# 1. Reutilización

Uno de los aspectos más importantes en el mundo de la programación es el de la reutilización. Consiste en utilizar algoritmos ya codificados (muchas veces ya compilados), que están *bien probados* y simplifican la producción de nuevo software, re-aprovechando lo que existe. Dentro de estos casos, en `C++` ya proporciona multitudes de estas porciones de código, muchos heredados del lenguaje `C`. Esto lo haremos a través de *funciones* que ya están a nuestra disposición.

Las funciones suelen estar agrupadas dentro de *bibliotecas* (*libraries* o simplemente *libs*), las cuales se encuentran agrupadas por su propósito o funcionalidad. Dentro de éstas podemos encontrar funciones matemáticas, como potenciación, redondeo, valor absoluto, seno, tangente, etc. Así como también pueden contener **valores constantes**, usualmente en forma de *etiquetas*, por ejemplo `M_PI` que contiene el valor de $\pi$.

Para utilizar alguna de estas bibliotecas, debemos agregar una cabecera asociada, para ello se recurre al comando `#include`, seguido del nombre de la misma encerrada entre corchetes angulares (<>). Esto usualmente se encuentra al principio del código fuente, en las primeras lineas, junto a la inclusión de la cabecera estándar `iostream`, pudiendo agregar sólo una a la vez y por linea:

```
1 #include <iostream>
2 #include <lib1>
3 #include <lib2>
4 // ...
5 #include <libX>
```

## 1.1. Cabeceras

Algunas bibliotecas estándar son heredadas de `C`, y sus cabeceras asociadas empiezan con "`c...`". Por ejemplo, la que provee funciones matemáticas se invoca como `cmath`. En `C` las cabeceras son archivos fuente terminados en `.h`, pero en `C++` se adoptó por no agregar esta terminación. Sin embargo, muchos archivos que funcionan y están adaptados para ambos lenguajes siguen utilizando la extensión `.h`. Si estas sólo son compatibles con `C++`, entonces se opta por utilizar `.hpp`. Lo mismo ocurre con los archivos fuentes que se han estado utilizando, los de `C` son terminados en `.c` y los de `C++` en `.cpp`, por ejemplo `main.cpp`.

Estos archivos de cabecera contienen únicamente declaraciones, es decir, no poseen el código o algoritmo que hace que funcione el código, eso ya suele encontrarse compilado y debe *conectarse* con un programa llamado *linker*. Luego, el compilador une el código fuente junto con las *libs* utilizadas para formar el archivo binario ejecutable. Todo esto ocurre automáticamente en la mayoría de los entornos de programación al invocar simplemente al compilador.

## 1.2. Invocar una función

Las funciones son invocadas o llamadas desde alguna parte de nuestro algoritmo para cumplimentar una tarea. Un ejemplo de llamado a función y su sintaxis sería el siguiente:

Programa 1: Distancia entre dos números

```cpp
#include <iostream>
#include <cmath> // incluye funciones matemáticas como abs()
using namespace std;
int main() {
  cout << "Ingrese dos números reales: ";
  float n1, n2;
  cin >> n1 >> n2;
  // Llamado a la función abs() "valor absoluto"
  float d = abs(n2-n1);
  cout << "La distancia entre ambos es: " << d << endl;
}
```

Como se observa en el Programa 1, para calcular la distancia entre dos números, se utiliza la función `abs()` la cual **recibe** un número como argumento (en este caso el resultado de `n2-n1`) y **retorna** o *devuelve* un valor que se almacena en la variable `d`. En ambos casos se trata de tipos `float`.

Al indagar dentro del archivo cabecera `cmath` observaremos al buscar `abs` una linea similar a esta:

```cpp
double abs(double x);
```

A esto se lo conoce como el **prototipo** o *declaración* de la función. Esta nos indica el tipo de valor es *retornado* por la función y el valor o valores que puede recibir y usualmente el nombre de dicho parámetro (por ejemplo x). En el Programa 1 hay un *casteo* automático de `float` a `double` tanto al dato que es enviado a `abs` como el devuelto y almacenado en `d` que en este caso sería de `double` a `float`.

