Resumen parcial 1 programacion.

**Punteros y Funciones**

Utilidad de una **función**:

La función es un recurso que permite la modularización de un programa. A través de ellas se puede descomponer un programa en varias funciones y así hacerlo mas simple, lo que facilita la construcción y arreglo del mismo.

Mas ventajas:

* Disminuir la cantidad de líneas de código.
* Disminuir la cantidad de errores.
* Reduce la complejidad del programa.
* Elimina código duplicado.
* Limita los efectos de los cambios.
* Ocultar detalles de implementación
* Promueve la reutilización de código.
* Mejora la legibilidad.
* Facilita la portabilidad de código.

Definición

Conjunto de instrucciones que realizan una tarea especifica. En general toman valores de entrada, llamados parámetros, y nos devuelven un valor de salida o de retorno. Al igual que con las variables, las funciones pueden declararse y definirse. Una declaración es simplemente una presentación, una definición contiene las instrucciones con las que realizará su trabajo al función.

Una vez definida, una función puede invocarse. En general, la definición de una función se compone de las siguientes secciones:

<tipo de retorno><nombre de la funcion>(<lista de parámetros formales>)

{

<Instrucciones>

Return <valor> (si es necesario)

}

Una vez invocada la función puede usarse en diferentes partes del programa para que realice una determinada tarea. Dependiendo de los parámetros usados la función arrojara diferentes resultados.

Definición de la función:

* Tipo de valor de ratorno
* Nombre de la función
* Lista de los parámetros entre paréntesis
* Cuerpo de la función

**PUNTEROS**

Direccion de memoria

Si bien utilizamos nombres para identificar una variable y manipular datos, en realidad, los datos se almacenan en direcciones de memoria. Las variables reemplazan a las direcciones de memoria para hacer la programación mas agradable y dejando al compilador la traducción de variable a dirección de memoria.

Un puntero es por definición una variable que almacena la dirección de memoria de otra variable. De sintaxis: tipo\_dato \* id\_variable;

En donde id\_variable guarda la dirección de tipo\_dato.

Ejemplo: int \* p;

Para poder acceder a una variable ya apuntada se utiliza el operador “\*”.

Entonces si “p” contiene la dirección de memoria de la variable \*p.

A las variables no se le puede asignar un valor arbitrario o una ubicación determinada por el programador. Para que esto suceda se utiliza el operador & o la función malloc()

& retorna la dirección de memoria de la variable

Malloc() busca una zona disponible de la memoria, la reserva y la retorna.

El pasaje de parámetros se realiza al momento de la invocación de la función

Existen dos tipos

PASAJE POR COPIA: las copias del parámetro real no tienen la capacidad de modificarlo en el transcurso de la implementación de la función. Solo se modifica el parámetro actual pero al momento de terminar la función el parámetro vuelve a valer lo que valia al principio.

PASAJE POR REFERENCIA:

Se utiliza una variable puntero como parámetro formal y el parámetro actual es una dirección de memoria. Lo que se copia es la dirección de memoria de la variable que oficia de parámetro actual. Entonces dentro de la función se puede acceder directamente al contenido de la variable a través del operador \*. Es posible modificar el contenido de la direccion de memoria pero no el parámetro.

**RECURSIVIDAD**

Una función recursiva es aquella que durante su ejecución se invoca directa o indirectamente a sí misma. Esta invocación depende de al menos una condición que actúa como condición de corte que genera el fin de la auto invocación. De esta manera llegamos a una solución de menor complejidad a la cual la llamamos Solucion Trivial.

En el momento de realizar una invocación recursiva, se suspende la ejecución de la función invocante hasta que se termina de resolver la función invocada. Si bien es la misma función la que se invoca, se genera un nuevo espacio de memoria para la resolución de la misma. Cada espacio tiene sus propias variables locales y parámetros, con valores propios.

REGLAS DE LA BUENA RECURSIVIDAD.

* Al menos una condición de corte con su consecuente solución trivial.
* Al menos una llamada recursiva.
* Que en cada llamada recursiva nos acerquemos a la solución trivial.
* Que al llegar a la solución trivial quede expresada la solución total.

USAMOS RECURSIVIDAD CUANDO.

* Cuando una función es repetitiva y haciendola recurrente escribimos menos código.
* Cuando la estructura de datos es recursiva.

Para recorrer una lista hay que tener en cuenta que no hace falta un ni WHILE ni FOR. Ni ninguna estructura de repetición.

* La recursión se encarga de repetir el algoritmo.
* La llamada recursiva con el acercamiento al corte hace que la función recorra la lista.
* La condición de corte se encarga de finalizar el recorrido de la lista.

La recursividad es costosa en términos de memoria ya que el programa debe resolver y almacenar cada una de los cálculos hechos para luego devolvlerlos, y no siempre es lo optimo al momento de buscar respuesta al problema. se recomienda el uso de la recursividad solo en caso que:

* La solución no pueda ser fácilmente expresada con iteración.
* La eficiencia de la recursividad sea positiva.

**ESTRUCTURAS DINAMICAS**

***¿Qué es una estructura de datos?***

Se trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays.

- Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, ficheros, etc.

***Listas vinculadas***

Conjunto de datos de acceso secuencial.

La lista vinculada es la estructura dinámica más simple, consiste de un conjunto de variables anónimas (creadas con malloc, y que no tienen nombre) que se vinculan unas a otras a través de un puntero.

Gráficamente, una lista (de cualquier tipo de dato) sería algo así:

dato

dato

dato

dato

NULL

lista

// variable

// estática

A cada uno de los elementos de la lista vinculada le vamos a llamar NODO. El acceso al primer nodo de la lista se hace a través de una variable (con nombre) de tipo puntero. El final de la lista se determina con NULL. Cada nodo de la lista contiene campos con información y uno con la dirección de memoria del siguiente nodo. El acceso a cada nodo de la lista se hace en forma secuencial, a través del campo que contiene la dirección del siguiente, así cada nodo, nos proporciona la dirección donde se encuentra el que le sigue.

**Operaciones básicas del manejo de listas:** Para poder manipular las listas en forma adecuada, lo haremos a través de un conjunto de funciones que se explicarán a continuación.

|  |  |
| --- | --- |
| typedef struct {   persona dato;  struct nodo \* siguiente;  } nodo; | typedef struct {   char nombre[20];   int edad;  } persona; |

nodo **\* lista; // variable estática definida en el main()**

*donde:*

***dato*** *es una variable de tipo persona que contiene información.*

***siguiente*** *es un campo puntero, que contiene la dirección de memoria de otra estructura similar.*

***lista*** *es una variable estática (la declaramos a nivel de main) que contiene la dirección de memoria del primer nodo de la lista.*

**Inicializar la lista**

Inicializa el puntero al primer nodo de la lista con el valor NULL.

|  |
| --- |
| nodo \* inicLista() {   return NULL;  } |

**Crear un nuevo nodo para luego agregar a la lista**

Función que recibe como parámetro los campos de información para la estructura, los agrega a la misma y retorna un puntero al NODO creado.

nodo \* crearNodo (persona dato) {

*//crea un puntero de tipo nodo*

nodo \* aux = (nodo\*) malloc(sizeof(nodo));

*//asigna valores a sus campos de información*

aux->dato = dato;

*//asigna valor NULL al campo que contiene la dirección de memoria del //siguiente nodo*

aux->siguiente = NULL; *//retorna la dirección de memoria del nuevo nodo, que deberá ser asignada a una variable de tipo "puntero a nodo".*return aux;

}

**Agregar un nodo ya creado al principio de la lista**

Función que recibe como parámetro una variable puntero al comienzo de la lista y otra variable puntero a un nuevo nodo. Agrega este nuevo nodo al comienzo de la lista y retorna la nueva posición de memoria.

|  |
| --- |
| nodo \* agregarPpio (nodo \* lista, nodo \* nuevoNodo) {  //si la lista está vacía, ahora apuntará al nuevo nodo.  if(lista == NULL) {   lista = nuevoNodo; }else  //si la lista no está vacía, inserta el nuevo nodo al comienzo de la //misma, y el viejo primer nodo pasa a ser el segundo de la lista.  {   nuevoNodo->siguiente = lista;  lista = nuevoNodo;  } return lista;  } |

Antes de la inserción:

Juan

18

Sole

30

María

45

Pepe

54

NULL

lista

Lucas

15

nuevo

NULL

Después de la inserción:

Juan

18

Sole

30

María

45

Pepe

54

NULL

lista

Lucas

15

**NOTA IMPORTANTE:** debido al tipo de pasaje de parámetros (por valor), el contenido del parámetro lista (*un puntero al primer nodo*) no puede ser alterado una vez que la función ha terminado, o sea, el parámetro actual siempre conserva su valor. No confundirse, lo que sí se puede alterar es **\*lista**.

*Lo que hacíamos antes, era cambiar el contenido de una variable apuntada, y no el contenido de la variable apuntadora (que contiene una dirección de memoria).*

**Buscar el último nodo**

Función que nos retorna la dirección de memoria del último nodo de la lista. Esta función solamente se invocará si la lista no está vacía (lista != NULL)

|  |
| --- |
| nodo \* buscarUltimo(nodo \* lista) {   nodo \* seg = lista;  if(seg != NULL)   while(seg->siguiente != NULL) {   seg = seg->siguiente;   }  return seg;  } |

*donde:*

*la variable local* ***seg*** *contiene al principio, la dirección de memoria de primer nodo de la lista. Luego, al ir ejecutándose la instrucción* ***seg = seg->siguiente;*** *dentro del ciclo de repetición,* ***seg*** *va tomando la dirección del nodo siguiente. De esta forma se avanza en el recorrido secuencial de la lista.*

Utilizamos una variable local **seg** para recorrer la lista en vez del mismo parámetro **lista** para no alterar el contenido del mismo, aunque, debido al tipo de pasaje de parámetros (por valor) el contenido de **lista** no podría ser alterado nunca. Sí se puede alterar el contenido de **\*lista.** Hacerlo de esta forma es una buena disciplina de programación que evita posibles errores.

