Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

UNIDAD 8
FUNCIONES
2023

UNIDAD 8

Funciones

Introducción

Para hallar la solución de un problema complejo, es generalmente conveniente dividirlo en pequeños problemas más simples y buscar la solución de cada uno de ellos en forma independiente. En el diseño de algoritmos computacionales y programas esta subdivisión en partes - que llamaremos ***funciones*** o ***subprogramas*** - constituye una herramienta muy importante que nos permite modularizar problemas grandes para reducir su complejidad.

Diremos que una ***función*** es un conjunto de acciones, diseñado generalmente en forma independiente, cuyo objetivo es resolver una parte del problema. Estas ***funciones*** pueden ser invocadas desde diferentes puntos de un mismo programa y también desde otras ***funciones***.

La finalidad del uso de ***funciones*** es simplificar el diseño, la codificación y la posterior depuración de programas complejos.

Las ventajas de usar subprogramas

- Reducen la complejidad del programa y permiten lograr mayor modularidad: un problema complejo se resuelve atacando por partes sub-problemas más simples.
- Facilitan el trabajo en equipo: distintos desarrolladores pueden implementar diferentes funciones o subprogramas de forma independiente.
- Facilitan la prueba de un programa: cada función puede ser probada previamente a su integración en el programa final, y en forma independiente de las demás.
- Permiten optimizar el uso y administración de recursos tales como la memoria: cada función puede gestionar localmente sus recursos auxiliares.
- Facilitan la reutilización del código: se pueden crear bibliotecas con funciones para reutilizarlas fácilmente desde múltiples programas.
- Evitan mezclar en un mismo código problemas correspondientes a diferentes niveles de abstracción: problemas de distinta naturaleza en un mismo programa se resuelven en funciones diferentes.

¿Cuándo emplear subprogramas?

Es conveniente emplear subprogramas cuando:

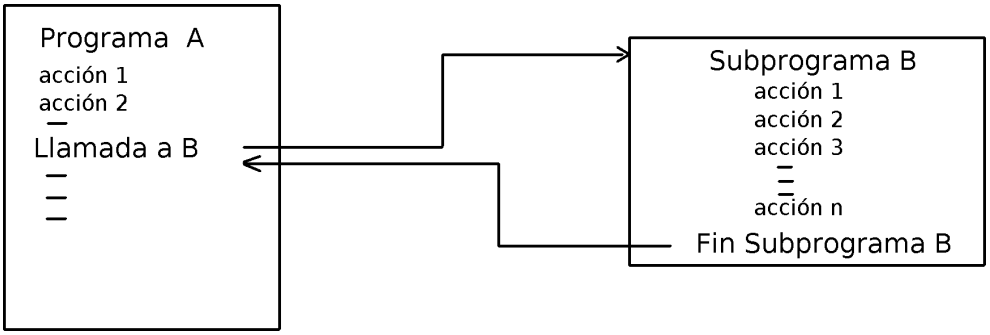
- Existe un conjunto de operaciones que se utilizan más de una vez en un mismo programa.

- Existe un conjunto de operaciones útiles que pueden ser utilizadas por otros programas.
- Se desea agrupar procesos para lograr una mayor claridad en el código del programa.
- Se pretende crear bibliotecas que permitan lograr mayor productividad en el desarrollo de futuros programas.
- Se deben resolver en un mismo programa problemas correspondientes a niveles de abstracción muy dispares, o problemas demasiado complejos.

Al plantear la solución a un problema que queremos resolver, diseñamos un programa al que llamaremos ***programa o función principal*** (de aquí el nombre "main", que significa "principal" en inglés, para la función con que inicia un programa C/C++)**.** Este podrá incluir entre sus acciones sentencias o expresiones que impliquen *llamar* o *invocar* a otra función o subprograma.

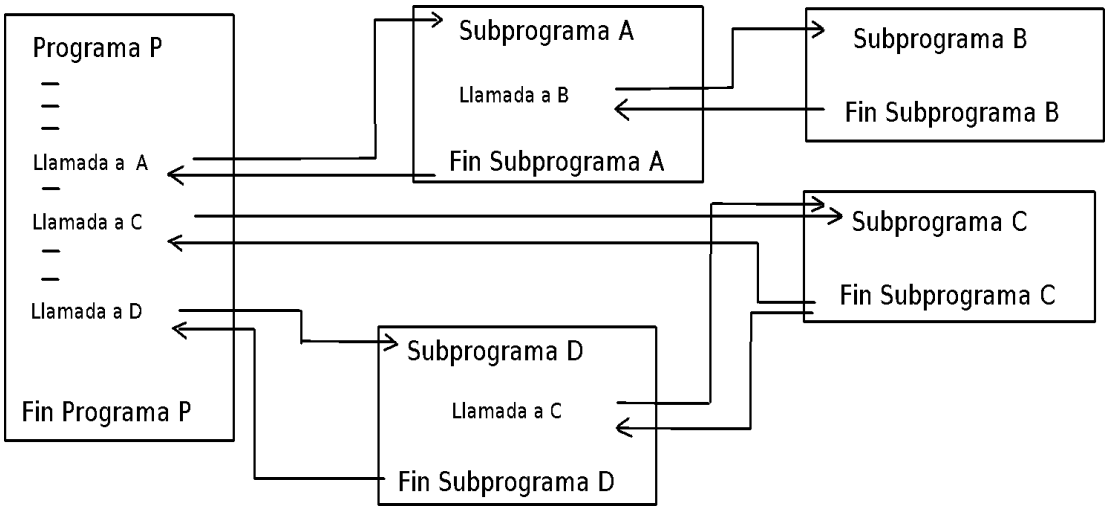
En la etapa de ejecución del programa, al encontrar la llamada a la función, se transfiere el control de ejecución a ésta y comienzan a ejecutarse las acciones previstas en la definición de la misma. Al finalizar la ejecución de la función, una vez obtenidos los resultados planeados, el control retorna al programa que produjo la llamada, transfiriendo dichos resultados, y continúa luego su ejecución.

