Unidad 06: Descomposiciones y funciones

6

Contenidos analíticos

Parte II: Constructos del Lenguaje: Expresiones, Declaraciones, Sentencias y Estructura de Programa

Unidad 5: Descomposiciones y funciones
La necesidad de la descomposición. Declaración y definición de funciones: argumentosy parámetros; invocación y declaración. La propiedad de la Reusabilidad. Ventajas de la generalidad de tipo y de criterio de selección.

Introducción

SI L = L 1 + L 2
Entonces
$Esfuerzo_{(L)} > Esfuerzo_{(L1)} + Esfuerzo_{(L2)}$

Modularizacion

Es conveniente, e importante descomponer por varias razones:

- Favorece la comprensión.
- Favorece el trabajo en equipo.
- Favorece el mantenimiento.
- Permite la reusabilidad del código.
- Permite la generalidad de tipos y de criterios
- Permite además separar la lógica de la algoritmia

Datos locales y globales

Unos se declaran en la sección de declaración del programa principal, los otros en la sección de declaración de cada modulo. Otros pueden ser declarados en un bloque determinado, por lo que solo tendrá visibisilidad en el mismo.

- El alcance de un identificador es el bloque del programa donde se lo declara.
- Si un identificador declarado en un bloque es declarado nuevamente en un bloque interno al primero el segundo bloque es excluido del alcance de la primera sección. Algunos lenguajes permiten la incorporación de operadores de ámbito.

Ocultamiento y protección de datos

Todo lo relevante para un modulo debe ocultarse a los otros módulos. De este modo se evita que en el programa principal se declaren datos que solo son relevantes para un modulo en particular y se protege la integridad de los datos.

Parámetros

Son variables cuya característica principal es que se utilizan para transferir información entre módulos.
Hay dos tipos de parámetros, los pasados por valor y los pasados por referencia o dirección.
Los parámetros pasados por referencia o dirección no envían una copia del dato sino envían la dirección de memoria donde el dato se encuentra por lo que tanto el proceso que lo llama como el proceso llamado pueden acceder a dicho dato para modificarlo.

Integridad de los datos

Es necesario conocer que datos utiliza con exclusividad cada modulo para declararlos como locales al modulo ocultándolo de los otros, si los datos pueden ser compartidos por ambos módulos debería conocerse cuales son, si el modulo los utiliza solo de lectura o puede modificarlos y es aquí donde se utilizan los parámetros.

Protección de datos

Si una variable es local a un modulo se asegura que cualquier otro modulo fuera del alcance de la variable no la pueda ver y por lo tanto no la pueda modificar.

Uso de parámetros para retornar valores

Si bien el pasaje por valor es útil para proteger a los datos, existen situaciones en las que se requiere hacer modificaciones sobre los datos y se necesitan conocer esas modificaciones. Para esto se deben utilizar parámetros pasados por referencia.

Utilidad del uso de parámetros

El uso de parámetros independiza a cada modulo del nombre de los identificadores que utilizan los demás. En el caso de lenguajes fuertemente tipiados solo importa la correspondencia en cantidad

tipo y orden entre los actuales del llamado y los formales de la implementación con independencia del nombre del identificador.

Reusabilidad

El uso de parámetros permite separar el nombre del dato, del dato en si mismo, lo que permite que el mismo código sea utilizado en distintas partes del programa simplemente cambiando la lista de parámetros actuales o argumentos.