Las funciones pueden tener tantos parámetros como resulten necesarios para cumplir con su propósito. Estas pueden ser de cualquier tipos de los que ya mencionados. Pero solo pueden retornar un único valor. Por ejemplo, veamos el siguiente caso.

Programa 2: Potenciación

```cpp
#include <iostream>
#include <cmath>
using namespace std;
int main() {
  cout << "Ingrese dos números: ";
  double n1, n2;
  cin >> n1 >> n2;
  double pot = pow(n1, n2); // n1^n2
  cout << n1 <<" elevado a la "<< n2 <<" es: "<< pot << endl;
}
```

Una posible salida de este programa podría ser:

```
Ingrese dos números: 2 3
2 elevado a la 3 es: 8
```

En esta oportunidad vemos que la función *potenciación* (`pow`) posee dos parámetros, los cuales se utilizan como base y exponente dentro del algoritmo que ejecuta. Es decir que una posible declaración de esta función es la siguiente:

```
double pow(double base, double exp);
```

Obsérvese que no es importante el nombre del parámetro, ya que simplemente es una variable donde se almacena un valor (*argumento*) el cual es necesario para operar.

Existe otro tipo de funciones que no necesitan recibir ningún tipo de argumento, ya que el algoritmo no depende de un valor externo. Se dice que son funciones que *no poseen parámetros* o que *no reciben argumentos*. Un ejemplo concreto de esto es la función estándar de `C` '`rand()`' que retorna valores enteros *pseudo-aleatorios* y puede declararse de dos maneras diferentes:

```
int rand();
// o
int rand(void);
```

Ambas declaraciones son válidas, en la primera simplemente se dejan los paréntesis vacíos de la función y en el otro se explicita la falta de parámetros con la palabra reservada `void` (*vacío*).

Por último, hay funciones que podrían **no retornar** o devolver ningún valor. Pero éstas serán estudiadas más adelante, la declaración de una función de este estilo se ve de la siguiente manera:

```
void abort(void);
```

En este caso, la función debe explicitar que no hay un resultado después de su ejecución con la palabra reservada `void`. Si la función además no recibe argumentos se las denominan **sub-procesos**. Sin embargo, podría recibir argumentos para operar o alterar su comportamiento, por ejemplo `exit()`, la cual posee un parámetro, pero no devuelve valores:

```
void exit(int exit_code);
```

## 2. Funciones y constantes matemáticas

Muchas de las actividades y ejercitaciones propuestas requieren de la utilización de procesos matemáticos como se ha expuesto en el apartado anterior con la distancia entre dos números reales. El uso de algoritmos donde están involucradas las matemáticas son innumerables, es por ello que ya hay muchos algoritmos de uso común dentro de la biblioteca estándar matemática a la cual accedemos a través de `<cmath>math`.

A continuación se emplearán algunas de estas funciones que se tienen a disposición.

## 2.1. Valores absolutos, Techo, Piso, Truncamiento y Redondeo

### 2.1.1. Prototipos

```
// Valor Absoluto: Retorna el valor númerico siempre positivo.
double abs(double x);
// Techo: retorna el valor redondeo al entero superior.
double ceil(double x);
// Piso: retorna el valor redondeo al entero inferior.
double floor(double x);
// Redondeo: retorna el valor redondeo al entero más próximo.
double round(double x);
// Truncamiento: retorna el valor entero.
double trunc(double x);
```