Observemos lo anterior gráficamente:



En el esquema, **A** es un programa que contiene una acción o *llamada* al ***subprograma*** **B** . Cuando el control de ejecución llega a la llamada, comienzan a ejecutarse las acciones descritas en **B**. Al finalizar **B**, el control vuelve al programa principal **A**, para continuar con las acciones restantes. Decimos que ***A*** es *cliente del subprograma* ***B.***

Este esquema simple: ***programa principal - subprograma*** , puede adquirir mayor complejidad con la existencia de otros ***subprogramas***. El control puede pasar del programa principal a cualquier subprograma, o de un subprograma a otro, pero siempre se retorna al lugar que produjo el llamado.



En el gráfico hemos representado el programa P que contiene 3 llamadas a subprogramas diferentes, A, C, D. A su vez, los subprogramas A y D son clientes de otros subprogramas: el subprograma A llama al B, y el D al C.

Durante la ejecución de P, se encuentra la acción de llamada a A, el control pasa a dicho subprograma y comienzan a ejecutarse las acciones que él describe hasta encontrar la llamada a B; en este momento, el control pasa al subprograma B, se ejecutan sus acciones y al finalizar éste retorna al punto desde donde fue llamado en el subprograma A. Luego se continúan ejecutando las acciones de A, y al finalizar, vuelve el control al punto de llamada en el programa principal P.
Continúa la ejecución de P hasta encontrar la llamada a C, pasando el control a este subprograma, se ejecutan sus acciones y retorna a P en el punto de llamada.
Continúa P hasta hallar la llamada a D; pasa el control a D, se ejecutan sus acciones hasta encontrar la invocación al subprograma C; comienzan a ejecutarse las acciones de C, y al terminar el control retorna a D en el mismo lugar donde se llamó a C. Continúa la ejecución de D, para luego retornar al programa principal P.

Observación: nótese que el mismo subprograma C fue llamado desde el programa principal P y desde el subprograma D. En otras palabras: P y D son ambos clientes del subprograma C.

Tipos de Funciones/Subprogramas

Casi todos los lenguajes de programación admiten funciones. Se los suele denominar **funciones**, **procedimientos**, **subprocesos o subrutinas**. En algunos lenguajes se utilizan diferentes denominaciones dependiendo de si el subprograma retorna o no un resultado a la función principal, o la función que realizo la llamada. En C++ no realizamos esta distinción, y empleamos siempre el término **función**.

Funciones en C++

Todo programa C++ consta de una o más funciones y una de ellas debe denominarse `main`. La ejecución de un programa C++ comienza por las acciones planteadas en la función `main`.

Si un programa C++ contiene varias funciones estas pueden definirse en cualquier lugar del programa (aún en archivos diferentes) pero en forma independiente una de otra (no incluir la definición de una función dentro de otra).

Como vimos en la introducción es posible acceder (llamar) a una función desde cualquier lugar del programa y hacerlo varias veces en un mismo programa. Al llamar a una función el control de ejecución pasa a la función y se llevan a cabo las acciones que la componen; luego el control retorna al punto de llamada.

Suele existir intercambio de información entre el programa o módulo que llama a la función (desde ahora "cliente") y la función misma. El programa puede enviarle información (entradas) a la función al realizar la llamada, y esta puede a su vez retornar un resultado al programa cliente.

Cuando una función genera un resultado que se envía al programa cliente, se dice que la función "devuelve" o "retorna" un valor. Una función que retorna un valor puede utilizarse en el programa cliente como parte de una expresión más compleja. Al evaluarse la expresión que invoca a una función, se evalúa (ejecuta) la función y el resultado de la misma "reemplaza" a la llamada, de la misma forma en que al evaluar una expresión que incluye una variable se "reemplaza" el identificador de la variable por su verdadero valor en ese momento de la ejecución.

```
float promedio3(int x, int y, int z); // declaración

int main( ) {
      int d1, d2, d3;
      cout <<"Ingrese el primer dato:" ; cin >> d1;
      cout <<"Ingrese el segundo dato:"; cin >> d2;
      cout <<"Ingrese el tercer dato:" ; cin >> d3;
      float p = promedio3(d1, d2, d3);
      cout <<"El promedio es:" << p << endl;
}

float promedio3(int x, int y, int z) { // definición
      float w=(x+y+z)/3.0 ;
      return(w);
}
```