Generalidad

Las funciones retornan valores en sus nombres, si ese valor es void, es un valor ausente, también puede retornar escalares, struct o punteros, en general esto es fuertemente tipado, es decir debe respetar el tipo. Muchas veces la lógica que desarrolla una función puede ser la misma solo cambiando el tipo de dato, es decir, supongamos que hacemos una función suma, que recibe dos parámetros pasados por valor y retorna su suma, si son enteros recibe dos enteros y retorna un valor entero con la suma de los mismos. Si se invocara con dos datos de tipo float, por ejemplo, la lógica de resolución es la misma, solo que difiere en el tipo de dato. Esto puede resolverse de dos formas: sobrecargando las funciones o utilizando plantillas. Esto lo abordaremos en este mismo documento, también la generalidad se puede dar, en cuestiones mas complejas según un criterio de selección, esto lo abordaremos cuando veamos estructuras de datos donde incorporaremos el termino puntero a funciones

Ejemplo en C++.

```
Funcion que retorna un escalar y evalua parámetros por valor
int suma1(int a, int b){
   return a+b;
}
float suma1(float a, float b){
   return a+b;
}
Int C=suma1(10*4,15*2);
```

Los argumentos vinculados con parametros valor pueden ser expresiones. Como las expresiones son enteras puede emparejar (matchear) con la definición de la primera función

```
float d = suma1(3.5, 8.4)
```

como la función esta “sobrecargada” hay otra definición con el mismo nombre, la invocación no empareja con la primera definición, pero si con la segunda, y esa se ejecuta

Utilizando plantillas

Definición de la función

```
template<typename T> T suma1(T a, T b){
        return a+b;
}
```

Invocacion

```
int c = suma1<int>(4,5);
float d = suma1<float> (3.4, 2.8);
```

el tipo de dato con el que se invoca entre los corchetes angulares se vincula con el identificador T declarado en el typename

```
void suma2(int a, int b, int& c){//&c por referencia, por dirección la modificacion dec altera x
   c = a + b;
   return;
}
Suma2(25, 82, x);
Argumento vincula parámetro pasado por referencia debe se valorL

void Intercambio ( int &a, int &b)
{// comienzo del bloque de declaración de la funcion
        int auxiliar;       //declaración de una variable auxiliar t
        auxiliar = *a;     //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = auxiliar;       //asignación a b del valor que contenía originalmente a.
} // fin del bloque de la funcion
```

Reformule los ejemplos anteriores para obtener generalidad de tipo

Beneficio del uso de acciones y funciones

Una acción o funcion tiene cuatro propiedades esenciales. Ellas son:

1. Generalidad
2. Ocultamiento de información
3. reusabilidad
4. Modularidad

De estas propiedades, se deducen una serie de beneficios muy importantes para el desarrollo de algoritmos.

1. Dominar la complejidad
2. Evitar repetir código
3. Mejorar la legibilidad
4. Facilitar el mantenimiento
5. Favorecer la corrección
6. Favorecer la reutilización

C++. agrega una característica de reutilización de código realmente poderosa que es el concepto de generalización de tipos mediante plantillas y criterios mediante punteros a funciones.

Las definiciones de las plantillas comienzan con la palabra template seguida de una lista de parámetros entre **<** y >, a cada parámetro que representa un tipo se debe anteponer la palabra

La función genérica según el tipo es:

```
//definicion de la plantilla de la función intercambio
template < typename T>
void Intercambio ( T &a, T &b) //con dos parámetros de tipo generico
{// comienzo del bloque de declaración de la funcion
        T t;      //declaración de una variable auxiliar t
        t = a;    //asignación a t del valor de a, para contenerlo
        a = b;    //asignación a a del valor de b para intercambiarlo
        b = t;    //asignación a b del valor que contenía originalmente a.
    return;
} // fin del bloque de la funcion
```

Invocación intercambio
Sea int a=10, b=5;
float c=3.0, d=5.0;
intercambio<int>(a,b);
intercambio<float>(c,d);

Concepto de recursividad:

Es un proceso que se basa en su propia definición. Una función puede invocarse a sí misma como parte de los tratamientos de cálculo que necesita para hacer su tarea
Parte de instancias complejas y las define en términos de instancias más simples del mismo problema, llegando a un punto donde las instancias más simples son definidas explícitamente.
Define el problema en términos de un problema más simple de la misma naturaleza.
Debe disminuir el espacio del problema en cada llamada recursiva
Hay una instancia particular que se conoce como caso base o caso degenerado
Divide el problema original en subproblemas más pequeños. Cuando es lo suficientemente chico se resuelve directamente y se combinan soluciones del subproblema hasta que queda resuelto el problema

