

Unidad I.

FUNCIONES Y PROCEDIMIENTOS (PROGRAMACION MODULAR)

- 1.1 Divide y vencerás.
- 1.2 Programación Modular.
- 1.3 Modularidad cohesión y acoplamiento
- 1.4 Variables locales y globales
- 1.5 Parámetros por valor y por referencia
- 1.6 Funciones y procedimientos
- 1.7 Ventajas y desventajas

COMPETENCIA A DESARROLLAR:

El estudiante:

- Comprende los fundamentos de programación, identificando el grado de modularidad, aplicando programación con funciones y procedimientos.

1. Divide y vencerás.

Paradigma bastante utilizado antiguamente como conquista territorial posteriormente se fue generalizando de manera que se fue aplicando en diversas situaciones.

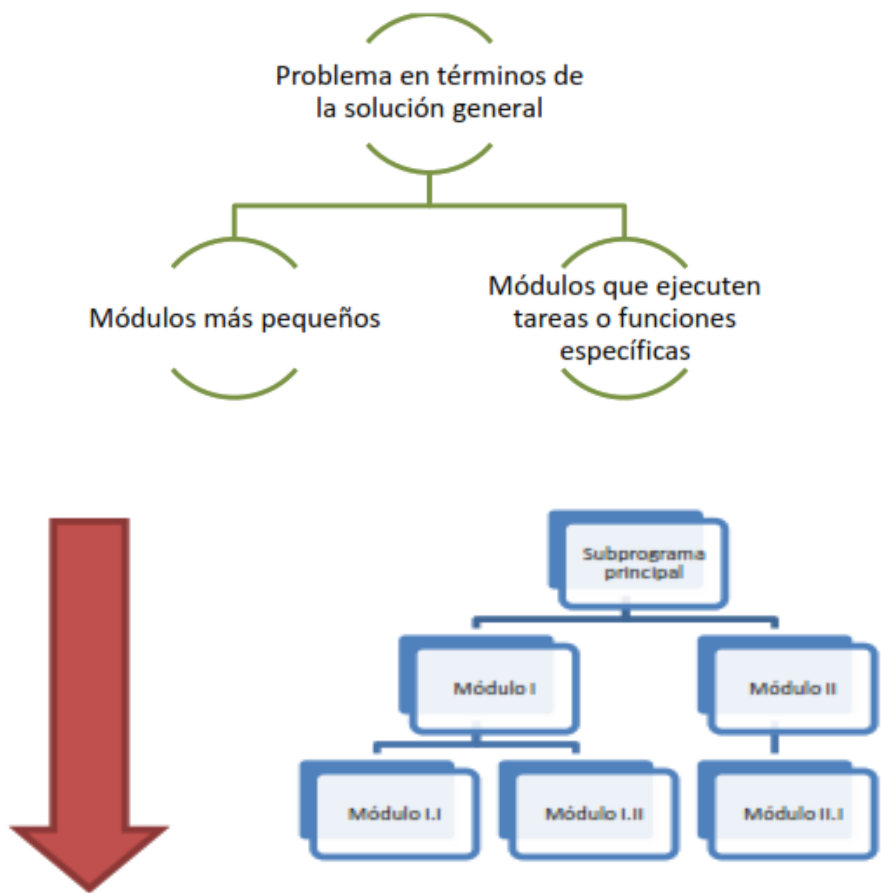


Actualmente esta técnica de dividir el problema principal en subproblemas se suele denominar paradigma "*divide y vencerás*" pues consiste en poder resolver un problema complejo dividiéndolo en subproblemas (problemas más sencillos) y a continuación dividir estos subproblemas en otros más simples, hasta que los problemas más pequeños sean fáciles de resolver y manejar.