Una función podría no retorna ningún valor, aunque este no será el caso mas habitual. Si esto ocurre, la función obviamente no podrá ser utilizada como parte

de una expresión, sino que la invocación será una instrucción completa en sí misma.

```
void saludar(string a_quien); // declaración

int main( ) {
      saludar("Mundo");
}

void saludar(string a_quien) { // definición
      cout << "Hola " << a_quien << endl;
}
```

Declarando y definiendo funciones en C++

C++ exige declarar una función antes de que sea utilizada. Declarar una función es especificar su interface. Esto es, describir la función como un sistema de caja negra: cómo se la denomina, qué datos debe recibir, y qué tipo de resultados produce. Para ello debemos escribir la cabecera de la función, que se conforma por: el tipo de resultado que devuelve la función, su nombre/identificador y, finalmente, sus argumentos entre paréntesis (tipo y nombre de cada uno, separados por comas). Esta cabecera recibe el nombre de **prototipo** de la función. Podemos tomar como ejemplo la función *promedio3* empleada en el recuadro anterior.

```
float promedio3(int x,int y,int z);
```

Usualmente se escribe el prototipo de la función antes de la función `main`, pero recordemos que en C++ es posible efectuar la declaración de un elemento en cualquier lugar del programa, con la única condición de hacerlo antes de invocar a dicho elemento. Cuando las funciones se encuentran en archivos fuente separados (a modo de bibliotecas), se escriben los prototipos en una archivo llamado "cabecera", que por lo general tendrá la extensión .h (h por "header", "cabecera" en inglés), y se coloca antes del main la directiva "#include" seguida del nombre del archivo de cabecera entre signos menor y mayor (si es una biblioteca del sistema) o entre comillas (si es una biblioteca propia).

Si bien el prototipo alcanza para que un compilador C++ compile una llamada a dicha función, el ejecutable no estará completo hasta que no contenga las definiciones correspondientes a todas las funciones invocadas. Para definir una función se debe ingresar su prototipo y a continuación, entre llaves (y sin el punto y coma) el cuerpo de la misma (es decir, el conjunto de acciones que componen esa función):

```
float promedio3(int x, int y, int z) {
      float w = (x+y+z)/3.0;
      return w;
}
```

Resultados de una función en C++

Si una función devuelve un resultado, se debe especificar su tipo antes del nombre o identificador de la función; y en el cuerpo debemos emplear una variable o expresión de igual tipo como argumento de la sentencia **`return`**.

```
float promedio3(int x,int y,int z) {
     float w = (x+y+z)/3.0;
     return w;
}
```

Obsérvese en el ejemplo que el tipo de la función **`promedio3`** y el tipo de la variable **`w`** que será retornada coinciden. También puede plantearse:

```
float promedio3(int x,int y,int z) {
    return (x+y+z)/3.0;
}
```

Cuando se ejecuta una sentencia **`return`** la función finaliza inmediatamente retornando el resultado indicado. Por esto, pueden programarse funciones con múltiples sentencias **`return`** utilizando estructuras de control. Por ejemplo, para determinar si un número es primo se deben buscar divisores para el mismo mayores a 1 y menores que el propio número. Al encontrar un divisor cualquiera en este rango ya podemos garantizar que el número *no* es primo sin tener que seguir buscando. Pero solo si ya hemos probado todos los divisores posibles podemos afirmar que *sí* es primo:

```
bool es_primo(int x) {
    for(int i=2;i<=sqrt(x);i++) {
        if (x%i==0) return false;
    }
    return true;
}
```

En cada invocación a la función `es_primo`, solo una de las dos sentencias `return` será ejecutada.

Es posible además que una función no devuelva resultados. En ese caso se especifica el tipo nulo `void` en su prototipo:

```
void promedio3(int x,int y,int z) {
    float w=(x+y+z)/3.0 ;
    cout << "el promedio es:" << w << endl;
}
```

Sin embargo, como se detallará más adelante, en general no es una buena práctica crear funciones que calculen resultados y los informen en pantalla en lugar de retornarlos.

Intercambio de información desde funciones C++

El empleo de funciones nos permite diseñar rutinas que pueden ser reutilizadas dentro del mismo programa o desde otros programas. Usualmente es necesario enviar información a la función desde el punto de llamada para que complete la tarea asignada. Esto se hace a través de sus parámetros o argumentos.

En el prototipo y en la definición de la función planteamos los **parámetros formales o de diseño**, y cuando invocamos a la función utilizamos **parámetros actuales o de llamada**.

```
float raiz_cuad(float x); // parámetro formal
int main() {
   .....
   float raiz_de_2 = raiz_cuad(2); // parámetro actual
   .....
}
```

En este ejemplo, x es el parámetro formal, y 2 es el actual. En la definición de una función se utilizan los parámetros formales. Estos serán en realidad variables locales que tomarán los valores indicados en los parámetros actuales al momento de invocar/ejecutar la función.

Si una función no requiere parámetros de entrada se la define con con paréntesis vacíos, y se la invoca igualmente con paréntesis vacíos:

```
char generar_letra_aleatoria( );
int main() {
   .....
   char c = generar_letra_aleatoria( );
   .....
}
```

Se debe notar que dado que la función se encuentra definida fuera del main, y viceversa, desde la función no se puede acceder a las variables locales del main, y desde el main tampoco se puede acceder a las variables locales de una función. Es por esto que el mecanismo para intercambiar información entre ambas se basa en los argumentos y el valor de retorno.