Tiene:

- ✓ Una ecuación de recurrencia, en función de términos anteriores $T_n = F(T_{n-1}, T_{n-2}, T_0)$.
- ✓ Uno o varios términos particulares que no dependen de los anteriores. Ti = $G_{(i)}$ (base)

Funcion Factorial

- ✓ Ecuación de recurrencia : $n! = n * (n-1)!$
- ✓ Condiciones particulares: $0! = 1$

Instancias que permanecen en memoria:

Funcion PotenciaNatural

- ✓ Ecuación de recurrencia : $a^n = a^{(n-1)} * a$ si $n > 1$
- ✓ Condiciones particulares: $a^0 = 1$

Funcion DivisionNatural

Dados dos valores num y den, con den <> 0 se puede definir el cálculo del cociente y el resto del siguiente modo:

- ✓ Si num < den → el cociente es = y el resto num.

- ✓ Si num <= den y si c y r son el cociente y resto entre num-den y den → cociente = c+1 y resto r.

Resumen:

Aquí se avanza sobre la necesidad de mayor abstracción procedural introduciendo conceptos claves de programación como acciones y funciones como la forma mas adecuada de estructurar problemas en el paradigma procedural. Es una introducción a la abstracción procedural y de datos que servirá de base para sustentar futuros conocimientos de estructuración de programas en clases cuando se aborde, en otra instancia, la programación orientada a objetos.
Se dio valor a términos como reusabilidad, ocultamiento de datos, polimorfismo y cohesión. Se introdujeron conceptos como parámetros actuales, argumentos, parámetros formales, pasaje por valor, pasaje por referencia. Si introdujo un concepto desde el punto de vista de la algoritmia importante como la programación genérica que permite el lenguaje que estamos abordando.

Resumen de funciones

Proposito de las funciones

1. Permite la descomposición como forma de alcanzar la solución
    a. Si L = L1 + L2 → Esf(L) > Esf(L1) + Esf(L2)
2. Promueve la modularidad
3. Favorece
    a. Comprension
    b. Trabajo en equipo
4. Facilita el codigo
    a. Evita repeticiones
5. Perimte
    a. Integridad y protección del dato
    b. Reusabilidad → uso de parametros
    c. Separar la lógica de la algoritmia → funciones de criterio
    d. Separar la lógica del tipo de dato → plantillas

Programacion Modular

1. Propone dividir el problema en modulos → funciones
2. Cada módulo permite aislar con mas precision el problema
3. En C/C++ todo programa ejecuta una función → main()
4. Dentro de la función principal se invocan otras funciones
5. Cada función invocada tiene asignada una tarea especifica.
6. La función, luego de ejecutarse vuelve al punto de invocación

Invocacion/declaración → distintos esquemas

Definicion anticipada	Declaracion definición	Bibliotecas propias

Definición de la función
int f(int a, int b){
return a + b;
};
int main()
{ ……..
Invocación
10 5
x = f(y,z);argumentos
f(x,y)
w = f (3*2,4);
….
}
	Declaración de la firma
	int f(int , int);
	
	int main()
	{ ……..
	invocación
	x = f(y,z);;
	…
	w = f (3,4);
	….
	}
	Definicion de la funcion
	int f(int a, int b){
	return a + b;
	};
		Inclusión de la biblioteca
		#include “miBiblio.h”
		
		int main()
		{ ……..
		
		invocación
		x = f(y,z);;
		…
		w = f (3,4);
		….
		}
		La biblioteca separa los prototipos en un archivo .h y el código en un archivo .c

Vimos que la declaración de la variable es darle al identificador que la señala el tipo de dato al que pertenece por ejemplo int a, con esto se le indica al compilador como debe interpretar a ese identificador, en este caso una variable de tipo entero.
La firma o prototipo de la función es también darle al compilador las características de modo de poder tener una buena interpretación del identificsdor de la funcion

