1. La programación modular

Como se mencionó en un primer encuentro, los programas pueden estar compuestos por uno o varios algoritmos. Los mismos, hasta ahora, fueron escritos secuencialmente dentro del cuerpo principal `main()`

Se puede considerar el simple hecho de leer un número y validar si está dentro de un rango como un *mini-algoritmo*, el cual podemos estar repitiendo varias veces a lo largo del programa. La programación modular, que consiste en particionar el código en *pequeños fragmentos reutilizables*, es una de las bases más importantes de la programación estructurada.

Entre las muchas ventajas, las más características son la generación de códigos más compactos, eficientes y legibles. En muchos lenguajes de programación esto se implementa a través de las denominadas **funciones**. Que son bloques de código que pueden ser **llamados** o *invocados* desde otros puntos del programa.

1.1. El concepto de una función

Al igual que en matemática, las funciones son nombradas y se utilizan para procesar algún tipo de información o procesar datos. Por ejemplo, usualmente a las funciones matemáticas se las suele **nombrar** f (de *función*) y el o los parámetros con los que opera entre paréntesis. Eso quiere decir que *f(x)* recibe un valor y luego se procesa (efectúa un calculo) y devuelve un resultado.

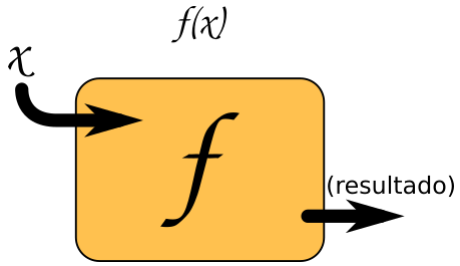

Figura 1: Representación gráfica de una función

Formalmente, en matemática se deben especificar todo lo que está formando parte de la misma. Aunque usualmente en el nivel medio/secundario se suele simplificar los detalles para priorizar los conceptos más importantes. Sin embargo, una función declarada correctamente se ve de la siguiente manera:

$$f : \forall x \in \mathbb{R} \rightarrow \mathbb{R} / f(x) = x^2$$

f es el nombre de la función. Aquí formalmente se nombra al parámetro x el cual *pertenece* al conjunto de los reales (Dominio). Lo que indica la $\rightarrow \mathbb{R}$, es que el resultado de la función está contenido dentro del conjunto de los reales (Co-Dominio). Luego se *define* la función (la implementación del algoritmo).

Los lenguajes de programación toman este mismo concepto, donde encontraremos que estas tienen un nombre, pueden recibir uno o más parámetros y dar como resultado un valor de ser requerido. Aquí un primer ejemplo de una función codificada en `C++`:

```
double al_cuadrado(double x) {
  double y = x*x;
  return y;
}
```

En el ejemplo anterior, la función recibe el nombre de `al_cuadrodo`, admite un parámetro llamdo `x`, el cual es una variable del tipo `float` declarada entre paréntesis después del nombre, y también se observa que la función está *"devolviendo"* un valor `float` que en este caso será el valor almacenado en `y`. Esto último es indicado por la palabra reservada `return`.

Aclaración

Este concepto no es nuevo, es una explicitación de lo mencionado: un programa es un conjunto de algoritmos. Cada uno de ellos puede estar sintetizado en un función. Entre más genéricas sean, más reutilizables y menos esfuerzo será necesario para hacer futuros programas.

El siguiente ejemplo es para reflexionar. En el mismo solo mostraremos el contenido del cuerpo principal `main()`.

Programa 1: Utilizando funciones

```
1 int main() {
2   mostrar_titulo();
3   double n = leer_numero();
4   n = calcular_cuadrado(n);
5   mostrar_resultado(n);
6 }
```

Aunque la persona que lea el fragmento de código anterior no sea un programador, puede intuir fácilmente de qué se trata este software. Ya que no se necesita ningún conocimiento muy especifico para comprenderlo.

2. Prototipo de una función

En la programación estructurada es común intentar colocar todo en funciones, a veces ya compiladas, es decir que no se necesita saber cómo funcionan internamente, simplemente utilizarlas como ya se ha practicado. Para ello debemos conocer al menos su **prototipo** o *definición*. Es decir qué tipo de dato retorna la función y que parámetros espera recibir.

```
// Función que recibe 3 números enteros y devuelve el promedio
double promedio(int n1, int n2, int n3);
```

Existen 4 posibilidades en cuanto a funciones:

- **Sub-proceso**: Es una función que no recibe ni retorna ningún tipo de dato.

- **Función sin retorno:** Es una función que tiene *parámetros*, pero no retorna valores, simplemente opera con los *argumentos* recibidos.
- **Función sin parámetros**: Es una función que no recibe datos pero retorna algún tipo de valor.
- **Función completa:** Posee parámetros y retorna algún tipo de valor, el valor retornado dependerá de los argumentos recibidos.