Pero, ¿qué pasa si una función modifica un parámetro? ¿Este cambio en un parámetro formal se ve reflejado en el parámetro actual? Es decir, ¿se pueden modificar desde una función variables del programa cliente si estas se corresponden con los parámetros? Esta pregunta tiene dos posibles respuestas, y ambas pueden ser válidas en C++, ya que existen dos mecanismos de pasaje de parámetros: por **valor** y por **referencia**.

Pasaje de parámetros por valor

En este caso, al producirse la llamada a la función los valores de los parámetros actuales son asignados en (copiados a) los parámetros formales. Los parámetros formales son entonces variables nuevas, locales a la función, e independientes de los parámetros actuales durante la ejecución de la misma. Cuando hay más de un parámetro, la correspondencia se realiza simplemente

por posición. Es decir, el valor del primer parámetro actual de la llamada se copia al primer parámetro formal del prototipo, el segundo al segundo, y así sucesivamente.

```
.....
int main() {
   .....
   float p = promedio3(d1, d2, d3);
   cout << "Datos:" << d1 << " " << d2 << " " << d3<< endl;
   cout << "Promedio:" << p <<endl;
   .....
}

float promedio3(int x,int y,int z) {
   float w=(x+y+z)/3.f;
   return w;
}
```

Si en este ejemplo los datos asignados a d1, d2, d3 son 10, 20, 46, la salida por consola del programa será:

Datos: 10 20 46
Promedio: 25.333333

Al finalizar las acciones de la función `promedio3()` se devuelve el control a la función principal `main()` retornandose el valor obtenido en `w`. El retorno también se hace por copia. Es decir, el valor de w (variable local dentro de la función que se utiliza para el return) se asignará a (copiará en) la variable p del programa cliente (local dentro del main). Al finalizar la ejecución de la instrucción que incluye la invocación (la declaración e inicialización de p en este ejemplo) todas las variables locales de la función se destruyen (por ejemplo, `w` deja de existir).

Pasaje de parámetros por referencia

La referencia consiste en utilizar como parámetro formal una referencia a la posición de memoria del parámetro actual o de llamada. Esto puede hacerse declarando un alias (mediante el símbolo &), o empleando punteros. El primer mecanismo es más claro sintácticamente y reduce la posibilidad de cometer errores, por lo que no ahondaremos en detalles sobre la definición y el uso de punteros en esta unidad. Con el símbolo & entonces es posible definir un alias para una variable. Un alias es otro nombre para la misma cosa. Es decir, podremos tener dos nombres para una misma variable (dos identificadores que hacen referencia a un único dato almacenado en una única posición de memoria determinada).

Observemos el siguiente ejemplo:

```
int  m=10;
int  &q = m;    // q es definido como alias de m
q++;            // se incrementa q en 1 y también m
cout << m; // se obtiene 11 como salida
```

En este caso q es un alias de m. Si modificamos q, y luego tomamos el valor de m, veremos el valor modificado, ya que en realidad m y q son dos nombre para una misma variable.

C++ emplea el símbolo **&** para realizar pasaje de parámetros por referencia. Esto quiere decir que ahora el parámetro actual y el formal corresponden a una misma variable. De esta forma, si la función modifica al argumento, en realidad está modificando una variable del programa cliente. Por esto, la referencia puede utilizarse como un mecanismo alternativo para que una función le pase un resultado al programa cliente. Veamos una alternativa para el cálculo del promedio utilizando pasaje por referencia:

```
void promedio3(int x, int y, int z, float &p) {
    p = (x+y+z)/3.0 ;
}
int main( ) {
   int d1,d2,d3;
   cin >> d1 >> d2 >> d3;
   float prom;
   promedio3(d1, d2, d3, prom);
   cout << "Datos:" << d1 << " " << d2 << " " << d3 << endl;
   cout << "Promedio:" << p << endl;
}
```

En este nuevo ejemplo, al invocar a la función `promedio3` desde el `main`, la variable `p` dentro de la función será un alias para la variable `prom` dentro de `main`. Entonces, al modificar `p` en la función, asignándole el resultado, se modifica `prom` en `main`. Así, luego de finalizada la función, el resultado permanece accesible en `main` mediante la variable `prom`. Muchas bibliotecas utilizan este mecanismo en sus funciones cuando necesitan retornar al main varios resultados, ya que el mecanismo del `return` como se mostró hasta el momento solo admite un único valor de retorno. Aún así, las reglas de estilo modernas **no recomiendan utilizar la referencia como primera opción para retornar indirectamente resultados**.

Más adelante en esta unidad se describe una solución alternativa para casos simples. Pero antes analicemos otra motivación importante para el uso de referencias: evitar realizar copias. La referencia será útil cuando un argumento sea una estructura de datos compleja cuya copia sea costosa y deba tratar de evitarse. Por ejemplo, cuando operemos con arreglos en la próxima unidad, podremos evitar tener que copiar uno por uno todos los elementos de un arreglo para que lo utilice una función. En estos casos, complementaremos la definición del argumento con el calificativo "const" para hacer evidente que el objetivo de la referencia es evitar la copia, y no permitir la modificación del valor.

```
int buscar(const vector<int> &v, int x) { … }
```

Nota: veremos en la siguiente unidad que `vector<int>` es una forma de declara un arreglo unidimensional de elementos de tipo int.