Declaracion de una funcion

Firma o prototipo
Tipo de retorno nombre (lista de tipo de dato del parametro y tipo de parametro)
int suma(int, int);

1. Tipos de retorno
    a. Escalares
    b. Srtruct
    c. Punteros
    d. Valor ausente (void)
2. Nombre → regla identificadores Letra(Letra+Digito)*
3. Argumentos
    a. Por valor tipo
    b. Por referencia o dirección tipo&

Definicion de una función

Tipo de retorno nombre (lista parámetros su tipo y tipo de dato){
int suma(int a, int b){
Cuerpo de la función
}

1. Tipos de retorno
    a. Escalares
    b. Srtruct
    c. Punteros
    d. Valor ausente (void)
2. Nombre → regla identificadores L(L+D)*
3. Parametros
    e. Por valor tipo identificador
    f. Por referencia o dirección tipo& identificador
2. Cuerpo de la función
    {
    Declaraciones locales;
    Acciones;
    Retorno;
    }

Invocaciones

1. Con valor ausente una invocación a si misma
    a. Nombre(lista de argumentos)
2. Retornando escalar, puntero o struct en una expresión
    a. Identificador = Nombre(lista de argumentos)
3. Los argumentos vinculados con parámetros variables deben ser valorL
4. Los argumentos vinculados con parámetro valor son expresiones, completas o incompletas
5. C++ pasa parámetros por valor o referencia, eso se evidencia en la declaración y definición pero no en la invocacion

Ejemplos

Funcion	Detalle
Firmas o prototipos	
void F1(int)	Firma de función que no retorna valor y recibe un parámetro por valor
void F2(int&)	No retorna valor pero evalua y modifica el argumento vinculado al parámetro
int F3(int)	Similar a F1 retornando un escalar
int F4(int&)	Similar a F2 retornando un escalar
int F5(int, int)	Evaluando dos parametros valor
int F6(int, int&)	Evaluando un parámetro por valor y otro por referencia
int F7(void)	Declaración explicita sin parámetros
Invocacionei	
F1(3*6) F1(a) F1(a+b)	Invocado con una expresión completa vinculada al parámetro valor Imvocado con un identificador (expresión incompleta)
F2 (a)	Invocado con un valor por el ásaje por referencia
A = F3(2*4) A = F3(b)	Un parámetro valor
A = F4(X)	Un parámetro por referencia
A = F5(x, 1+2)	Dos parametros valor
A = F6(14, w)	Un parámetro valor y otro por referencia
A = F7()	Sin parámetros

Definiciones: Ejemplos

```
int suma1 (int a, int b){
return a + b;
}
suma1(3*4, 8)
X=suma1(y,z)

void suma2 (int a, int  b, int& c){
 c = a+b;
return;
}
          10 5 ?
Suma2(x ,  y,   z)
Intercambio(x,y)
void intercambio (int &a, int &b){
int c = a;
a = b;
b =  c;
return;
}
int suma1(int a , int b)
{
return a+b; }
```

```
Int y;
Int main(){
Int x;

X = f(y,z)

}
void suma2( int a, int b, int& c){
 c = a +b;
return;
}

Int main(){
Int h=10, i=15, j, k=20, l=30, m, n;
J=suma1(h,i);
m=suma1(suma1(h,i),3*4);h,i)

suma2(h,i,j);
cout<< j;
```

Pasaje de cadenas como parametros

int F (char* s) o int F (char s[])
Pasaje de vectores y matrices
int F (int v[])
int F (int m[][columnas]

Reusabilidad ejemplos

A = suma1(B, C);
D = suma1(D, E);
F = suma1 (2*3, 4-2);
Se invoca en distintos puntos del programa con conjunto de argumentos diferentes respetando orden yipo y características. Recordando que los que se vinculan con parámetros variables deben ser valor, los que se vinculan con parámetros valor spn expresiones