Este paradigma también es utilizado en el mundo de la programación veamos como:

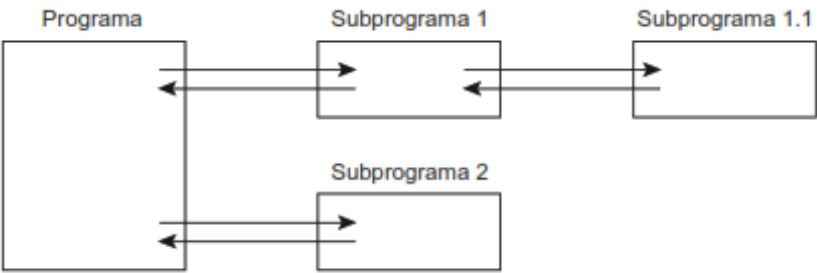
2. Programación Modular.

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible, manejable y reutilizable. También denominado diseño descendente(top-down), cada subprograma ejecuta una única función o actividad.



La programación modular permite:

- a) Dividir la complejidad de un problema convirtiendo problemas complejos en un conjunto de problemas más simples y por tanto más sencillos de implementar.



- b) Reutilizar el código de un programa en cualquier momento de la ejecución del mismo.

```
1  #include <iostream>
2  using namespace std;
3
4  /*PROCEDIMIENTO LEER HORA*/
5  void leer_hora(int &ht){
6      cout<<"Ingrese horas de trabajo de la semana ";
7      cin>>ht;
8  }
9  /*FUNCION SALARIO*/
10 float salario_semanal(float ht ){
11     float sdia=0;
12     if (ht<=40)
13         sdia=20*ht;
14     else
15         sdia=40*20+(ht-40)*25;
16     return sdia;
17 }//fin funcion
18
19 /*PROGRAMA PRINCIPAL*/
20 int main()
21 { float semp=0;
22   int ht;
23   cout<<"Salario de obrero 1"<<endl;
24   leer_hora(ht);
25   semp=salario_semanal(ht);
26   cout<<"El sueldo semanal del obrero 1 es ="<<semp<<endl;
27   cout<<"salario de obrero 2"<<endl;
28   leer_hora(ht);
29   semp=salario_semanal(ht);
30   cout<<"El sueldo semanal del obrero 2 es ="<<semp<<endl;
31   cout<<"salario de obrero 3"<<endl;
32   leer_hora(ht);
33   semp=salario_semanal(ht);
34   cout<<"El sueldo semanal del obrero 3 es ="<<semp<<endl;
35   return 0;
36 }//fin principal
37
```

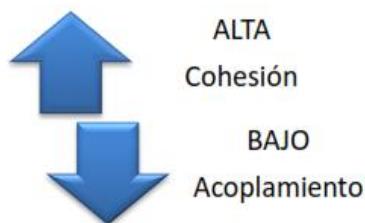
Reutilización de funciones
leer_hora() y
salario_semanal()

3. Modularidad, cohesión y acoplamiento.

Los módulos deben cumplir dos características básicas para ser candidatos a una buena división del problema.

- **Cohesión:** Las instrucciones contenidas dentro de un módulo deben contribuir a la ejecución de la misma tarea, es decir cada módulo debe realizar una única tarea o proceso.

- **Acoplamiento:** La dependencia entre módulos debe ser mínima, es decir mide el grado de relación de un módulo con los demás. Con un menor acoplamiento el modulo es más sencillo de diseñar, programar, probar y mantener.



4. Variables locales y globales.

Cuando se consideran varios subprogramas (funciones o procedimientos) y no un solo programa principal, se pueden declarar variables tanto en el contexto global del programa, como de manera local en cada subprograma a lo que se le conoce como alcance de las variables. Desde este punto de vista, las variables pueden ser de dos tipos: locales o globales.

1.4.1 variables locales

Las variables locales sólo existen en un ámbito determinado del programa, por ejemplo, en un subprograma o en un bloque de sentencias. Solo pueden ser utilizadas en el subprograma donde fueron definidas.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

/* Prototipo de la funcion*/
double cuadrado (double numero);

int main()
{
    double cuad=0; }
    double numero;
    cout<<" Introduce el valor de un número \n";
    cin>>numero;
    cuad = cuadrado(numero);
    cout<<" El cuadrado del número es "<<cuad;
    system("PAUSE");
    return 0;
}
```

Variables Locales