Como excepción a la regla, eventualmente se encontrará con casos donde tal vez sea conveniente que la función opere "*in place*" (esto es, directamente sobre el parámetro), y entonces deba utilizar la referencia para modificar el argumento. Utilice este recurso solo cuando el efecto de la función sea obvio sin analizar su implementación:

```
void insertar(vector<int> &v, int pos, int x_nuevo) { … }
```

Parámetros por defecto

Es posible proponer, en el prototipo de la función, parámetros formales inicializados con valores. Estos valores serán asumidos por defecto en el cuerpo de la función **cuando no se indique parámetros actuales en la llamada** para tales argumentos.

```
float promedio3(int x,int y,int z=10);
.....
void main( )
  {
     .....
     float p=promedio3(d1, d2); // x=d1, y=d2, z=10
     .....
     float q=promedio3(d1,d2,d3); // x=d1, y=d2, z=d3
  }
```

La única restricción sintáctica para estos parámetros por defecto, es el hecho de que deben ubicarse al final (a la derecha) de la lista de parámetros formales. De acuerdo a esto último, el siguiente prototipo de función C++ sería causa de error en una compilación:

```
float promedio3(int x,int y=5,int z): // ERROR!
```

Sobrecarga de Funciones

Dos funciones diferentes pueden tener el mismo nombre si el prototipo varía en sus parámetros. Esto significa que se puede nombrar de la misma manera dos o más funciones si tienen distinta cantidad de parámetros y/o diferentes tipos de parámetros. Por ejemplo,

```
// Ejemplo de sobrecarga de funciones
int dividir (int a, int b) {
   return (a/b);
}
float dividir (float a, float b) {
  return (a/b);
}
int main () {
  int x=5,y=2;
  float n=5.f,m=2.f;
  cout << dividir (x,y) << endl;
  cout << dividir (n,m) << endl ;
}
```

La salida del programa será:

2
2.5

En este caso se han definido dos funciones con el mismo nombre, pero una de ellas acepta parámetros de tipo **int** y la otra acepta parámetros de tipo **float**. El compilador decide cual debe emplear en cada llamada a partir de los tipos de

sus parámetros actuales. Por simplicidad, ambas funciones tienen el mismo código, pero esto no es estrictamente necesario. Se pueden hacer dos funciones con el mismo nombre pero con comportamientos totalmente diferentes.

Otro ejemplo de sobrecarga válida, utilizando diferentes cantidades de argumentos:

```
float promedio(int a, int b) {
   return (a+b)/2.0;
}
float promedio(int a, int b, int c) {
  return (a+b+c)/3.0;
}
int main () {
  int x=5,y=2,z=7;
  cout << promedio(x,y) << endl;
  cout << promedio(x,y,z) << endl;
}
```

Obsérvese un caso erróneo de aplicación de sobrecarga en funciones:

```
// Ejemplo erróneo de sobrecarga
int dividir(int,int);
float dividir(int,int);
int main () {
  int a,b;
  ...
  int x = dividir(a,b);
  float x = dividir(a,b);
}
```

En la primer función se propone un división entera y en la segunda una división entre enteros que arroja un flotante. Pero ambas funciones están definidas con el mismo tipo y cantidad de parámetros. Cuando el programa cliente invoque a `división()` no podrá discernir a cual de las 2 funciones se refiere la llamada. Recuerde que para resolver una asignación, primero se evalúa la expresión a la derecha del =, y luego se asigna el resultado a la variable. Esta variable no influye en el primer paso (en la evaluación de la expresión), por lo que no puede ayudar a decidir qué sobrecarga utilizar.

Múltiples valores de retorno

Ya se advirtió que no es posible colocar dos o más valores en una sentencia `return`. Y aunque una función sí puede tener varias sentencias `return`, solo una de ellas será ejecutada en cada invocación (dado que al llegar a un `return` la función finaliza inmediatamente). Cuando se quiere diseñar una función que retorne dos o más resultados, existen múltiples mecanismos.

Resolveremos por ahora solo el caso en que se quieran retornar 2 valores. Para casos donde se deban retornar muchos valores será recomendable utilizar

estructuras de datos como vectores o structs, que estudiaremos en la próxima unidad.

Para el caso de 2 valores, utilizaremos el tipo `std::pair` (par). Un par se forma por dos valores agrupados en una sola variable. Entonces, se puede plantear una función que retorne un solo dato, pero que ese dato sea un par. Es como enviar desde la función al programa cliente un único "paquete", que al abrirlo dentro contiene dos elementos. Dentro del par, el primer elemento siempre se denomina `first` y el segundo `second`.

Considere estas dos formas de implementar una función para obtener cociente y resto de una división entera:

```
pair<int,int> division_entera(int dividendo, int divisor) {
    int resultado = dividendo / divisor;
    int resto = dividendo % divisor;
    return {resultado,resto};
}

pair<int,int> division_entera(int dividendo, int divisor) {
    pair<int,int> r;
    r.first = dividendo / divisor;
    r.second = dividendo % divisor;
    return r;
}
```

Ambas responden al mismo prototipo, por lo que son equivalentes desde el punto de vista de un programa cliente.