Generalidad

1. Según el tipo de dato → plantillas
2. Según el criterio de selección → punteros a funciones

Generalidad por tipo de dato	
Sin generalizar	**Generalidad con plantilla**

int suma1 (int a, int b){
int c = a+b;
return c;
}
 	
float suma1 (float a, float b){
float c = a+b;
return c;
}
 	
Invocación
X = suma1 (y,z)
	template < typename T>
	T suma1 (T a, T b){
	return a + b;
	}
	
	Invocación
	X = suma1<int> (y,z)
	X = suma1<float>(y,z)

Logros Pedagógicos: Aplicar las abstracciones de los lenguajes de programación para la implementación de los algoritmos mediante la descomposición a través de funciones.

Si optara
Parte II: Constructos del Lenguaje: Expresiones, Declaraciones, Sentencias y Estructura de Programa

Para la verificación de los mismos sugerimos estos principios de auto evaluacion. Tenga en cuenta que la formación es incremental asi como la adquisición de sus competencias en el área de programación como elemento constitutivo de su formación como Ingeniero en Sistemas de Información de UTN.BA

Consigna 1

Dado un conjunto de ternas de valores determinar e imprimir, cuantas de esas ternas no forman triángulos, cuantas forman triángulos equiláteros, cuantas triángulos isósceles y cuantas escalenos

Recuerde que llevar del mundo real al mundo computacional requiere un importante nivel de abstracción, interpretacion y definiciones precisas.
Por ejemplo, que significan en el universo computacional términos tales como: Dado, Determinar, Imprimir. Como se obtiene una terna? Cuantas ternas se evaluan? Es preciso el tamaño del lote? Como resolvería el hecho de alguna ambigüedad? Cuantos identificadores requiere para las ternas, los necesarios para una sola terna o para las muchas que evalua? Además de la decisión de los identificadores de las ternas, que otros identificadores requiere para poder “determinar” lo que se pide? Cuales son las condiciones para que formen triangulo? Estratégicamente le es mas útil resolver todo junto o estima mejor descomponer en dos subgrupos: determinar si forman triangulo por un lado y al tener certeza de esto determinar que tipo es? Si optara por esta segunda alternativa

dende resolvería la condición de no negatividad de los valores de la terna? Si optara por la descomposición que precondición cree que podría agregar?

Desarrollo las siguientes funciones, haciendo referencia a reusabilidad y generalidad en caso que pueda corresponder. Desarrolle un programa con ejemplo de invocaciones. Tenga en cuenta relaciones entre argumentos y parámetros asi como una correcta utilización de parámetros variables y parámetros valor

1. Desarrollar una función que reciba tres enteros de un digito (Unidad, Decena, Centena) en el rango [0..9] y retorne un entero de tres dígitos con la forma CDU.
2. Idem anterior recibiendo cinco enteros de un digito y componiendo un entero de cinco dígitos
3. Desarrollar una función que reciba un entero de tres dígitos en el rango [100..999] y lo descomponga en cada uno de los tres dígitos
4. Idem anterior con cinco enteros de un digito componiendo uno de cinco dígitos
5. Desarrollar una función que dados tres valores enteros que representan la fecha de nacimiento de una persona retorne un entero único con la fecha con formato AAAAMMDD
6. Desarrollar una función que dada la fecha de nacimiento con formato AAAAMMDD devuelva tres enteros de representen el año, el mes y el dia
7. Desarrolle una función que dados el dia de nacimiento, mes de nacimiento y año de nacimiento de dos personas indique quien es el mayor. Plantee dos estrategias de solución diferentes
8. Desarrolle una función que reciba un valor que representa el radio de una circunferencia y devuelva el perímetro de la misma y el área del circulo que genera
9. Desarrolle una función iterativa que reciba dos enteros (base y exponente) y calcule la potencia
10. Desarrolle la consigna anterior con una función recursiva
11. Desarrolle una función que reciba tres valores del mismo tipo (mayor, medio y menor) y que los devuelva ordenados (el mayor valor en mayor, el menor valor en menor y el otro en medio. Especifique precondiciones