```
/* Funcion para el cálculo del cuadrado de un número*/  
double cuadrado (double número)  
{  
    double cuadrado;  
    cuadrado = numero * numero;  
    return cuadrado;  
}
```

Variable Local

1.4.2 variables globales

Son las que son accesibles desde cualquier punto del programa y se pueden usar desde cualquier módulo o subprograma, esto lleva a considerar que la variable puede usarse en cualquier parte del programa y su valor se puede alterar incontroladamente, lo cual puede ocasionar errores en el programa. La programación modular sugiere que se evite el uso de variables globales

```
main.cpp x  
1  #include <iostream>  
2  using namespace std;  
3  int ht;  
4  /*PROCEDIMIENTO LEER HORA*/  
5  void leer_hora() {  
6      cout<<"Ingrese horas de trabajo de la semana ";  
7      cin>>ht;  
8  }  
9  /*FUNCION SALARIO*/  
10 float salario_semanal(int ht ) {  
11     float sdia=0;  
12     if (ht<=40)  
13         sdia=20*ht;  
14     else  
15         sdia=40*20+(ht-40)*25;  
16     return sdia;  
17 } //fin funcion  
18  
19 /*PROGRAMA PRINCIPAL*/  
20 int main()  
21 { float semp=0;  
22     leer_hora();  
23     semp=salario_semanal(ht);  
24     cout<<"El sueldo semanal del empleado es "<<semp;  
25     return 0;  
26 } //fin principal  
27
```

Variable Global

5. Parámetros por valor y por referencia

Entre los subprogramas que componen la estructura completa de un programa, se establece una serie de comunicaciones a través de determinadas interfaces (llamadas a los módulos) que permiten el paso de información de un módulo a otro, así como la transferencia del control de la ejecución del programa.

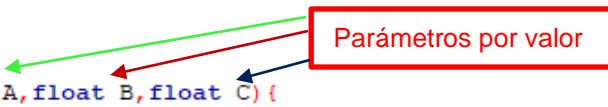
De manera más formal se puede definir un parámetro como un valor que se pasa al subproceso cuando éste es invocado.

- En la definición del subprograma los nombres de los parámetros definidos se denominan parámetros formales.
- En las invocaciones al subprograma desde algún punto del programa, los valores que se pasan al módulo se denominan parámetros actuales (reales).

1.5.1 Parámetros por valor

En este caso el subprograma recibe una copia del valor que se le pasa. La variable original (parámetro actual) no cambia de valor, independientemente de que en el subprograma se cambie el contenido del parámetro formal.

```
1  #include <iostream>
2  using namespace std;
3
4  //PROTOTIPO FUNCION
5  float area_pahuichi(float, float, float);
6  /*PROGRAMA PRINCIPAL*/
7  int main()
8  {
9      float A,B,C;
10     cout << "valor de A?"; cin>>A;
11     cout << "valor de B?"; cin>>B;
12     cout << "valor de C?"; cin>>C;
13     cout << "El area del terreno A es = "<< area_pahuichi(A,B,C);
14     return 0;
15 }
16 /*funcion area pahuichi*/
17 float area_pahuichi(float A, float B, float C){
18     float area=0,D=0;
19     D=A-C;
20     area= B*C+(D*B/2);
21     return area;
22 }
23
```



1.5.2 Parámetros por referencia

En el paso por referencia no se pasa una copia del valor sino la identificación de la zona de memoria donde se almacena dicho valor.

```

1  #include <iostream>
2  using namespace std;
3
4  /*PROCEDIMIENTO LEER HORA*/
5  void leer_hora(int &ht){
6      cout<<"Ingrese horas de trabajo de la semana ";
7      cin>>ht;
8  }
9  /*FUNCION SALARIO*/
10 float salario_semanal(int ht ){
11     float sdia=0;
12     if (ht<=40)
13         sdia=20*ht;
14     else
15         sdia=40*20+(ht-40)*25;
16     return sdia;
17 } //fin funcion
18
19 /*PROGRAMA PRINCIPAL*/
20 int main()
21 { float semp=0;
22   int ht=0;
23   leer_hora(ht);
24   semp=salario_semanal(ht);
25   cout<<"El sueldo semanal del empleado es "<<semp;
26   return 0;
27 } //fin principal
28
  
```

Parámetro por referencia