- La primera genera los dos resultados en variables independientes, y las agrupa para conformar un par al momento de hacer el return[1].
- La segunda versión, en cambio, declara una variable `p` de tipo par y asigna sus dos valores accediendo a ellos con p.`first` y `p.second`. Ambos mecanismos son válidos.

Veamos ahora dos formas de recibir los datos del par en el programa cliente

```
int main() {
    int a, b;
    cin >> a >> b;
    pair<int,int> r = division_entera(a,b);
    cout << "Resultado: " << r.first << endl;
    cout << "Resto: " << r.second << endl;
}

int main() {
    int a, b;
    cin >> a >> b;
    int resultado, resto;
```

[1] En otros textos podría encontrar ejemplos donde el agrupamiento se haga con una función llamada `make_pair` en lugar de usar simplemente llaves como aquí. Es también una opción válida.

```
    tie(resultado,resto) = division_entera(a,b);
    cout << "Resultado: " << resultado << endl;
    cout << "Resto: " << resto << endl;
}
```

Ambas versiones generan a ojos del usuario el mismo programa.

- En la primera, el resultado de la función se recibe en una variable `r` de tipo par, y se accede a los dos datos mediante `r.first` y `r.second`.
- En la segunda implementación se declaran dos variables independientes `resultado` y `resto`, que se unen "temporalmente" para parecer un par y permitir la asignación, gracias a la función `tie`. Luego de asignado el resultado de la función, las variables seguirán siendo independientes.

Nota: hay una generalización del par, llamada tupla (`std::tuple`) que puede servir para retornar un número arbitrario de elementos. Sin embargo, tanto par como tupla, no dejan claro qué significa cada valor, ya que solo podemos especificar el tipo, pero no darle nombre. Por esto no es buena idea generalizar este mecanismo para retornar por ejemplo 5 valores. Será menos propenso a errores definir un `struct`, que es una estructura de datos que estudiaremos en la próxima unidad y nos dará la posibilidad de ponerle nombres a las partes para que sea más evidente qué es cada una.

Recursividad

La recursividad es una técnica que permite definir a una función en términos de sí misma. En otras palabras: una función es recursiva cuando se invoca a sí misma.

C++ admite el uso de funciones recursivas; cualquier función C++ puede incluir en su código una invocación a sí misma, a excepción de `main()`.

Como ventaja de esta técnica podemos destacar que permite en algunos casos resolver elegantemente algoritmos complejos. Como desventaja debemos decir que los procedimientos recursivos son menos eficientes –en términos de velocidad de ejecución- que los no recursivos, y que la cantidad de llamadas recursivas que una función puede realizar se encuentra limitada.

Los algoritmos recursivos surgen naturalmente de muchas definiciones que se plantean conceptualmente como recurrentes. Obsérvese el caso del factorial de un número (n!): por definición es el producto de dicho número por todos los factores consecutivos y decrecientes a partir de ese número, hasta la unidad:
n!= n*.(n-1)*(n-2)*…*2*1.
Por ejemplo: 5!=5*4*3*2*1= 120
Pero el producto 4*3*2*1 es 4!
Por lo tanto podemos escribir: 5!= 5* 4!
Y en general: n! = n*(n-1)!

Como vemos en la línea anterior, hemos expresado el factorial en función de sí mismo. La clave es que para calcular el factorial de un número, utilizamos el factorial de otro. En algún, punto, deberíamos llegar al factorial de algún numero que podamos determinar fácilmente y entregar un resultado directamente sin continuar la recursión. Es decir, el problema se va transformando hasta llegar a un caso conocido en el cual se corta la recursión evitando así que la función continue llamándose infinitamente (o hasta que la memoria se agote). Es exactamente lo que podemos plantear algorítmicamente usando funciones en C++. La solución recursiva del factorial de un número puede expresarse de la siguiente forma:

```
long factorial(unsigned int x) {
   if  (x==0)
         return 1; // caso fácil y conocido
   else
          return x*factorial(x-1); // caso recursivo
}
```

Obsérvese que en la función recursiva existe una condición (x==0) que permite abandonar el proceso recursivo cuando la expresión relacional arroje verdadero; de otro modo el proceso sería infinito.

Entonces, para implementar una función recursiva debemos identificar dos cosas: una definición recursiva o recurrente, que plantee la solución a una

versión del problema en términos de otras; y una caso base al cual arriben tarde o temprano todos los demás casos y pueda resolverse directamente.

Notar que la versión no recursiva del factorial utilizaría un bucle para realizar los n productos que el cálculo implica, pero en la versión recursiva no se observa bucle alguno. El mecanismo de recursión reemplaza al bucle. La explicación es la siguiente: en la versión no recursiva se invoca una sola vez a la función que realiza n productos; mientras que en la versión recursiva la función se invoca n veces, realizando solo un producto en cada invocación.

Condiciones para que una función sea recursiva

Toda función recursiva debe.

