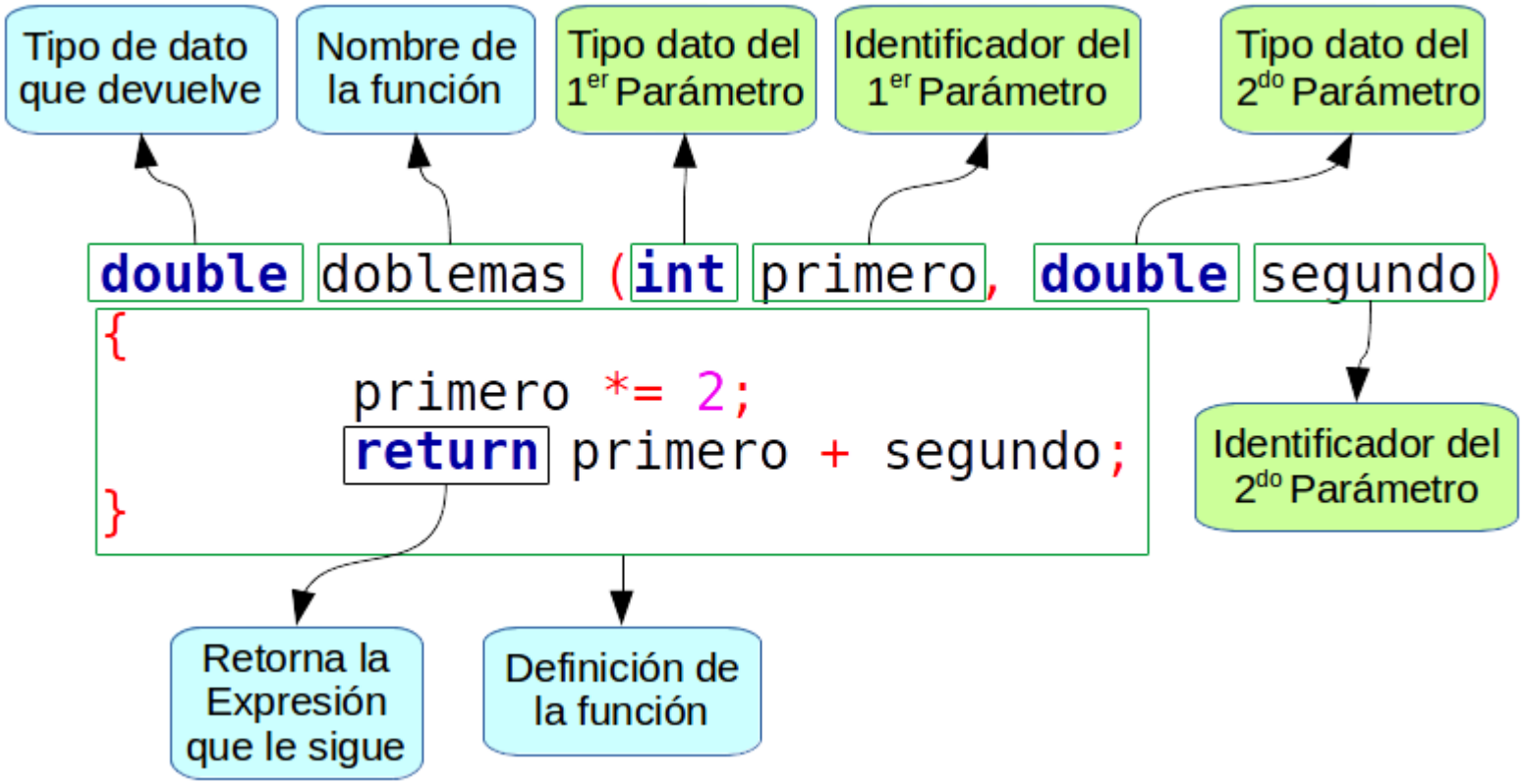
Separación del código en partes

- Muchas veces hay código que se repite, por ejemplo un cálculo particular o una búsqueda en una estructura determinada.
- El código que se repite se lo aparta en una subrutina que luego es invocada en los lugares donde haga falta. Los motivos para hacer esto son:
  - Achicar el código (fuente y ejecutable)
  - Mantenimiento y corrección de errores: modifico o corrijo en un único lugar.
- Algunos distinguen las subrutinas según devuelvan un resultado o no. En C++ este concepto se implementa mediante funciones, que siempre devuelven un valor, pero este puede ser “vacío”.

Funciones (implementación)

- La definición de una función tiene la siguiente forma:



Funciones (declaración)

- La declaración es similar a la definición con dos diferencias
  - El código encerrado entre llaves se reemplaza por punto y coma:

```
double doblemas (int primero,
                 double segundo);
```

  - Los identificadores de los parámetros son optativos, pueden no estar pero es una buena práctica ponerlos (como auto documentación)
- Antes de usar una función, al igual que con una variable, esta debe haber sido definida o al menos declarada.

Parámetros

- Al invocar una función se debe pasar la misma cantidad de parámetros que esta tiene definidos y cada uno con el tipo de dato correspondiente.
- Se hace corresponder por posición, al primer parámetro formal (el definido en la función) se le asigna el primer parámetro real (también llamado actual), es decir el usado en la invocación

```
doblemas(int primero, double segundo) { …

resul = doblemas(nro, sumando);
```

Parámetros – tipos de pasaje

- En C++ hay dos tipos de pasaje de parámetros: por valor y por referencia
- Por Valor:
  - El parámetro formal recibe una **copia** del valor del parámetro real. Esto permite que el parámetro real, además de una variable, pueda ser una constante o una expresión. Las modificaciones hechas al parámetro formal durante la ejecución de la función no afectan al parámetro real.
- Por referencia:
  - Requiere que el parámetro real sea una variable. El parámetro formal pasa a ser un “alias” del parámetro real. Durante la ejecución de la función los cambios realizados en el parámetro formal se hacen realmente con el parámetro actual.