```

void dividir(int dividendo, int divisor, int& coc, int& resto)
{
    coc = dividendo / divisor ;
    resto = dividendo % divisor ;
}

int main()
{
    int cociente ;
    int resto ;
    dividir(7, 3, cociente, resto) ;
    // ahora 'cociente' valdrá 2 y 'resto' valdrá 1
}
  
```

Parámetros por referencia

6. Funciones y procedimientos

1.6.1 Funciones.

Una función es son un conjunto de instrucciones que realizan una tarea específica, usualmente reciben parámetros cuyos valores se utilizan para efectuar operaciones y retornar un solo valor.

Tenemos:

a) Funciones predefinidas estándar de C++

Función	Descripción	Tipo Dato Argumento	Tipo Dato devuelto	Ejemplo	Valor devuelto ejemplo	Librería (")
sqrt	Raíz cuadrada	double	double	sqrt(4.0)	2.0	math.h
		float	float	sqrt(4.0)	2.0	
		long	long	sqrt(4)	2	
pow (*)	Potencias	double	double	pow(2.0, 3.0) pow(2.0, 3)	8.0 8.0	math.h
		float	float	pow(2.0, 3.0) pow(2.0, 3)	8.0 8.0	
		long double	long double	pow(20.0, 3)	8000.0	
abs	Valor Absoluto	double	double	abs(-7.0) abs(7.0)	7 7	math.h
		float	float	abs(-7.0) abs(7.0)	7 7	
		int	int	abs(-7) abs(7)	7 7	
		long	long	abs(-70000) abs(70000)	70000 70000	stdlib.h
		long double	long double	abs(-70000.0) abs(70000.0)	70000.0 70000.0	
ceil	Techo(redondeo arriba)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	math.h
		float	float	ceil(3.2) ceil(3.9)	4.0 4.0	
		long double	long double	ceil(30000.2) ceil(30000.9)	30001.0 30001.0	
floor	Piso(redondeo abajo)	double	double	floor(3.2) floor(3.9)	3.0 3.0	math.h
		float	float	floor(3.2) floor(3.9)	3.0 3.0	
		long double	long double	floor(30000.2) floor(30000.9)	30000.0 30000.0	
exp	Exponencial	double	double	exp(1.0)	2.7	math.h
		float	float	exp(1.0)	2.7	
		long double	long double	exp(10.0)	22026.5	
log	Logaritmo natural(ln)	double	double	log(5.0)	1.6	math.h
		float	float	log(5.0)	1.6	
		long double	long double	log(1000.0)	6.9	
log10	Logaritmo base 10 (ln)	double	double	log10(10.0)	1.0	math.h
		float	float	log10(10.0)	1.0	
		long double	long double	log10(1000.0)	3.0	

b) Funciones definidas por el usuario

SINTAXIS:

- Prototipo de la función.

Un prototipo de función le indica al compilador el tipo de dato que regresa la función, el número de parámetros que la función espera recibir, los tipos de dichos parámetros, y el orden en el cual se esperan dichos parámetros. El compilador utiliza los prototipos de funciones para verificar las llamadas de función.

- Cuerpo de la función.

El formato de una definición de función es:

tipo_de_valor_de_retorno < nombre_la_función> (tipo param1, tipo param2,...)