Cuando una función no retorna valores se indica con la palabra reservada `void` (*vacío*), y en caso de no tener parámetros puede indicarse con paréntesis vacíos o con la palabra `void` encerrada entre los mismos.

```
// No retorna, no recibe (sub-proceso)
void mostrar_titulo(void); // El segundo void es opcional

// No retorna, si recibe
void mostrar_barra_de_estrellas(int cantidad);

// Retorna, no recibe
double leer_numero();

// Retorna y recibe
int leer_rango(int minimo, int maximo);
```

Al igual que las variables deben tener nombres que clarifiquen su utilización, las funciones deben ser nombradas clarificando el comportamiento esperado por las mismas. Sus **parámetros**, al ser variables, deben indicar claramente cual es su rol dentro del proceso o el sub-algoritmo que están conformando.

Parámetros vs. Argumentos

Este es un concepto que gran parte de los programadores confunden. El **parámetro** de una función **es la declaración de la variable** que almacenará el dato con el cual va a operar.

El **argumento es el valor que tomará el parámetro** al momento de invocar a la función (cuando está será utilizada).

3. Implementación de la función

La *implementación o definición* de la función es el código correspondiente al algoritmo que modela. No hay diferencias entre lo que se escribía en el cuerpo principal `main()`.

La definición de la función es el prototipo sumado al cuerpo principal el cual está encerrado entre llaves.

```
void mostrar_titulo(void) {
  cout << "*** Un gran programa ***\n\n";
}
```

Cuando las funciones reciben parámetros estos van dentro del paréntesis y separados por coma:

```
// Función con un solo parámetro:
void mostrar_barra_de_estrellas(int cantidad) {
  for(int i=0; i < cantidad; i++) {
    cout << '*';
  }
  cout << endl;
}

// Función con 2 parámetros:
void mostrar_suma(double a, double b) {
  cout << a << "+" << b << "=" << (a+b) << endl;
}
```

Las funciones que devuelven algún tipo de valor deben hacerlo utilizando la palabra reservada `return`. Y tiene la limitación de que solo puede retornar un único valor, indicando de qué tipo es al principio de la declaración:

```
// Función que devuelve un número:
double leer_numero() {
  double n;
  cout << "Ingrese un número: ";
  cin >> n;
  return n;
}

// Función que recibe 2 enteros y devuelve un entero:
int leer_rango(int min, int max) {
  int n;
  do {
    cout<<"Ingrese un número entre "<<min<<" y "<<max<< ": ";
    cin >> n;
  } while (n < min or n > max);
  return n;
}
```

4. Invocar a la función

Nada de lo anterior es fructífero al menos que utilicemos la función que definimos. En algún punto del programa debemos *invocar o llamar* a la función como se hizo en el Programa 1. Al momento de hacer esta operación debemos especificar los argumentos y opcionalmente almacenar los valores devueltos por estas.

Parámetros vs. Argumentos

Este es un concepto que gran parte de los programadores confunden. El **parámetro** de una función **es la declaración de la variable** que almacenará el dato con el cual va a operar.

El **argumento es el valor que tomará el parámetro** al momento de invocar a la función (cuando está será utilizada).

Es posible hacer llamadas a función dentro de otras funciones, incrementando la modularización del programa.

```
// Recibe un número y devuelve su parte entera
int parte_entera(double n) {
  return int(n);
}

// Recibe un número y devuelve su parte decimal
double parte_decimal(double n) {
  return (n - parte_entera(n));
}
```

Cuando las funciones devuelven algún valor, y este nos resulta de utilidad, podemos almacenarlo en una variable o mostrarlo en pantalla de ser necesario:

Programa 2: Particionar un número decimal

```
#include <iostream>

using namespace std;

/* Funciones implementadas: */
int parte_entera(double n) {
  return int(n);
}

bool es_entero(double n) {
  return (parte_entera(n) == n);
}

double parte_decimal(double n) {
  return (n - parte_entera(n));
}

double leer_numero_con_coma() {
  double n;
  do {
    cout << "Ingrese un número con al menos un decimal: ";
    cin >> n;
  } while (es_entero(n));
  return n;
}
```

```
26
27 int main() {
28   double n = leer_numero_con_coma();
29   cout << "Parte entera: " << parte_entera(n) << "\n";
30   cout << "Parte decimal: "<< parte_decimal(n) << "\n";
31 }
```

4.1. Conocer antes de usar

En la formalidad, las funciones deben ser declaradas o implementadas antes de ser utilizadas. Es decir que debemos conocer, al menos, su prototipo antes de poder utilizarlas. De lo contrario podemos recibir el error o advertencia de que utilizamos *implícitamente* una función. La mejor práctica es declarar los prototipos de las funciones y luego implementarlas, aunque por practicidad podemos implementar las funciones directamente antes de llamarlas en pequeños ejemplos o códigos no muy largos como en el Programa 2. Lo más habitual es hacerlo de la siguiente manera.