### 2.1.2. Ejemplos

```
#include <iostream> // para utilizar cout
#include <cmath>    // para utilizar abs, round, floor, etc.
using namespace std;
int main() {
  cout << "VALOR ABSOLUTO\n";
  cout << "abs(3.77) -> " << abs(3.77) << "\n";
  cout << "abs(-2.33) -> " << abs(-2.33) << "\n\n";

  cout << "REDONDEO\n";
  cout << "round(3.77) -> " << round(3.77) << "\n";
  cout << "round(-2.33) -> " << round(-2.33) << "\n\n";

  cout << "PISO\n";
  cout << "floor(3.77) -> " << floor(3.77) << "\n";
  cout << "floor(-2.33) -> " << floor(-2.33) << "\n\n";

  cout << "TECHO\n";
  cout << "ciel(3.77) -> " << ceil(3.77) << "\n";
  cout << "ciel(-2.33) -> " << ceil(-2.33) << "\n\n";

  cout << "TRUNCAMIENTO\n";
  cout << "trunc(3.77) -> " << trunc(3.77) << "\n";
  cout << "trunc(-2.33) -> " << trunc(-2.33) << "\n\n";
}
```

## 2.2. Funciones trigonométricas

## 2.3. Prototipos

Las funciones trigonométricas reciben un parámetro cuyo valor debe estar expresado en radianes.

```
double acos(double x);  /* Arco-coseno */
double asin(double x);  /* Arco-seno */
double atan(double x);  /* Arco-tangente */
double atan2(double x); /* Arco-tangente cuadrada */
double cos(double x);   /* Coseno */
double cosh(double x);  /* Coseno hiperbólico */
double sin(double x);   /* Seno */
double sinh(double x);  /* Seno hiperbólico */
double tan(double x);   /* Tangente */
double tanh(double x);  /* Tangente hiperbólica */
```

### 2.3.1. Ejemplos

```
#include <iostream> // cout
#include <cmath>    // sin, cos, tan, asin, acos, atan

using namespace std;

int main() {

  cout << "Trigonométricas básicas: " << endl;
  cout << "sin(pi/2) -> " << sin(M_PI/2) << endl;
  cout << "con(pi)   -> " << cos(M_PI) << endl;
  cout << "tan(pi/4) -> " << tan(M_PI/4) << endl;
  cout << endl;

  cout << "Funciones inversas: " << endl;
  cout << "asin(1)  -> " << asin(1) << endl;
  cout << "acos(-1) -> " << acos(-1) << endl;
  cout << "atan(1)  -> " << atan(1) << endl;
  cout << endl;

}
```

## 2.4. Exponencial, Logaritmos, Potenciación y Raíz Cuadrada

### 2.4.1. Prototipos

```
/* Función exponencial */
double exp(double x);   // Base e
double exp10(double x); // Base 10
double exp2(double x);  // Base 2

/* Logaritmos */
double log(double x);   // Logaritmo natural (base e)
double log10(double x); // Logaritmo base 10
```

```
/* Potenciación */
double pow(double b, double e); // b elevado a e

/* Raíz cuadrada */
double sqrt(double x);
```

### 2.4.2. Ejemplos

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
  cout << "Ingrese un número: ";
  double x;
  cin >> x;
  cout << "e elevado a la " << x << " es " << exp(x) << endl;
}
```

## 2.5. Constantes

Para evitar cálculos innecesarios o simplemente para hacer uso de algunos valores útiles, la biblioteca matemática tiene las siguiente **etiquetas**, las cuales se declaran utilizando el comando de pre-procesador `define`.

```
#define M_E         2.71828182845904523540  /* e          */
#define M_LOG2E     1.44269504088896340740  /* log_2 e    */
#define M_LOG10E    0.43429448190325182765  /* log_10 e   */
#define M_LN2       0.69314718055994530942  /* log_e 2    */
#define M_LN10      2.30258509299404568402  /* log_e 10   */
#define M_PI        3.14159265358979323846  /* pi         */
#define M_PI_2      1.57079632679489661923  /* pi/2       */
#define M_PI_4      0.78539816339744830962  /* pi/4       */
#define M_1_PI      0.31830988618379067154  /* 1/pi       */
#define M_2_PI      0.63661977236758134308  /* 2/pi       */
#define M_2_SQRTPI  1.12837916709551257390  /* 2/sqrt(pi) */
#define M_SQRT2     1.41421356237309504880  /* sqrt(2)    */
#define M_SQRT1_2   0.70710678118654752440  /* 1/sqrt(2)  */
```

# 3. Comprobación de caracteres

Existe una biblioteca en particular que tiene varias funciones que son útiles para la comprobación y manejo de caracteres que se detallan a continuación y se encuentran todas en `<cctype>`.

### 3.1. Prototipos

