```
#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

Functions

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Functions

- Functions in C++ have many similarities with C
- The main exceptions are the following
  - Argument passing
    - By value, address (pointer), reference
  - Varying parameters
    - Number of parameters
  - Overloading
  - Default arguments
  - Pointers to functions

Argument passing

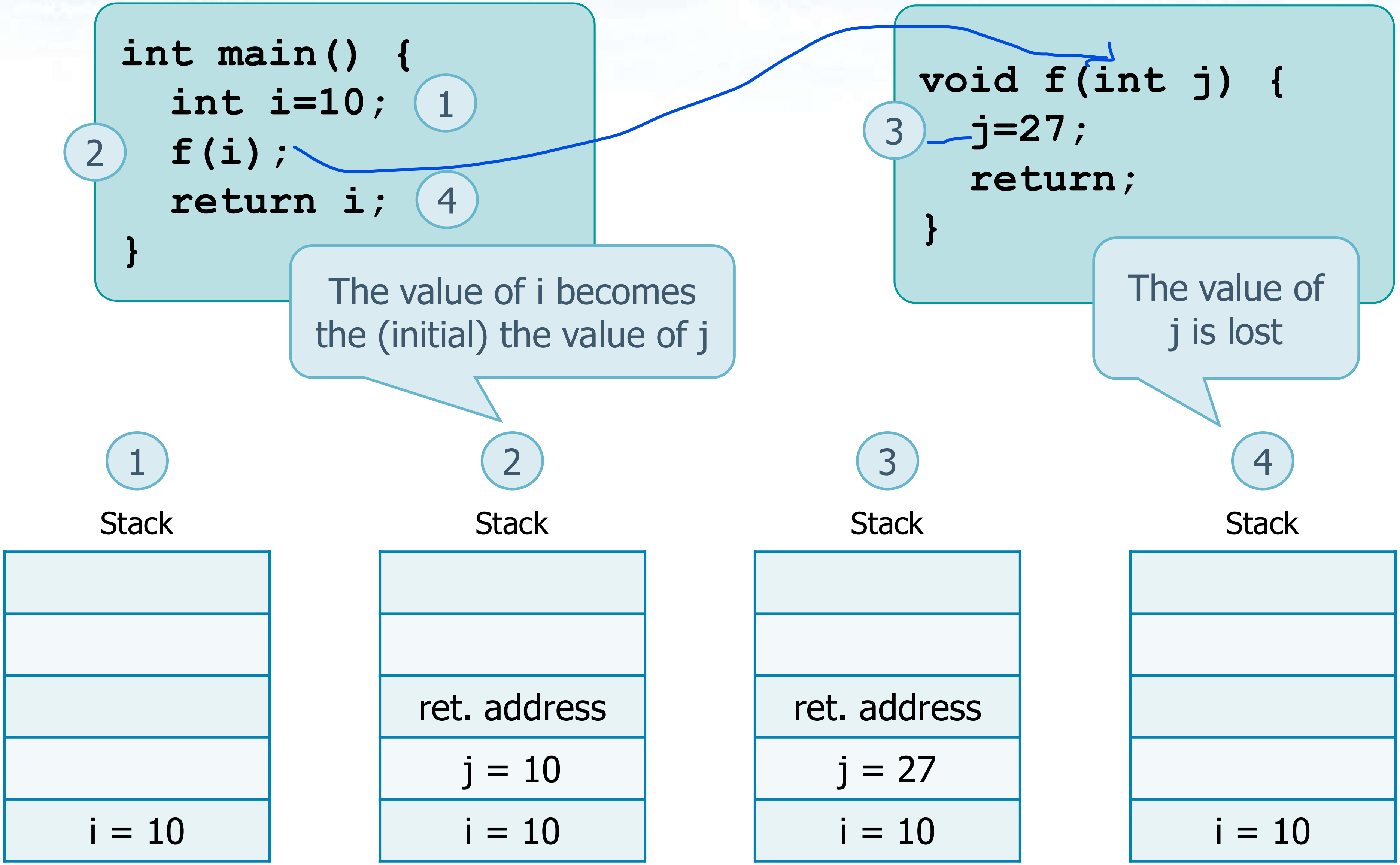
- In C++ functions may have 3 different types of parameters
  - By value
  - By address (pointer)
  - By reference
- The last one is not present in C

In C, these were parameters by reference

Arguments by value

- For the arguments passed by value
  - We provide a value to the function parameter when the function is called
    - The local parameter is equivalent to a local variable
    - This variable holds a **copy** of the parameter
  - Changes to the local parameter will not affect the original variable

Example