```
{  
    Declaraciones;  
    Sentencias;  
}
```

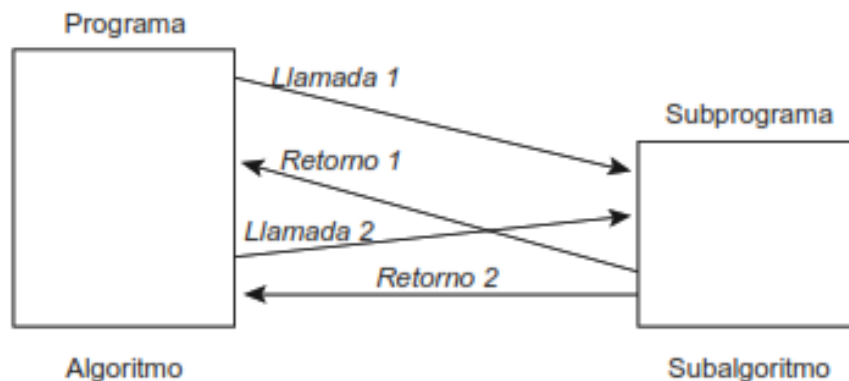
Donde:

tipo_de_valor_de_regreso: El tipo de valor de regreso el valor que una función devolverá.

nombre_de_la_función: Es cualquier identificador valido.

lista_de_parámetros: Lista separada por comas que contiene las declaraciones de los parámetros recibidas por la función al ser llamada.

Declaraciones y Sentencias: forman el cuerpo de la función. El cuerpo de la función también se conoce como un bloque.



- **Llamado o invocación a una función**

Cuando definimos una función solo le indicamos al programa que esta función existe, pero una definición de función no implica la realización de las instrucciones que la constituyen. Para hacer uso de una función, el programa principal la debe llamar o invocar.

Por ejemplo:

```
1  #include<iostream>
2  #include<math.h>
3  using namespace std;
4  //function formula
5  float formula(int x,int y,int z ){
6  float resul;
7  resul=(sqrt(3*pow(x,2)+2*pow(y,2)*z)+5*pow(x,22)*y)/(3+pow(x,2)*pow(y,2)*pow(z,2));
8  return resul;
9  }//fin funcion
10
11 int main(){
12 int x,y,z;
13 float r;
14 cout<<"Ingrese valor de x ";cin>>x;
15 cout<<"Ingrese valor de y ";cin>>y;
16 cout<<"Ingrese valor de z ";cin>>z;
17 r=formula(x,y,z);
18 cout<<"\nEL VALOR DE LA FORMULA ES="<<r;
19 return 0;
20 }
21
```

Llamada a la función formula

1.6.2 Procedimientos.

Un procedimiento es un conjunto de instrucciones que realiza una tarea específica que proporciona cero, uno o varios valores en función de los parámetros definidos en su formato.

En lenguaje C++ un procedimiento es básicamente una función void, que no utiliza una sentencia return.

Para trabajar con procedimientos es el mismo proceso que para funciones descrito anteriormente, para invocarlo, es decir, para hacer que se ejecute, basta con escribir su nombre en el cuerpo de otro modulo o en el programa principal.

SINTAXIS:

void nombre_proc (tipo param1, tipo param2,...)

```
{  
    Declaraciones;  
    Sentencias;  
}
```

- La programación modular requiere más memoria y tiempo de ejecución.

```
1  #include <iostream>  
2  #include<string.h>  
3  using namespace std;  
4  /*Prototipos de procedimientos*/  
5  void leer(string &usuario,string &pass);  
6  void clave();  
7  /*PROGRAMA PRINCIPAL*/  
8  int main()  
9  {  
10     clave(); ← Llamada al Procedimiento clave  
11     return 0;  
12 }  
13  
14 /*PROCEDIMIENTO LEER*/  
15 void leer(string &usuario,string &pass){  
16     cout<<"Ingrese usuario ";  
17     cin>>usuario;  
18     cout<<"Ingrese clave ";  
19     cin>>pass;  
20 }  
21 /*PROCEDIMIENTO CLAVE*/  
22 void clave(){  
23     string usuario=" ",pass=" ";  
24     leer(usuario,pass); ← Llamada al Procedimiento leer  
25     if(usuario == "ADMIN" && pass == "EMI2019")  
26     {  
27         cout<<"Acceso Permitido!!";  
28     }  
29     else{  
30         cout<<"Acceso Denegado!!!";  
31     }  
32 }  
33
```

7. Ventajas y desventajas de la programación modular

ventajas

Facilidad de mantenimiento y corrección de errores.

Permite la reutilización de código.

Favorece la división del trabajo en un equipo de programadores.

Programas más fáciles de entender.

Desventajas

No se dispone de algoritmos formales de modularidad, por lo que a veces los programadores no tienen claras las ideas de los módulos.

La programación modular requiere más memoria y tiempo de ejecución.