1. Realizar llamadas a sí misma para efectuar versiones reducidas de la misma tarea.
2. Incluir uno o más casos donde la función realice su tarea sin emplear una llamada recursiva, permitiendo detener la secuencia de llamadas (condición de detención o stop)

Analizando el ejemplo del factorial de un número, podemos ver que la expresión `x*factorial(x-1)` corresponde al requisito 1 y la expresión `x==0` al segundo requisito.

La recursión es una herramienta muy potente y utilizada correctamente permite en ciertos casos generar algoritmos muy eficientes. Lamentablemente a esta altura de la materia, no hemos aprendido lo suficiente como para plantear muchos ejemplos donde se aprecie la verdadera potencia de la recursión. Por ahora en muchos casos se aplicará forzadamente solo para entender y practicar el concepto, pero más adelante en la carrera encontrará infinidad de aplicaciones.

Como ejemplo de cómo la recursión puede reducir drásticamente la cantidad de pasos a ejecutarse en un algoritmo, puede pensar qué sucede si se implementa una función para calcular potencias con exponentes enteros utilizando la regla recursiva $a^n = a^{n/2} * a^{n/2}$ cuando n es par (por ej: para calcular 2^{20} puede calcular primero 2^{10} y luego multiplicar el resultado por sí mismo).

Identificación y diseño de funciones

Veremos ahora algunos criterios para realizar divisiones en funciones que resulten convenientes, tanto para simplificar la resolución un problema, como para poder reutilizar luego las funciones en otros programas.

Diseño de prototipos

Un prototipo de función bien diseñado permite deducir rápidamente qué hace la función y qué significan cada uno de sus argumentos. En general, una vez resuelto un problema mediante una función, el programador ya no debería necesitar volver a los detalles de la solución, sino que debería poder utilizar fácilmente la función como caja negra y concentrar su atención en otras partes del programa.

Por ejemplo, considere estos dos prototipos para una misma operación:

```
a) float aplicar_descuento (float monto_inicial,
                            unsigned int porcentaje_descuento)
b) float ap_des(float a, float b)
```

En el primer caso, el nombre de la función es totalmente descriptivo, mientras que en el segundo solo tiene sentido si recuerdan las abreviaciones y se conoce el contexto para las mismas[2]. Además, en el primero queda más claro qué representa cada argumento, mientras que en el segundo no es obvio el orden de los mismos. Más aún, un porcentaje de, por ejemplo, 25% se representa a veces mediante el entero 25, mientras que en otros casos se utiliza el real 0.25 (25/100). Utilizar int para el segundo argumento permite deducir rápidamente que se utilizará la primer representación sin tener que buscar en la documentación o analizar la implementación de la función, y calificarlo como `unsigned` indicar que el porcentaje se ingresa como positivo por más que se trate de un descuento. Conforme avance en el estudio de C++ irá confirmando que los tipos de datos se tornan cada vez más importantes y constituyen una gran herramienta para brindar información para la detección de errores no solo a otro programador, sino también al compilador.

Pasaje por referencia y múltiples valores de retorno

Habiendo presentado dos mecanismos para que una función retorne más de un valor al programa cliente, el pasaje por referencia y el uso de pares, para tal fin se recomienda en C++ moderno utilizar el segundo. Esto se debe a que permite separar mejor cual es la entrada y cual es la salida para dicha función: los argumentos serán todos de entrada, y toda la salida estará en el valor de retorno.

En un código que utiliza referencia para retornar valores, al ver un prototipo que recibe un valor por referencia, no es simple determinar si se trata de un argumento de entrada (se está intentando evitar la copia), o un valor de salida, o peor aún, ambas cosas a la vez (se recibe un valor y luego se usa esa misma variable para retornar otro). El único caso en que esta decisión está clara y no se requiere inspeccionar la implementación, es el caso en que la referencia se declara como constante:

```
int sumar_todo(const tipo_de_arreglo_muy_muy_grande &v);
```

En este ejemplo, una función que suma todos los elementos de un arreglo, se utiliza la referencia para evitar tener que copiar todo un arreglo que puede tener

[2] Por ejemplo, en el contexto de redes wi-fi, ap podría significar “Access-Point” en lugar de “APlicar” y des referenciar al protocola Data-Encription Standar en lugar de “DEScuento”.

una dimensión muy grande, y se agrega el calificativo `const` para que esta intención quede clara. El calificativo `const` indica que la función no podrá modificar al arreglo, y por lo tanto desde el programa cliente podemos invocarla con la seguridad de que no alterará su contenido.

En conclusión, se recomienda usar pares para funciones que retornen dos valores, y dejar la referencia solo para el caso en que se pretende evitar la copia, explicitando siempre esta intención mediante el calificativo `const`.

Cuando la cantidad de valores de retorno, o de argumentos de entrada sea muy elevada, ninguno de estos mecanismos resultará totalmente adecuado. En general una función que recibe 20 argumentos, o retorna 10 valores diferentes resulta muy propensa a causar errores y confusiones. Para estos casos, será recomendable utilizar una nueva estructura (denominada struct) que será objeto de estudio de la próxima unidad de la materia.