```
int main() {
  int i=10;    1
2 f(i);
  return i;    4
}
```

```
void f(int j) {
3 j=27;
  return;
}
```

The value of i becomes the (initial) the value of j

The value of j is lost

1 Stack	2 Stack	3 Stack	4 Stack
	ret. address	ret. address	
	j = 10	j = 27	
i = 10	i = 10	i = 10	i = 10

Arguments by address

- Pointers behave like any other type
  - To pass a pointer "by value," we copy the pointer
    - After the copy, the two pointers are distinct
    - However, the pointer may give indirect access to the object to which it points
  - By dereferencing the address, the function may access and modify the original data
    - Passing a **pointer by value** implies passing the pointed **object by reference**
    - Again, a copy of the address is done, not a copy of the data referenced by the address
  - Unfortunately, an address can be **nullptr**, thus a variable by reference can be **invalid**

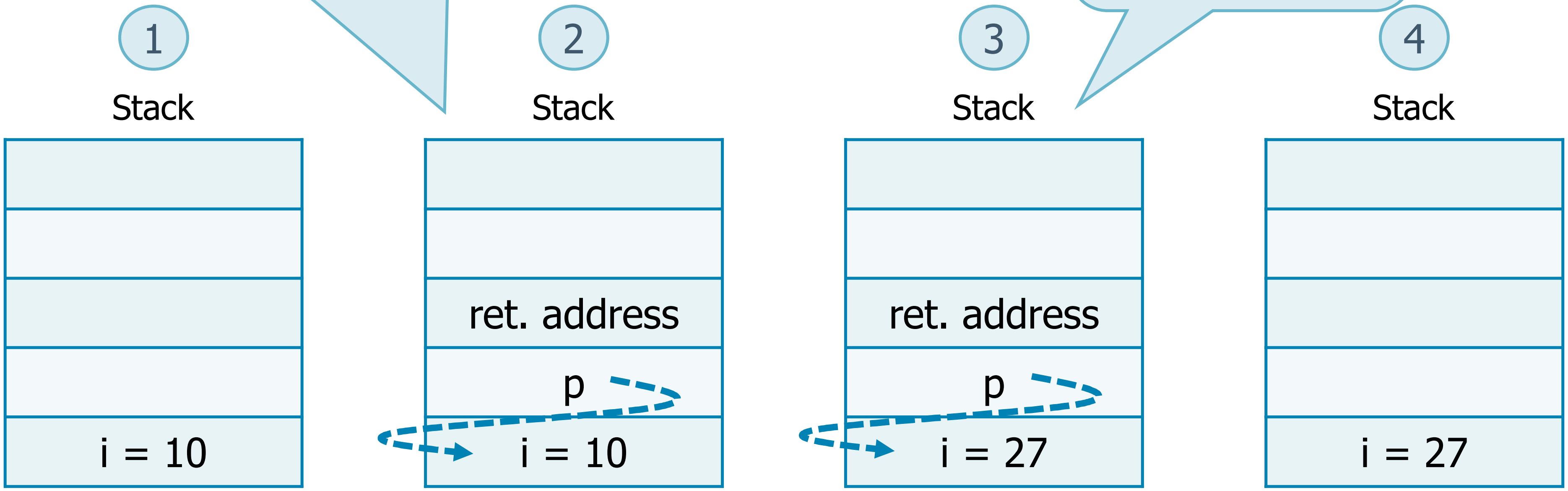
Example