Programa 3: Particionar un número decimal v2

```
1  #include <iostream>
2
3  using namespace std;
4
5  /********* Declaración de funciones (prototípos) *********/
6  bool es_entero(double n);
7  int parte_entera(double n);
8  double parte_decimal(double n);
9  double leer_numero_con_coma();
10
11 int main() {
12   double n = leer_numero_con_coma();
13   cout << "Parte entera: " << parte_entera(n) << '\n';
14   cout << "Parte decimal: "<< parte_decimal(n) << '\n';
15 }
16
17 /****************** Implementaciones ******************/
18 int parte_entera(double n) {
19   return int(n);
20 }
21
22 bool es_entero(double n) {
23   return (parte_entera(n) == n);
24 }
25
26 double parte_decimal(double n) {
27   return (n - parte_entera(n));
28 }
29
30 double leer_numero_con_coma() {
31   double n;
```

```
32    do {
33      cout << "Ingrese un número con al menos un decimal: ";
34      cin >> n;
35    } while (es_entero(n));
36    return n;
37  }
```

5. Buenas prácticas

5.1. Nombrar funciones

Al igual que con las variables, las funciones deben describir exactamente lo que hacen. A su vez será propicio que los nombres no sean demasiado largos y que al ser acciones que se van a ejecutar en lo posible sean o empiecen **con un verbo**. También es posible, si es claro lo que hace la función, que se nombre como un *sustantivo*, en este último caso, solamente si retorna un valor. Por último, al igual que las variables booleanas, sería correcto que estas empiecen con los verbos ser/estar, poder y tener.

```
void mostrar_titulo(string t); // Correcto, empieza con verbo
double promedio(int a, int b, int c); // Correcto... pero
double promediar_enteros(int a, int b, int c); // es mejor...

// Correcto, sustantivo, devuelve el máximo:
int maximo(int a, int b);
// Correcto, verbo, pero más largo:
int obtener_maximo(int a, int b);

// MAL ¿qué calculo?¿en qué contexto?
int hacer_calculo(int a, int b);

// MAL sustantivo, pero no retorna ningún valor
void saldo(int nro_cuenta);

// Bien si es obvio que se trata de un sistema de usuarios
bool es_admin(string usuario);
```

5.2. Responsabilidad, longitud y dependencia

Las funciones deben tener una sola responsabilidad y ser lo más breves posibles. Si una función crece demasiado, lo mejor es separarla en pequeñas funciones siempre que sea posible. Además, deben ser lo más específicas que se pueda dentro del contexto. Si todo eso se logra se dice que hay una **alta cohesión** en el software.

Es deseable que las funciones no dependan demasiado de otros códigos o módulos de software fuera de las funcionalidades estándar. Esto aumentaría su reutilización, ayudando

a mantener y extender el software más fácilmente. A esta característica se la denomina **acoplamiento**.

Un buen diseño de software se caracteriza por tener *alta cohesión y bajo acoplamiento*. Es decir, funcionalidades muy específicas que tiene una única responsabilidad dentro del software que componen y además que no dependen exageradamente de otros módulos para su correcto funcionamiento.

6. Ámbito de una variable

Las variables son espacios reservados en la memoria principal de nuestro programa (RAM). El cual es un recurso limitado, y que debemos economizar si deseamos hacer aplicaciones óptimas.

El ámbito de una variable está dada por su **visibilidad** y *existencia*. La visibilidad es desde donde se puede acceder a la variable para leerla y escribirla. La existencia es desde su creación, es decir, cuando se reserva memoria, hasta que libera el espacio que ocupa.

6.1. Locales

Como se puede observar en los ejemplos anteriores, las variables dentro de una función (incluyendo sus parámetros), pueden tener el mismo nombre que una utilizada en otra función. Esto es debido a la visibilidad que tienen. Estas sólo existen dentro de sus *ámbitos*. Es por ello que no hay problema si más de una variable tiene el mismo nombre en estos casos:

```
int sumar(int a, int b);
int maximo(int a, int b, int c);
double promedio(double a, double b, double c);
```

Como se observa en los prototipos de las funciones, sus parámetros tiene los mismo nombres, incluso pueden ser de tipos de datos diferentes. Pero cada una existe dentro de su propia función. Por eso se dice que **son locales** dentro de las mismas.

```
int maximo(int a, int b, int c) {
  int m;
  if(a > b and a > c) {
    m = a;
  } else if(b > c) {
    m = b;
  } else {
    m = c;
  }
  return m;
}
```

En el código anterior, observamos que hay una variable local llamada `m` en conjunto a los parámetros `a, b` y `c`. Las variables locales se crean al momento de invocar la función

y liberan el espacio al finalizar al misma. Es decir, dejan de existir. Para un uso eficiente de la memoria **es recomendable adoptar el uso de variables locales y funciones**. De esta manera, la memoria podrá utilizar los mismos espacios para diferentes variables, utilizando y liberando el espacio correspondiente.