Entrada/salida en funciones

En general, no es conveniente emplear operaciones de entrada y salida en funciones. Es mejor operar a través de parámetros y que la entrada y salida la realice el programa cliente de la función. Esto es así para independizar la función del tipo de entorno en que se ejecutará el programa cliente que la utilizará. Por ejemplo: si en una función incluimos operaciones de salida empleando el modo consola en C++ a través de los objetos `cin/cout`, no podremos emplear esta función en un programa C++ que opere en un entorno gráfico donde la entrada y salida se realizan a través de componentes visuales (ventanas y cuadros de texto), ni tampoco cuando los datos se guarden en otros medios como archivos, se deban enviar por una conexión de red a un servidor, o se necesiten como parte (resultados intermedios) de un cálculo más complejo.

Observemos en el ejemplo de abajo la diferencia entre una función que realiza una salida y otra que sólo devuelve el resultado. La función `volumen_cilindro1()` calcula el volumen de un cilindro y produce una salida con ese resultado

```
void volumen_cilindro1(float radio, float altura) {
   float vol = 3.14*radio*radio*altura;
   cout<<”El volumen del cilindro es:”<<vol;
}
```

La función `volumen_cilindro2()` solo devuelve el resultado obtenido al cliente que invoque la función.

```
float volumen_cilindro2(float radio, float altura) {
   return 3.14*radio*radio*altura;
}
```

La función `volumen_cilindro2()` puede ser reutilizada en programas cuya entrada y salida se realice a través de componentes visuales de un entorno gráfico. La función `volumen_cilindro1()` solo puede emplearse en programas C++ que operen en modo consola, y solo cuando el resultado deba mostrarse con el mensaje prefijado (¿qué pasaría, por ejemplo, si el programa cliente está pensado para usuarios que hablan francés en lugar de español?). Además, la función `volumen_cilindro2()` garantiza la separación del código relacionado al cálculo respecto del código relacionado a los formatos de entrada/salida, dos tareas diferentes de un mismo programa.

¿Esto significa que nunca debemos usar cin/cout dentro de una función? No. Significa que no debemos hacerlo cuando el objetivo de la función es calcular algo. Pero si la función solo sirve para cargar un valores, o solo sirve para mostrar valores, entonces sí es válido hacerlo. Por ej, una función:

```
void imprimir_vector(const vector<int> &v) { … }
```

La siguiente sección fundamenta mejor esta idea.

División de responsabilidades

Recordemos que uno de los objetivos iniciales de la división en funciones es el de separar el programa en partes más pequeñas para que sean más fáciles de resolver. Para que esto efectivamente se cumpla, estas partes deben ser independientes. En este caso, podría haber un conjunto de funciones que sólo realice cálculos, y otro conjunto que solo se encargue de la entrada/salida de datos.

Por ejemplo:

```
float obtener_radio() {
    cout << "Ingrese el radio: ";
    float r; cin >> r; return r;
}
void informar_resultados(float area, float circulo) {
    cout << "Area: " << area << endl;
    cout << "Perimetro: " << perimetro << endl;
}

float area_circulo(float radio) {
    return M_PI*radio*radio;
}
float perimetro_circulo(float radio) {
    return 2*M_PI*radio;
}

int main() {
    float r = obtener_radio();
    float p = perimetro_circulo(r), a = area_circulo(r);
    informar_resultados(p,a);
}
```

En este ejemplo tenemos dos conjuntos de funciones. Uno (area_circulo y perimetro_circulo) que se encarga solamente de resolver la parte matemática del problema, y otro (obtener_radio e informar_resultado) que se encarga exclusivamente del mecanismo de entrada y salida. Las fórmulas utilizadas para los cálculos de área y perímetro no dependen de los formatos de entrada/salida, así como los mensajes para el usuario no dependen de las fórmulas matemáticas. Por esto, ambos conjuntos de funciones pueden ser desarrollados y probados por separado. Más aún, se pueden cambiar las implementaciones de cualquiera de ellos y estos cambios no afectarán al otro. Por ejemplo, si se reemplaza las dos funciones de entrada/salida por otro par similar que utilice otro mecanismo, u otro idioma.

Es importante destacar que cada una de estas cuatro funciones tiene una y solo una responsabilidad. No es recomendable asignar múltiples responsabilidades a una misma función. Si ese fuera el caso en algún punto del programa, esa función con múltiples responsabilidades deberá descomponerse a su vez de sub-funciones que resuelvan cada parte de ese problema compuesto, de forma que estas tendrán una sola responsabilidad, y que la responsabilidad de la función descompuesta sea simplemente orquestar esa delegación de responsabilidades (así como la función main del ejemplo delega operaciones de entrada/salida y cálculos a otras funciones más concretas).

Finalmente, si todas las funciones están correctamente diseñadas, el trabajo que queda por realizar en la función main es casi trivial, ya que se reduce a “conectar” sus interfaces. Es decir pasar resultados de unas a argumentos de otras. Para implementar esto, no es necesario conocer los detalles de implementación de ninguna función, sino solo sus prototipos. Es por esto que en muchos casos se desarrolla primero el programa cliente (la función main en este ejemplo), para determinar cómo deberían ser los prototipos ideales, y luego respetando esos prototipos se codifican las funciones. Así, a medida que se avanza en la resolución se va bajando en el nivel de abstracción y atacando sub-problemas cada vez más concretos, de forma equivalente a cómo se describió en la unidad 1 con el método de refinamientos sucesivos.