::cout << "&"; }
};
int main() {
  int i=3;
  std::vector<A> v(i);
}
```

This is wrong, answer is aaa*** checked by running the code

```
a***
```