Pasaje por Valor

```
#include <iostream>

using namespace std;

double doblemas (int primero, double segundo)
{
        primero *= 2;
        return primero + segundo;
}

int main()
{
        int nro = 2;
        double sumando {5.4};
        cout << "nro = " << nro << " - sumando = " << sumando << endl;
        //nro = 2 - sumando = 5.4
        cout << "doblemas(nro, sumando) = " <<  doblemas(nro, sumando) << endl;
        //doblemas(nro, sumando) = 9.4
        cout << "nro = " << nro << " - sumando = " << sumando << endl;
        //nro = 2 - sumando = 5.4 (nro sige siendo 2, no 4)
        cout << "doblemas(nro*3, 3.2) = " <<  doblemas(nro*3, 3.2) << endl;
        //doblemas(nro*3, 3.2) = 15.2
        return 0;
}
```

Pasaje por Referencia


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

double doblemas (int& primero, double& segundo)
{
        primero *= 2;
        return primero + segundo++;
}

int main()
{
        int nro = 2;
        double sumando {5.4};
        cout << "nro = " << nro << " - sumando = " << sumando << endl;
        //nro = 2 - sumando = 5.4
        cout << "doblemas(nro, sumando) = " <<  doblemas(nro, sumando) << endl;
        //doblemas(nro, sumando) = 9.4
        cout << "nro = " << nro << " - sumando = " << sumando << endl;
        //nro = 4 - sumando = 6.4 !!
        cout << "doblemas(nro*3, 3.2) = " <<  doblemas(nro*3, 3.2) << endl;
        /*error: invalid initialization of non-const reference of type ‘int&’
          from an rvalue of type ‘int’ (más otro mensaje similar por double*/
        return 0;
}
```

Ámbito (Scope)

- El ámbito (scope) de un identificador es la porción de código en la cuál está asociado a un determinado objeto (variable, función, etc). Esto define su alcance, es decir, en que partes del programa el identificador puede ser usado
- Las variables declaradas dentro de un bloque son por defecto automáticas. Son creadas cuando el programa ingresa en el bloque que las define y se destruyen al salir del mismo. Al crear la variable su valor es indeterminado, el programador debe asignar un valor explícitamente
- Las variables declaradas a nivel de archivo son estáticas y existen durante toda la ejecución del programa. Si el programador no las inicializa explícitamente son puestas “a cero” antes de comenzar el mismo

Ámbito (Scope)

- Las variables declaradas dentro de un bloque con clase de almacenamiento **static** pasan a ser estáticas, como las declaradas a nivel de archivo, y por tanto vale para ellas el lo dicho en el punto anterior.
  - Existen durante toda la ejecución del programa
  - Inicializadas en cero por defecto
- Para poder usar una variable declarada a nivel de archivo en un fuente distinto a donde se la definió, se usa la clase de almacenamiento **extern** para indicar que no se define una nueva variable, sino que se quiere usar una ya definida en otro fuente.
- Si una variable a nivel de archivo se la declara con clase de almacenamiento **static**, entonces no es posible usarla desde otro fuente, ni aún con **extern** (técnicamente pasa a tener vinculación interna)

Ámbito - Ejemplo

```
void f1(int param) {
        cout << "global = " << global;//'global' was not declared in this scope
        cout << "param = " << param; //Ok, muestra 5 cuando la llame desde main
}

int global;

int main() {
        int nro = 5;
        f1(nro);
        cout << "global = " << global; //Ok, muestra 0
        cout << "nro = " << nro; //Ok, muestra 5
        cout << "local = " << local; //'local' was not declared in this scope
        int local = 7;
        cout << "local = " << local; //Ok, muestra 7
        nro = f2(); //error: 'f2' was not declared in this scope
        for (int i = 0; i < 10; ++i) {
                int local = 3;
                cout << "local = " << local; //Ok, muestra 3
        }
        cout << "local = " << local; //Ok, vuelve a mostrar 7
        return 0;
}
```

Recursividad

- Una función es recursiva si en su código se llama a si misma, una o más veces.
- La idea general dividir un problema grande en la composición de uno más problemas similares pero más chicos.
- Es necesario tener un caso base, es decir, uno que no se dividirá más y servirá al resto para construir la solución.
- Los algoritmos recursivos suelen ser elegantes y simples de entender, pero no eficientes, por eso se los suele convertir a una versión iterativa equivalente

Recursividad Ejemplo

- Suponga que no tiene el operador de multiplicación, y quiere hacer una función recursiva en base a la suma.
- Partiendo de que a*b = a + a+ … + a (b veces)
- Podemos partir en a*b = a + a*(b-1)
- Agregando el caso base (cero) llegamos a:

$$a * b = \begin{cases} a + (a * (b-1)) & Si\ b > 0 \\ 0 & Si\ b = 0 \end{cases}$$

Multiplicación

Recursiva

```
int multiplicar(int a, int b)
{
        if (b > 0)
                return a + multiplicar(a, b - 1);
        else //CASO BASE
                return 0;
}
```

Iterativa

```
int multIter(int a, int b)
{
        int mult = 0;
        for (int i = 0; i < b; i++) {
                mult += a;
        }
        return mult;
}
```

Licencia