6.2. Globales

En algunos programas hay variables de las cuales se hace necesario accederlas o modificarlas a lo largo de todo el código y, si bien, sería posible pasarlas como argumento entre funciones, podría ser algo bastante engorroso. Para esos casos, podemos utilizar **variables globales**, las cuales son accesibles desde cualquier punto del archivo fuente que estemos codificando.

Para declarar una variable global, simplemente debemos hacerlo por fuera de cualquier función. Esto comúnmente se hace al principio del código, para tenerlas visualizadas desde el comienzo.

```
#include <iostream>

using namespace std;
/****************** Variables Globales ******************/
int ingresos = 0; // cant. de datos ingresados por el usuario

/********* Declaración de funciones (prototípos) *********/
double leer_numero_entre(double min, double max);

int main() {
  cout << "Haga los siguientes ingresos:\n";
  double a = leer_numero_entre(-1, 1);
  double b = leer_numero_entre(0, 100);
  double c = leer_numero_entre(-50, 50);
  cout <<"Números ingresados: "<<a<<", "<<b<<", "<<c<<"\n";
  // Se muestran la cantidad de datos ingresos por el usuario
  // incluyendo los que no pasaron las validaciones:
  cout <<"Datos ingresados por el usuario: "<<ingresos<<"\n";
}
/****************** Implementaciones ******************/
double leer_numero_entre(double min, double max) {
  double n;
  do {
    cout << "Ingrese un número (" << min <<"-"<< max <<"): ";
    cin >> n;
    ingresos++; // modifico la variable global.
  } while(num < min or num > max);
  return num;
}
```

En el ejemplo anterior, la variable `ingresos` incrementa su valor cada vez que se lee un dato, independientemente de si el usuario puso un valor dentro del rango requerido. Como ésta fue declarada fuera de toda función, puede ser modificada por esta o varias funciones sin perder su valor.

Es importante entender que las variables globales se crean junto con el programa y solo liberan su espacio al finalizar la ejecución del mismo. Es decir que no dejan de existir hasta que no se finaliza la ejecución. Es por esta razón que es aconsejable evitarlas en la medida de lo posible.

7. Ejercitación

7.1. Complete los siguientes programas

7.1.1. Implemente las funciones declaradas para que el siguiente programas funcione

```cpp
#include <iostream>
using namespace std;
/* Declaración de funciones */
// Lee un valor entero validando que sea mayor a 0.
int leer_entero_positivo();
// Lee un texto no vacío.
string leer_texto();
// Repite el parámetro 'texto' la cantidad indicada por el
// parámetro 'veces'
void repetir(string texto, int veces);

int main() {
  int n = leer_entero_positivo();
  string t = leer_texto();
  repetir(t, n);
}
// Implementación:
// ...
```

7.1.2. Complete los espacios en blanco de las siguientes porciones de código (*snippets*)

```cpp
..... al_cuadrado(double n) {
   double resultado = n*n;
   ........ resultado;
}
double promedio_enteros(int a, int b, .... c) {
  ..... promedio = (a+b+c)/....;
  return promedio;
}
..... saludar(string nombre) {
  cout << "Hola, " << ....... << ".\n";
  cout << "¡Bienvenido!\n";
}
```

7.2. Implemente las siguientes funciones y haga programas de prueba para verificar su funcionamiento

1. La función recibe un parámetro entero positivo, muestra los pares positivos hasta llegar al número ingresado por parámetro. En caso de que el parámetro no sea positivo, se debe mostrar un error.

2. La función recibe dos números cualquiera y devuelve el valor del menor.

3. La función "`redondeo()`" recibe un número decimal y lo devuelve redondeado al entero más próximo.

4. La función "`piso()`", la cual recibe un número decimal y devuelve el valor entero menor más próximo. (P.e.: `piso(3.77)` → `3`; `piso(-2.22)` → `-3`).

5. La función "`techo()`", la cual recibe un número decimal y devuelve el valor entero mayor más próximo.

6. La función recibe un número cualquiera y devuelve el valor absoluto.

7. La función recibe dos números cualquiera y devuelve la distancia entre ellos.

8. La función recibe un texto y dos valores numéricos indicando mínimo y máximo valor esperado. Muestra el texto de su parámetro y lee del teclado un número entre los valores mínimo y máximo devolviendo el valor ingresado por el usuario.

9. Recibe un número entero positivo, muestra la cantidad de números de la serie de Fibonacci indicados por el parámetro.