```
/* Funciones booleanas */
bool islower(char c); // true si es letra minúscula
bool isupper(char c); // true si es letra mayúscula
bool isalpha(char c); // Equivale a (islower() or isupper())
bool isdigit(char c); // true si es caracter numérico (0 a 9)
bool isalnum(char c); // Equivale a (isalpha() or isdigit())
bool isblank(char c); // true si es espacio(' ') o tab('\t')
bool isspace(char c); // true si ' ','\t','\v','\r','\n','\f'
bool isgraph(char c); // true si es imprimible excepto ' '
bool isprint(char c); // true imprimible y espacio (' ')
bool ispunct(char c); // true si es simbolo o puntuación
bool isxdigit(char c);// true si es hexa (0-9 o A-F o a-f)
/* Procesamiento de datos */
int toupper(char c); // Devuelve el ASCII mayúscula
int tolower(char c); // Devuelve el ASCII minúscula
```

### 3.2. Ejemplos

```
#include <iostream> // cout
#include <cctype>
using namespace std;
int main() {
  char c = toupper('c'); // se guarda el valor ASCII de 'C'
  cout << "Mayúscula de 'c' es '" << c << "'\n";
  char G = tolower('G'); // se guarda el valor ASCII de 'g'
  cout << "Minúscula de 'G' es '" << G << "'\n";
  if(isaplpha('h')) {
    cout << "'h' es letra...\n";
  }
  if(isdigit('9')) {
    cout << "'9' es número...\n";
  }
}
```

## 4. Manejo de entrada/salida segura

Aquí se expondrá una técnica para evitar errores al momento de validar datos ingresados por el usuario, es decir lo que ocurre cuando ingresan un carácter en vez de un número, ya sea entero o decimal.

### 4.1. getline()

El operador >> de cin tiene la limitación de leer un solo valor, es decir, que almacena un solo valor por cada variable y siempre en variables separadas. Esto es deseable con datos del

tipo numérico o caracteres sueltos, pero en cuanto a los textos (`string`) quizás deseemos que puedan almacenar más de una palabra. Para ello existe la función `getline()`, la cual recibe 2 argumentos. El *stream* de origen (en este caso `cin`) y la variable donde se almacenará la linea leída.

Programa 3: Leer una línea entera

```
#include <iostream>

using namespace std;

int main() {
  cout << "Ingrese su nombre completo: ";
  string nombre;
  getline(cin, nombre);
  cout << "¡Mucho gusto, " << nombre << "!" << endl;
}
```

Una posible salida de este ejemplo es:

```
Ingrese su nombre completo: Maria Elsa Payo Campante
¡Mucho gusto, Maria Elsa Payo Campante!
```

Esto sería de utilidad, ya que verdaderamente lo *entrante y saliente* por la consola es texto. No es recomendable mezclar la operación `cin >>` con `getline()`, es por ello que se optará por utilizar siempre este último.

## 4.2. Transformación de tipos

Desde la *C Standar Librady*, la cual se accede a través de la cabecera `cstdlib`, podemos encontrar las funciones que se encarga de transformar *Strings* en tipos numéricos, los cuales son conocidos como *ASCII to ...*[1] y se enumeran a continuación.