```
  int main() {
    int i=10;    1
2   f(&i);
    return i;    4
  }
```

```
  void f(int *p) {
3   if (p!=nullptr)
      *p = 27;
  }
```

Access to the location referenced by the pointer p

The new location does not contain an integer but a pointer



Arguments by reference

New parameter passing strategy

- With a parameter by
  - Value, we may need to **copy** a lot of memory, and we cannot modify extern objects
  - Address, we can have **null** pointers
- With a parameter by **reference**, we pass a pointer to a verified variable
  - The parameter is accessed directly without dereferencing the pointer
    - The syntax is simpler
  - We never have **nullptr** pointers

Example

```
  int main() {
     int i=10;     1
2    f(i);
     return i;     4
  }
```

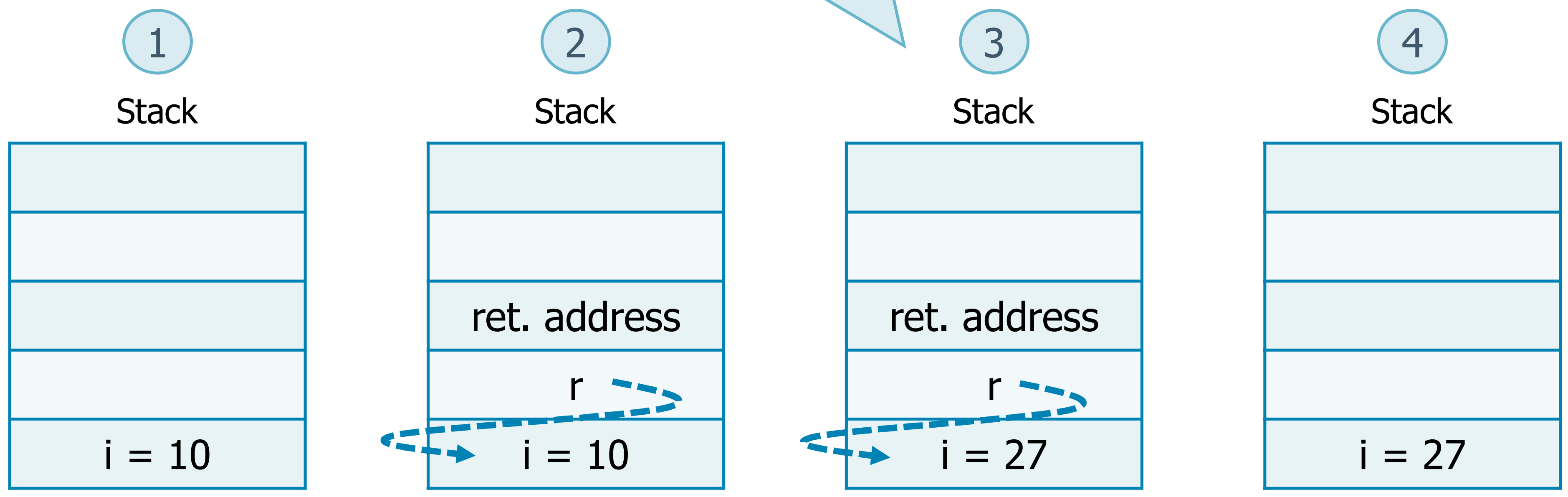
No &

No * but &

```
  void f(int &r) {
3   r = 27;
  }
```

r is another name for i

The code is "by value"
The effect is "by reference"



Examples

```
void reset (int &i) {
  i=0;
}

...

int j=10;
reset (j);
cout << j;
```

Passed by reference

The value is 0

We will see strings in Unit 04

```
bool is_shorter (const string &s1, const string &s2) {
  return s1.size() < s2.size();
}

...

if (is_shorted (s1, s2)) ...
```

It can be inefficient to copy large objects. Moreover, some objects cannot be copied. In those cases, we can use references

Example

The classical **swap** function