Notar que en el ciclo de repetición, la condición es **(seg->siguiente != NULL)**, esto es porque debemos detener nuestro avance al llegar al último nodo, o sea, al nodo cuyo siguiente es NULL. Por este motivo evaluamos al siguiente y no al propio nodo.

**Buscar un nodo según el valor de un campo**

Función que retorna la dirección de memoria de un nodo que contiene el campo **dato.nombre** con el mismo valor que el parámetro **nombre**

nodo \* buscarNodo(nodo \* lista, char nombre[20]) {   
 //busca un nodo por nombre y retorna su posición de memoria   
 //si no lo encuentra retorna NULL.   
  
 nodo \* seg; //apunta al nodo de la lista que está siendo procesado  
 seg = lista; //con esto evito cambiar el valor de la variable   
 //lista, que contiene un puntero al primer nodo de la   
 //lista vinculada

while ((seg != NULL) && ( strcmp(nombre, seg->dato.nombre)!=0 )) {   
//busco mientras me quede lista por recorrer y no haya encontrado el nombre   
 seg=seg->siguiente; //avanzo hacia el siguiente nodo.   
 }  
//en este punto puede haber fallado alguna de las dos condiciones   
//del while. si falla la primera es debido a que no encontró lo   
//que buscaba (seg es NULL), si falla la segunda es debido a que se   
//encontró el nodo buscado.

return seg;   
}

**Agregar un nuevo nodo al final de la lista**

Esta función nos permite agregar al final de la lista un nuevo nodo.

|  |
| --- |
| nodo \* agregarFinal(nodo \* lista, nodo \* nuevoNodo) {  if(lista == NULL) {   lista = nuevoNodo;   } else {   nodo \* ultimo = buscarUltimo(lista);   ultimo->siguiente = nuevoNodo;   }  return lista;  } |

Notar que se invoca a la función **buscarUltimo(nodo \*)** ya definida previamente.

La variable local **ultimo** contiene la dirección de memoria del último nodo de la lista, y a través de la misma, accedo a su campo **siguiente**, actualizando su valor. Esto agrega un nodo más a la lista.

El retorno de la función siempre es el mismo, excepto la primera vez, cuando la lista está vacía y se le agrega el primer elemento.

**Agregar un nodo nuevo manteniendo el orden (según un campo)**

Esta función agrega un nodo nuevo a la lista, insertándolo en el lugar correspondiente según un orden preestablecido por un campo. Por ejemplo: insertar un nuevo nodo en la lista ordenada por el campo nombre. Retorna un puntero al primer nodo de la lista, que se modifica solamente cuando el nuevo nodo se inserta en la primera posición.

|  |
| --- |
| nodo \* agregarEnOrden(nodo \* lista, nodo \* nuevoNodo) {  // agrega un nuevo nodo a la lista manteniendo el orden.  //si la lista está vacía agrego el primer elemento.   if(lista == NULL) {   lista = nuevoNodo;   }else {   //si el nuevo elemento es menor que el primero de la lista,  //agrego al principio  if(strcmp(nuevoNodo->dato.nombre,lista->dato.nombre)<0){   lista = agregarPpio(lista, nuevoNodo);   } else {   //busco el lugar en donde insertar el nuevo elemento.   //necesito mantener la dirección de memoria del nodo anterior   //al nodo que tiene un nombre mayor al del nuevo nodo.   nodo \* ante = lista;   nodo \* seg = lista->siguiente;   while((seg != NULL)  &&(strcmp(nuevoNodo->dato.nombre,seg->dato.nombre)>0)) {   ante = seg;   seg = seg->siguiente;   }  // inserto el nuevo nodo en el lugar indicado.   nuevoNodo->siguiente = seg;  ante->siguiente = nuevoNodo;   }  }   return lista;  } |

ejemplo: agregar el nodo “Luis”

Juan

18

Laura

30

María

45

Pepe

54

NULL

lista

seg

ante

Luis

35

NULL

nuevoNodo

Juan

18

Laura

30

María

45

Pepe

54

NULL

lista

seg

ante

Luis

35

*ante->siguiente=nuevoNodo*

*nuevoNodo->siguiente=seg*

***Listas doblemente vinculadas***

Estructura de datos que consiste en un conjunto de nodos enlazados secuencialmente. Cada nodo contiene dos campos, llamados enlaces, que son referencias al nodo siguiente y al anterior en la secuencia de nodos. El enlace al nodo anterior del primer nodo y el enlace al nodo siguiente del último nodo, apuntan a un tipo de nodo que marca el final de la lista, normalmente un puntero null, para facilitar el recorrido de la lista.

El doble enlace de los nodos permite recorrer la lista en cualquier dirección. Mientras que agregar o eliminar un nodo en una lista doblemente enlazada requiere cambiar más enlaces que en estas mismas operaciones en una lista simplemente enlazada, las operaciones son más simples porque no hay necesidad de mantener guardado el nodo anterior durante el recorrido, ni necesidad de recorrer la lista para hallar el nodo anterior, la referencia al nodo que se quiere eliminar o insertar es lo único necesario.

PASAJES