```
// ASCII a punto flotante (doble precisión)
double atof([c_str] s);
// ASCII a etnero
int atoi([c_str] s);
```

Si alguna de estas funciones no puede convertir el texto en número, en la variable se almacenará un '`0`'.

En rigor de verdad, `c_str` no es un tipo existente, pero los *strings* no son iguales en `C` que en `C++`. Pero, afortunadamente, los `strings` de `C++` saben como hacerse compatibles con `C` invocando a la función `c_str()` como se observa en el siguiente ejemplo:

Programa 4: Entrada segura de números

```
#include <iostream> // cout, cin
#include <cstdlib>  // atoi()

```

[1] Esta era la codificación estándar en el momento en que se implementaron estas funciones.

```cpp
using namespace std;

int main() {
  cout << "Ingrese un número entero: ";
  // Lectura de la entrada del usuario:
  string entrada;
  getline(cin, entrada);
  // Transformación a string de C y luego a entero:
  int num = atoi(entrada.c_str());
  // Se verifica si hubo un error en el ingreso:
  if(num == 0 and entrada != "0") {
    cout << "¡Error de conversión!" << endl;
  } else {
    cout << "Todo bien con " << num << endl;
  }
}
```

Posibles salidas del Programa 4 serían las siguientes:

```
Ingrese un número entero: -5
Todo bien con -5
```

```
Ingrese un número entero: 0
Todo bien con 0
```

```
Ingrese un número entero: w
¡Error de conversión!
```

Ahora, si necesitamos validar una entrada en un cierto rango podemos utilizar el siguiente algoritmo:

```cpp
string entrada;
double num;
do {
  cout << "Ingrese un número entre -2.5 y 2.5: ";
  getline(cin, entrada);
  num = atof(entrada.c_str());
} while((num==0 and entrada!="0") or num<-2.5 or num>2.5);
```

# 5. Utilitarias

Hay una cantidad basta de funciones en las bibliotecas estándar tanto de C como C++. Aquí se expondrán solo algunas que podrían ser de interés para ciertos algoritmos.

## 5.1. Tiempo

### 5.1.1. `time()`

Dentro de la biblioteca estándar de `C` se encuentran las funciones de manejo de tiempo accedidas a través de la cabecera `ctime`. Será de interés para el curso utilizar las funciones `time()` y `sleep()`. La función `time()` retornará el valor calendario actual del sistema sobre el cual se está ejecutando en un formato numérico entero y representan los segundos transcurridos desde *"The Epoch"* que es la hora 00:00UTC del 1 de enero de 1970[2].

Programa 5: Leer el tiempo calendario del sistema.

```
#include <iostream> // cout
#include <ctime>    // time y time_t

using namespace std;

int main() {
  time_t s = time(0);

  cout <<
        "Han transcurridos " << s << " segundos "
        "desde las 0hs UTC del 1 de enero de 1970." << endl;
}
```

Como se observa, time nos devuelve un valor del tipo `time_t`, el cual es, en el fondo, un entero no signado. No se hará mención del argumento de `time()`, siempre utilizaremos 0 para invocarlo.

Este dato es útil para contabilizar duraciones, implementar una aplicación que necesite de marcas de tiempo, agendas, calendarios, etc. Por ejemplo, podremos saber cuánto tiempo estuvo ejecutándose un programa.

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

int main() {
  time_t init = time(0);
  /* ... resto del programa ... */
  cout << "Tiempo de ejecución "
       << time(0)-init          // calculo tiempo transcurrido
       << " segundos." << endl;
}
```

### 5.1.2. `sleep()`

Otra función que podría llegar a ser de utilidad en cuanto al manejo del tiempo es la función `sleep()`. Ésta depende íntimamente de cómo está implementada desde el sistema

[2]Para más información puede consultar en https://es.wikipedia.org/wiki/Tiempo_Unix

operativo, es por este motivo que dependiendo sobre qué plataforma se esté programando debemos incluir una biblioteca distinta:

```
#include <windows.h> // Para MS Windows
#include <unistd.h>  // GNU/Linux, MacOS y otros basados en Unix
```

Su funcionamiento es sencillo, cuando se encuentre esta función en la lógica del programa este "se detendrá" por la cantidad de segundos pasadas por parámetro:

Programa 6: Escribir segundos transcurridos

```
#include <iostream>
#include <unistd.h> // sleep()
using namespace std;

int main() {
  for(int i=0; i < 10; i++) {
    sleep(1); // esperar 1 segundo
    cout << i+1 << "s transcurrido(s)...\n";
  }
  cout << "Fin del programa\n";
}
```

## 5.2. Números aleatorios

En algunas ocasiones será de interés generar números *al azar*, ya sea porque son parte de un algoritmo o para generar diferentes comportamientos cada vez que se ejecute nuestro programa.