```
void swap(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 10;
  int secondNum = 20;

  cout << "Before swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  // Call the function swap
  swap(firstNum, secondNum);

  cout << "After swap: " << "\n";
  cout << firstNum << secondNum << "\n";

  return 0;
}
```

Constant parameters

- Parameters that the function does not change should be defined as constant
  - Not doing that would give the impression that the function does modify the parameter

```
void find_char (string &s) { ... }
...
find_char ("hello word");
```

This is an error
We must define the
string as const

```
void find_char (const string &s) { ... }
```

Check documentation
for more details

Varying parameters

- Like in C it is possible to write functions with a variable number of parameters
- In the C++ standard this is possible using two strategies
  - If all arguments have the same type, it is possibile to use the library **initializer_list**
  - Otherwise, it is possibile to use a special parameter type called **ellipsis**

Reported for the sake of completeness. You may ignore it !

Overloading

- In C++, it is possible to have multiple definitions for the same function in the same scope
- Overloaded functions have
  - The same name
  - Different parameter lists
  - Appear in the same scope region
- Overloading
  - Eliminates the necessity to remind different names
  - Is implemented by the compiler
    - The compiler calls the function that best matches the actual argument list

Overloading

- The definitions of the function must differ from each other for
  - The types of its arguments
  - The number of its arguments
  - You **cannot** overload function declarations that differ only by the return type

Examples

- The following functions perform the same action but on different object types
  - Definitions (and declarations)

```
void print (char c) { ... }
void print (char *s) { ... }
void print (int v[], int n) { ... }
```

  - Function calls

```
print (‘A’);
print ("string"):
print (v, size);
```

Example

Function overloading

```
int plus_func(int x, int y) {
  return x + y;
}

double plus_func(double x, double y) {
  return x + y;
}

int main() {
  int my1 = plus_func(8, 5);
  double my2 = plus_func(4.3, 6.26);
  cout << "Int: " << my1 << "\n";
  cout << "Double: " << my2;
  return 0;
}
```

Default arguments

- Functions may have parameters that have a particular value in most, but not all, calls
- In those cases, we can declare that value as a default argument
  - Each parameter can have a single default value in a given scope
  - If a parameter has a default argument, **all** the parameter that follow it **must** also have a default value

Example

- The following functions perform the same action but on different object types
  - Declarations

```
void myf (int i, int j, char c);
void myf (int i, int j, char c=‘a’);
void myf (int i, int j=20, char c=‘a’);
void myf (int i=10, int j=20, char c=‘a’);
void myf (int i=10, int j=20, char c);
```

No default

Error: If a parameter has a default value all parameters that follow it must have a default value

Example

- The following functions perform the same action but on different object types
  - Declarations

```
void myf (int i, int j, char c);
void myf (int i, int j, char c=‘a’);
void myf (int i, int j=20, char c=‘a’);
void myf (int i=10, int j=20, char c=‘a’);
void myf (int i=10, int j=20, char c);
```

Error

  - Calls

```
myf ();
myf (12):
myf (12, 34):
myf (12, 34, ‘z’);
```

```
myf (10, 20, ‘a’);
myf (12, 20, ‘a’):
myf (12, 34, ‘a’):
myf (12, 34, ‘z’);
```

Example

Default arguments

```
#include<iostream>
using namespace std;

int sum(int x, int y, int z=0, int w=0) {
  return (x + y + z + w);
}

int main() {
  cout << sum(10, 15) << endl;
  cout << sum(10, 15, 25) << endl;
  cout << sum(10, 15, 25, 30) << endl;
  return 0;
}
```

Output

```
25
50
80
```

Pointers to functions

- A function pointer is just like any other pointer but is denotes a function
- The name of a function is automatically converted into the function pointer
- Function pointer can be
  - Passed to a function as a parameter
  - Returned by a function

Reported for the sake of completeness. Some example may follow.