Aquí se optará por utilizar de la biblioteca estándar `random` un tipo de dato llamado `random_device`, el cual consulta al sistema operativo para generar números *pseudo-aletarios*. Luego la variable se utiliza como si se tratara de una función.

Programa 7: Generar un número aletario al estiolo de C++

```
#include <iostream> // cout
#include <random>   // random_device

using namespace std;

int main() {
  random_device rand;
  int n = rand();
  cout << "Número generado aleatoreamente: " << n << endl;
}
```

Como se observa el valor generado es un entero, si se desea otro rango o números decimales, se deberá operar matemáticamente sobre el mismo para lograr los valores deseados como se observa en el siguiente ejemplo.

```cpp
// Números entre 0 y 9:
int n = rand() % 10;

// Números entre -100 y 100:
int n = rand()%201-100;

// Números entre 0.00 y 1.00
double n = rand()%101 / 100.0;

// Números entre -1.00 y 1.00
double n = (rand()%201-100) / 100.0;
```

# 6. Ejercitación

Intente resolver los siguientes problemas planteando los algoritmos en diagramas y codifíquelos en C++ **validando siempre las entradas por parte del usuario**, volviendo a pedir los datos ingresados en caso de error.

1. Pida ingresar dos números reales $a$ y $b$, ambos positivos o cero. Calcule el resultado de $\sqrt{a^b}$ y muéstrelo en pantalla redondeado a 2 decimales.
2. Pida ingresar los valores de 2 puntos en el plano $(x_1, y_1)$ y $(x_2, y_2)$. Calcule y muestre la distancia entre dichos puntos redondeados a 3 decimales aplicando el teorema de pitágoras $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
3. Pida ingresar un ángulo en radianes y el lado adyacente de un triangulo rectángulo. Calcule y muestre por pantalla el valor de la la hipotenusa y el opuesto de dicho triangulo. Recuerde que: $\sin(\theta) = \frac{OP}{HIP}$; $\cos(\theta) = \frac{AD}{HIP}$; $\tan(\theta) = \frac{OP}{AD}$
4. Repita el punto anterior agregando una opción previa al usuario con la cual decida si quiere ingresar el ángulo en radianes o grados. Y en caso de que elija grados haga la conversión pertinente para el cálculo.
5. Pida ingresar 3 variables reales $a$, $b$ y $c$ pertenecientes a una función cuadrática $(a.x^2 + b.x + c)$ e indique si posee raíces reales (2 simples o una doble) o no posee raíces reales. Calcule y muestre dichas raíces.
6. Pida al usuario ingresar el radio $r$ y el centro en el plano $c(h, k)$ de una circunferencia. Calcule y muestre su perímetro y área. Luego pida ingresar un punto cualquiera $p(x, y)$ e indique si se encuentra dentro, fuera o en el perímetro del círculo.
7. Implemente un programa que pida una cantidad de dados y de lados de dichos dados. Luego, *tire los dados*, muestre el valor de cada uno y el valor acumulado.
8. Implemente el programa del punto anterior agregando *cantidad de tiradas* y ejecútelas cada 5 segundos hasta finalizar. Los valores acumulados a mostrar deben ser de cada tirada, **no es relevante** el valor acumulado total de todas las tiradas.
9. Desarrolle una función que reciba un máximo y un mínimo, ambos enteros, y devuelva un número al azar dentro de ese rango incluyéndolos.